

Tema 1

Compiladores, Linguagens e Gramáticas

Introdução

Compiladores, 2º semestre 2022-2023

Enquadramento

Linguagens de programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do código

Linguagens: Definição como Conjunto

Conceito básicos e terminologia

Operações sobre palavras

Operações sobre linguagens

Introdução às gramáticas

Hierarquia de Chomsky

Autómatos

Máquina de Turing

Autómatos linearmente limitados

Autómatos de pilha

Miguel Oliveira e Silva, Artur Pereira
DETI, Universidade de Aveiro

Enquadramento

Linguagens de
programação

**Compiladores:
Introdução**

**Estrutura de um
Compilador**

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

**Implementação de um
Compilador**

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

**Linguagens: Definição
como Conjunto**

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

**Introdução às
gramáticas**

Hierarquia de Chomsky
Autómatos

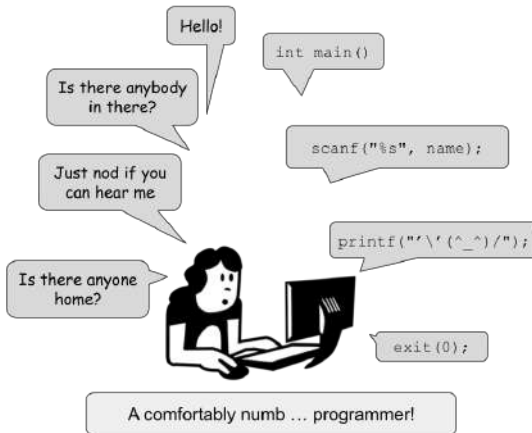
Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Enquadramento

Enquadramento

- Nesta disciplina vamos falar sobre **linguagens** – o que são e como as podemos definir – e sobre **compiladores** – ferramentas que as reconhecem e que permitem realizar acções como consequência desse processo.



Enquadramento

Linguagens de programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical
Análise Sintáctica
Análise Semântica
Síntese

Implementação de um Compilador

Análise léxica
Análise sintáctica
Análise semântica
Síntese: interpretação do código


Linguagens: Definição como Conjunto

Conceito básicos e terminologia
Operações sobre palavras
Operações sobre linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos
Máquina de Turing
Autómatos linearmente limitados
Autómatos de pilha

Enquadramento (2)

- Se tivesse que definir **linguagem** como é que o faria?
 - Permite **expressar, transmitir e receber ideias**.
 - **Comunicação** entre pessoas ou seres vivos em geral.
 - Inclui a comunicação com e entre máquinas.
 - Requer várias entidade comunicantes, um código e regras para que a comunicação seja inteligível.
- Necessário: codificação e um conjunto de regras comuns, e interlocutores que as conheçam.
- Vejamos, como exemplo, algumas linguagens naturais.
- Palavras diferentes, em linguagens diferentes, podem ter o mesmo significado:
 - “adeus”, “goodbye”, “au revoir”,

- Por outro lado, também existem palavras iguais com significados diferentes (dependendo do contexto):
 - *morro, rio, caminho,*

Enquadramento

Linguagens de
programação

Compiladores:
Introdução

Estrutura de um
Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um
Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição
como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às
gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Enquadramento (3)

- Diferentes linguagens podem utilizar símbolos (letras ou caracteres) diferentes, ou partilhar muitos deles.
- Compreensão de uma palavra é feita letra a letra, mas isso não acontece com um texto.
- Assim, podemos ver uma linguagem natural como o português como sendo composta por mais do que uma linguagem:
 - Uma que explicita as regras para construir palavras a partir do alfabeto das letras:

$a + d + e + u + s \rightarrow \text{adeus}$

- E outra que contém as regras gramaticais para construir frases a partir das palavras resultantes da linguagem anterior:

$\text{adeus} + e + \text{até} + \text{amanhã} \rightarrow \text{adeus e até amanhã}$

Neste caso o alfabeto deixa de ser o conjunto de letras e passa a ser o conjunto de palavras existentes.

Enquadramento

Linguagens de
programação

Compiladores:
Introdução

Estrutura de um
Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um
Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição
como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às
gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Enquadramento (4)

- Inerente às linguagens, é a necessidade de decidir se uma sequência de símbolos do alfabeto é válida.
 - **correcto:**
 $a + d + e + u + s \rightarrow \text{adeus}$
 $\text{adeus} + e + \text{até} + \text{amanhã} \rightarrow \text{adeus e até amanhã}$
 - **incorrecto:**
 $e + d + u + a + s \rightarrow \text{edues}$
 $\text{até} + \text{adeus} + \text{amanhã} + e \rightarrow \text{até adeus amanhã e}$
- Só sequências válidas é que permitem uma comunicação correcta.
- Por outro lado, essa comunicação tem muitas vezes um efeito.
- Seja esse efeito uma resposta à mensagem inicial, ou o despoletar de uma qualquer acção.

Enquadramento

Linguagens de
programação

Compiladores:
Introdução

Estrutura de um
Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um
Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição
como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às
gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Linguagens de programação

- As linguagens de comunicação com computadores – designadas por linguagens de programação – partilham todas estas características.
- Diferem, no facto de não poderem ter nenhuma **ambiguidade**, e de as acções despoletadas serem muitas vezes a mudança do estado do sistema computacional, podendo este estar ligado a entidades externas como sejam outros computadores, pessoas, sistemas robóticos, máquinas de lavar, etc..
- Vamos ver que podemos definir linguagens de programação por estruturas formais bem comportadas.
- Para além disso, veremos também que essas definições nos ajudam a implementar acções interessantes.

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

Enquadramento

Linguagens de programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do código

Linguagens: Definição como Conjunto

Conceito básicos e terminologia

Operações sobre palavras

Operações sobre linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente limitados
Autómatos de pilha

Enquadramento

Linguagens de
programação

**Compiladores:
Introdução**

Estrutura de um
Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um
Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição
como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às
gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Compiladores: Introdução

Enquadramento

Linguagens de
programação

**Compiladores:
Introdução**

**Estrutura de um
Compilador**

Análise Lexical
Análise Sintáctica
Análise Semântica
Síntese

**Implementação de um
Compilador**

Análise léxica
Análise sintáctica
Análise semântica
Síntese: interpretação do
código

**Linguagens: Definição
como Conjunto**

Conceito básicos e
terminologia
Operações sobre palavras
Operações sobre
linguagens

**Introdução às
gramáticas**

Hierarquia de Chomsky
Autómatos
Máquina de Turing
Autómatos linearmente
limitados
Autómatos de pilha

Compiladores

Compreensão, interpretação e/ou tradução automática de linguagens.

Compiladores (Processadores de Linguagens)

- Os **compiladores** são programas que permitem:
 - decidir sobre a correcção de sequências de símbolos do respectivo alfabeto;
 - despoletar acções resultantes dessas decisões.
- Frequentemente, os compiladores “limitam-se” a fazer a tradução entre linguagens.



- 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*).
- Nestes casos, na inexistência de erros, é gerado um programa composto por código executável directa ou indirectamente pelo sistema computacional:



Exemplo: Java *bytecode*

```
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
}
```

Exemplo 2: Calculadora

- Código fonte:

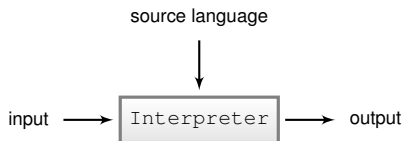
```
1+2*3:4
```

- Uma possível compilação para 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);  
    }  
}
```

Compiladores: Processadores de Linguagens (2)

- Uma variante possível consiste num **interpretador**:



- Neste caso a execução é feita instrução a instrução.
- `Python` e `bash` são exemplos de linguagens interpretadas.
- Existem também aproximações híbridas 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`.
- Em geral os compiladores processam código fonte em formato de *texto*, havendo uma grande variedade no formato do código gerado (texto, binário, interpretado, ...).

Exemplo: Calculadora

- Código fonte:



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

- Uma possível interpretação:

```
2.5
```

Enquadramento

Linguagens de
programação

**Compiladores:
Introdução**

**Estrutura de um
Compilador**

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

**Implementação de um
Compilador**

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

**Linguagens: Definição
como Conjunto**

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

**Introdução às
gramáticas**

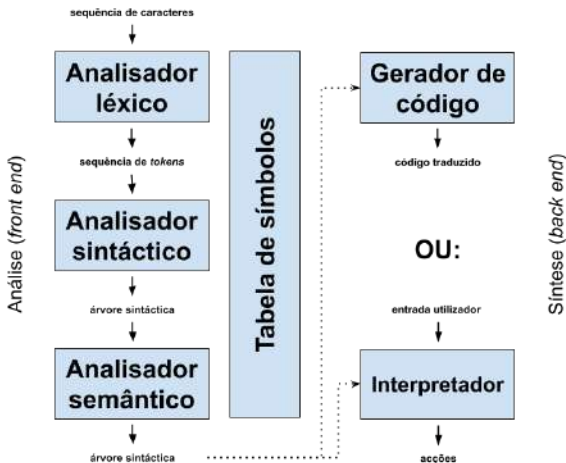
Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Estrutura de um Compilador

Estrutura de um Compilador



Compiladores,
Linguagens e
Gramáticas

Enquadramento

Linguagens de
programação

Compiladores:
Introdução

Estrutura de um
Compilador

Análise Lexical
Análise Sintáctica
Análise Semântica
Síntese

Implementação de um
Compilador

Análise léxica
Análise sintáctica
Análise semântica
Síntese: interpretação do
código

Linguagens: Definição
como Conjunto

Conceito básicos e
terminologia
Operações sobre palavras
Operações sobre
linguagens

Introdução às
gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados
Autómatos de pilha

Estrutura de um Compilador (2)

- 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 sintáctica**: composição de *tokens* numa estrutura sintáctica adequada.
 - **análise semântica**: verificação se a estrutura sintáctica tem significado.
- As acções consistem na geração do programa na linguagem destino e podem envolver também diferentes fases de geração de código e optimização.

- Conversão da sequência de caracteres de entrada numa sequência de elementos lexicais.
- Esta estratégia simplifica brutalmente a gramática da análise sintáctica, e permite uma implementação muito eficiente do analisador léxico (mais tarde veremos em detalhe porquê).
- Cada elemento lexical pode ser definido por um tuplo com uma identificação do elemento e o seu valor (o valor pode ser omitido quando não se aplica):

```
<token_name , attribute_value >
```

- Exemplo 1:

```
pos = pos + vel * 5;
```

pode ser convertido pelo analisador léxico (*scanner*) em:

```
<id , pos> <=> <id , pos> <+> <id , vel> <*> <int , 5> <;>
```

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Análise Lexical (2)

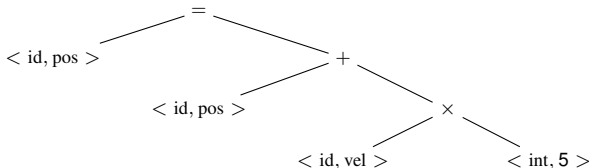
- Em geral os espaços em branco, e as mudanças de linha e os comentários não são relevantes nas linguagens de programação, pelo que podem ser eliminados pelo analisador lexical.
- Exemplo 2: esboço de linguagem de processamento geométrico:

```
distance ( 0 , 0 ) ( 4 , 3 )
```

pode ser convertido pelo analisador léxico (*scanner*) em:

```
<distance> <( > <num,0> <,> <num,0> <)>  
<( > <num,4> <,> <num,3> <)>
```

- Após a análise lexical segue-se a chamada análise sintáctica (*parsing*), onde se verifica a conformidade da sequência de elementos lexicais com a estrutura sintáctica da linguagem.
- Nas linguagens que se pretende sintacticamente processar, podemos sempre fazer uma aproximação à sua estrutura formal através duma representação tipo *árvore*.
- Para esse fim é necessário uma *gramática* que especifique a estrutura desejada (voltaremos a este problema mais à frente).
- No exemplo 1 (`pos = pos + vel * 5;`):



Enquadramento

Linguagens de programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica
Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do código

Linguagens: Definição como Conjunto

Conceito básicos e terminologia

Operações sobre palavras

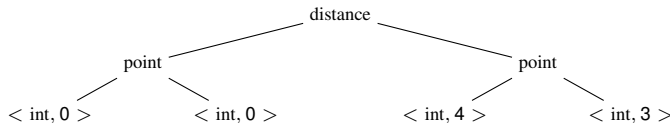
Operações sobre linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente limitados
Autómatos de pilha

- No exemplo 2 (`distance (0 , 0) (4 , 3)`):



- Chama-se a atenção para duas características das árvores sintáticas:
 - não incluem alguns elementos lexicais (que apenas são relevantes para a estrutura formal);
 - definem sem ambiguidade a ordem das operações (havemos de voltar a este problema).

- A parte final do *front end* do compilador é a *análise semântica*.
- Nesta fase são verificadas, tanto quando possível, restrições que não é possível (ou sequer desejável) que sejam feitas nas duas fases anteriores.
- Por 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 objecto 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).

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados
Autómatos de pilha

Análise Semântica (2)

- Utiliza a árvore sintáctica da análise sintáctica 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).

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

- 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.

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Geração de código: exemplo

- 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*.
- Para o exemplo 1 (`pos = pos + vel * 5;`) poderíamos ter:

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

- Este código poderia depois ser otimizado 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
```

Linguagens: Definição como Conjunto

**Compiladores,
Linguagens e
Gramáticas**

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

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 subjectividade e ambiguidade, as linguagens de programação requerem total objectividade.

Compiladores,
Linguagens e
Gramáticas

Enquadramento

Linguagens de
programação

Compiladores:
Introdução

Estrutura de um
Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um
Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição
como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às
gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Linguagens (2)

- Como definir linguagens de forma sintética e objectiva?
- 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**.

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Conceito básicos

Enquadramento

Linguagens de
programação

**Compiladores:
Introdução**

**Estrutura de um
Compilador**

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

**Implementação de um
Compilador**

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

**Linguagens: Definição
como Conjunto**

**Conceito básicos e
terminologia**

Operações sobre palavras

Operações sobre
linguagens

**Introdução às
gramáticas**

Hierarquia de Chomsky

Autómatos

Máquina de Turing

Autómatos linearmente
limitados

Autómatos de pilha

Conceito 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.
- Por exemplo:
$$\{n \mid n \in \mathbb{N} \wedge n \leq 9\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Conceito básicos e terminologia (2)

- Um **símbolo** (ou **letra**) é a unidade atômica (indivisível) das linguagens.
- Em linguagens assentes em texto, um símbolo será um carácter.
- Um **alfabeto** é um conjunto finito não vazio de símbolos.
- Por exemplo:
 - $A = \{0, 1\}$ é o alfabeto dos algarismos binários.
 - $A = \{0, 1, \dots, 9\}$ é o alfabeto dos algarismos decimais.
- Uma **palavra** (*string* ou cadeia) é uma sequência de símbolos sobre um dado alfabeto A .

$$U = a_1 a_2 \cdots a_n, \quad \text{com } a_i \in A \wedge n \geq 0$$

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Conceito básicos e terminologia (3)

- Por exemplo:
 - $A = \{0, 1\}$ é o alfabeto dos algarismos binários.
01101, 11, 0
 - $A = \{0, 1, \dots, 9\}$ é o alfabeto dos algarismos decimais.
2016, 234523, 999999999999999, 0
 - $A = \{0, 1, \dots, 0, a, b, \dots, z, @, \dots\}$
mos@ua.pt, Bom dia!

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky

Autómatos

Máquina de Turing

Autómatos linearmente
limitados

Autómatos de pilha

Conceito básicos e terminologia (4)

- A **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 .
- Por 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

Conceito básicos e terminologia (5)

- 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.
- Por exemplo:
 - $\{0, 1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$
 - $\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}^* = \{\epsilon, \clubsuit, \diamondsuit, \heartsuit, \spadesuit, \clubsuit\diamondsuit, \dots\}$
- 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: $L \subseteq A^*$

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky

Autómatos

Máquina de Turing

Autómatos linearmente
limitados

Autómatos de pilha

Conceito básicos e terminologia (6)

- Exemplo de linguagens sobre o alfabeto $A = \{0, 1\}$
 - $L_1 = \{u \mid u \in A^* \wedge |u| \leq 2\} = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$
 - $L_2 = \{u \mid u \in A^* \wedge \forall_i u_i = 0\} = \{\varepsilon, 0, 00, 000, 0000, \dots\}$
 - $L_3 = \{u \mid u \in A^* \wedge u.\text{count}(1) \bmod 2 = 0\} = \{000, 11, 000110101, \dots\}$
 - $L_4 = \{\} = \emptyset$ (conjunto vazio)
 - $L_5 = \{\varepsilon\}$
 - $L_6 = A$
 - $L_7 = A^*$
- Note que $\{\}$, $\{\varepsilon\}$, A e 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, etc.



Operações sobre palavras

**Compiladores,
Linguagens e
Gramáticas**

Enquadramento

Linguagens de
programação

**Compiladores:
Introdução**

**Estrutura de um
Compilador**

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

**Implementação de um
Compilador**

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

**Linguagens: Definição
como Conjunto**

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

**Introdução às
gramáticas**

Hierarquia de Chomsky

Autómatos

Máquina de Turing

Autómatos linearmente
limitados

Autómatos de pilha

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*):

$$u : \{1, 2, \dots, n\} \rightarrow A, \quad \text{com } n = |u|$$

em que u_i representa o *i*ésimo 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

$$u = \{u_1, u_2, \dots, u_n\} \implies u^R = \{u_n, \dots, u_2, u_1\}$$

Operações sobre palavras (2)

- 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)
 - $u \neq \varepsilon \wedge v \neq \varepsilon \wedge u \neq v \implies u.v \neq v.u$ (não comutativo)
- A **potência** de ordem n , com $n \geq 0$, de uma palavra u denota-se por u^n e representa a concatenação de n réplicas de u , ou seja, $\underbrace{uu \cdots u}_{n \times}$.
- $u^0 = \varepsilon$

Operações sobre linguagens

**Compiladores,
Linguagens e
Gramáticas**

Enquadramento

Linguagens de
programação

Compiladores:
Introdução

Estrutura de um
Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um
Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição
como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

**Operações sobre
linguagens**

Introdução às
gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Operações sobre linguagens: reunião

- A **reunião** de duas linguagens L_1 e L_2 denota-se por $L_1 \cup L_2$ e é dada por:

$$L_1 \cup L_2 = \{u \mid u \in L_1 \vee u \in L_2\}$$

- Por exemplo, se definirmos as linguagens L_1 e L_2 sobre o alfabeto $A = \{a, b\}$:

$$L_1 = \{u \mid u \text{ começa por } a\} = \{a w \mid w \in A^*\}$$

$$L_2 = \{u \mid u \text{ termina com } a\} = \{w a \mid w \in A^*\}$$

- qual será o resultado da reunião destas linguagens?

$$L = L_1 \cup L_2 = ?$$

- Resposta:

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

Operações sobre linguagens: intercepção

- A **intercepção** de duas linguagens L_1 e L_2 denota-se por $L_1 \cap L_2$ e é dada por:

$$L_1 \cap L_2 = \{u \mid u \in L_1 \wedge u \in L_2\}$$

- Por exemplo, se definirmos as linguagens L_1 e L_2 sobre o alfabeto $A = \{a, b\}$:

$$L_1 = \{u \mid u \text{ começa por } a\} = \{a w \mid w \in A^*\}$$

$$L_2 = \{u \mid u \text{ termina com } a\} = \{w a \mid w \in A^*\}$$

- qual será o resultado da intercepção destas linguagens?

$$L = L_1 \cap L_2 = ?$$

- Resposta:

$$L = \{a w a \mid w \in A^*\} \cup \{a\}$$

Operações sobre linguagens: diferença

- A **diferença** de duas linguagens L_1 e L_2 denota-se por $L_1 - L_2$ e é dada por:

$$L_1 - L_2 = \{u \mid u \in L_1 \wedge u \notin L_2\}$$

- Por exemplo, se definirmos as linguagens L_1 e L_2 sobre o alfabeto $A = \{a, b\}$:

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

$$L_2 = \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\}$$

- qual será o resultado da diferença destas linguagens?

$$L = L_1 - L_2 = ?$$

- Resposta:

$$L = \{awx \mid w \in A^* \wedge x \in A \wedge x \neq a\}$$

- ou:

$$L = \{awb \mid w \in A^*\}$$

Operações sobre linguagens: complementação

- A **complementação** da linguagem L denota-se por \bar{L} e é dada por:

$$\bar{L} = A^* - L = \{u \mid u \notin L\}$$

- Por exemplo, se definirmos a linguagem L_1 sobre o alfabeto $A = \{a, b\}$:

$$L_1 = \{u \mid u \text{ começa por } a\} = \{a w \mid w \in A^*\}$$

- qual será o resultado da complementação desta linguagem?

$$L = \bar{L_1} = ?$$

- Resposta:

$$L = \{x w \mid w \in A^* \wedge x \in A \wedge x \neq a\} \cup \{\varepsilon\}$$

- ou:

$$L = \{b w \mid w \in A^*\} \cup \{\varepsilon\}$$

Operações sobre linguagens: concatenação

- A **concatenação** de duas linguagens L_1 e L_2 denota-se por $L_1.L_2$ e é dada por:

$$L_1.L_2 = \{uv \mid u \in L_1 \wedge v \in L_2\}$$

- Por exemplo, se definirmos as linguagens L_1 e L_2 sobre o alfabeto $A = \{a, b\}$:

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

$$L_2 = \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\}$$

- qual será o resultado da concatenação destas linguagens?

$$L = L_1.L_2 = ?$$

- Resposta:

$$L = \{aw a \mid w \in A^*\}$$

Operações sobre linguagens: potenciação

- A **potência** de ordem n da linguagem L denota-se por L^n e é definida indutivamente por:

$$\begin{aligned}L^0 &= \{\varepsilon\} \\ L^{n+1} &= L^n.L\end{aligned}$$

- Por exemplo, se definirmos a linguagem L_1 sobre o alfabeto $A = \{a, b\}$:

$$L_1 = \{u \mid u \text{ começa por } a\} = \{a w \mid w \in A^*\}$$

- qual será o resultado da potência de ordem 2 desta linguagem?

$$L = L_1^2 = ?$$

- Resposta:

$$L = \{a w_1 a w_2 \mid w_1, w_2 \in A^*\}$$

Operações sobre linguagens: fecho de Kleene

- O **fecho de Kleene** da linguagem L denota-se por L^* e é dado por:

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

- Por exemplo, se definirmos a linguagem L_1 sobre o alfabeto $A = \{a, b\}$:

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

- qual será o fecho de Kleene desta linguagem?

$$L = L_1^* = ?$$

- Resposta:

$$L = L_1 \cup \{\varepsilon\}$$

- Note que para $n > 1$ $L_1^n \subset L_1$

Operações sobre linguagens: 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 respectivamente sobre os alfabetos A_1 e A_2 , então o alfabeto resultante da aplicação duma qualquer operação binária sobre as linguagens é: $A_1 \cup A_2$

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical
Análise Sintáctica
Análise Semântica
Síntese

Implementação de um Compilador

Análise léxica
Análise sintáctica
Análise semântica
Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia
Operações sobre palavras

Operações sobre linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos
Máquina de Turing
Autómatos linearmente
limitados
Autómatos de pilha

Introdução às gramáticas

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

- 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 objectividade 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**.

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky

Autómatos

Máquina de Turing

Autómatos linearmente
limitados

Autómatos de pilha

Gramáticas (2)

- Por exemplo, em português a frase “O cão ladra” pode ser gramaticalmente descrita por:

frase	→	sujeito predicado
sujeito	→	artigo substantivo
predicado	→	verbo
artigo	→	O Um
substantivo	→	cão lobo
verbo	→	ladra 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:
 - T é um conjunto finito não vazio designado por alfabeto terminal, onde cada elemento é designado por símbolo **terminal**;
 - N é um conjunto finito não vazio, disjunto de T ($N \cap T = \emptyset$), cujos elementos são designados por símbolos **não terminais**;
 - $S \in N$ é um símbolo não terminal específico designado por **símbolo inicial**;
 - 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.

Gramáticas: exemplos

- Formalmente, a gramática anterior será:

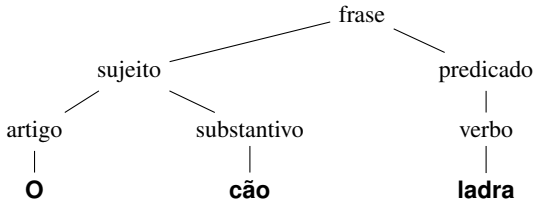
$$G = (\{\mathbf{O, Um, cão, lobo, ladra, uiva}\}, \\ \{\text{frase, sujeito, predicado, artigo, substantivo, verbo}\}, \\ \text{frase}, P)$$

- P é constituído pelas regras já apresentadas:

frase \rightarrow sujeito predicado
sujeito \rightarrow artigo substantivo
predicado \rightarrow verbo
artigo \rightarrow **O | Um**
substantivo \rightarrow **cão | lobo**
verbo \rightarrow **ladra | uiva**

Gramáticas: exemplos (2)

- Podemos descrever a frase “O cão ladra” com a seguinte árvore (denominada sintáctica).



Gramáticas: exemplos (3)

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

$$S \rightarrow 0S$$

$$S \rightarrow 0A$$

$$A \rightarrow 0A1$$

$$A \rightarrow \varepsilon$$

- Qual será a linguagem definida por esta gramática?

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

Gramáticas: exemplos (4)

- Sendo $A = \{a, b\}$, defina uma gramática para a seguinte linguagem:

$$L_1 = \{a w \mid w \in A^*\}$$

- A gramática $G = (\{a, b\}, \{S, X\}, S, P)$, onde P é constituído pelas regras:

$$S \rightarrow aX$$

$$X \rightarrow aX$$

$$X \rightarrow bX$$

$$X \rightarrow \varepsilon$$

ou:

$$S \rightarrow aX$$

$$X \rightarrow aX \mid bX \mid \varepsilon$$

Gramáticas: exemplos (5)

- Sendo $A = \{0, 1\}$, defina uma gramática para a seguinte linguagem:

$$L_3 = \{u \mid u \in A^* \wedge u.\text{count}(1) \bmod 2 = 0\}$$

- A gramática $G = (\{0, 1\}, \{S, A\}, S, P)$, onde P é constituído pelas regras:

$$S \rightarrow S1S1S \mid A$$

$$A \rightarrow 0A \mid \varepsilon$$

Hierarquia de Chomsky

**Compiladores,
Linguagens e
Gramáticas**

Enquadramento

Linguagens de
programação

Compiladores:
Introdução

Estrutura de um
Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um
Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição
como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às
gramáticas

Hierarquia de Chomsky

Autómatos

Máquina de Turing

Autómatos linearmente
limitados

Autómatos de pilha

Autómatos finitos

- Restrições sobre α e 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 cada regra $\alpha \rightarrow \beta$ de P obedece a $|\alpha| \leq |\beta|$ (com a exceção de também poder existir a produção vazia: $S \rightarrow \varepsilon$).
 - 3 G será do **tipo-2**, ou gramática **independente, ou livre, do contexto**, se cada regra $\alpha \rightarrow \beta$ 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: $A \rightarrow cB$, $A \rightarrow c$ ou $A \rightarrow \varepsilon$, 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).

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky

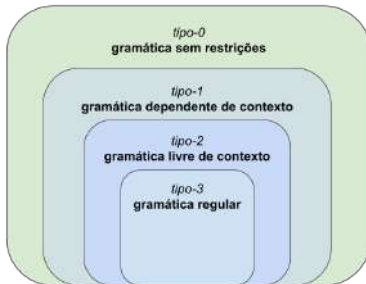
Autómatos

Máquina de Turing

Autómatos linearmente
limitados

Autómatos de pilha

Hierarquia de Chomsky (2)



- 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.

Enquadramento

Linguagens de programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical
Análise Sintáctica
Análise Semântica
Síntese

Implementação de um Compilador

Análise léxica
Análise sintáctica
Análise semântica
Síntese: interpretação do código

Linguagens: Definição como Conjunto

Conceito básicos e terminologia
Operações sobre palavras
Operações sobre linguagens

Introdução às gramáticas

Hierarquia de Chomsky

Autómatos
Máquina de Turing
Autómatos linearmente limitados
Autómatos de pilha

Hierarquia de Chomsky (3)

- 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 **autômatos linearmente limitados** processam gramáticas dependentes do contexto (tipo-1);
 - 3 Os **autômatos de pilha** processam gramáticas independentes do contexto (tipo-2);
 - 4 Os **autômatos finitos** processam gramáticas regulares (tipo-3).

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky

Autômatos

Máquina de Turing

Autômatos linearmente
limitados

Autômatos de pilha

Autômatos finitos

Autómatos

Compiladores, Linguagens e Gramáticas

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky

Autómatos

Máquina de Turing

Autómatos linearmente
limitados

Autómatos de pilha

Autómatos finitos

Máquina de Turing

- (Alan Turing, 1936)
- Modelo abstracto 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.

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

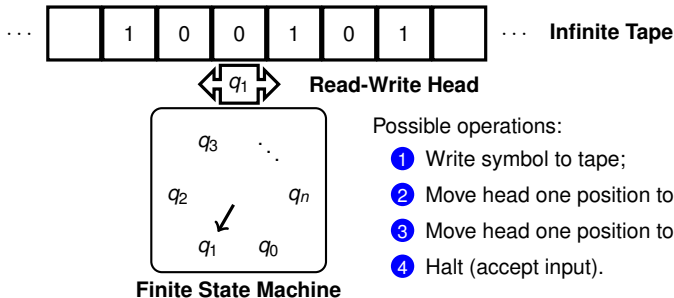
Hierarquia de Chomsky
Autómatos

Máquina de Turing

Autómatos linearmente
limitados

Autómatos de pilha

Máquina de Turing (2)



Possible operations:

- 1 Write symbol to tape;
- 2 Move head one position to the right;
- 3 Move head one position to the left;
- 4 Halt (accept input).

- A máquina de estados finita (FSM) tem acesso ao símbolo actual e decide a próxima acção a ser realizada.
- A acção consiste na transição de estado e qual a operação sobre a fita.
- Se não for possível nenhuma acção, a entrada é rejeitada.

Máquina de Turing: 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.

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing

Autómatos linearmente
limitados

Autómatos de pilha

Máquina de Turing: exemplo (2)

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

Estado	Entrada	
	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

- $E_1 \dots 0111101110 \dots \rightarrow E_1 \dots 0111101110 \dots \xrightarrow{*} E_2 \dots 0111101110 \dots \rightarrow$
 $E_3 \dots 0111111110 \dots \rightarrow E_3 \dots 0111111110 \dots \xrightarrow{*} E_3 \dots 0111111110 \dots \rightarrow$
 $E_4 \dots 0111111110 \dots \rightarrow E_5 \dots 0111111100 \dots \rightarrow E_5 \dots 0111111100 \dots \rightarrow$
 $E_6 \dots 0111111000 \dots \rightarrow E_7 \dots 0111111000 \dots \xrightarrow{*} E_7 \dots 0111111000 \dots$

Autômatos linearmente limitados

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

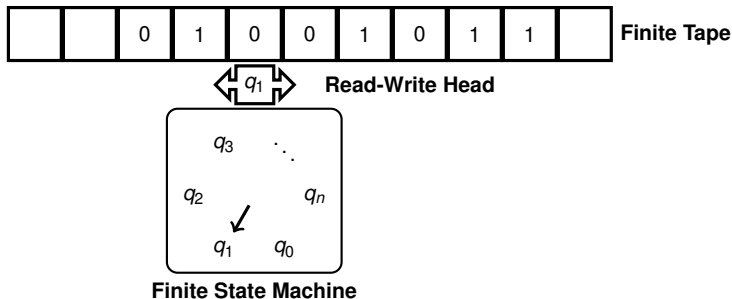
Hierarquia de Chomsky

Autômatos

Máquina de Turing

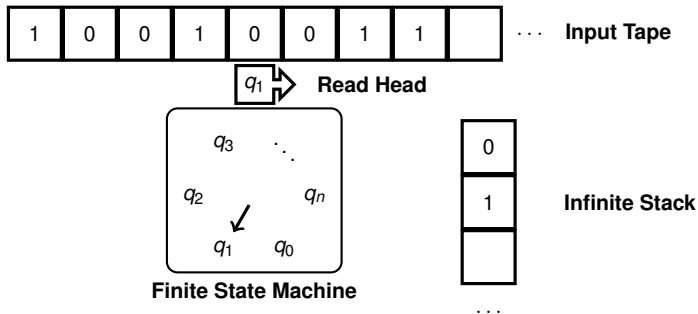
Autômatos linearmente
limitados

Autômatos finitos



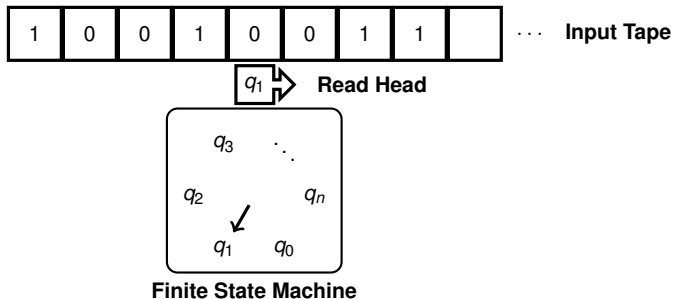
- Diferem das MT pela finitude da fita.

Autómatos de pilha



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

Autómatos finitos



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

Enquadramento

Linguagens de
programação

Compiladores: Introdução

Estrutura de um Compilador

Análise Lexical

Análise Sintáctica

Análise Semântica

Síntese

Implementação de um Compilador

Análise léxica

Análise sintáctica

Análise semântica

Síntese: interpretação do
código

Linguagens: Definição como Conjunto

Conceito básicos e
terminologia

Operações sobre palavras

Operações sobre
linguagens

Introdução às gramáticas

Hierarquia de Chomsky
Autómatos

Máquina de Turing
Autómatos linearmente
limitados

Autómatos de pilha

Tema 2

ANTLR4

Introdução, Estrutura, Aplicação

Compiladores, 2^o semestre 2022-2023

Miguel Oliveira e Silva, Artur Pereira
DETI, Universidade de Aveiro

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de
gramáticas

Especificação de
gramáticas

ANTLR4: Estrutura
léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras
léxicas

Padrões léxicos típicos

Operador léxico "não
ganancioso"

ANTLR4: Estrutura
sintáctica

Secção de *tokens*

Ações no preâmbulo da
gramática

ANTLR4: Regras
sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Apresentação

Exemplos

*Hello**Expr*Exemplo *figuras*Exemplo *visitor*Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Acções

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Acções no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Apresentação

ANTLR4: apresentação

- *ANother Tool for Language Recognition*
- O ANTLR é um gerador de processadores de linguagens que pode ser utilizado para ler, processar, executar ou traduzir linguagens.
- Desenvolvido por Terrence Parr:
 - 1988: tese de mestrado (YUCC)
 - 1990: PCCTS (ANTLR v1). Programado em C++.
 - 1992: PCCTS v 1.06
 - 1994: PCCTS v 1.21 e SORCERER
 - 1997: ANTLR v2. Programado em Java.
 - 2007: ANTLR v3 (LL (*), *auto-backtracking*, yuk!).
 - 2012: ANTLR v4 (ALL (*), *adaptive LL*, yep!).
- Terrence Parr, The Definitive ANTLR 4 Reference, 2012, The Pragmatic Programmers.
- Terrence Parr, Language Implementation Patterns, 2010, The Pragmatic Programmers.
- <https://www.antlr.org>

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4: instalação

- Descarregar o ficheiro `antlr4-install.zip` do *elearning*.
- Executar o *script* `./install.sh` no directório `antlr4-install`.
- Há dois ficheiros `jar` importantes:

`antlr-4.*-complete.jar` e `antlr-runtime-4.*.jar`

- O primeiro é *necessário* para **gerar** processadores de linguagens, e o segundo é o *suficiente* para os **executar**.
- Para experimentar basta:

```
java -jar antlr-4.*-complete.jar
```

ou:

```
java -cp .:antlr-4.*-complete.jar org.antlr.v4.Tool
```

- O ANTLR4 fornece uma ferramenta de teste muito flexível (implementada com o script `antlr4-test`):

```
java org.antlr.v4.gui.TestRig
```

- Podemos executar uma gramática sobre uma qualquer entrada, e obter a lista de *tokens* gerados, a árvore sintáctica (num formato tipo `LISP`), ou mostrar graficamente a árvore sintáctica.

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no préambulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

- Nesta disciplina são disponibilizados vários comandos (em `bash`) para simplificar (ainda mais) a geração de processadores de linguagens:

<code>antlr4</code>	compilação de gramáticas ANTLR-v4
<code>antlr4-test</code>	depuração de gramáticas
<code>antlr4-clean</code>	eliminação dos ficheiros gerados pelo ANTLR-v4
<code>antlr4-main</code>	geração da classe <code>main</code> para a gramática
<code>antlr4-visitor</code>	geração de uma classe <code>visitor</code> para a gramática
<code>antlr4-listener</code>	geração de uma classe <code>listener</code> para a gramática
<code>antlr4-build</code>	compila gramáticas e o código <code>java</code> gerado
<code>antlr4-run</code>	executa a classe <code>*Main</code> associada à gramática
<code>antlr4-jar-run</code>	executa um ficheiro jar (incluindo os jars do antlr)

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4: instalação (3)

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

`antlr4-javac` compilador java (jar do antlr no CLASSPATH)

`antlr4-java` máquina virtual java (jar do antlr no CLASSPATH)

`java-clean` eliminação dos ficheiros binários java

`view-javadoc` abre a documentação de uma classe java no browser

`st-groupfile2string` converte um STGroupFile num STGroupString

- Estes comandos estão disponíveis no *elearning* e fazem parte da instalação automática.

Exemplos

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4: Hello

- ANTLR4:



- Exemplo:

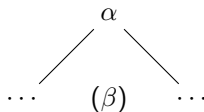
```
// (this is a line comment)
grammar Hello ;           // Define a grammar called Hello
// parser (first letter in lower case) :
r : 'hello' ID ;         // match keyword hello followed by an identifier
// lexer (first letter in upper case) :
ID : [a-z]+ ;             // match lower-case identifiers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

- As duas gramáticas – lexical e sintáctica – são expressas com instruções com a seguinte estrutura:

$$\alpha : \beta;$$

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

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



- Podemos agora gerar o processador desta linguagem e experimentar a gramática utilizando o programa de teste do ANTLR4.

```
antlr4 Hello.g4
antlr4-javac Hello*.java
echo "hello compiladores" | antlr4-test Hello r -tokens
```

- Utilização:

```
antlr4-test [<Grammar> <rule>] [-tokens | -tree | -gui]
```

ANTLR4: Ficheiros gerados

- Executando o comando `antlr4` sobre esta gramática obtemos os seguintes ficheiros:

```
Hello.g4   $\xrightarrow{\text{antlr4}}$   HelloLexer.java  
                                   HelloLexer.tokens  
                                   Hello.tokens  
                                   HelloParser.java  
                                   HelloListener.java  
                                   HelloBaseListener.java
```

lexer

parser

syntax-tree

traversal

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

- Ficheiros gerados:
 - `HelloLexer.java`: código Java com a análise léxica (gera *tokens* para a análise sintáctica)
 - `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 sintáctica)
 - `HelloParser.java`: código Java com a análise sintáctica (gera a árvore sintáctica 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.

ANTLR4: Ficheiros gerados (2)

- 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.

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

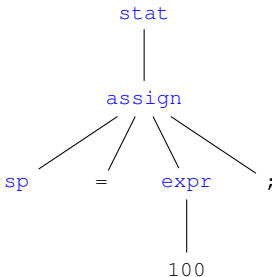
- Exemplo:

```
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:

```
sp = 100;
```

- Vamos obter a seguinte árvore sintática:



Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintática

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

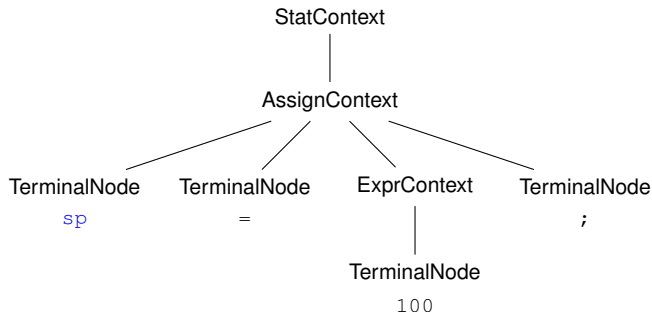
Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

- 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 respectivos objectos) com o contexto de todas as regras da gramática:



Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4: contexto automático (2)

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

`(grammar Expr);` → classes: ExprLexer and ExprParser

`(stat): assign ;` → class StatContext in ExprParser

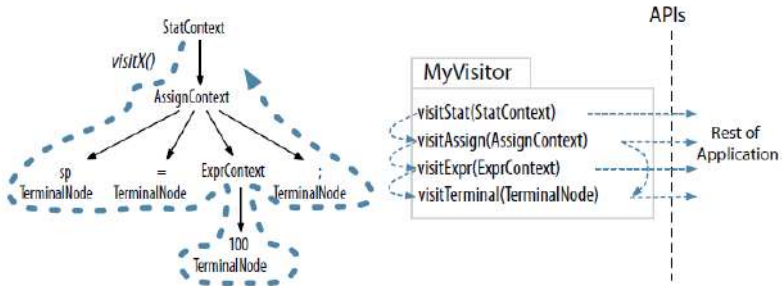
`(assign): ID '=' expr ';' ;` → class AssignContext in ExprParser

`(expr): INT ;` → class ExprContext in ExprParser

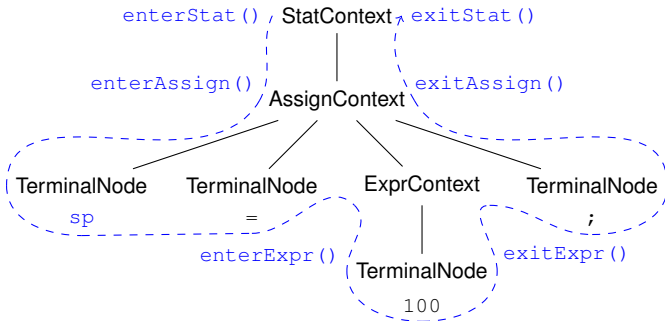
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [\t\r\n]+ -> skip ;

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

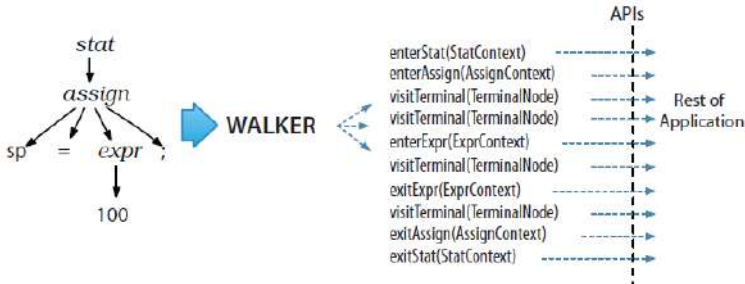
- Os objectos de contexto têm a si associada toda a informação relevante da análise sintáctica (*tokens*, referência aos nós filhos da árvore, etc.)
- Por exemplo o contexto `AssignContext` contém métodos `ID` e `expr` para aceder aos respectivos nós.
- No caso do código gerado automaticamente do tipo *visitor* o padrão de invocação é ilustrado a seguir:



- O código gerado automaticamente do tipo *listener* tem o seguinte padrão de invocação:



- A sua ligação à restante aplicação é a seguinte:



ANTLR4: atributos e acções

- É possível associar **atributos** e **acções** 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 acções ocorre durante a análise sintáctica.
- A execução de cada acção ocorre no contexto onde ela é declarada. Assim se uma acção estiver no fim de uma regra (como exemplificado acima), a sua execução ocorrerá após o respectivo reconhecimento.
- A linguagem a ser executada na acção não tem de ser necessariamente Java (existem muitas outras possíveis, como C++ e python).

ANTLR4: atributos e acções (2)

- 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.

Exemplo figuras

[Apresentação](#)

[Exemplos](#)

Hello

Expr

[Exemplo figuras](#)

Exemplo *visitor*

Exemplo *listener*

[Construção de gramáticas](#)

Especificação de gramáticas

[ANTLR4: Estrutura léxica](#)

Comentários

Identificadores

Literais

Palavras reservadas

Ações

[ANTLR4: Regras léxicas](#)

Padrões léxicos típicos

Operador léxico "não ganancioso"

[ANTLR4: Estrutura sintáctica](#)

Secção de *tokens*

Ações no preâmbulo da gramática

[ANTLR4: Regras sintácticas](#)

Padrões sintácticos típicos

Precedência

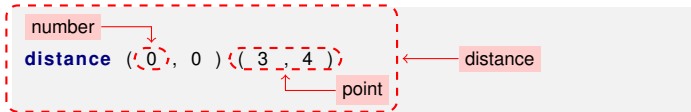
Associatividade

Herança de gramáticas

ANTLR4: Figuras

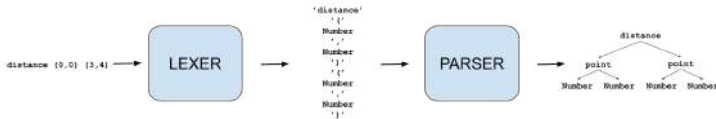
ANTLR4

- Recuperando o exemplo das figuras.



- Gramática inicial para figuras:

```
grammar Shapes;  
// parser rules:  
distance: 'distance' point point;  
point: '(' x=NUM ',' y=NUM ')';  
// lexer rules:  
NUM: [0-9]+;  
WS: [ \t\n\r]+ -> skip;
```



[Apresentação](#)

[Exemplos](#)

Hello

Expr

[Exemplo figuras](#)

[Exemplo visitor](#)

[Exemplo listener](#)

[Construção de gramáticas](#)

Especificação de gramáticas

[ANTLR4: Estrutura léxica](#)

Comentários

Identificadores

Literais

Palavras reservadas

Ações

[ANTLR4: Regras léxicas](#)

Padrões léxicos típicos

Operador léxico "não ganancioso"

[ANTLR4: Estrutura sintática](#)

Secção de *tokens*

Ações no préambulo da gramática

[ANTLR4: Regras sintáticas](#)

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Integração num programa

```
import java.io.IOException;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
public class ShapesMain {
    public static void main(String[] args) {
        try {
            // create a CharStream that reads from standard input:
            CharStream input = CharStreams.fromStream(System.in);
            // create a lexer that feeds off of input CharStream:
            ShapesLexer lexer = new ShapesLexer(input);
            // create a buffer of tokens pulled from the lexer:
            CommonTokenStream tokens = new CommonTokenStream(lexer);
            // create a parser that feeds off the tokens buffer:
            ShapesParser parser = new ShapesParser(tokens);
            // begin parsing at distance rule:
            ParseTree tree = parser.distance();
            if (parser.getNumberOfSyntaxErrors() == 0) {
                // print LISP-style tree:
                // System.out.println(tree.toStringTree(parser));
            }
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
        catch (RecognitionException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Exemplo *visitor*

- Uma primeira versão (limpa) de um *visitor* pode ser gerada com o script `antlr4-visitor`
- Depois podemos alterá-la, por exemplo, da seguinte forma:

```
import org.antlr.v4.runtime.tree.AbstractParseTreeVisitor;

public class ShapesMyVisitor extends ShapesBaseVisitor<Object> {
    @Override
    public Object visitDistance (ShapesParser.DistanceContext ctx) {
        double res;
        double[] p1 = (double[]) visit (ctx.point (0));
        double[] p2 = (double[]) visit (ctx.point (1));
        res = Math.sqrt (Math.pow (p1[0]-p2[0], 2) +
                             Math.pow (p1[1]-p2[1], 2));
        System.out.println ("visitDistance: "+res);
        return res;
    }

    @Override
    public Object visitPoint (ShapesParser.PointContext ctx) {
        double[] res = new double[2];
        res[0] = Double.parseDouble (ctx.x.getText ());
        res[1] = Double.parseDouble (ctx.y.getText ());

        return (Object) res;
    }
}
```

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Exemplo *visitor* (2)

- Para utilizar esta classe:

```
public static void main(String[] args) {  
    ...  
    // visitor:  
    ShapesMyVisitor visitor = new ShapesMyVisitor();  
    System.out.println("distance: "+visitor.visit(tree));  
    ...  
}
```

- O comando `antlr4-main` permite a geração automática deste código no método `main`.

```
antlr4-main <Grammar> <start-rule>  
        -v <nome-da-classe-ou-ficheiro-visitor> ...
```

- Note que podemos criar o método `main` com os *listeners* e *visitors* que quisermos (a ordem especificada nos argumentos do comando é mantida).

Exemplo *listener*

```
import static java.lang.System.*;

import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.tree.ErrorNode;
import org.antlr.v4.runtime.tree.TerminalNode;

public class ShapesMyListener extends ShapesBaseListener {
    @Override
    public void enterPoint(ShapesParser.PointContext ctx) {
        int x = Integer.parseInt(ctx.x.getText());
        int y = Integer.parseInt(ctx.y.getText());
        out.println("enterPoint x="+x+",y="+y);
    }

    @Override
    public void exitPoint(ShapesParser.PointContext ctx) {
        int x = Integer.parseInt(ctx.x.getText());
        int y = Integer.parseInt(ctx.y.getText());
        out.println("exitPoint x="+x+",y="+y);
    }
}
```

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Exemplo *listener* (2)

- Para utilizar esta classe:

```
public static void main(String[] args) {  
    ...  
    // listener:  
    ParseTreeWalker walker = new ParseTreeWalker();  
    ShapesMyListener listener = new ShapesMyListener();  
    walker.walk(listener, tree);  
    ...  
}
```

- O comando `antlr4-main` permite a geração automática deste código no método `main`.

```
antlr4-main <Grammar> <start-rule>  
           -l <nome-da-classe-ou-ficheiro-listener> ...
```

Construção de gramáticas

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

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 sintáctica; 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.

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literals

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Construção de gramáticas (2)

- Surpreendentemente, o número de padrões base é relativamente baixo:
 - 1 **Sequência**: sequência de elementos;
 - 2 **Optativo**: aplicação optativa do elemento (zero ou uma ocorrência);
 - 3 **Repetitivo**: aplicação repetida do elemento (zero ou mais, uma ou mais);
 - 4 **Alternativa**: escolha entre diferentes alternativas (como por exemplo, diferentes tipos de instruções);
 - 5 **Recursão**: definição directa ou indirectamente recursiva de um elemento (por exemplo, instrução condicional é uma instrução que selecciona para execução outras instruções);
- É de notar que 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.

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Construção de gramáticas (3)

- Considere o seguinte programa em Java:

```
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;
    }
}
```

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literals

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Construção de gramáticas (4)

- Mesmo sem uma gramática definida explicitamente, podemos neste programa inferir todos os padrões atrás referidos:
 - 1 **Sequência**: a instrução atribuição de valor é definida como sendo um identificador, seguido do carácter =, seguido de uma expressão.
 - 2 **Optativo**: a instrução condicional pode ter, ou não, a selecção de código para a condição falsa.
 - 3 **Repetitivo**: (1) uma classe é uma repetição de membros; (2) um algoritmo é uma repetição de comandos.
 - 4 **Alternativa**: diferentes instruções podem ser utilizadas onde uma instrução é esperada.
 - 5 **Recursão**: a instrução composta é definida como sendo uma sequência de instruções delimitada por chavetas; qualquer uma dessas instruções pode ser também uma instrução composta.

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

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> ::= <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> : <meaning> ;`

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4:

Estrutura Léxica

[Apresentação](#)

[Exemplos](#)

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

[Construção de gramáticas](#)

Especificação de gramáticas

[ANTLR4: Estrutura léxica](#)

Comentários

Identificadores

Literais

Palavras reservadas

Ações

[ANTLR4: Regras léxicas](#)

Padrões léxicos típicos

Operador léxico "não ganancioso"

[ANTLR4: Estrutura sintáctica](#)

Secção de *tokens*

Ações no preâmbulo da gramática

[ANTLR4: Regras sintácticas](#)

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Estrutura léxica: comentários

- A estrutura léxica do ANTLR4 deverá ser familiar para a maioria dos programadores já que se aproxima da sintaxe das linguagens da família do C (C++, Java, etc.).
- Os comentários são em tudo semelhantes aos do Java permitindo a definição de comentários de linha, multilinha, ou tipo JavaDoc.

```
/**  
 * Javadoc alike comment!  
 */  
grammar Name;  
/*  
 multiline comment  
 */  
  
/** parser rule for an identifier */  
id: ID ; // match a variable name
```

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Estrutura léxica: identificadores

- O primeiro carácter dos identificadores tem de ser uma letra, seguida por outras letras dígitos ou o carácter `_`
- Se a primeira letra do identificador é minúscula, então este identificador representa uma regra sintáctica; caso contrário (i.e. letra maiúscula) então estamos na presença duma regra léxica.

```
ID, LPAREN, RIGHT_CURLY, Other // lexer token names  
expr, conditionalStatment      // parser rule names
```

- Como em Java, podem ser utilizados caracteres Unicode.

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

- Em ANTLR4 não há distinção entre literais do tipo carácter e do tipo *string*.
- Todos os literais são delimitados por aspas simples.
- Exemplos: `'if'`, `'>='`, `'assert'`
- Como em Java, os literais podem conter sequências de escape tipo Unicode (`'\u3001'`), assim como as sequências de escape habituais (`'\r\t\n'`)

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Estrutura léxica: palavras reservadas

- O ANTLR4 tem a seguinte lista de palavras reservadas (i.e. que não podem ser utilizadas como identificadores):

```
import , fragment , lexer ,  
parser , grammar , returns ,  
locals , throws , catch ,  
finally , mode , options ,  
tokens , skip
```

- Mesmo não sendo uma palavra reservada, não se pode utilizar a palavra `rule` já que esse nome entra em conflito com os nomes gerados no código.

[Apresentação](#)

[Exemplos](#)

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

[Construção de gramáticas](#)

Especificação de gramáticas

[ANTLR4: Estrutura léxica](#)

Comentários

Identificadores

Literais

[Palavras reservadas](#)

Ações

[ANTLR4: Regras léxicas](#)

Padrões léxicos típicos

Operador léxico "não ganancioso"

[ANTLR4: Estrutura sintática](#)

Secção de *tokens*

Ações no preâmbulo da gramática

[ANTLR4: Regras sintáticas](#)

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Estrutura léxica: acções

- As acções são blocos de código escritos na linguagem destino (Java por omissão).
- As acções podem ter múltiplas localizações dentro da gramática, mas a sintaxe é sempre a mesma: texto delimitado por chavetas: { . . . }
- Se por caso existirem *strings* ou comentários (ambos tipo C/Java) contendo chavetas não há necessidade de incluir um carácter de escape ({ . . . " } " . / * } * / . . . }).
- O mesmo acontece se as chavetas foram balanceadas ({ { { . . . { } . . . } } }).
- Caso contrário, tem de se utilizar o carácter de escape ({ \ { } , { \ } }).
- O texto incluído dentro das acções tem de estar conforme com a linguagem destino.

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Acções

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Acções no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Estrutura léxica: acções (2)

- As acções podem aparecer nas regras léxicas, nas regras sintácticas, na especificação de excepções da gramática, nas secções de atributos (resultado, argumento e variáveis locais), em certas secções do cabeçalho da gramática e em algumas opções de regras (predicados semânticos).
- Pode considerar-se que cada acção será executada no contexto onde aparece (por exemplo, no fim do reconhecimento duma regra).

```
grammar Expr ;
stat :
    { System.out.println ("[ stat]: before assign"); } assign
  | expr { System.out.println ("[ stat]: after expr"); }
  ;
assign :
  ID
  { System.out.println ("[ assign]: after ID and before =!"); }
  '=' expr ';' ;
expr : INT { System.out.println ("[ expr]: INT!"); } ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Acções

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Acções no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4:

Regras Léxicas

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

- A gramática léxica é composta por regras (ou produções), em que cada regra define um *token*.
- As regras léxicas têm de começar por uma letra maiúscula, e podem ser visíveis apenas dentro do analisador léxico:

```
INT: DIGIT+ ; // visible in both parser and lexer
fragment DIGIT: [0-9]; // visible only in lexer
```

- Como, por vezes, a mesma sequência de caracteres pode ser reconhecida por diferentes regras (por exemplo: identificadores e palavras reservadas), o ANTLR4 estabelece critérios que permitem eliminar esta ambiguidade (e dessa forma, reconhecer um, e um só, *token*).

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintática

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Regras léxicas (2)

- Esses critérios são essencialmente dois (na ordem seguinte):

- 1 Reconhece *tokens* que consomem o máximo possível de caracteres.

Por exemplo, num reconhecedor léxico para Java, o texto `if a` é reconhecido com um único *token* tipo identificador, e não como dois *tokens* (palavra reservada `if` seguida do identificador `a`).

- 2 Dá prioridade às regras definidas em primeiro lugar.
Por exemplo, na gramática seguinte:

```
ID : [a-z]+;  
IF : 'if' ;
```

o *token* `IF` nunca vai ser reconhecido!

- O ANTLR4 também considera que os *tokens* definidos implicitamente em regras sintáticas, estão definidos *antes* dos definidos explicitamente por regras léxicas.
- A especificação destas regras utiliza **expressões regulares**.

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintática

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Expressões regulares em ANTLR4

ANTLR4

[Apresentação](#)

[Exemplos](#)

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

[Construção de gramáticas](#)

Especificação de gramáticas

[ANTLR4: Estrutura léxica](#)

Comentários

Identificadores

Literais

Palavras reservadas

Ações

[ANTLR4: Regras léxicas](#)

Padrões léxicos típicos

Operador léxico "não ganancioso"

[ANTLR4: Estrutura sintáctica](#)

Secção de tokens

Ações no preâmbulo da gramática

[ANTLR4: Regras sintáticas](#)

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Syntax

Description

R : ... ;

Define lexer rule R

X

Match lexer rule element X

'literal'

Match literal text

[char-set]

Match one of the chars in char-set

'x'..'y'

Match one of the chars in the interval

X Y ... Z

Match a sequence of rule lexer elements

(...)

Lexer subrule

X?

Optionally match rule element X

*X**

Match rule element X zero or more times

X+

Match rule element X one or more times

Expressões regulares em ANTLR4 (2)

ANTLR4

[Apresentação](#)

[Exemplos](#)

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

[Construção de gramáticas](#)

Especificação de gramáticas

[ANTLR4: Estrutura léxica](#)

Comentários

Identificadores

Literais

Palavras reservadas

Ações

[ANTLR4: Regras léxicas](#)

Padrões léxicos típicos

Operador léxico "não ganancioso"

[ANTLR4: Estrutura sintáctica](#)

Secção de *tokens*

Ações no preâmbulo da gramática

[ANTLR4: Regras sintácticas](#)

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Syntax	Description
$\sim X$	Match one of the chars NOT in the set defined by x
$.$	Match any char
$X*? Y$	Match X until Y appears (non-greedy match)
$\{ \dots \}$	Lexer action
$\{p\}?$	Evaluate semantic predicate p (if false, the rule is ignored)
$x \mid \dots \mid z$	Multiple alternatives

Padrões léxicos típicos

ANTLR4

Token category Possible implementation

Identifiers

```
ID: LETTER (LETTER | DIGIT) *;  
fragment LETTER: 'a'..'z'|'A'..'Z'|'_';  
    // same as: [a-zA-Z_]  
fragment DIGIT: '0'..'9';  
    // same as: [0-9]
```

Numbers

```
INT: DIGIT+;  
FLOAT: DIGIT+ '.' DIGIT+ | '.' DIGIT+;
```

Strings

```
STRING: '"' (ESC | . ) *? '"';  
fragment ESC: '\\\"' | '\\\\\\';
```

Comments

```
LINE_COMMENT: '//' .*? '\n' -> skip;  
COMMENT: '/*' .*? '*/' -> skip;
```

Whitespace

```
WS: [ \t\n\r]+ -> skip;
```

[Apresentação](#)

[Exemplos](#)

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

[Construção de gramáticas](#)

Especificação de gramáticas

[ANTLR4: Estrutura léxica](#)

Comentários

Identificadores

Literais

Palavras reservadas

Ações

[ANTLR4: Regras léxicas](#)

Padrões léxicos típicos

Operador léxico "não ganancioso"

[ANTLR4: Estrutura sintáctica](#)

Secção de *tokens*

Ações no preâmbulo da gramática

[ANTLR4: Regras sintácticas](#)

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Operador léxico “não ganancioso”

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico “não ganancioso”

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Operador léxico “não ganancioso”

- Por omissão, a análise léxica é “gananciosa”.
- Isto é, os *tokens* são gerados com o maior tamanho possível.
- Esta particularidade é em geral a desejada, mas pode trazer problemas em alguns casos.
- Por exemplo, se quisermos reconhecer um *string*:

```
STRING: " " . * " " ;
```

- (No analisador léxico o ponto (.) reconhece qualquer carácter excepto o EOF.)
- Esta regra não funciona, porque, uma vez reconhecido o primeiro carácter " , o analisador léxico vai reconhecer todos os caracteres como pertencendo ao `STRING` até ao último carácter " .
- Este problema resolve-se com o operador *non-greedy*:

```
STRING: " " . * ? " " ; // match all chars until a " appears!
```

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico “não ganancioso”

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4: Estrutura Sintáctica

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de
gramáticas

Especificação de
gramáticas

ANTLR4: Estrutura
léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras
léxicas

Padrões léxicos típicos

Operador léxico "não
ganancioso"

ANTLR4: Estrutura
sintáctica

Secção de *tokens*

Ações no preâmbulo da
gramática

ANTLR4: Regras
sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

- As gramáticas em ANTLR4 têm a seguinte estrutura sintáctica:

```
grammar Name;           // mandatory
options { ... }         // optional
import ... ;            // optional
tokens { ... }          // optional
@actionName { ... }     // optional
rule1 : ... ;           // parser and lexer rules
...
```

- As regras léxicas e sintácticas pode aparecer misturadas e distinguem-se por a primeira letra do nome da regra ser minúscula (analizador sintáctico), ou maiúscula (analizador léxico).
- Como já foi referido, a ordem pela qual as regras léxicas são definidas é muito importante.

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4: Estrutura sintáctica (2)

- É possível separar as gramáticas sintácticas das léxicas precedendo a palavra reservada `grammar` com as palavras reservadas `parser` ou `lexer`.

```
parser grammar NameParser;  
...
```

```
lexer grammar NameLexer;  
...
```

- A secção das **opções** permite definir algumas opções para os analisadores (e.g. origem dos *tokens*, e a linguagem de programação de destino).

```
options { tokenVocab=NameLexer; }
```

- Qualquer opção pode ser redefinida por argumentos na invocação do ANTLR4.
- A secção de `import` relaciona-se com herança de gramáticas (que veremos mais à frente).

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

- A secção de **tokens** permite associar identificadores a *tokens*.
- Esses identificadores devem depois ser associados a regras léxicas, que podem estar na mesma gramática, noutra gramática, ou mesmo ser directamente programados.

```
tokens { «Token1», ... , «TokenN» }
```

- Por exemplo:

```
tokens { BEGIN, END, IF, ELSE, WHILE, DO }
```

- Note que não é necessário ter esta secção quando os tokens tem origem numa gramática lexical antlr4 (basta a secção **options** com a variável **tokenVocab** correctamente definida).

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Acções no preâmbulo da gramática

- Esta secção permite a definição de **acções** no preâmbulo da gramática (como já vimos, também podem existir acções noutras zonas da gramática).
- Actualmente só existem dois acções possíveis nesta zona (com o Java como linguagem destino): `header` e `members`

```
grammar Count;  
@header {  
package foo;  
}  
@members {  
int count = 0;  
}
```

- A primeira injecta código no início de ficheiros, e a segunda permite que se acrescentem membros às classes do analisador sintáctico e/ou léxico.
- Eventualmente podemos restringir estas acções ou ao analisador sintáctico (`@parser::header`) ou ao analisador léxico (`@lexer::members`)

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Acções

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Acções no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4: Regras Sintáticas

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de
gramáticas

Especificação de
gramáticas

ANTLR4: Estrutura
léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras
léxicas

Padrões léxicos típicos

Operador léxico "não
ganancioso"

ANTLR4: Estrutura
sintáctica

Secção de *tokens*

Ações no preâmbulo da
gramática

ANTLR4: Regras
sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Construção de regras: síntese

ANTLR4

<i>Syntax</i>	<i>Description</i>
---------------	--------------------

<i>r : ... ;</i>	<i>Define rule r</i>
------------------	----------------------

<i>x</i>	<i>Match rule element x</i>
----------	-----------------------------

<i>x y ... z</i>	<i>Match a sequence of rule elements</i>
------------------	--

<i>(...)</i>	<i>Subrule</i>
--------------	----------------

<i>x?</i>	<i>Match rule element x</i>
-----------	-----------------------------

<i>x*</i>	<i>Match rule element x zero or more times</i>
-----------	--

<i>x+</i>	<i>Match rule element x one or more times</i>
-----------	---

<i>x ... z</i>	<i>Multiple alternatives</i>
--------------------	------------------------------

A rule element is a token (lexical, or terminal rule), a syntactical rule (non-terminal), or a subrule.

- As regras podem ser *recursivas*.
- No entanto, só pode haver recursividade à esquerda se for directa (i.e. definida na própria regra).

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Regras sintáticas: movendo informação

- Em ANTLR4 cada regra sintáctica pode ser vista como uma espécie de método, havendo mecanismos de comunicação similares: **argumentos** e **resultado**, assim como **variáveis locais** à regra.
- Podemos também anotar regras com um nome alternativo:

```
expr : e1=expr '+' e2=expr  
      | INT ;
```

- Podemos também etiquetar com nomes, diferentes alternativas duma regra:

```
expr : expr '*' e2=expr # ExprMult  
      | expr '+' e2=expr # ExprAdd  
      | INT             # ExprInt  
      ;
```

- O ANTLR4 irá gerar informação de contexto para cada nome (incluindo métodos para usar no *listener* e/ou nos *visitors*).

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Regras sintáticas: movendo informação (2)

```
grammar Info ;

@header {
import static java.lang.System.*;
}

main: seq1=seq[true] seq2=seq[false] {
    out.println("average(seq1): "+$seq1.average);
    out.println("average(seq2): "+$seq2.average);
}
;

seq[boolean crash] returns[double average=0]
    locals[int sum=0, int count=0]:
    '(' ( INT {$sum+=$INT.int;$count++;} ) * ')' {
        if ($count > 0)
            $average = (double)$sum/$count;
        else if ($crash) {
            err.println("ERROR: divide by zero!");
            exit(1);
        }
    }
;

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

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

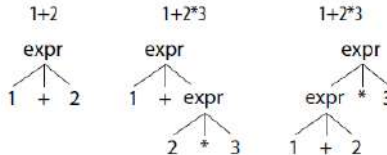
Herança de gramáticas

Padrões sintáticos típicos

ANTLR4

Pattern name	Possible implementation	Apresentação
Sequence	<pre>x y ... z [' INT+ ']' [' INT* ']</pre>	Exemplos Hello Expr Exemplo figuras Exemplo visitor Exemplo listener
Sequence with terminator	<pre>(instruction ';') * // program sequence (row '\n') * // lines of data</pre>	Construção de gramáticas Especificação de gramáticas
Sequence with separator	<pre>expr (',' expr) * // function call arguments (expr (',' expr) *) ? // optional arguments</pre>	ANTLR4: Estrutura léxica Comentários Identificadores Literais Palavras reservadas Ações
Choice	<pre>type: 'int' 'float'; instruction: conditional loop ... ;</pre>	ANTLR4: Regras léxicas Padrões léxicos típicos Operador léxico "não ganancioso"
Token dependence	<pre>(' expr ')' // nested expression ID '[' expr ']' // array index {' instruction+ '}' // compound instruction '<' ID (',' ID)* '>' // generic type specifier</pre>	ANTLR4: Estrutura sintática Secção de tokens Ações no preâmbulo da gramática
Recursivity	<pre>expr: '(' expr ')' ID; classDef: 'class' ID '{' (classDef method field)* '}' ;</pre>	ANTLR4: Regras sintáticas Padrões sintáticos típicos Precedência Associatividade Herança de gramáticas

- Por vezes, formalmente, a interpretação da ordem de aplicação de operadores pode ser subjectiva:



- Em ANTLR4 esta ambiguidade é resolvida dando primazia às sub-regras declaradas primeiro:

```
expr : expr '*' expr // higher priority
    | expr '+' expr
    | INT              // lower priority
    ;
```

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Associatividade

- Por omissão, a associatividade na aplicação do (mesmo) operador é feita da esquerda para a direita:

$$a + b + c = ((a + b) + c)$$

- No entanto, há operadores, como é o caso da potência, que podem requerer a associatividade inversa:

$$a \uparrow b \uparrow c = a^{b^c} = a^{(b^c)}$$

- Este problema é resolvido em ANTLR4 de seguinte forma:

```
expr : <assoc=right> expr '^' expr
    | expr '*' expr // higher priority
    | expr '+' expr
    | INT           // lower priority
    ;
```



Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Herança de gramáticas

- A secção de **import** implementa um mecanismo de herança entre gramáticas.
- Por exemplo as gramáticas:

```
grammar ELang;  
stat : (expr ';'*) EOF ;  
expr : INT ;  
INT : [0-9]+ ;  
WS : [ \r\t\n]+ -> skip ;
```

```
grammar MyELang;  
import ELang;  
expr : INT | ID ;  
ID : [a-z]+ ;
```

- Geram a gramática MyELang equivalente:

```
grammar MyELang;  
stat : (expr ';'*) EOF ;  
expr : INT | ID ;  
ID : [a-z]+ ;  
INT : [0-9]+ ;  
WS : [ \r\t\n]+ -> skip ;
```

- Isto é, as regras são herdadas, excepto quando são redefinidas na gramática descendente.

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Herança de gramáticas (2)

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintática

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

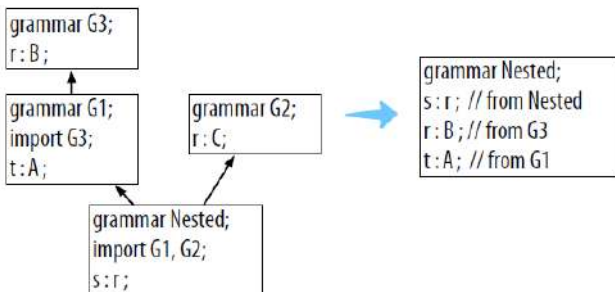
Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

- Este mecanismo permite herança múltipla:



- Note-se a importância na ordem dos `imports` na gramática `Nested`.
- A regra `r` vem da gramática `G3` e não da gramática `G2`.

ANTLR4:

Mais sobre acções

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Acções

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Acções no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Mais sobre acções

- Já vimos que é possível acrescentar directamente na gramática acções (expressas na linguagem destino) que são executadas durante a fase de análise sintáctica (na ordem expressa na gramática).
- Podemos também associar a cada regra dois blocos especiais de código – `@init` e `@after` – cuja execução, respectivamente, precede ou sucede ao reconhecimento da regra.
- O bloco `@init` pode ser útil, por exemplo, para inicializar variáveis.
- O bloco `@after` é uma alternativa a colocar a acção no fim da regra.

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Acções

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Acções no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Exemplo: tabelas CSV

[Apresentação](#)

[Exemplos](#)

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

[Construção de gramáticas](#)

Especificação de gramáticas

[ANTLR4: Estrutura léxica](#)

Comentários

Identificadores

Literais

Palavras reservadas

Ações

[ANTLR4: Regras léxicas](#)

Padrões léxicos típicos

Operador léxico "não ganancioso"

[ANTLR4: Estrutura sintáctica](#)

Secção de *tokens*

Ações no preâmbulo da gramática

[ANTLR4: Regras sintácticas](#)

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Exemplo

- Exemplo: gramática para ficheiros tipo CSV com os seguintes requisitos:
 - 1 A primeira linha indica o nome dos campos (deve ser escrita sem nenhuma formatação em especial);
 - 2 Em todas as linhas que não a primeira associar o valor ao nome do campo (devem ser escritas com a associação explícita, tipo atribuição de valor com `field = value`).

grammar CSV;

file : line line* EOF;

line : field (SEP field)* '\r'? '\n';

field : TEXT | STRING | ;

SEP : ',' ; // (',' | '\t')*

STRING : [\t]* '"' .*? '"' [\t]*;

TEXT : ~[,\r\n]~[,\r\n]*;

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Exemplo

```
grammar CSV;
@header {
import static java.lang.System.*;
}
@parser::members {
protected String[] names = new String[0];
public int dimNames() { ... }
public void addName(String name) { ... }
public String getName(int idx) { ... }
}

file: line[true] line[false]* EOF;

line[boolean firstLine]
locals[int col = 0]
@after { if (!firstLine) out.println(); }
: field[$firstLine, $col++] (SEP field[$firstLine, $col++]* '\r'? '\n';

field[boolean firstLine, int col]
returns[ String res = "" ]
@after {
if ($firstLine)
addName($res);
else if ($col >= 0 && $col < dimNames())
out.print(" " + getName($col) + ": " + $res);
else
err.println("\nERROR: invalid field \"" + $res + "\" in column " + ($col + 1));
}
:
(TEXT { $res = $TEXT.text.trim(); }) |
(STRING { $res = $STRING.text.trim(); }) |
;

SEP: ' '; // ( ' ' / '\t ')*
STRING: [ \t]* ' '.*? ' ' [ \t]*;
TEXT: ~[ , "\r\n]~[ , \r\n]*;
```

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintática

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

[Apresentação](#)[Exemplos](#)*Hello**Expr*

Exemplo figuras

Exemplo *visitor*Exemplo *listener*[Construção de gramáticas](#)

Especificação de gramáticas

[ANTLR4: Estrutura léxica](#)

Comentários

Identificadores

Literais

Palavras reservadas

Acções

[ANTLR4: Regras léxicas](#)

Padrões léxicos típicos

Operador léxico "não ganancioso"

[ANTLR4: Estrutura sintáctica](#)Secção de *tokens*

Acções no preâmbulo da gramática

[ANTLR4: Regras sintácticas](#)

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4:

Gramáticas ambíguas

- A definição de gramáticas presta-se, com alguma facilidade, a gerar ambiguidades.
- Esta característica nas linguagens humanas é por vezes procurada, mas geralmente é um problema.

“Para o meu orientador, para quem nenhum agradecimento é demasiado.”

“O professor falou aos alunos de engenharia”

“What rimes with orange? . . . No it doesn’t!”

- No caso das linguagens de programação, em que os efeitos são para ser interpretados e executados por máquinas (e não por nós), não há espaço para ambiguidades.
- Assim, seja por construção da gramática, seja por regras de prioridade que lhe sejam aplicadas por omissão, as gramáticas não podem ser ambíguas.
- Em ANTLR4 a definição e construção de regras define prioridades.

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico “não ganancioso”

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Gramáticas ambíguas: analisador léxico

- Se as gramáticas léxicas fossem apenas definidas por expressões regulares que competem entre si para consumir os caracteres de entrada, então elas seriam naturalmente ambíguas.

```
...
conditional: 'if' '(' expr ')' 'then' stat; // incomplete
ID: [a-zA-Z]+;
...
```

- Neste caso a sequência de caracteres `if` tanto pode dar um identificador como uma palavra reservada.
- O ANTLR4 utiliza duas regras fora das expressões regulares para lidar com ambiguidade:
 - 1 Por omissão, escolhe o *token* que consome o máximo número de caracteres da entrada;
 - 2 Dá prioridade aos *tokens* definidos primeiro (sendo que os definidos implicitamente na gramática sintáctica têm precedência sobre todos os outros).

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Gramáticas ambíguas: analisador sintáctico

- Já vimos que nas regras sintácticas também pode haver ambiguidade.
- Os dois excertos seguintes exemplificam gramáticas ambíguas:

```
stat : ID '=' expr
      | ID '=' expr
expr : NUM
      ;
```

```
stat : expr ';'
      | ID '(' ')' ';'
expr : ID '(' ')'
      | NUM
      ;
```

- Em ambos os casos a ambiguidade resulta de ser ter uma sub-regra repetida, directamente, no primeiro caso, e indirectamente, no segundo caso.

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

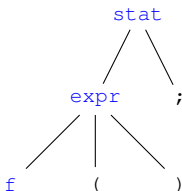
Associatividade

Herança de gramáticas

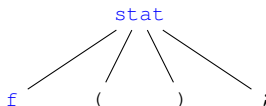
Gramáticas ambíguas: analisador sintáctico (2)


- A gramática diz-se ambígua porque, para a mesma entrada, poderíamos ter duas árvores sintáticas diferentes.

Expressão `f () ;`



Instrução `f () ;`



- Outros exemplos de ambiguidade são os da precedência e associatividade de operadores .

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintática

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Gramáticas ambíguas: analisador sintático (3)

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintática

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

- O ANTLR4 tem regras adicionais para eliminar ambiguidades sintáticas.
- Tal como no analisador léxico, regras *Ad hoc* fora da notação das gramática independentes de contexto, garantem a não ambiguidade.
- Essas regras são as seguintes:
 - 1 As alternativas, directa ou indirectamente, definidas primeiro têm precedência sobre as restantes.
 - 2 Por omissão, a associatividade de operadores é à esquerda.
- Das duas árvores sintáticas apresentadas no exemplo anterior, a gramática definida impõe a primeira alternativa.

Gramáticas ambíguas: analisador sintáctico (4)

- A linguagem C tem ainda outro exemplo prático de ambiguidade.
- A expressão $i * j$ tanto pode ser uma multiplicação de duas variáveis, como a declaração de uma variável j como ponteiro para o tipo de dados i .
- Estes dois significados tão diferentes podem também ser resolvidos em gramáticas ANTLR4 com os chamados **predicados semânticos**.

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4: Predicados semânticos

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de
gramáticas

Especificação de
gramáticas

ANTLR4: Estrutura
léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras
léxicas

Padrões léxicos típicos

Operador léxico "não
ganancioso"

ANTLR4: Estrutura
sintáctica

Secção de *tokens*

Ações no preâmbulo da
gramática

ANTLR4: Regras
sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

- Em ANTLR4 é possível utilizar informação semântica (expressa na linguagem destino e injetada na gramática), para orientar o analisador sintático.
- Essa funcionalidade chama-se **predicados semânticos**:
 $\{ \dots \} ?$
- Os predicados semânticos permitem seletivamente activar/desactivar porções das regras gramaticais durante a própria análise sintáctica.
- Vamos, como exemplo, desenvolver uma gramática para analisar sequências de números inteiros, mas em que o primeiro número não pertence à sequência, mas indica sim a dimensão da sequência:
- Assim a lista 2 4 1 3 5 6 7 indicaria duas sequências:
 $(4, 1)$ $(5, 6, 7)$

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

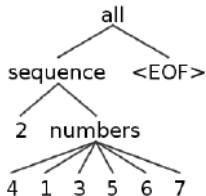
Associatividade

Herança de gramáticas

Exemplo

```
grammar Seq;  
  
all: sequence* EOF;  
  
sequence: INT numbers;  
  
numbers: INT*;  
  
INT: [0-9]+;  
WS: [ \t\r\n]+ -> skip;
```

Com esta gramática, a árvore sintáctica gerada para a entrada
2 4 1 3 5 6 7 é:



Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Exemplo

```
grammar Seq;

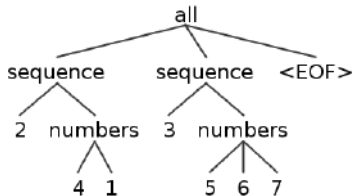
all: sequence* EOF;

sequence
    @init { System.out.print("("); }
    @after { System.out.println(")"); }
    : INT numbers[$INT.int];

numbers[int count] locals [int c = 0]
    : ( { $c < $count }? INT
        { $c++; System.out.print(($c == 1 ? "" : " ") + $INT.text); }
        )* ;

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

Agora a árvore sintáctica já corresponde ao pretendido:



Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4:

Separar *lexer* do *parser*

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Separar analisador léxico do analisador sintático

- Muito embora se possa definir a gramática completa, juntando a análise léxica e a sintática no mesmo módulo, podemos também separar cada uma dessas gramáticas.
- Isso facilita, por exemplo, a reutilização de analisadores léxicos.
- Existem também algumas funcionalidades do analisador léxico, que obrigam a essa separação (“ilhas” lexicais).
- Para que a separação seja bem sucedida há um conjunto de regras que devem ser seguidas:
 - 1 Cada gramática indica o seu tipo no cabeçalho:
 - 2 Os nomes das gramáticas devem (respectivamente) terminar em `Lexer` e `Parser`
 - 3 Todos os *tokens* implicitamente definidos no analisador sintático têm de passar para o analisador léxico (associando-lhes um identificador para uso no *parser*).
 - 4 A gramática do analisador léxico deve ser compilada pelo ANTLR4 antes da gramática sintática.
 - 5 A gramática sintática tem de incluir uma opção (`tokenVocab`) a indicar o analisador léxico.

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico “não ganancioso”

ANTLR4: Estrutura sintática

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Separar analisador léxico do analisador sintático (2)

ANTLR4

[Apresentação](#)

[Exemplos](#)

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

[Construção de gramáticas](#)

Especificação de gramáticas

[ANTLR4: Estrutura léxica](#)

Comentários

Identificadores

Literais

Palavras reservadas

Ações

[ANTLR4: Regras léxicas](#)

Padrões léxicos típicos

Operador léxico "não ganancioso"

[ANTLR4: Estrutura sintática](#)

Secção de *tokens*

Ações no preâmbulo da gramática

[ANTLR4: Regras sintáticas](#)

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

```
lexer grammar NAMELexer;
```

```
...
```

```
parser grammar NAMEParser;
```

```
options {  
    tokenVocab=NAMELexer;  
}
```

```
...
```

- No teste da gramática deve utilizar-se o nome sem o sufixo:

```
antlr4-test NAME rule
```

Exemplo

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

```
lexer grammar CSVLexer;
```

```
COMMA: ',';
```

```
EOL: '\r'? '\n';
```

```
STRING: '"' ( '"' | ~ '"' )* '"';
```

```
TEXT: ~[',"\r\n']~[',\r\n']*;
```

```
parser grammar CSVParser;
```

```
options {  
    tokenVocab=CSVLexer;  
}
```

```
file: firstRow row* EOF;
```

```
firstRow: row;
```

```
row: field (COMMA field)* EOL;
```

```
field: TEXT | STRING | ;
```

```
antlr4 CSVLexer.g4
```

```
antlr4 CSVParser.g4
```

```
antlr4-javac CSV*.java
```

```
// ou apenas: antlr4-build
```

```
antlr4-test CSV file
```

ANTLR4:

“Ilhas” lexicais

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico “não ganancioso”

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

- Outra característica do ANTLR4 é a possibilidade de reconhecer um conjunto diferente de *tokens* consoante determinados critérios.
- Para esse fim existem os chamados *modos* lexicais.
- Por exemplo, em XML, o tratamento léxico do texto deve ser diferente consoante se está dentro duma “marca” (*tag*) ou fora.
- Uma restrição desta funcionalidade é o facto de só se poderem utilizar modos lexicais em gramáticas léxicas.
- Ou seja, torna-se obrigatória a separação entre os dois tipos de gramáticas.
- Os modos lexicais são geridos pelos comandos:
mode (NAME) , pushMode (NAME) , popMode
- O modo lexical por omissão é designado por:
DEFAULT_MODE

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico “não ganancioso”

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Exemplo

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

```
lexer grammar ModesLexer;
```

```
// default mode
```

```
ACTION_START: '{' -> mode(INSIDE_ACTION);
```

```
OUTSIDE_TOKEN: ~'{' +;
```

```
mode INSIDE_ACTION;
```

```
ACTION_END: '}' -> mode(DEFAULT_MODE);
```

```
INSIDE_TOKEN: ~'}' +;
```

```
parser grammar ModesParser;
```

```
options {
```

```
    tokenVocab=ModesLexer;
```

```
}
```

```
all: ( ACTION_START | OUTSIDE_TOKEN | ACTION_END |  
      INSIDE_TOKEN ) * EOF;
```

Exemplo (2)

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

```
lexer grammar ModesLexer;
```

```
// default mode
```

```
ACTION_START: '{' -> pushMode(INSIDE_ACTION);
```

```
OUTSIDE_TOKEN: ~'{' +;
```

```
mode INSIDE_ACTION;
```

```
ACTION_END: '}' -> popMode;
```

```
INSIDE_ACTION_START: '{' -> pushMode(INSIDE_ACTION);
```

```
INSIDE_TOKEN: ~[{}]+;
```

```
parser grammar ModesParser;
```

```
options {  
    tokenVocab=ModesLexer;  
}
```

```
all: ( ACTION_START | OUTSIDE_TOKEN | ACTION_END |  
      INSIDE_ACTION_START | INSIDE_TOKEN ) * EOF;
```

ANTLR4:

Enviar *tokens* para canais diferentes

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Enviar *tokens* para canais diferentes

- Nos exemplos de gramáticas que temos vindo a apresentar, tem-se optado pela acção `skip` quando na presença dos chamados espaços em branco ou de comentários.
- Esta acção faz desaparecer esses *tokens* simplificando a análise sintáctica.
- O preço a pagar (geralmente irrelevante) é perder o texto completo que lhes está associado.
- No entanto, em ANTLR4 é possível ter dois em um. Isto é, retirar *tokens* da análise sintáctica, sem no entanto fazer desaparecer completamente esses *tokens* (podendo-se recuperar o texto que lhe está associado).
- Esse é o papel dos chamados **canais léxicos**.

```
WS: [ \t\n\r ]+      -> skip; // make token disappear
COMMENT: '/*' .*? '*/' -> skip; // make token disappear
```

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Enviar *tokens* para canais diferentes (2)

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintácticos típicos

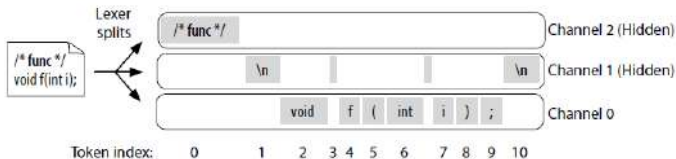
Precedência

Associatividade

Herança de gramáticas

```
WS: [ \t\n\r ]+      -> channel(1); // redirect to channel 1
COMMENT: '/*' .*? '*/' -> channel(2); // redirect to channel 2
```

- A classe `CommonTokenStream` encarrega-se de juntar os tokens de todos os canais (o visível – canal zero – e os escondidos).



- (É possível ter código para aceder aos *tokens* de um canal em particular.)

Exemplo: declaração de função

```
grammar Func;
```

```
func: type=ID function=ID '(' varDecl* ')' ';' ;  
varDecl: type=ID variable=ID ;
```

```
ID: [a-zA-Z_]+;
```

```
WS: [ \t\r\n]+ -> channel(1);
```

```
COMMENT: '/*' .*? '*/' -> channel(2);
```

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4: Reescrever a entrada

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Reescrever a entrada

- O ANTLR4 facilita a geração de código que resulte de uma reescrita do código de entrada. Isto é, inserir, apagar, e/ou modificar partes desse código.
- Para esse fim existe a classe `TokenStreamRewriter` (que têm métodos para inserir texto antes ou depois de *tokens*, ou para apagar ou substituir texto).
- Vamos supor que se pretende fazer algumas alterações de código fonte `Java`, por exemplo, acrescentar um comentário imediatamente antes da declaração de uma classe..
- Podemos ir buscar a gramática disponível para a versão 8 do `Java`: `Java8.g4`
(procurar em: <https://github.com/antlr/grammars-v4>)
- Para que a reescrita apenas acrescente o comentário, é necessário substituir o `skip` dos *tokens* que estão a ser desprezados, redireccionando-os para um canal escondido.
- Agora podemos criar um *listener* para resolver este problema.

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Exemplo

```
import org.antlr.v4.runtime.*;

public class AddClassCommentListener extends Java8BaseListener {

    protected TokenStreamRewriter rewriter;

    public AddClassCommentListener(TokenStream tokens) {
        rewriter = new TokenStreamRewriter(tokens);
    }

    public void print() {
        System.out.print(rewriter.getText());
    }

    @Override public void enterNormalClassDeclaration(
        Java8Parser.NormalClassDeclarationContext ctx) {
        rewriter.insertBefore(ctx.start, "/*\n * class "+
                               ctx.Identifier().getText()+
                               "\n */\n");
    }
}
```

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

[Apresentação](#)[Exemplos](#)*Hello**Expr*Exemplo *figuras*Exemplo *visitor*Exemplo *listener*[Construção de gramáticas](#)

Especificação de gramáticas

[ANTLR4: Estrutura léxica](#)

Comentários

Identificadores

Literais

Palavras reservadas

Ações

[ANTLR4: Regras léxicas](#)

Padrões léxicos típicos

Operador léxico "não ganancioso"

[ANTLR4: Estrutura sintáctica](#)Secção de *tokens*

Ações no preâmbulo da gramática

[ANTLR4: Regras sintácticas](#)

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4:

Desacoplar código da gramática

ParseTreeProperty

- Já vimos que podemos manipular a informação gerada na análise sintáctica de múltiplas formas:
 - Directamente na gramática recorrendo a acções e associando atributos a regras (argumentos, resultado, variáveis locais);
 - Utilizando *listeners*;
 - Utilizando *visitors*;
 - Associando atributos à gramática fazendo a sua manipulação dentro dos *listeners* e/ou *visitors*.
- Para associar informação extra à gramática, podemos acrescentar atributos à gramática (sintetizados, herdados ou variáveis locais às regras), ou utilizando os resultados dos métodos `visit`.

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Acções

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Acções no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Desacoplar código da gramática (2)

- Alternativamente, o ANTLR4 fornece outra possibilidade: a sua biblioteca de *runtime* contém um *array* associativo que permite associar nós da árvore sintáctica com atributos – `ParseTreeProperty`.
- Vamos ver um exemplo com uma gramática para expressões aritméticas:

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Exemplo

```
grammar Expr;  
  
main: stat* EOF;  
  
stat: expr;  
  
expr: expr '*' expr # Mult  
    | expr '+' expr # Add  
    | INT           # Int  
    ;  
  
INT: [0-9]+;  
WS: [ \t\r\n]+ -> skip;
```

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Exemplo

```
import org.antlr.v4.runtime.tree.ParseTreeProperty;

public class ExprSolver extends ExprBaseListener {
    ParseTreeProperty<Integer> mapVal = new ParseTreeProperty<>();
    ParseTreeProperty<String> mapTxt = new ParseTreeProperty<>();

    public void exitStat(ExprParser.StatContext ctx) {
        System.out.println(mapTxt.get(ctx.expr()) + " = " +
                           mapVal.get(ctx.expr()));
    }

    public void exitAdd(ExprParser.AddContext ctx) {
        int left = mapVal.get(ctx.expr(0));
        int right = mapVal.get(ctx.expr(1));
        mapVal.put(ctx, left + right);
        mapTxt.put(ctx, ctx.getText());
    }

    public void exitMult(ExprParser.MultContext ctx) {
        int left = mapVal.get(ctx.expr(0));
        int right = mapVal.get(ctx.expr(1));
        mapVal.put(ctx, left * right);
        mapTxt.put(ctx, ctx.getText());
    }

    public void exitInt(ExprParser.IntContext ctx) {
        int val = Integer.parseInt(ctx.INT().getText());
        mapVal.put(ctx, val);
        mapTxt.put(ctx, ctx.getText());
    }
}
```

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4: gestão de erros

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

- Por omissão o ANTLR4 faz uma gestão de erros automática, que, em geral, responde bem às necessidades.
- No entanto, por vezes é necessário ter algum controlo sobre este processo.
- No que diz respeito à apresentação de erros, por omissão o ANTLR4 formata e envia essa informação para a saída *standard* da consola.
- Esse comportamento pode ser redefinido com a interface `ANTLRErrorListener`.
- Como o nome indica, o padrão de software utilizado é o de um *listener*, e tal como nos temos habituado em ANTLR existe uma classe base (com os métodos todos implementados sem código): `BaseErrorListener`
- O método `syntaxError` é invocado pelo ANTLR na presença de erros e aplica-se ao analisador sintáctico.

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

Relatar erros: exemplo 1

- Como exemplo podemos definir um *listener* que escreva também a pilha de regras do parser que estão activas.

```
import org.antlr.v4.runtime.*;
import java.util.List;
import java.util.Collections;

public class VerboseErrorListener extends BaseErrorListener {
    @Override public void syntaxError(Recognizer<?, ?> recognizer,
        Object offendingSymbol,
        int line, int charPositionInLine,
        String msg,
        RecognitionException e)
    {
        Parser p = ((Parser) recognizer);
        List<String> stack = p.getRuleInvocationStack();
        Collections.reverse(stack);
        System.err.println("rule stack: "+stack);
        System.err.println("line "+line+": "+charPositionInLine+
            " at "+offendingSymbol+": "+msg);
    }
}
```

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Relatar erros: exemplo 1 (2)

- Podemos agora desactivar os *listeners* definidos por omissão e activar o novo *listener*:

```
...
AParser parser = new AParser(tokens);
parser.removeErrorListeners(); // remove ConsoleErrorListener
parser.addErrorListener(new VerboseErrorListener()); // add ours
parser.mainRule(); // parse as usual
...
```

- Note que podemos detectar a existência de erros após a análise sintáctica (já feito pelo `antlr4-main`):

```
...
parser.mainRule(); // parse as usual
if (parser.getNumberOfSyntaxErrors() > 0) {
    ...
}
```

- Podemos também passar todos os erros de reconhecimento de *tokens* para a análise sintáctica:

```
grammar AParser;
...
/**
Last rule in grammar to ensure all errors are passed to the parser
*/
ERROR: . ;
```

[Apresentação](#)

[Exemplos](#)

[Hello](#)

[Expr](#)

[Exemplo figuras](#)

[Exemplo visitor](#)

[Exemplo listener](#)

[Construção de gramáticas](#)

[Especificação de gramáticas](#)

[ANTLR4: Estrutura léxica](#)

[Comentários](#)

[Identificadores](#)

[Literais](#)

[Palavras reservadas](#)

[Acções](#)

[ANTLR4: Regras léxicas](#)

[Padrões léxicos típicos](#)

[Operador léxico "não ganancioso"](#)

[ANTLR4: Estrutura sintáctica](#)

[Secção de tokens](#)

[Acções no preâmbulo da gramática](#)

[ANTLR4: Regras sintácticas](#)

[Padrões sintácticos típicos](#)

[Precedência](#)

[Associatividade](#)

[Herança de gramáticas](#)

Relatar erros: exemplo 2

- Outro *listener* que escreva os erros numa janela gráfica:

```
import org.antlr.v4.runtime.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;

public class DialogErrorListener extends BaseErrorListener {
    @Override public void syntaxError(Recognizer<?, ?> recognizer,
        Object offendingSymbol, int line, int charPositionInLine,
        String msg, RecognitionException e)
    {
        Parser p = ((Parser)recognizer);
        List<String> stack = p.getRuleInvocationStack();
        Collections.reverse(stack);
        StringBuilder buf = new StringBuilder();
        buf.append("rule stack: "+stack+" ");
        buf.append("line "+line+": "+charPositionInLine+" at "+
            offendingSymbol+": "+msg);
        JDialog dialog = new JDialog();
        Container contentPane = dialog.getContentPane();
        contentPane.add(new JLabel(buf.toString()));
        contentPane.setBackground(Color.white);
        dialog.setTitle("Syntax error");
        dialog.pack();
        dialog.setLocationRelativeTo(null);
        dialog.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        dialog.setVisible(true);
    }
}
```

[Apresentação](#)

[Exemplos](#)

[Hello](#)

[Expr](#)

[Exemplo figuras](#)

[Exemplo visitor](#)

[Exemplo listener](#)

[Construção de gramáticas](#)

[Especificação de gramáticas](#)

[ANTLR4: Estrutura léxica](#)

[Comentários](#)

[Identificadores](#)

[Literais](#)

[Palavras reservadas](#)

[Ações](#)

[ANTLR4: Regras léxicas](#)

[Padrões léxicos típicos](#)

[Operador léxico "não ganancioso"](#)

[ANTLR4: Estrutura sintáctica](#)

[Secção de tokens](#)

[Ações no preâmbulo da gramática](#)

[ANTLR4: Regras sintáticas](#)

[Padrões sintáticos típicos](#)

[Precedência](#)

[Associatividade](#)

[Herança de gramáticas](#)

- A recuperação de erros é a operação que permite que o analisador sintáctico continue a processar a entrada depois de detectar um erro, por forma a se poder detectar mais do que um erro em cada compilação.
- Por omissão o ANTLR4 faz uma recuperação automática de erros que funciona razoavelmente bem.
- As estratégias seguidas pela ANTLR4 para esse fim são as seguintes:
 - inserção de *token*;
 - remoção de *token*;
 - ignorar *tokens* até sincronizar novamente a gramática com o fim da regra actual.
- (Não vamos detalhar mais este ponto.)

Apresentação

Exemplos

Hello

Expr

Exemplo *figuras*

Exemplo *visitor*

Exemplo *listener*

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de *tokens*

Ações no preâmbulo da gramática

ANTLR4: Regras sintácticas

Padrões sintácticos típicos

Precedência

Associatividade

Herança de gramáticas

ANTLR4: alterar estratégia de gestão de erros

- Por omissão a estratégia de gestão de erros do ANTLR4 tenta recuperar a análise sintáctica utilizando uma combinação das estratégias atrás sumariamente apresentadas.
- A interface `ANTLRErrorStrategy` permite a definição de novas estratégias, existindo duas implementações na biblioteca de suporte: `DefaultErrorStrategy` e `BailErrorStrategy`.
- A estratégia definida em `BailErrorStrategy` assenta na terminação imediata da análise sintáctica quando surge o primeiro erro.
- A documentação sobre como lidar com este problema pode ser encontrada na classe `Parser`.
- Para definir uma nova estratégia de gestão de erros utiliza-se o seguinte código:

```
...
AParser parser = new AParser(tokens);
parser.setErrorHandler(new BailErrorStrategy());
...
```

- Alternativamente pode-se colocar um `exit` na classe `ErrorListener` utilizada.

ANTLR4

Apresentação

Exemplos

Hello

Expr

Exemplo figuras

Exemplo visitor

Exemplo listener

Construção de gramáticas

Especificação de gramáticas

ANTLR4: Estrutura léxica

Comentários

Identificadores

Literais

Palavras reservadas

Ações

ANTLR4: Regras léxicas

Padrões léxicos típicos

Operador léxico "não ganancioso"

ANTLR4: Estrutura sintáctica

Secção de tokens

Ações no preâmbulo da gramática

ANTLR4: Regras sintáticas

Padrões sintáticos típicos

Precedência

Associatividade

Herança de gramáticas

Tema 3

Análise Semântica

Gramáticas de atributos, tabela de símbolos

Compiladores, 2^o semestre 2022-2023

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restringidas
por contexto

Miguel Oliveira e Silva, Artur Pereira
DETI, Universidade de Aveiro

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

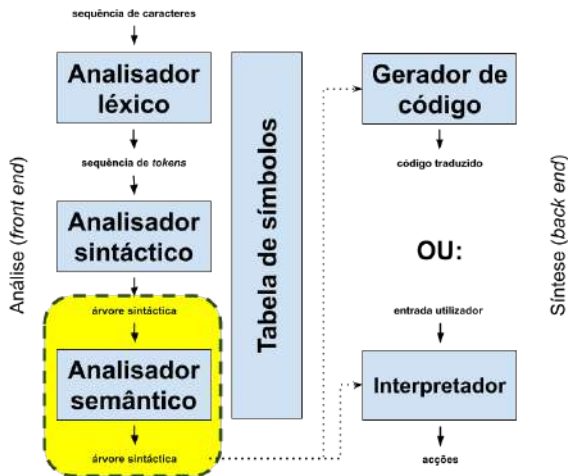
Agrupando símbolos em
contextos

Instruções restringidas
por contexto

Análise semântica

Análise semântica: Estrutura de um Compilador

- Vamos agora analisar com mais detalhe a fase de análise semântica:



- No processamento de uma linguagem a análise semântica deve garantir, tanto quanto possível, que o programa fonte faz sentido (mediante as regras definidas na linguagem).
- Erros semânticos comuns:
 - Variável/função não definida;
 - Tipos incompatíveis (e.g. atribuir número real a uma variável inteira, ou utilizar uma expressão não booleana na condições de uma instrução condicional);
 - Definir instrução num contexto errado (e.g. utilizar em `Java` a instrução `break` fora de um ciclo ou `switch`).
 - Aplicação sem sentido de instrução (e.g. importar uma *package* inexistente em `Java`).
- Em alguns casos, estes erros podem ser avaliados ainda durante a análise sintáctica; noutros casos, só é possível fazer essa verificação após uma análise sintáctica bem sucedida, fazendo uso da informação retirada dessa análise.

Análise semântica: Estrutura de um Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restringidas por contexto

- No processamento de linguagens, a avaliação semântica pode ser feita associando informação e acções às regras sintácticas da gramática (i.e. aos nós da **árvore sintáctica**).
- Este procedimento designa-se por **avaliação dirigida pela sintaxe**.
- Por exemplo, numa gramática para expressões aritméticas podemos associar aos nós da árvore uma variável com o tipo da expressão, e acções que permitam verificar a sua correcção (e não permitir, por exemplo, que se tente somar um booleano com um inteiro).
- Em ANTLR4, a associação de atributos e acções à árvore sintáctica, pode ser feita durante a própria análise sintáctica, e/ou posteriormente recorrendo a *visitors* e/ou *listeners*.

- A verificação de cada propriedade semântica de uma linguagem pode ser feita em dois tempos distintos:
 - Em **tempo dinâmico**: isto é, durante o **tempo de execução**;
 - Em **tempo estático**: isto é, durante o **tempo de compilação**.
- Só em compiladores fazem sentido verificações estáticas de propriedades semânticas.
- Em interpretadores as fases de análise e síntese da linguagem são ambas feitas em tempo de execução, pelo que as verificações são sempre dinâmicas.
- A verificação estática tem a vantagem de garantir, em tempo de execução, que certos erros nunca vão ocorrer (dispensando a necessidade de proceder à sua depuração e teste).

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Detecção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restritas
por contexto

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restringidas
por contexto

Sistema de tipos

- O sistema de tipos de uma linguagem de programação é um sistema lógico formal, com um conjunto de regras semânticas, que por associação de uma propriedade (tipo) a entidades da linguagem (expressões, variáveis, métodos, etc.) permite a detecção de uma classe importante de erros semânticos: *erros de tipos*.
- A verificação de erros de tipo, é aplicável nas seguintes operações:
 - Atribuição de valor: $v = e$
 - Aplicação de operadores: $e_1 + e_2$ (por exemplo)
 - Invocação de funções: $f(a)$
 - Utilização de classes/estruturas: $o.m(a)$ ou $data.field$
- Outras operações, como por exemplo a utilização arrays, podem também envolver verificações de tipo. No entanto, podemos considerar que as operações sobre arrays são atribuições de valor e aplicação de métodos especiais.

- Diz-se que qualquer uma destas operações é válida quando existe **conformidade** entre as propriedades de tipo das entidades envolvidas.
- A conformidade indica se um tipo T_2 pode ser usado onde se espera um tipo T_1 . É o que acontece quando $T_1 = T_2$.
 - Atribuição de valor ($v = e$).
O tipo de e tem de ser conforme com o tipo de v
 - Aplicação de operadores ($e_1 + e_2$).
Existe um operador $+$ aplicável aos tipos de e_1 e e_2
 - Invocação de funções ($f(a)$).
Existe uma função global f que aceita argumentos a conformes com os argumentos formais declarados dessa função.
 - Utilização de classes/estruturas ($o.m(a)$ ou $data.field$).
Existe um método m na classe correspondente ao objecto o , que aceita argumentos a conformes com os argumentos formais declarados desse método; e existe um campo $field$ na estrutura/classe de $data$.

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restringidas
por contexto

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restringidas
por contexto

Gramáticas de atributos

- Já vimos que atribuir sentido ao código fonte de uma linguagem requer, não só, correcção sintáctica (assegurada por gramáticas independentes de contexto) como também correcção semântica.
- Nesse sentido, é de toda a conveniência ter acesso a toda a informação gerada pela análise sintáctica, i.e. à árvore sintáctica, e poder associar nova informação aos respectivos nós.
- Este é o objectivo da **gramática de atributos**:
 - Cada símbolo da gramática da linguagem (terminal ou não terminal) pode ter a si associado um conjunto de zero ou mais **atributos**.
 - Um atributo pode ser um número, uma palavra, um tipo, ...
 - O cálculo de cada atributo tem de ser feito tendo em consideração a dependência da informação necessária para o seu valor.

Análise semântica: Estrutura de um Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restringidas por contexto

- Entre os diferentes tipos de atributos, existem alguns cujo valor depende apenas da sua vizinhança sintática.
 - Um desses exemplos é o valor de uma expressão aritmética (que para além disso, depende apenas do próprio nó e, eventualmente, de nós descendentes).
- Existem também atributos que (podem) depender de informação remota.
 - É o caso, por exemplo, do tipo de dados de uma expressão que envolva a utilização de uma variável ou invocação de um método.

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintática

Tabela de símbolos

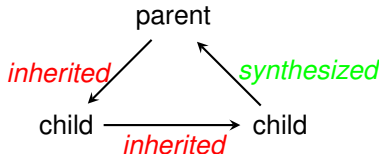
Agrupando símbolos em
contextos

Instruções restringidas
por contexto

Dependência local: classificação de atributos

- Os atributos podem ser classificados duas formas, consoante as dependências que lhes são aplicáveis:
 - 1 Dizem-se **sintetizados**, se o seu valor depende apenas de nós descendentes (i.e. se o seu valor depende apenas dos símbolos existentes no respectivo corpo da produção).
 - 2 Dizem-se **herdados**, se depende de nós "irmãos" ou de nós ascendentes.

parent \rightarrow child child



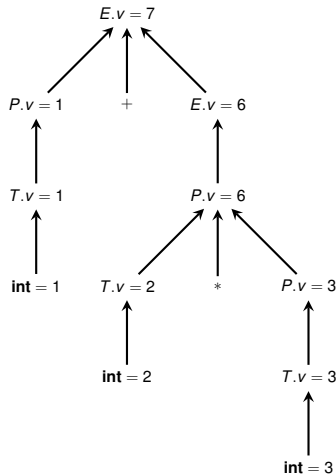
- Formalmente podem-se designar os atributos anotando com uma seta no sentido da dependência (para cima, nos atributos sintetizados; e para baixo nos herdados).

Exemplo dependência local: expressão aritmética

- Considere a seguinte gramática:

$$E \rightarrow P + E \mid P$$
$$P \rightarrow T * P \mid T$$
$$T \rightarrow (E) \mid \text{int}$$

- Se quisermos definir um atributo v para o valor da expressão, temos um exemplo de um atributo sintetizado.
- Por exemplo, para a entrada $1 + 2 * 3$ — temos a seguinte árvore sintática anotada:



Exemplo dependência local: expressão aritmética (2)

$$E \rightarrow P + E \mid P$$

$$P \rightarrow T * P \mid T$$

$$T \rightarrow (E) \mid \text{int}$$

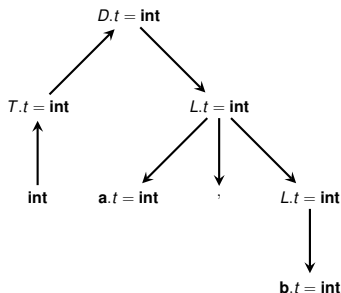
Produção	Regra semântica
$E_1 \rightarrow P + E_2$	$E_1.v = P.v + E_2.v$
$E \rightarrow P$	$E.v = P.v$
$P_1 \rightarrow T * P_2$	$P_1.v = T.v * P_2.v$
$P \rightarrow T$	$P.v = T.v$
$T \rightarrow (E)$	$T.v = E.v$
$T \rightarrow \text{int}$	$T.v = \text{int.value}$

Exemplo dependência local: declaração

- Considere a seguinte gramática:

$$D \rightarrow T L$$
$$T \rightarrow \text{int} \mid \text{real}$$
$$L \rightarrow \text{id}, L \mid \text{id}$$

- Se quisermos definir um atributo τ para indicar o tipo de cada variável **id**, temos um exemplo de um atributo herdado.
- Por exemplo, para a entrada — **int** a, b — temos a seguinte árvore sintática anotada:



Exemplo dependência local: declaração (2)

$$D \rightarrow T L$$
$$T \rightarrow \text{int} \mid \text{real}$$
$$L \rightarrow \text{id}, L \mid \text{id}$$

Produção	Regra semântica
$D \rightarrow T L$	$D.t = T.t$ $L.t = T.t$
$T \rightarrow \text{int}$	$T.t = \text{int}$
$T \rightarrow \text{real}$	$T.t = \text{real}$
$L_1 \rightarrow \text{id}, L_2$	$\text{id}.t = L_1.t$ $L_2.t = L_1.t$
$L \rightarrow \text{id}$	$\text{id}.t = L.t$

ANTLR4: Declaração de atributos associados à árvore sintáctica

- Podemos declarar atributos de formas distintas:
 - 1 Directamente na gramática independente de contexto recorrendo a argumentos e resultados de regras sintácticas;

```
expr[ String type ] returns[ int value ]: // type not used
    e1=expr '+' e2=expr
    { $value = $e1.value + $e2.value ;}          #ExprAdd
    |
    INT
    { $value = Integer.parseInt($INT.text); } #ExprInt
    ;
```

- 2 Indirectamente fazendo uso do *array* associativo `ParseTreeProperty`:

```
protected ParseTreeProperty<Integer> value =
    new ParseTreeProperty<>();
...
@Override public void exitInt(ExprParser.IntContext ctx){
    value.put(ctx, Integer.parseInt(ctx.INT().getText()));
}
...
@Override public void exitAdd(ExprParser.AddContext ctx){
    int left = value.get(ctx.e1);
    int right = value.get(ctx.e2);
    value.put(ctx, left + right);
}
```

- Podemos ainda utilizar o resultado dos métodos `visit`.

- Este *array* tem como chave nós da árvore sintáctica, e permite simular quer argumentos, quer resultados, de regras.
- A diferença está nos locais onde o seu valor é atribuído e acedido.
- Para simular a passagem de **argumentos** basta atribuir-lhe o valor **antes** de percorrer o respectivo nó (nos *listeners* usualmente nos métodos `enter...`), sendo o acesso feito no **próprio** nó.
- Para simular **resultados**, faz-se como no exemplo dado (i.e. atribui-se o valor no **próprio** nó, e acede-se nos nós **ascendentes**).

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restringidas
por contexto

- Podemos associar três tipos de informação a regras sintáticas:
 - 1 Informação com origem em regras utilizadas no corpo da regra (atributos sintetizados);
 - 2 Informação com origem em regras que utilizam esta regra no seu corpo (atributos herdados);
 - 3 Informação local à regra.
- Em ANTLR4 a utilização directa de todos estes tipos de atributos é muito simples e intuitiva:
 - 1 Atributos sintetizados: resultado de regras;
 - 2 Atributos herdados: argumentos de regras;
 - 3 Atributos locais.
- Alternativamente, podemos utilizar o *array* associativo `ParseTreeProperty` (que se justifica apenas para as duas primeiras, já que para a terceira podemos utilizar variáveis locais ao método respectivo); ou o resultado dos métodos `visit` (no caso de se utilizar *visitors*) para atributos sintetizados.

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restringidas
por contexto

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restringidas
por contexto

Tabela de símbolos

Tabela de símbolos

- A gramática de atributos é adequada para lidar com atributos com dependência local.
- No entanto, podemos ter informação cuja origem não tem dependência directa na árvore sintáctica (por exemplo, múltiplas aparições duma variável), ou que pode mesmo residir no processamento de outro código fonte (por exemplo, nomes de classes definidas noutra ficheiro).
- Assim, sempre que a linguagem utiliza símbolos para representar entidades do programa – como sejam: variáveis, funções, registos, classes, etc. – torna-se necessário associar à identificação do símbolo (geralmente um identificador) a sua definição (categoria do símbolo, tipo de dados associado).
- É para esse fim que existe a **tabela de símbolos**.
- A tabela de símbolos é um *array* associativo, em que a chave é o nome do símbolo, e o elemento um objecto que define o símbolo.
- As tabelas de símbolos podem ter um alcance global, ou local (por exemplo: a uma bloco de código ou a uma função).

Tabela de símbolos (2)

- A informação associada a cada símbolo depende do tipo de linguagem definida, assim como de estarmos na presença de um interpretador ou de um compilador.
- São exemplos dessas propriedades:
 - **Nome:** nome do símbolo (chave do *array* associativo);
 - **Categoria:** o que é que o símbolo representa, classe, método, variável de objecto, variável local, etc.;
 - **Tipo:** tipo de dados do símbolo;
 - **Valor:** valor associado ao símbolo (apenas no caso de interpretadores).
 - **Visibilidade:** restrição no acesso ao símbolo (para linguagens com encapsulamento).

Tabela de símbolos: implementação

- Numa aproximação orientada por objectos podemos definir a classe abstracta `Symbol`:

```
public abstract class Symbol {  
    public Symbol(String name, Type type) { ... }  
    public String name() { ... }  
    public Type type() { ... }  
}
```

- Podemos agora definir uma variável:

```
public class VariableSymbol extends Symbol {  
    public VariableSymbol(String name, Type type) {  
        super(name, type);  
    }  
}
```

Tabela de símbolos (3)

- A classe `Type` permite a identificação e verificação da conformidade entre tipos:

```
public abstract class Type {  
    protected Type(String name) { ... }  
    public String name() { ... }  
    public boolean subtype(Type other) {  
        assert other != null;  
        return name.equals(other.name());  
    }  
}
```

- Podemos agora implementar tipos específicos:

```
public class RealType extends Type {  
    public RealType() { super("real"); }  
}
```

```
public class IntegerType extends Type {  
    public IntegerType() { super("integer"); }  
  
    public boolean subtype(Type other) {  
        return super.subtype(other) ||  
            other.name().equals("real");  
    }  
}
```

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restringidas
por contexto

Agrupando símbolos em contextos

- Se a linguagem é simples, contendo um único contexto de definição de símbolos, então o tempo de vida dos símbolos está ligado ao tempo de vida do programa, sendo suficiente uma única tabela de símbolos.
- No entanto, se tivermos a possibilidade de definir símbolos em contextos diferentes, então precisamos de resolver o problema dos símbolos terem tempos de vida (e/ou visibilidade) que dependem do contexto dentro do programa.

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restritas
por contexto

Agrupando símbolos em contextos (2)

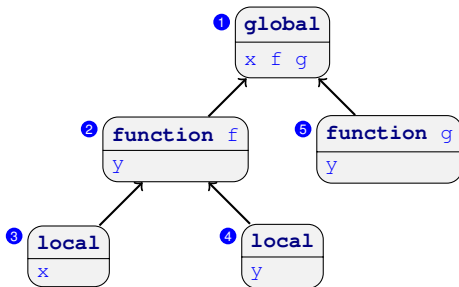
- Considere como exemplo o seguinte código (na linguagem C):

```
❶ // start of global scope
   int x;           // define variable x in global scope
❷ void f() {        // define function f in global scope
   int y;           // define variable y in local scope of f
❸ { int x; }        // define variable x in nested local scope
❹ { int y; }        // define variable y in another nested local scope
}
❺ void g() {        // define function g in global scope
   int y;           // define variable y in local scope of g
}
...
```

- A numeração identifica os diferentes contextos de símbolos.
- Um aspecto muito importante é o facto dos contextos poderem ser definidos dentro de outros contextos.
- Assim o contexto ❷ está definido dentro do contexto ❶; e, por sua vez, o contexto ❸ está definido dentro do ❷.
- Em ❹ o símbolo `x` está definido em ❶.

Agrupando símbolos em contextos (3)

- Para representar adequadamente esta informação estrutura-se as diferentes tabelas de símbolos numa árvore onde cada nó representa uma pilha de tabelas de símbolos a começar nesse nó até à raiz (tabela de símbolos global).



Agrupando símbolos em contextos (4)

- Consoante o ponto onde estamos no programa. temos uma pilha de tabelas de símbolos definida para resolver os símbolos.
- Pode haver repetição de nomes de símbolos, valendo o definido na tabela mais próxima (no ordem da pilha).
- Caso seja necessário percorrer a árvore sintáctica várias vezes, podemos registar numa lista ligada a sequência de pilhas de tabelas de símbolos que são aplicáveis em cada ponto do programa.

Análise semântica:
Estrutura de um
Compilador

Avaliação dirigida pela
sintaxe

Deteção estática ou
dinâmica

Sistema de tipos

Gramáticas de
atributos

Dependência local:
classificação de atributos

ANTLR4: Declaração de
atributos associados à
árvore sintáctica

Tabela de símbolos

Agrupando símbolos em
contextos

Instruções restringidas
por contexto

Instruções restringidas por contexto

Instruções restringidas por contexto

- Algumas linguagens de programação restringem a utilização de certas instruções a determinados contexto.
- Por exemplo, em Java as instruções `break` e `continue` só podem ser utilizadas dentro de ciclos ou da instrução condicional `switch`.
- A verificação semântica desta condição é muito simples de implementar, podendo ser feita durante a análise sintáctica recorrendo a predicados semânticos e um contador (ou uma pilha) que registre o contexto.

```
@parser::members {  
    int acceptBreak=0;  
}  
...  
forLoop: 'for' '(' expr ';' expr ';' expr ')'  
    { acceptBreak++;}  
    instruction  
    { acceptBreak--;}  
    ;  
break: { acceptBreak > 0}? 'break' ';' ;  
    ;  
instruction: forLoop | break | ...  
    ;
```

Tema 4

Síntese

Geração de código e gestão de erros

Compiladores, 2^o semestre 2022-2023

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

Miguel Oliveira e Silva, Artur Pereira
DETI, Universidade de Aveiro

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

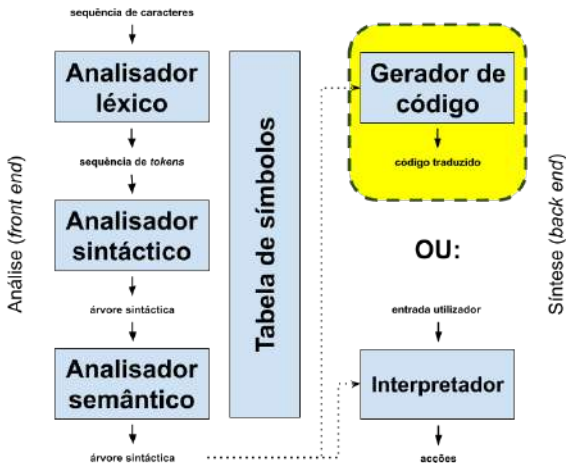
TAC: Endereços e instruções

Controlo de fluxo

Funções

Síntese: geração de código

Síntese: geração de código



Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

Síntese: geração de código (2)

- Podemos definir o objectivo de um compilador como sendo *traduzir* o código fonte de uma linguagem para outra linguagem.



- A geração do código para a linguagem destino pode ser feita por diferentes fases (podendo incluir fases de optimização), mas nós iremos abordar apenas uma única fase.
- A estratégia geral consiste em identificar **padrões de geração de código**, e após a análise semântica percorrer novamente a árvore sintáctica (mas já com a garantia muito importante de inexistência de erros sintácticos e semânticos) gerando o código destino nos pontos apropriados.

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

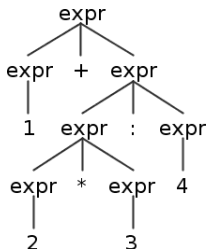
Controlo de fluxo

Funções

Exemplo: Calculadora

- Código fonte:

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



- Uma possível compilação para 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);
    }
}
```

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código:
padrões comuns

Geração de código para
expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de
expressões binárias

TAC: Endereços e
instruções

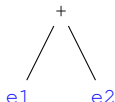
Controlo de fluxo

Funções

- Tradicionalmente, o ensino de processadores de linguagens tende a dar primazia à geração de código baixo nível (linguagem máquina, ou *assembly*).
- A larga maioria da bibliografia mantém esse enfoque.
- No entanto, do ponto de vista prático serão poucos os programadores que, fazendo uso de ferramentas para gerar processadores de linguagens, necessitam ou ambicionam este tipo de geração de código.
- Nesta disciplina vamos, alternativamente, discutir a geração de código numa perspectiva mais abrangente, incluindo a geração de código em linguagens de alto nível.

- No que diz respeito à geração de código em linguagens de baixo nível, é necessário um conhecimento robusto em arquitectura de computadores e lidar com os seguintes aspectos:
 - Representação e formato da informação (formato para números inteiros, reais, estruturas, *array*, etc.);
 - Gestão e endereçamento de memória;
 - Implementação de funções (passagem de argumentos e resultado, suporte para recursividade com pilha de chamadas e *frame pointers*);
 - Alocação de registos do processador.
- (Consultar a bibliografia recomendada para estudar este tipo de geração de código.)

- Seja qual for o nível da linguagem destino, uma possível estratégia para resolver este problema consiste em identificar sem ambiguidade **padrões de geração de código** associados a cada **elemento gramatical da linguagem**.
- Para esse fim, é necessário definir o contexto de geração de código para cada elemento (por exemplo, geração de instruções na linguagem destino, ou atribuir a valor a uma variável), e depois garantir que o mesmo é compatível com todas as utilizações do elemento.



$$\begin{aligned} & \dots (e_1) \\ & v_1 = e_1 \\ & \dots (e_2) \\ & v_2 = e_2 \\ & v_+ = v_1 + v_2 \end{aligned}$$

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

- Como a larguíssima maioria das linguagens destino são textuais, esses padrões de geração de código consistem em padrões de geração de texto.
- Assim sendo, em `Java`, poderíamos delegar esse problema no tipo de dados `String`, `StringBuilder`, ou mesmo na escrita directa de texto em em ficheiro (ou no *standard output*).
- No entanto, também aí o ambiente `ANTLR4` fornece uma ajuda mais estruturada, sistemática e modular para lidar com esse problema.

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

String Template

String Template

- A biblioteca (Java) *String Template* fornece uma solução estruturada para a geração de código textual.
- O software e documentação podem ser encontrados em <http://www.stringtemplate.org>
- Para ser utilizada é apenas necessário o pacote ST-4.?.jar (a instalação feita do antlr4 já incluiu este pacote).
- Vejamos um exemplo simples:

```
import org.stringtemplate.v4.*;
...
// code gen. pattern definition with <name> hole:
ST hello = new ST("Hello, <name>");
// hole pattern definition:
hello.add("name", "World");
// code generation (to standard output):
System.out.println(hello.render());
```

- Mesmo sendo um exemplo muito simples, podemos já verificar que a definição do padrão de texto, está separada do preenchimento dos “buracos” (atributos ou expressões) definidos, e da geração do texto final.

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

- Podemos assim delegar em partes diferentes do gerador de código, a definição dos padrões (que passam a pertencer ao contexto do elemento de código a gerar), o preenchimento dos “buracos” definidos, e a geração do texto final de código.
- Os padrões são blocos de texto e expressões.
- O texto corresponde a código destino literal, e as expressões são em “buracos” que podem ser preenchidos com o texto que se quiser.
- Sintaticamente, as expressões são identificadores delimitados por `<expr>` (ou por `$`).

```
import org.stringtemplate.v4.*;
...
ST assign = new ST("<var> = <expr>;\n");
assign.add("var", "i");
assign.add("expr", "10");
String output = assign.render();
System.out.println(output);
```

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

- Podemos também agrupar os padrões numa espécie de funções (módulo `STGroup`):

```
import org.stringtemplate.v4.*;
...
STGroup group = new STGroupString(
    "assign(var,expr) ::= \"<var> = <expr>;\" "
);
ST assign = group.getInstanceOf("assign");
assign.add("var", "i");
assign.add("expr", "10");
String output = assign.render();
System.out.println(output);
```

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

- Podemos também colocar cada função num ficheiro:

```
// file assign.st
assign(var, expr) ::= "<var> = <expr>;"
```

```
import org.stringtemplate.v4.*;
...
// assuming that assign.st is in current directory:
STGroup group = new STGroupDir(".");
ST assign = group.getInstanceOf("assign");
assign.add("var", "i");
assign.add("expr", "10");
String output = assign.render();
System.out.println(output);
```

- Uma melhor opção é optar por ficheiros modulares contendo grupos de funções/padrões:

```
// file templates.stg

templateName(arg1, arg2, ..., argN) ::= "single-line template"

templateName(arg1, arg2, ..., argN) ::= <<
multi-line template preserving indentation and newlines
>>

templateName(arg1, arg2, ..., argN) ::= <%
multi-line template that ignores indentation and newlines
%>
```

```
import org.stringtemplate.v4.*;
...
// assuming that templates.stg is in current directory:
STGroup allTemplates = new STGroupFile("templates.stg");
ST st = group.getInstanceOf("templateName");
...
```

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

String Template: dicionários e condicionais

- Nestes módulos podemos ainda definir dicionários (arrays associativos).

```
typeValue ::= [  
  "integer": "int",  
  "real": "double",  
  "boolean": "boolean",  
  default: "void"  
]
```

- Na definição de padrões podemos utilizar uma instrução condicional que só aplica o padrão caso o atributo seja adicionado:

```
stats(stat) ::= <<  
<if (stat)><stat; separator="\n"><endif>  
>>
```

- O campo `separator` indica que em cada operação `add` em `stat`, se irá utilizar esse separador (no caso, uma mudança de linha).

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

- Podemos ainda definir padrões utilizando outros padrões (como se fossem funções).

```
module(name, stat) ::= <<
public class <name> {
    public static void main(String[] args) {
        <stats (stat)>
    }
}
>>

conditional(stat, var, stat_true, stat_false) ::= <<
<stats (stat)>
if (<var>) {
    <stat_true>
}< if (stat_false)>
else {
    <stat_false>
}< endif>
>>
```

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

- Também existe a possibilidade de utilizar listas para concatenar texto e argumentos de padrões:

```
binaryExpression ( type , var , e1 , op , e2 ) ::=  
    "<decl ( type , var , [ e1 , \" \" , op , \" \" , e2 ] ) > "
```

- OU:

```
binaryExpression ( type , var , e1 , op , e2 ) ::= <<  
<decl ( type , var , [ e1 , \" \" , op , \" \" , e2 ] ) >  
>>
```

- Para mais informação sobre as possibilidades desta biblioteca devem consultar a documentação existente em:
<http://www.stringtemplate.org>.

Geração de código: padrões comuns

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

Geração de código: padrões comuns

- Uma geração de código modular requer um contexto uniforme que permita a inclusão de qualquer combinação de código a ser gerado.
- Na sua forma mais simples, o padrão comum pode ser simplesmente uma sequência de instruções.

```
stats ( stat ) ::= <<
< if ( stat ) > < stat ; separator = "\n" > < endif >
>>

module ( name , stat ) ::= <<
public class < name >
{
    public static void main ( String [] args )
    {
        < stats ( stat ) >
    }
}
>>
```

- Com este padrão, podemos inserir no lugar do “buraco” `stat` a sequência de instruções que quisermos.
- Naturalmente, que para uma geração de código mais complexa podemos considerar a inclusão de buracos para membros de classe, múltiplas classes, ou mesmo vários ficheiros.

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

Geração de código: padrões comuns (2)

- Para a linguagem C, teríamos o seguinte padrão para um módulo de compilação:

```
stats(stat) ::= <<
<if (stat)><stat; separator="\n"><endif>
>>

module(name, stat) ::= <<
#include <stdio.h>
#include <math.h>

int main()
{
    <stats(stat)>
}
>>
```

- Se a geração de código for guiada pela árvore sintáctica (como normalmente acontece), então os padrões de código a ser gerados devem ter em conta as definições gramaticais de cada símbolo, permitindo a sua aplicação modular em cada contexto.

Geração de código para expressões

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

Geração de código para expressões

- Para ilustrar a simplicidade e poder de abstração do *String Template* vamos estudar o problema de geração de código para expressões.
- Para resolver este problema de uma forma modular, podemos utilizar a seguinte estratégia:
 - 1 considerar que qualquer expressão tem a si associada uma variável (na linguagem destino) com o seu valor;
 - 2 para além dessa associação, podemos também associar a cada expressão um `ST (stats)` com as instruções que atribuem o valor adequado à variável.
- Como habitual, para fazer estas associações podemos definir atributos na gramática, fazer uso do resultados das funções de um *Visitor* ou utilizar a classe `ParseTreeProperty`
- Desta forma, podemos fácil e de uma forma modular, gerar código para qualquer tipo de expressão.

Geração de código para expressões (2)

- Padrões para expressões (para Java) podem ser:

```
typeValue ::= [  
    "integer":"int", "real":"double",  
    "boolean":"boolean", default:"void"  
]  
  
init(value) ::= "<if(value)> = <value><endif>"  
decl(type, var, value) ::=  
    "<typeValue.(type)> <var><init(value)>;"  
  
binaryExpression(type, var, e1, op, e2) ::=  
    "<decl(type, var, [e1, \" \", op, \" \", e2])>"
```

- Para C apenas seria necessário mudar o padrão

typeValue:

```
typeValue ::= [  
    "integer":"int", "real":"double",  
    "boolean":"int", default:"void"  
]
```

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

Exemplo: compilador simples

Código de triplo endereço

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

- O padrão para expressões é um exemplo duma representação muito utilizada para geração de código baixo nível (em geral, intermédio, e não final), designada por **codificação de triplo endereço** (TAC).
- Esta designação tem origem nas instruções com a forma:
 $x = y \text{ op } z$
- No entanto, para além desta operação típica de expressões binárias, esta codificação contém outras instruções (ex: operações unárias e de controlo de fluxo).
- No máximo, cada instrução tem três operandos (i.e. três variáveis ou endereços de memória).
- Tipicamente, cada instrução TAC realiza uma operação elementar (e já com alguma proximidade com as linguagens de baixo nível dos sistemas computacionais).

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

TAC: Exemplo de expressões binárias

- Por exemplo a expressão $a + b * (c + d)$ pode ser transformada na sequência TAC:

```
t8 = d ;  
t7 = c ;  
t6 = t7 + t8 ;  
t5 = t6 ;  
t4 = b ;  
t3 = t4 * t5 ;  
t2 = a ;  
t1 = t2 + t3 ;
```

- Esta sequência – embora fazendo uso desregrado no número de registos (o que, num compilador gerador de código máquina, é resolvido numa fase posterior de optimização) – é codificável em linguagens de baixo nível.

- Nesta codificação, um endereço pode ser:
 - Um nome do código fonte (variável, ou endereço de memória);
 - Uma constante (i.e. um valor literal);
 - Um nome temporário (variável, ou endereço de memória), criado na decomposição TAC.
- As instruções típicas do TAC são:
 - 1 Atribuições de valor de operação binária: $x = y \text{ op } z$
 - 2 Atribuições de valor de operação unária: $x = \text{op } y$
 - 3 Instruções de cópia: $x = y$
 - 4 Saltos incondicionais e etiquetas: **goto** L e **label** L :
 - 5 Saltos condicionais: **if** x **goto** L ou **ifFalse** x **goto** L
 - 6 Saltos condicionais com operador relacional:
if x **relop** y **goto** L (o operador pode ser de igualdade ou ordem)
 - 7 Invocações de procedimentos (**param** $x_1 \dots \text{param } x_n$;
call p, n ; $y = \text{call } p, n$; **return** y)
 - 8 Instruções com arrays (i.e. o operador é os parêntesis rectos, e um dos operandos é o índice inteiro).
 - 9 Instruções com ponteiros para memória (como em C)

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

- As instruções de controlo de fluxo são as instruções condicionais e os ciclos.
- Em linguagens de baixo nível muitas vezes estas instruções não existem.
- O que existe em alternativa é a possibilidade de dar “saltos” dentro do código recorrendo a endereços (*labels*) e a instruções de salto (*goto*, ...).

```
if (cond) {  
    A;  
}  
else {  
    B;  
}
```

```
ifFalse cond goto l1  
A  
goto l2  
label l1 :  
B  
label l2 :
```

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

Controlo de fluxo (2)

- De forma similar podemos gerar código para ciclos:

```
while ( cond ) {  
    A;  
}
```

```
label l1 :  
ifFalse cond goto l2  
A  
goto l1  
label l2 :
```

- A geração de código para funções pode ser feita recorrendo a uma estratégia tipo “macro”, ou implementando módulos algorítmicos separados.
- Neste último caso, é necessária a definição de um bloco algorítmico separado, assim como implementar a passagem de argumentos/resultado para/de a função.
- A passagem de argumentos pode seguir diferentes estratégias: passagem por valor, passagem por referência de variáveis, passagem por referência de objectos/registos.
- Para termos implementações recursivas é necessário que se definam novas variáveis em cada invocação da função.
- A estrutura de dados que nos permite fazer isso de uma forma muito eficiente e simples é a pilha de execução.
- Esta pilha armazena os argumentos, variáveis locais à função e o resultado da função (permitindo ao código que invoca a função não só passar os argumentos à função como ir buscar o seu resultado).

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções

Funções (2)

- Geralmente as arquitecturas de linguagens de baixo nível (CPU's) têm instruções específicas para lidar com esta estrutura de dados.
- Vamos exemplificar esse procedimento:

```
// use:  
... f(x,y);  
...  
// define:  
int f(int a, int b) {  
    A;  
    return r;  
}
```

```
// use:  
push 0 // result  
push x  
push y  
call f,2  
pop r // result  
...  
// define:  
label f:  
pop b  
pop a  
pop r  
store stack-position  
A  
// reset stack to stack-position  
restore stack-position  
push r  
return
```

Síntese: geração de código

Geração de código máquina

Geração de código

String Template

Geração de código: padrões comuns

Geração de código para expressões

Síntese: geração de código intermédio

Código de triplo endereço

TAC: Exemplo de expressões binárias

TAC: Endereços e instruções

Controlo de fluxo

Funções



Compiladores

Linguagens Regulares, Expressões Regulares e Gramáticas Regulares

Artur Pereira <artur@ua.pt>,
Miguel Oliveira e Silva <mos@ua.pt>

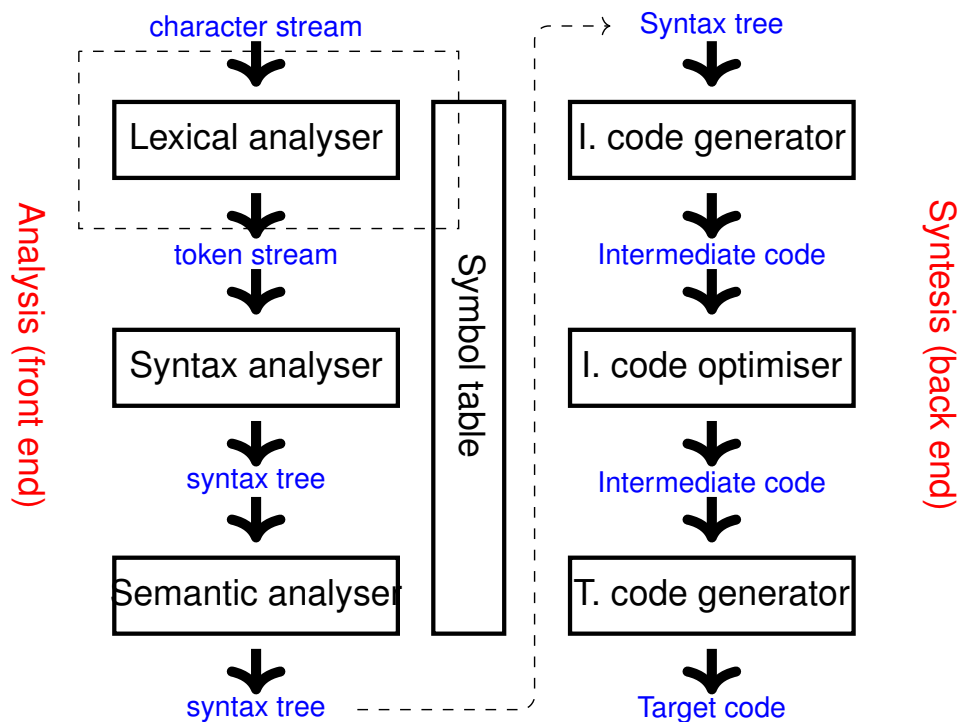
DETI, Universidade de Aveiro

Ano letivo de 2022-2023

Sumário

- ① Análise lexical revisitada
- ② Linguagens regulares
- ③ Expressões regulares
- ④ Gramáticas regulares
- ⑤ Equivalência entre expressões regulares e gramáticas regulares

Papel da análise lexical



Papel da análise lexical

- Converte a sequência de caracteres numa sequência de *tokens*
- Um *token* é um tuplo `<token-name, attribute-value>`
 - `token-name` é um símbolo (abstrato) representando um tipo de entrada
 - `attribute-value` representa o valor corrente desse símbolo

- Exemplo:

`pos = pos + vel * 5;`

é convertido em

`<ID, "pos"> <=> <ID, "pos"> <+> <ID, "vel">
<*> <INT, 5>`

- Tipicamente, alguns símbolos são descartados pelo analisador lexical
- O conjunto dos *tokens* corresponde a uma linguagem regular
 - os *tokens* são descritos usando expressões regulares e/ou gramáticas regulares
 - são reconhecidos usando autómatos finitos

Linguagem regular

Definição

A classe das **linguagens regulares** sobre o alfabeto A define-se indutivamente da seguinte forma:

- 1 O conjunto vazio, \emptyset , é uma linguagem regular (LR).
- 2 Qualquer que seja o $a \in A$, o conjunto $\{a\}$ é uma LR.

Note que:

- em $a \in A$, a é uma letra do alfabeto
- em $\{a\}$, a é uma palavra com apenas uma letra
- Numa analogia Java, o primeiro é um 'a' e o segundo um "a"

Linguagem regular

Definição

A classe das **linguagens regulares** sobre o alfabeto A define-se indutivamente da seguinte forma:

- 1 O conjunto vazio, \emptyset , é uma linguagem regular (LR).
- 2 Qualquer que seja o $a \in A$, o conjunto $\{a\}$ é uma LR.
- 3 Se L_1 e L_2 são LR, então a sua reunião ($L_1 \cup L_2$) é uma LR.

Exemplo:

- Seja $L_1 = \{ab, c\}$, uma LR sobre o alfabeto $A = \{a, b, c\}$
- e $L_2 = \{bb, c\}$, outra LR sobre o mesmo alfabeto A
- então, $L_3 = L_1 \cup L_2 = \{ab, bb, c\}$ é uma LR sobre o mesmo alfabeto A

Linguagem regular

Definição

A classe das **linguagens regulares** sobre o alfabeto A define-se indutivamente da seguinte forma:

- 1 O conjunto vazio, \emptyset , é uma linguagem regular (LR).
- 2 Qualquer que seja o $a \in A$, o conjunto $\{a\}$ é uma LR.
- 3 Se L_1 e L_2 são LR, então a sua reunião ($L_1 \cup L_2$) é uma LR.
- 4 Se L_1 e L_2 são LR, então a sua concatenação ($L_1 \cdot L_2$) é uma LR.

Exemplo:

- Seja $L_1 = \{ab, c\}$, uma LR sobre o alfabeto $A = \{a, b, c\}$
- e $L_2 = \{bb, c\}$, outra LR sobre o mesmo alfabeto A
- então, $L_3 = L_1 \cdot L_2 = \{abbb, abc, cbb, cc\}$ é uma LR sobre o mesmo alfabeto A

Linguagem regular

Definição

A classe das **linguagens regulares** sobre o alfabeto A define-se indutivamente da seguinte forma:

- 1 O conjunto vazio, \emptyset , é uma linguagem regular (LR).
- 2 Qualquer que seja o $a \in A$, o conjunto $\{a\}$ é uma LR.
- 3 Se L_1 e L_2 são LR, então a sua reunião ($L_1 \cup L_2$) é uma LR.
- 4 Se L_1 e L_2 são LR, então a sua concatenação ($L_1 \cdot L_2$) é uma LR.
- 5 Se L_1 é uma LR, então o seu fecho de Kleene (L_1^*) é uma LR.

Exemplo:

- Seja $L_1 = \{ab, c\}$, uma LR sobre o alfabeto $A = \{a, b, c\}$
- então, $L_2 = L_1^* = \{\varepsilon, ab, c, abab, abc, cab, cc, \dots\}$ é uma LR sobre o mesmo alfabeto

Linguagem regular

Definição

A classe das **linguagens regulares** sobre o alfabeto A define-se indutivamente da seguinte forma:

- ① O conjunto vazio, \emptyset , é uma linguagem regular (LR).
- ② Qualquer que seja o $a \in A$, o conjunto $\{a\}$ é uma LR.
- ③ Se L_1 e L_2 são LR, então a sua reunião ($L_1 \cup L_2$) é uma LR.
- ④ Se L_1 e L_2 são LR, então a sua concatenação ($L_1 \cdot L_2$) é uma LR.
- ⑤ Se L_1 é uma LR, então o seu fecho de Kleene (L_1^*) é uma LR.
- ⑥ Nada mais é LR.

Note que

- $\{\varepsilon\}$ é uma LR, uma vez que $\{\varepsilon\} = \emptyset^*$.

Definição de linguagem regular

exemplo #1

Q Mostre que a linguagem L , constituída pelo conjunto dos números binários começados em 1 e terminados em 0 é uma LR sobre o alfabeto $A = \{0, 1\}$

R

- pela regra 2 (elementos primitivos), $\{0\}$ e $\{1\}$ são LR
- pela regra 3 (união), $\{0, 1\} = \{0\} \cup \{1\}$ é uma LR
- pela regra 5 (fecho), $\{0, 1\}^*$ é uma LR
- pela regra 4 (concatenação), $\{1\} \cdot \{0, 1\}^*$ é uma LR
- pela regra 4, $(\{1\} \cdot \{0, 1\}^*) \cdot \{0\}$ é uma LR
- logo, $L = \{1\} \cdot \{0, 1\}^* \cdot \{0\}$ é uma LR

Expressões regulares

Definição

O conjunto das **expressões regulares** sobre o alfabeto A define-se indutivamente da seguinte forma:

- 1 \emptyset é uma expressão regular (ER) que representa a LR $\{\}$.
- 2 Qualquer que seja o $a \in A$, a é uma ER que representa a LR $\{a\}$.
- 3 Se e_1 e e_2 são ER representando respetivamente as LR L_1 e L_2 , então $(e_1|e_2)$ é uma ER representando a LR $L_1 \cup L_2$.
- 4 Se e_1 e e_2 são ER representando respetivamente as LR L_1 e L_2 , então (e_1e_2) é uma ER representando a LR $L_1.L_2$.
- 5 Se e_1 é uma ER representando a LR L_1 , então $(e_1)^*$ é uma ER representando a LR $(L_1)^*$.
- 6 Nada mais é expressão regular.

-
- É habitual representar-se por ε a ER \emptyset^* . Representa a linguagem $\{\varepsilon\}$.

Expressões regulares

Precedência dos operadores regulares

- Na escrita de expressões regulares assume-se a seguinte precedência dos operadores:
 - fecho ($*$)
 - concatenação
 - escolha ($|$).
- O uso destas precedências permite a queda de alguns parêntesis e consequentemente uma notação simplificada.

-
- Exemplo: a expressão regular

$e_1|e_2 e_3^*$

recorre a esta precedência para representar a expressão regular

$(e_1)|(e_2 ((e_3)^*))$

Expressões regulares

Exemplos

Q Determine uma ER que represente o conjunto dos números binários começados em 1 e terminados em 0.

R $1(0|1)^*0$

Q Determine uma ER que represente as sequências definidas sobre o alfabeto $A = \{a, b, c\}$ que satisfazem o requisito de qualquer b ter um a imediatamente à sua esquerda e um c imediatamente à sua direita.

R O a pode aparecer sozinho; o c também; o b , se aparecer, tem de ter um a à sua esquerda e um c à sua direita. Ou seja, pode considerar-se que as palavras da linguagem são sequências de 0 ou mais a , c ou abc .

$(a|abc|c)^*$

Q Determine uma ER que represente as sequências binárias com um número par de zeros.

R $(1^*01^*01^*)^*|1^* = 1^*(01^*01^*)^*$

Expressões regulares

Propriedades da operação de escolha

- A operação de escolha goza das propriedades:
 - comutativa: $e_1 | e_2 = e_2 | e_1$
 - associativa: $e_1 | (e_2 | e_3) = (e_1 | e_2) | e_3 = e_1 | e_2 | e_3$
 - idempotência: $e_1 | e_1 = e_1$
 - existência de elemento neutro: $e_1 | \emptyset = \emptyset | e_1 = e_1$

- Exemplo:

- comutativa: $a | ab = ab | a$
- associativa: $a | (b | ca) = (a | b) | ca = a | b | ca$
- idempotência: $ab | ab = ab$
- não há interesse prático em fazer uma união com o conjunto vazio

Expressões regulares

Propriedades da operação de concatenação

- A operação de concatenação goza das propriedades:
 - associativa: $e_1(e_2e_3) = (e_1e_2)e_3 = e_1e_2e_3$
 - existência de elemento neutro: $e_1\varepsilon = \varepsilon e_1 = e_1$
 - existência de elemento absorvente: $e_1\emptyset = \emptyset e_1 = \emptyset$
 - **não goza da propriedade comutativa**

-
- Exemplo: seja $e_1 = a$, $e_2 = bc$, e $e_3 = c$
 - associativa: $a(bc\ c) = (a\ bc)c = a\ bc\ c$

Expressões regulares

Propriedades distributivas

- A combinação das operações de concatenação e escolha gozam das propriedades:
 - distributiva à esquerda da concatenação em relação à escolha:
$$e_1(e_2 \mid e_3) = e_1e_2 \mid e_1e_3$$
 - distributiva à direita da concatenação em relação à escolha:
$$(e_1 \mid e_2)e_3 = e_1e_3 \mid e_2e_3$$

-
- Exemplo:
 - distributiva à esquerda da concatenação em relação à escolha:
$$ab(a \mid cc) = aba \mid abcc$$
 - distributiva à direita da concatenação em relação à escolha:
$$(ab \mid a)cc = abcc \mid acc$$

Expressões regulares

Propriedades da operação de fecho de Kleene

- A operação de fecho goza das propriedades:

- $(e^*)^* = e^*$
- $(e_1^* \mid e_2^*)^* = (e_1 \mid e_2)^*$
- $(e_1 \mid e_2^*)^* = (e_1 \mid e_2)^*$
- $(e_1^* \mid e_2)^* = (e_1 \mid e_2)^*$

- Mas atenção:

- $(e_1 \mid e_2)^* \neq e_1^* \mid e_2^*$
- $(e_1 e_2)^* \neq e_1^* e_2^*$

- Exemplo:

- $b(a^*)^* = b a^*$
- $(a^* \mid b^*)^* = (a \mid b)^*$
- $(a \mid b^*)^* = (a \mid b)^*$
- $(a^* \mid b)^* = (a \mid b)^*$
- $(a \mid b)^* \neq a^* \mid b^*$
- $(ab)^* \neq a^* b^*$

Expressões regulares

Exemplos

- Q Sobre o alfabeto $A = \{0, 1\}$ construa uma expressão regular que represente a linguagem

$$L = \{\omega \in A^* : \#(0, \omega) = 2\}$$

$$\mathcal{R} \ 1^* 0 1^* 0 1^*$$

- Q Sobre o alfabeto $A = \{a, b, \dots, z\}$ construa uma expressão regular que represente a linguagem

$$L = \{\omega \in A^* : \#(a, \omega) = 3\}$$

$$\mathcal{R} \ (b|c|\dots|z)^* a (b|c|\dots|z)^* a (b|c|\dots|z)^* a (b|c|\dots|z)^*$$

- Na última resposta, onde estão as reticências (...) deveriam estar todas as letras entre d e y. Parece claro que faz falta uma forma de simplificar este tipo de expressões

Expressões regulares

Extensões notacionais comuns

- uma ou mais ocorrências:

$$e^+ = e.e^*$$

- uma ou nenhuma ocorrência:

$$e? = (e|\varepsilon)$$

- um símbolo do sub-alfabeto dado:

$$[a_1a_2a_3 \cdots a_n] = (a_1 | a_2 | a_3 | \cdots | a_n)$$

- um símbolo do sub-alfabeto dado:

$$[a_1 - a_n] = (a_1 | \cdots | a_n)$$

- um símbolo do alfabeto fora do conjunto dado:

$$[\hat{a}_1a_2a_3 \cdots a_n], \quad [\hat{a}_1 - a_n]$$

Em ANTLR:

- $x..y$ é equivalente a $[x-y]$
- $\sim[abc]$ é equivalente a $[\hat{a}bc]$

Expressões regulares

Outras extensões notacionais

- n ocorrências de:

$$e\{n\} = \underbrace{e.e.\cdots.e}_n$$

- de n_1 a n_2 ocorrências:

$$e\{n_1, n_2\} = \underbrace{e.e.\cdots.e}_{n_1, n_2}$$

- n ou mais ocorrências:

$$e\{n, \} = \underbrace{e.e.\cdots.e}_{n,}$$

- $.$ representa um símbolo qualquer

- \wedge representa palavra vazia no início de linha

- $\$$ representa palavra vazia no fim de linha

- $\backslash <$ representa palavra vazia no início de palavra

- $\backslash >$ representa palavra vazia no fim de palavra

Em ANTLR:

- Pode ser feito através de predicados semânticos

Expressões regulares

Exemplos de extensões notacionais

- Q Sobre o alfabeto $A = \{0, 1\}$ construa uma expressão regular que reconheça a linguagem

$$L = \{\omega \in A^* : \#(0, \omega) = 2\}$$

$$\mathcal{R} \ 1^* 0 1^* 0 1^* = (1^* 0) (1^* 0) 1^* = (1^* 0) \{2\} 1^*$$

- Q Sobre o alfabeto $A = \{a, b, \dots, z\}$ construa uma expressão regular que reconheça a linguagem

$$L = \{\omega \in A^* : \#(a, \omega) = 3\}$$

$$\begin{aligned} \mathcal{R} \ (b|c|\dots|z)^* a (b|c|\dots|z)^* a (b|c|\dots|z)^* a (b|c|\dots|z)^* \\ = ([b-z]^* a) ([b-z]^* a) ([b-z]^* a) [b-z]^* \\ = ([b-z]^* a) \{3\} [b-z]^* \end{aligned}$$

Gramáticas regulares

Introdução

- Exemplo de gramática regular

$$\begin{array}{l} S \rightarrow a \ X \\ X \rightarrow a \ X \\ \quad | \ b \ X \\ \quad | \ \epsilon \end{array}$$

- Exemplo de gramática **não** regular

$$\begin{array}{l} S \rightarrow a \ S \ a \\ \quad | \ b \ S \ b \\ \quad | \ a \end{array}$$

- Letras minúsculas representam símbolos terminais e letras maiúsculas representam símbolos não terminais (o contrário do ANTLR)
- Nas gramáticas regulares os símbolos não terminais apenas podem aparecer no fim

Gramáticas regulares

Definição

Uma **gramática regular** é um quádruplo $G = (T, N, P, S)$, onde

- T é um conjunto finito não vazio de símbolos terminais;
- N , sendo $N \cap T = \emptyset$, é um conjunto finito não vazio de símbolos não terminais;
- P é um conjunto de produções (ou regras de rescrita), cada uma da forma $\alpha \rightarrow \beta$, onde
 - $\alpha \in N$
 - $\beta \in T^* \cup T^* N$
- $S \in N$ é o símbolo inicial.

-
- A linguagem gerada por uma gramática regular é regular
 - Logo, é possível converter-se uma gramática regular numa expressão regular que represente a mesma linguagem e vice-versa

Gramáticas regulares

Operações sobre gramáticas regulares

- As gramáticas regulares são fechadas sob as operações de
 - reunião
 - concatenação
 - fecho
 - intersecção
 - complementação
- As operações de intersecção e complementação serão abordadas mais adiante através de autómatos finitos

Reunião de gramáticas regulares

Exemplo

Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática regular que represente a linguagem

$$L = L_1 \cup L_2$$

sabendo que

$$L_1 = \{\omega a : \omega \in T^*\} \quad L_2 = \{a\omega : \omega \in T^*\}$$

R

$$\begin{array}{ll} S_1 \rightarrow a S_1 & S_2 \rightarrow a X_2 \\ | b S_1 & X_2 \rightarrow a X_2 \\ | c S_1 & | b X_2 \\ | a & | c X_2 \\ & | \varepsilon \end{array}$$

- Comece-se por obter as gramáticas regulares que representam L_1 e L_2 .

Reunião de gramáticas regulares

Exemplo

Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática regular que represente a linguagem

$$L = L_1 \cup L_2$$

sabendo que

$$L_1 = \{\omega a : \omega \in T^*\} \quad L_2 = \{a\omega : \omega \in T^*\}$$

R

$$\begin{array}{lll} S_1 \rightarrow a S_1 & S_2 \rightarrow a X_2 & S \rightarrow S_1 \mid S_2 \\ | b S_1 & X_2 \rightarrow a X_2 & S_1 \rightarrow a S_1 \mid b S_1 \mid c S_1 \\ | c S_1 & | b X_2 & | a \\ | a & | c X_2 & S_2 \rightarrow a X_2 \\ & | \varepsilon & X_2 \rightarrow a X_2 \mid b X_2 \mid c X_2 \\ & & | \varepsilon \end{array}$$

- E acrescentam-se as transições $S \rightarrow S_1$ e $S \rightarrow S_2$ que permitem escolher as palavras de L_1 e de L_2 , sendo S o novo símbolo inicial.

Reunião de gramáticas regulares

Algoritmo

\mathcal{D} Sejam $G_1 = (T_1, N_1, P_1, S_1)$ e $G_2 = (T_2, N_2, P_2, S_2)$ duas gramáticas regulares quaisquer, com $N_1 \cap N_2 = \emptyset$. A gramática $G = (T, N, P, S)$ onde

$$T = T_1 \cup T_2$$

$$N = N_1 \cup N_2 \cup \{S\} \quad \text{com} \quad S \notin (N_1 \cup N_2)$$

$$P = \{S \rightarrow S_1, S \rightarrow S_2\} \cup P_1 \cup P_2$$

é regular e gera a linguagem $L = L(G_1) \cup L(G_2)$.

- Para $i = 1, 2$, a nova produção $S \rightarrow S_i$ permite que G gere a linguagem $L(G_i)$

Concatenação de gramáticas regulares

Exemplo

\mathcal{Q} Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática regular que represente a linguagem

$$L = L_1 \cdot L_2$$

sabendo que

$$L_1 = \{\omega a : \omega \in T^*\} \quad L_2 = \{a\omega : \omega \in T^*\}$$

\mathcal{R}

$S_1 \rightarrow a \ S_1$	$S_2 \rightarrow a \ X_2$
$b \ S_1$	$X_2 \rightarrow a \ X_2$
$c \ S_1$	$b \ X_2$
a	$c \ X_2$
	ε

- Comece-se por obter as gramáticas regulares que representam L_1 e L_2 .

Concatenação de gramáticas regulares

Exemplo

- Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática regular que represente a linguagem

$$L = L_1 \cdot L_2$$

sabendo que

$$L_1 = \{\omega a : \omega \in T^*\} \quad L_2 = \{a\omega : \omega \in T^*\}$$

R

$S_1 \rightarrow a S_1$	$S_2 \rightarrow a X_2$	$S_1 \rightarrow a S_1 \mid b S_1 \mid c S_1$
$\mid b S_1$	$X_2 \rightarrow a X_2$	$\mid a S_2$
$\mid c S_1$	$\mid b X_2$	$S_2 \rightarrow a X_2$
$\mid a$	$\mid c X_2$	$X_2 \rightarrow a X_2 \mid b X_2 \mid c X_2$
	$\mid \varepsilon$	

- A seguir substitui-se $S_1 \rightarrow a$ por $S_1 \rightarrow a S_2$, de modo a impor que a segunda parte das palavras têm de pertencer a L_2

Concatenação de gramáticas regulares

Algoritmo

- D Sejam $G_1 = (T_1, N_1, P_1, S_1)$ e $G_2 = (T_2, N_2, P_2, S_2)$ duas gramáticas regulares quaisquer, com $N_1 \cap N_2 = \emptyset$. A gramática $G = (T, N, P, S)$ onde

$$T = T_1 \cup T_2$$

$$N = N_1 \cup N_2$$

$$P = \{A \rightarrow \omega S_2 : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^*\} \\ \cup \{A \rightarrow \omega : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^* N_1\} \\ \cup P_2$$

$$S = S_1$$

é regular e gera a linguagem $L = L(G_1) \cdot L(G_2)$.

- As produções da primeira gramática do tipo $\beta \in T^*$ ganham o símbolo inicial da segunda gramática no fim
- As produções da primeira gramática do tipo $\beta \in T^* N$ mantêm-se inalteradas
- As produções da segunda gramática mantêm-se inalteradas

Fecho de gramáticas regulares

Exemplo

Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática regular que represente a linguagem

$$L = L_1^*$$

sabendo que

$$L_1 = \{\omega a : \omega \in T^*\}$$

R

$$\begin{array}{l} S_1 \rightarrow a S_1 \\ \quad | \quad b S_1 \\ \quad | \quad c S_1 \\ \quad | \quad a \end{array}$$

- Começa-se pela obtenção da gramática regular que representa L_1 .

Fecho de gramáticas regulares

Exemplo

Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática regular que represente a linguagem

$$L = L_1^*$$

sabendo que

$$L_1 = \{\omega a : \omega \in T^*\}$$

R

$$\begin{array}{l} S_1 \rightarrow a S_1 \\ \quad | \quad b S_1 \\ \quad | \quad c S_1 \\ \quad | \quad a \end{array}$$

$$\begin{array}{l} S \rightarrow \varepsilon \quad | \quad S_1 \\ S_1 \rightarrow a S_1 \quad | \quad b S_1 \quad | \quad c S_1 \\ \quad \quad \quad | \quad a S \end{array}$$

- Acrescentando-se a transição $S \rightarrow S_1$ e substituindo-se $S_1 \rightarrow a$ por $S_1 \rightarrow a S$, permite-se iterações sobre S_1
- Acrescentando-se $S \rightarrow \varepsilon$, permite-se 0 ou mais iterações

Fecho de gramáticas regulares

Algoritmo

\mathcal{D} Seja $G_1 = (T_1, N_1, P_1, S_1)$ uma gramática regular qualquer. A gramática $G = (T, N, P, S)$ onde

$$T = T_1$$

$$N = N_1 \cup \{S\} \text{ com } S \notin N_1$$

$$P = \{S \rightarrow \varepsilon, S \rightarrow S_1\}$$

$$\cup \{A \rightarrow \omega S : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^*\}$$

$$\cup \{A \rightarrow \omega : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^* N_1\}$$

é regular e gera a linguagem $L = (L(G_1))^*$.

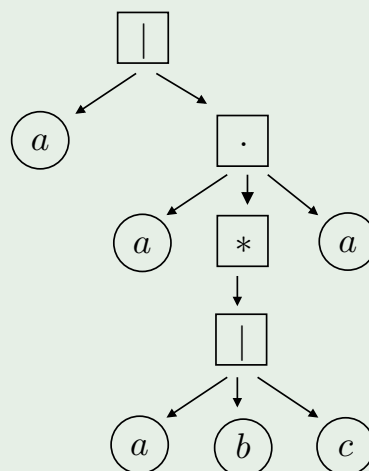
- As novas produções $S \rightarrow \varepsilon$ e $S \rightarrow S_1$ garantem que $(L(G_1))^n \subseteq L(G)$, para qualquer $n \geq 0$
- As produções que só têm terminais ganham o novo símbolo inicial no fim
- As produções que terminam num não terminal mantêm-se inalteradas

Conversão de uma ER em uma GR

exemplo

\mathcal{Q} Construa uma GR equivalente à ER $e = a|a(a|b|c)^*a$.

\mathcal{R}



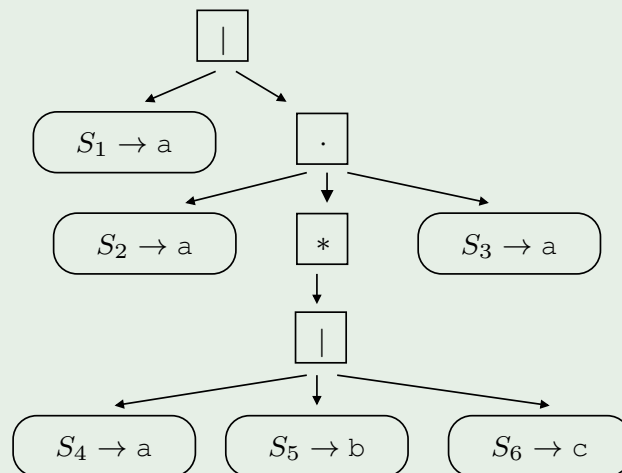
- Coloque-se de forma arbórea

Conversão de uma ER em uma GR

exemplo

Q Construa uma GR equivalente à ER $e = a|a(a|b|c)^*a$.

R



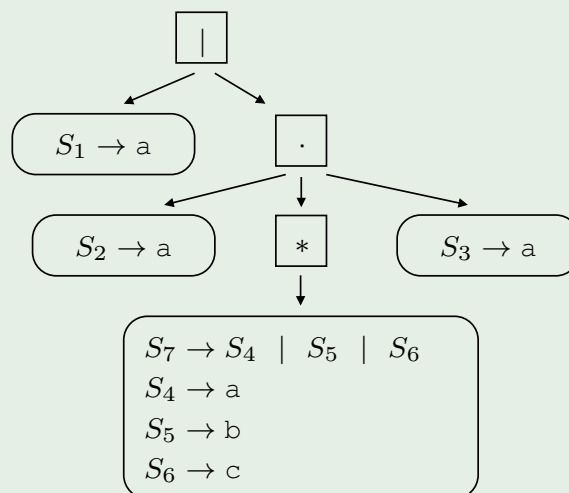
- Após converter as folhas (elementos primitivos) em GR

Conversão de uma ER em uma GR

exemplo

Q Construa uma GR equivalente à ER $e = a|a(a|b|c)^*a$.

R



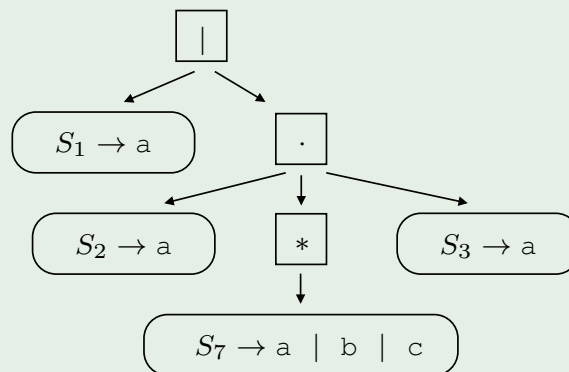
- Após aplicar a escolha (reunião) de baixo

Conversão de uma ER em uma GR

exemplo

Q Construa uma GR equivalente à ER $e = a|a(a|b|c)^*a$.

R



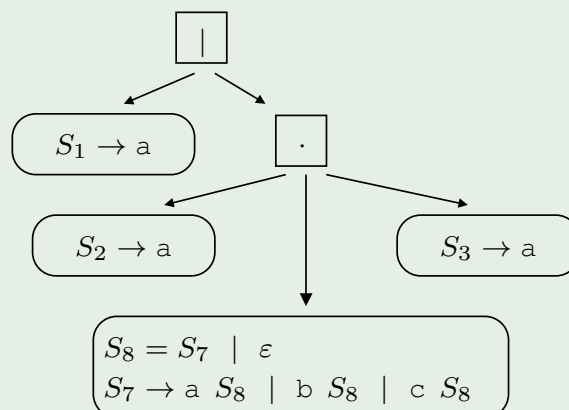
- Simplificando

Conversão de uma ER em uma GR

exemplo

Q Construa uma GR equivalente à ER $e = a|a(a|b|c)^*a$.

R



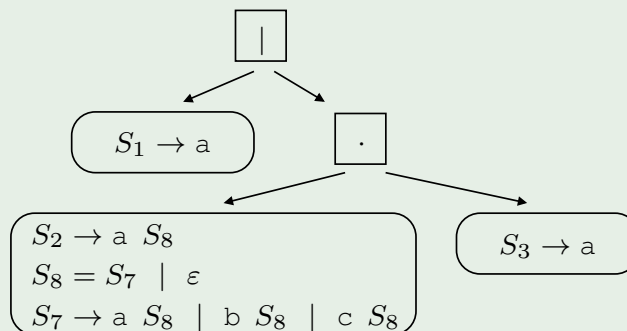
- Após aplicar o fecho

Conversão de uma ER em uma GR

exemplo

Q Construa uma GR equivalente à ER $e = a|a(a|b|c)^*a$.

R



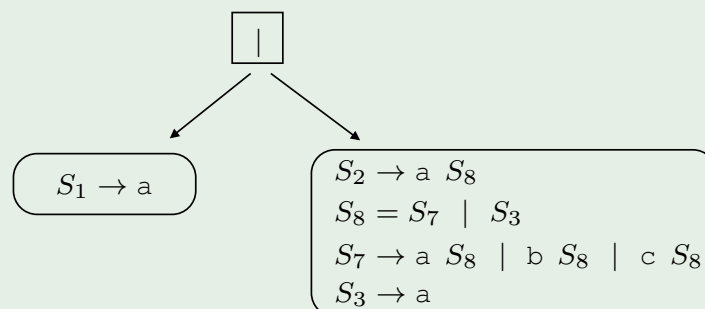
- Após aplicar a concatenação da esquerda

Conversão de uma ER em uma GR

exemplo

Q Construa uma GR equivalente à ER $e = a|a(a|b|c)^*a$.

R



- Após aplicar a concatenação da direita

Conversão de uma ER em uma GR

exemplo

Q Construa uma GR equivalente à ER $e = a|a(a|b|c)^*a$.

R

$$\begin{aligned} S &\rightarrow S_1 \mid S_2 \\ S_1 &\rightarrow a \\ S_2 &\rightarrow a S_8 \\ S_8 &\rightarrow S_7 \mid S_3 \\ S_7 &\rightarrow a S_8 \mid b S_8 \mid c S_8 \\ S_3 &\rightarrow a \end{aligned}$$

e simplificando

$$\begin{aligned} S &\rightarrow a \mid a S_8 \\ S_8 &\rightarrow a S_8 \mid b S_8 \mid c S_8 \mid a \end{aligned}$$

- Finalmente após aplicar escolha (reunião) de cima

Conversão de uma ER em uma GR

Abordagem

- Dada uma expressão regular qualquer ela é:
 - ou um elemento primitivo;
 - ou uma expressão do tipo e^* , sendo e uma expressão regular qualquer;
 - ou uma expressão do tipo $e_1.e_2$, sendo e_1 e e_2 duas expressões regulares quaisquer;
 - ou uma expressão do tipo $e_1|e_2$, sendo e_1 e e_2 duas expressões regulares quaisquer;
- Identificando-se as GR equivalentes às ER primitivas, tem-se o problema resolvido, visto que se sabe como fazer a reunião, a concatenação e o fecho de GR.

expressão regular	gramática regular
ε	$S \rightarrow \varepsilon$
a	$S \rightarrow a$

Conversão de uma ER em uma GR

Algoritmo de conversão

- 1 Se a ER é do tipo primitivo, a GR correspondente pode ser obtido da tabela anterior.
- 2 Se é do tipo e^* , aplica-se este mesmo algoritmo na obtenção de uma GR equivalente à expressão regular e e, de seguida, aplica-se o fecho de GR.
- 3 Se é do tipo $e_1.e_2$, aplica-se este mesmo algoritmo na obtenção de GR para as expressões e_1 e e_2 e, de seguida, aplica-se a concatenação de GR.
- 4 Finalmente, se é do tipo $e_1|e_2$, aplica-se este mesmo algoritmo na obtenção de GR para as expressões e_1 e e_2 e, de seguida, aplica-se a reunião de GR.

- Na realidade, o algoritmo corresponde a um processo de decomposição arbórea a partir da raiz seguido de um processo de construção arbórea a partir das folhas.

Conversão de uma GR em uma ER

Exemplo

Q Obtenha uma ER equivalente à gramática regular seguinte

$$\begin{aligned} S &\rightarrow a S \mid c S \mid aba X \\ X &\rightarrow a X \mid c X \mid \varepsilon \end{aligned}$$

R Abordagem admitindo expressões regulares nas produções das gramáticas

$$\begin{aligned} E &\rightarrow \varepsilon S \\ S &\rightarrow a S \mid c S \mid (aba) X \\ X &\rightarrow a X \mid c X \mid \varepsilon \end{aligned}$$

$$\begin{aligned} E &\rightarrow \varepsilon S \\ S &\rightarrow (a|c) S \mid (aba) X \\ X &\rightarrow (a|c) X \mid \varepsilon \end{aligned}$$

$$\begin{aligned} E &\rightarrow \varepsilon (a|c)^* (aba) X \\ X &\rightarrow (a|c) X \mid \varepsilon \end{aligned}$$

$$E \rightarrow \varepsilon (a|c)^* (aba) (a|c)^* \varepsilon$$

- acrescentou-se um novo símbolo inicial de forma a garantir que não aparece do lado direito
- transformou-se $S \rightarrow a S$ e $S \rightarrow c S$ em $S \rightarrow (a|c) S$
- fez-se algo similar com o X
- transformaram-se as produções $E \rightarrow \varepsilon S$, $S \rightarrow (a|c) S$ e $S \rightarrow aba X$ em $E \rightarrow (a|c)^* aba X$
- Note que o $(a|c)$ passou a $(a|c)^*$
- repetiu-se com o X , obtendo-se a ER desejada: $(a|c)^* aba(a|c)^*$

Conversão de uma GR em uma ER

Exemplo

Q Obtenha uma ER equivalente à gramática regular seguinte

$$S \rightarrow a S \mid c S \mid aba X$$

$$X \rightarrow a X \mid c X \mid \varepsilon$$

R Abordagem transformando a gramática num conjunto e triplos

$$\{(E, \varepsilon, S), \\ (S, a, S), (S, c, S), (S, aba, X), \\ (X, a, X), (X, c, X), (X, \varepsilon, \varepsilon)\}$$

- *converte-se a gramática num conjunto de triplos, acrescentando um inicial*

$$\{(E, \varepsilon, S), (S, (a|c), S), (S, aba, X), \\ (X, (a|c), X), (X, \varepsilon, \varepsilon)\}$$

- *transformou-se $(S, a, S), (S, c, S)$ em $(S, (a|c), S)$*
- *fez-se algo similar com o X*

$$\{(E, (a|c)^* aba, X), \\ (X, (a|c), X), (X, \varepsilon, \varepsilon)\}$$

- *transformou-se o triplo de triplos $(E, \varepsilon, S), (S, (a|c), S), (S, aba, X)$ em $(E, (a|c)^* aba, X)$*
- *Note que o $(a|c)$ passou a $(a|c)^*$*

$$\{(E, (a|c)^* aba(a|c)^*, \varepsilon)\}$$

- *repetiu-se com o X , obtendo-se a ER desejada: $(a|c)^* aba(a|c)^*$*

Conversão de uma GR em uma ER

Algoritmo

- Uma expressão regular e que represente a mesma linguagem que a gramática regular G pode ser obtida por um processo de transformações de equivalência.

- Primeiro, converte-se a gramática $G = (T, N, P, S)$ no conjunto de triplos seguinte:

$$\begin{aligned} \mathcal{E} &= \{(E, \varepsilon, S)\} \\ &\cup \{(A, \omega, B) : (A \rightarrow \omega B) \in P \wedge B \in N\} \\ &\cup \{(A, \omega, \varepsilon) : (A \rightarrow \omega) \in P \wedge \omega \in T^*\} \end{aligned}$$

com $E \notin N$.

- A seguir, removem-se, por transformações de equivalência, um a um, todos os símbolos de N , até se obter um único triplo da forma (E, e, ε) .
- O valor de e é a expressão regular pretendida.

Conversão de uma GR em uma ER

Algoritmo de remoção dos símbolos de N

- 1 Substituir todos os triplos da forma (A, α_i, A) , com $A \in N$, por um único (A, ω_2, A) , onde $\omega_2 = \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_m$
- 2 Substituir todos os triplos da forma (A, β_i, B) , com $A, B \in N$, por um único (A, ω_1, B) , onde $\omega_1 = \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$
- 3 Substituir cada tripla de triplos da forma $(A, \omega_1, B), (B, \omega_2, B), (B, \omega_3, C)$, com $A, B, C \in N$, pelo triplo $(A, \omega_1 \omega_2^* \omega_3, C)$
- 4 Repetir os passos anteriores enquanto houver símbolos intermédios

-
- Note que, se não existir qualquer triplo do tipo (A, α_i, A) , ω_2 representa o conjunto vazio e consequentemente $\omega_2^* = \varepsilon$



Compiladores

Autómatos finitos

Artur Pereira <artur@ua.pt>,
Miguel Oliveira e Silva <mos@ua.pt>

DETI, Universidade de Aveiro

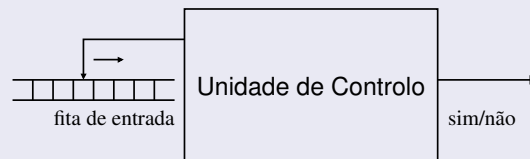
Ano letivo de 2022-2023

Sumário

- ① Autómatos finitos deterministas (AFD)
- ② Redução de autómato finito determinista
- ③ Autómatos finitos não deterministas (AFND)
- ④ Equivalência entre AFD e AFND
- ⑤ Operações sobre autómatos finitos (AF)
- ⑥ Equivalência entre ER e AF
- ⑦ Equivalência entre GR e AF

Autômato finito

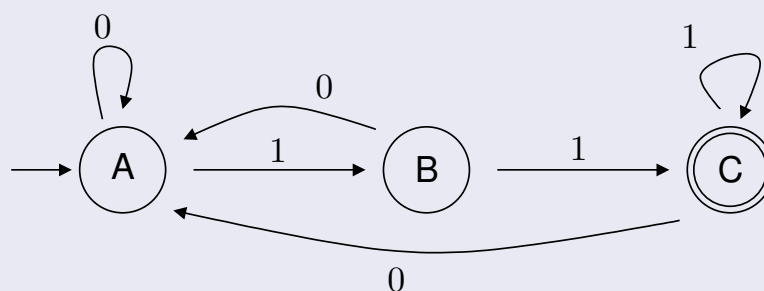
Um **autômato finito** é um mecanismo reconhecedor das palavras de uma linguagem regular



- A unidade de controlo é baseada nas noções de estado e de transição entre estados
 - número finito de estados
- A fita de entrada é só de leitura, com acesso sequencial
- A saída indica se a palavra é ou não aceite (reconhecida)
- Os autômatos finitos podem ser **deterministas**, **não deterministas** ou **generalizados**

Autômato finito determinista

Um **autômato finito determinista** é um autômato finito

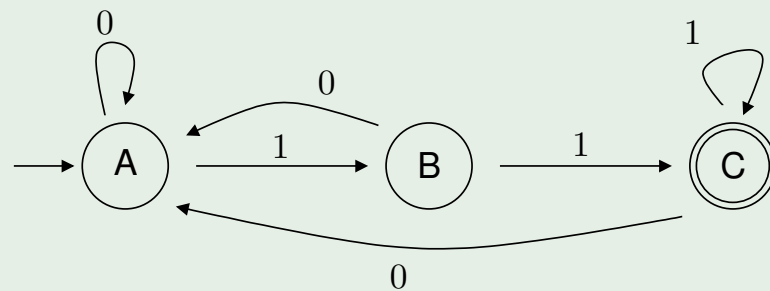


onde

- as transições estão associadas a símbolos individuais do alfabeto;
- de cada estado sai **uma e uma só** transição por cada símbolo do alfabeto;
- há um estado inicial;
- há 0 ou mais estados de aceitação, que determinam as palavras aceites;
- os caminhos que começam no estado inicial e terminam num estado de aceitação representam as palavras aceites (reconhecidas) pelo autômato.

Autômato finito determinista: exemplo (1)

Q Que palavras binárias são reconhecidas pelo autômato seguinte?

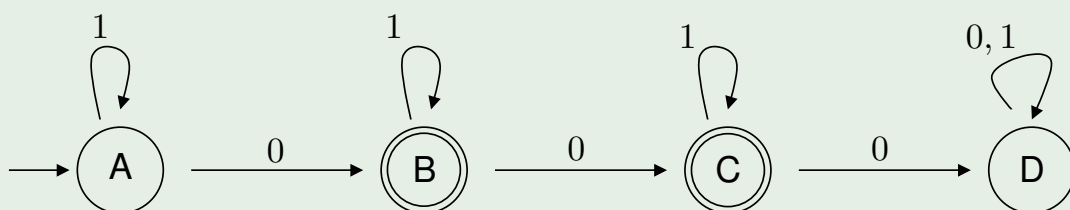


R Todas as palavras terminadas em 11.

E Obtenha uma expressão regular que represente a mesma linguagem.

Autômato finito determinista: exemplo (2)

Q Que palavras binárias são reconhecidas pelo autômato seguinte?

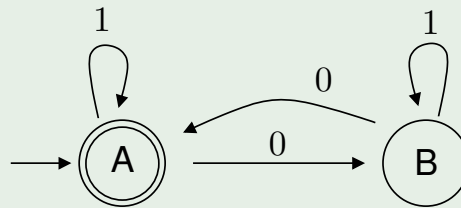


R Todas as palavras com apenas 1 ou 2 zeros.

E Obtenha uma expressão regular que represente a mesma linguagem.

Autômato finito determinista: exemplo (3)

Q Que palavras binárias são reconhecidas pelo autômato seguinte?



R as sequências binárias com um número par de zeros.

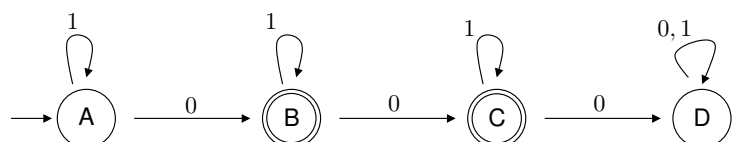
E Obtenha uma expressão regular que represente a mesma linguagem.

Definição de autômato finito determinista

D Um **autômato finito determinista** (AFD) é um quintuplo $M = (A, Q, q_0, \delta, F)$, em que:

- A é o alfabeto de entrada;
- Q é um conjunto finito não vazio de estados;
- $q_0 \in Q$ é o estado inicial;
- $\delta : Q \times A \rightarrow Q$ é uma função que determina a transição entre estados; e
- $F \subseteq Q$ é o conjunto dos estados de aceitação.

- $A = \{0, 1\}$
- $Q = \{A, B, C, D\}$
- $q_0 = A$
- $F = \{B, C\}$
- Como representar δ ?



Definição de autômato finito determinista

D Um **autômato finito determinista** (AFD) é um quintuplo $M = (A, Q, q_0, \delta, F)$, em que:

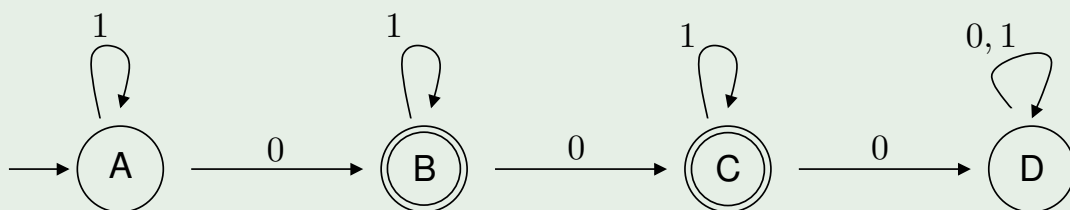
- A é o alfabeto de entrada;
- Q é um conjunto finito não vazio de estados;
- $q_0 \in Q$ é o estado inicial;
- $\delta : Q \times A \rightarrow Q$ é uma função que determina a transição entre estados; e
- $F \subseteq Q$ é o conjunto dos estados de aceitação.

Q Como representar a função δ ?

- Matriz de $|Q|$ linhas por $|A|$ colunas. As células contêm elementos de Q .
- Conjunto de pares $((q, a), q) \in (Q \times A) \times Q$
 - ou equivalentemente conjunto de triplos $(q, a, q) \in Q \times A \times Q$

Autômato finito determinista: exemplo (4)

Q Represente textualmente o AFD seguinte.



R

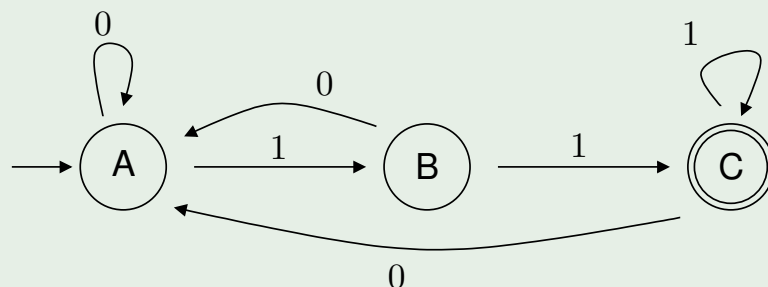
$M = (A, Q, q_0, \delta, F)$ com

- $A = \{0, 1\}$
- $Q = \{A, B, C, D\}$
- $q_0 = A$
- $F = \{B, C\}$
- $\delta = \{$
 - $(A, 0, B), (A, 1, A),$
 - $(B, 0, C), (B, 1, B),$
 - $(C, 0, D), (C, 1, C),$
 - $(D, 0, D), (D, 1, D)\}$

	0	1
A	B	A
B	C	B
C	D	C
D	D	D

Autômato finito determinista: exemplo (5)

Q Represente textualmente o AFD seguinte.



R

$M = (A, Q, q_0, \delta, F)$ com

- $A = \{0, 1\}$

- $Q = \{A, B, C\}$

- $q_0 = A$

- $F = \{C\}$

- $\delta = \{$
 $(A, 0, A), (A, 1, B),$
 $(B, 0, A), (B, 1, C),$
 $(C, 0, A), (C, 1, C),$

- $\delta =$

	0	1
A	A	B
B	A	C
C	A	C

Linguagem reconhecida por um AFD (1)

- Diz-se que um AFD $M = (A, Q, q_0, \delta, F)$, **aceita** uma palavra $u \in A^*$ se u se puder escrever na forma $u = u_1 u_2 \cdots u_n$ e existir uma sequência de estados s_0, s_1, \dots, s_n , que satisfaça as seguintes condições:

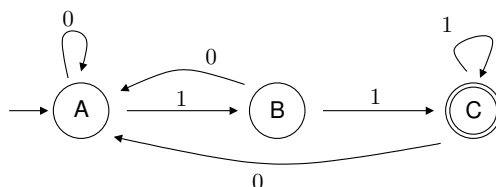
- 1 $s_0 = q_0$;

- 2 qualquer que seja o $i = 1, \dots, n$, $s_i = \delta(s_{i-1}, u_i)$;

- 3 $s_n \in F$.

Caso contrário diz-se que M **rejeita** a sequência de entrada.

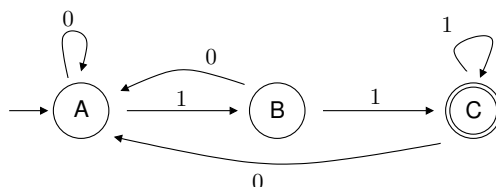
- A palavra $\omega_1 = 0101$ faz o caminho $A \xrightarrow{0} A \xrightarrow{1} B \xrightarrow{0} A \xrightarrow{1} B$
 - como B não é de aceitação, ω_1 não pertence à linguagem
- A palavra $\omega_2 = 0011$ faz o caminho $A \xrightarrow{0} A \xrightarrow{0} A \xrightarrow{1} B \xrightarrow{1} C$
 - como C é de aceitação, ω_2 pertence à linguagem



Linguagem reconhecida por um AFD (2)

- Seja $\delta^* : Q \times A^* \rightarrow Q$ a extensão de δ definida indutivamente por
 - $\delta^*(q, \varepsilon) = q$
 - $\delta^*(q, av) = \delta^*(\delta(q, a), v)$, com $a \in A \wedge v \in A^*$
- M aceita u se $\delta^*(q_0, u) \in F$.
- $L(M) = \{u \in A^* : M \text{ aceita } u\} = \{u \in A^* : \delta^*(q_0, u) \in F\}$

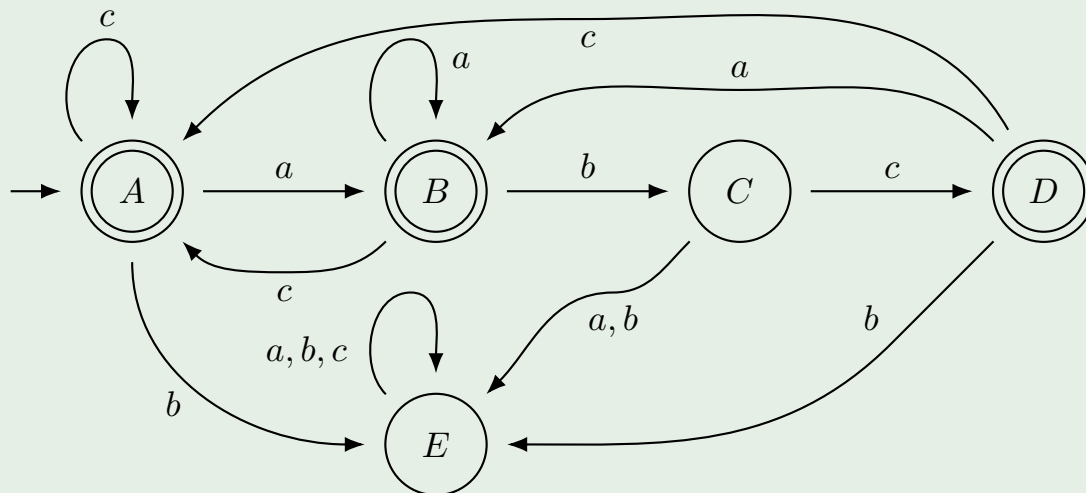
- $\delta^*(A, 0101) = \delta^*(\delta(A, 0), 101) = \delta^*(A, 101)$
 $= \delta^*(\delta(A, 1), 01) = \delta^*(B, 01)$
 $= \delta^*(\delta(B, 0), 1) = \delta^*(A, 1) = \delta^*(B, \varepsilon) = B$
- $\delta^*(A, 0011) = \delta^*(\delta(A, 0), 011) = \delta^*(A, 011)$
 $= \delta^*(\delta(A, 0), 11) = \delta^*(A, 11)$
 $= \delta^*(\delta(A, 1), 1) = \delta^*(B, 1) = \delta^*(C, \varepsilon) = C$



Autômato finito determinista: exemplo (6)

- \mathcal{Q} Sobre o alfabeto $A = \{a, b, c\}$ considere a linguagem
- $$L = \{\omega \in A^* : (\omega_i = b) \Rightarrow ((\omega_{i-1} = a) \wedge (\omega_{i+1} = c))\}$$
- Projecte um autômato que reconheça L .

\mathcal{R}



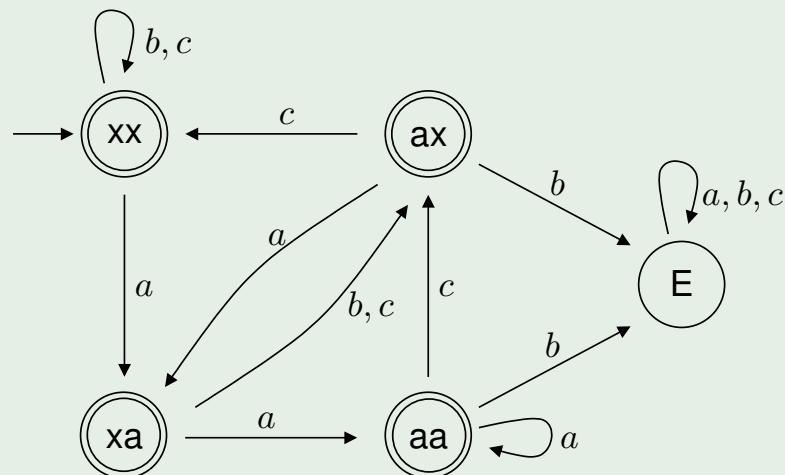
Autómato finito determinista: exemplo (7)

Q Sobre o alfabeto $A = \{a, b, c\}$ considere a linguagem

$$L = \{\omega \in A^* : (\omega_i = a) \Rightarrow (\omega_{i+2} \neq b)\}$$

Projecte um autómato que reconheça L .

R



Autómato finito determinista: exemplo (8)

Q Sobre o alfabeto $A = \{a, b, c\}$ considere a linguagem

$$L = \{\omega \in A^* : (\omega_i = a) \Rightarrow (\omega_{i+2} = b)\}$$

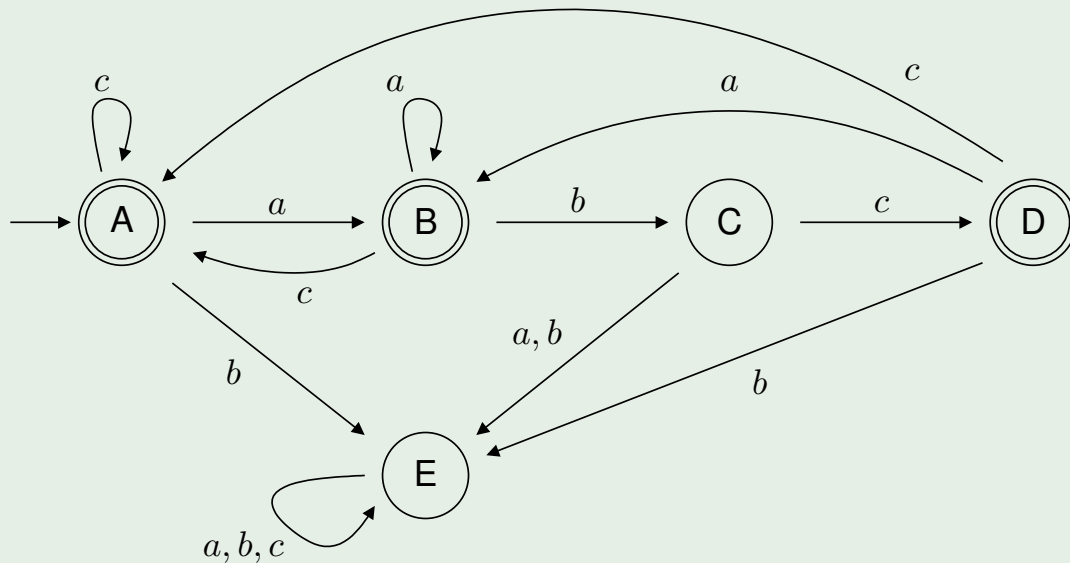
Projecte um autómato que reconheça L .

R

???

Redução de autômato finito determinista (1)

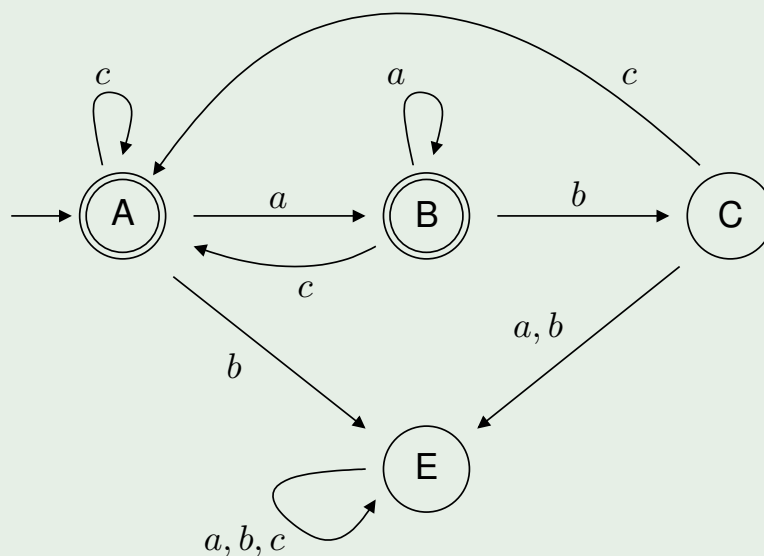
- Q Considere o autômato seguinte (o do exemplo 6) e compare os estados A e D. Que pode concluir ?



- São equivalentes. Por conseguinte, podem ser fundidos

Redução de autômato finito determinista (2)

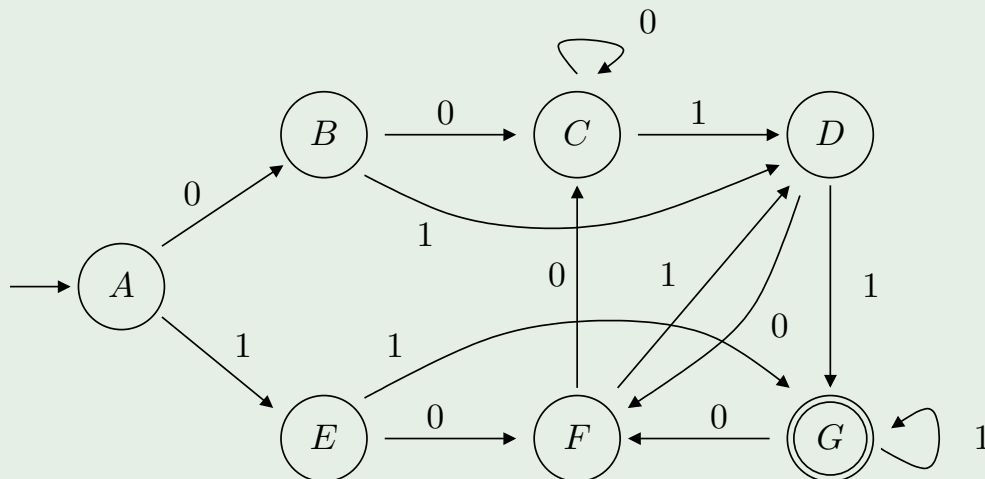
- O que resulta em



- Este, pode provar-se, não tem estados redundantes.
- Está no estado **reduzido**

Algoritmo de Redução de AFD (1)

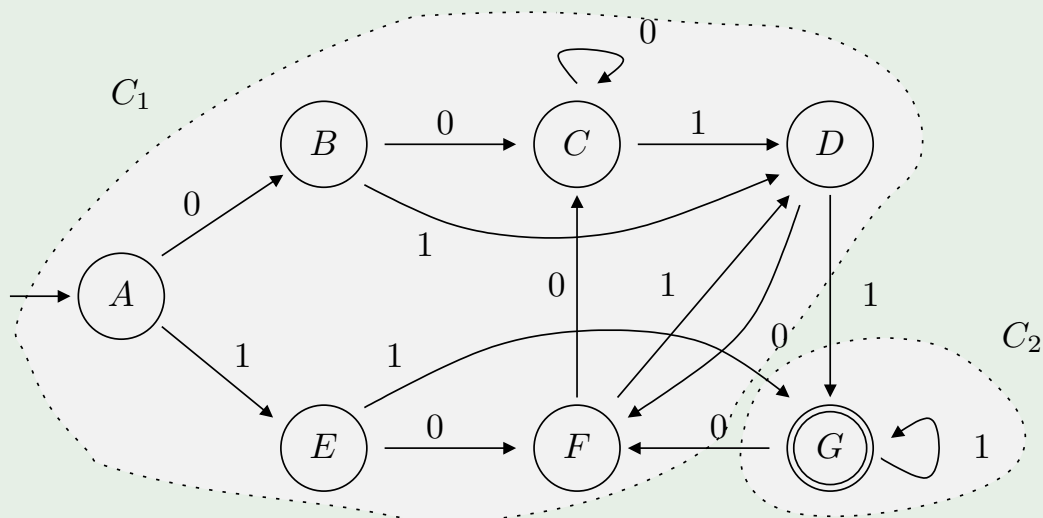
- Como proceder para reduzir um AFD?



- Primeiro, dividem-se os estados em dois conjuntos, um contendo os estados de aceitação e outro os de não-aceitação.

Algoritmo de Redução de AFD (2)

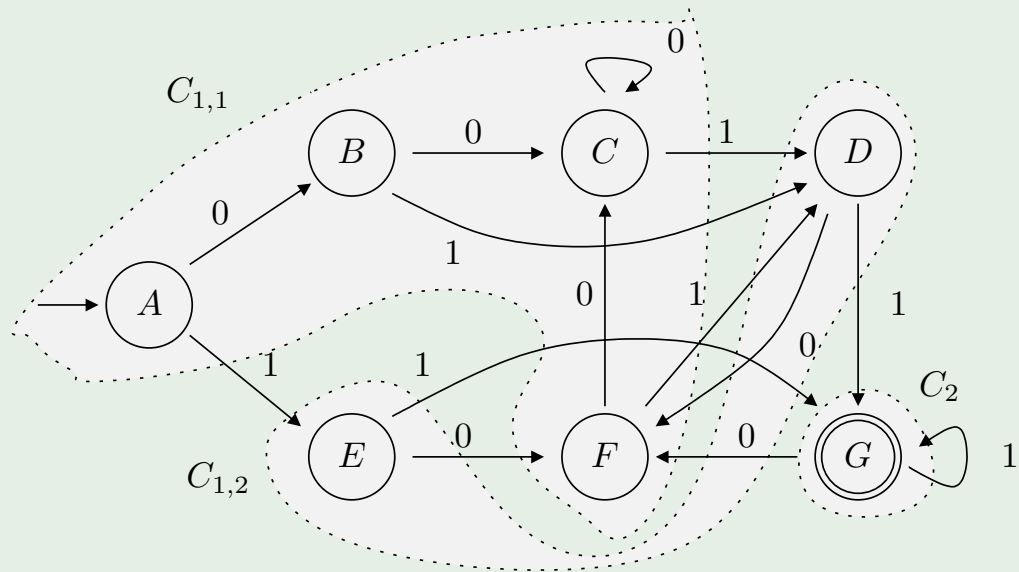
- Obtêm-se $C_1 = \{A, B, C, D, E, F\}$ e $C_2 = \{G\}$.



- Em C_1 , as transições em 0 são todas internas, mas as em 1 podem ser internas ou provocar uma ida para C_2 . Logo, não representa uma classe de equivalência e tem de ser dividido.

Algoritmo de Redução de AFD (3)

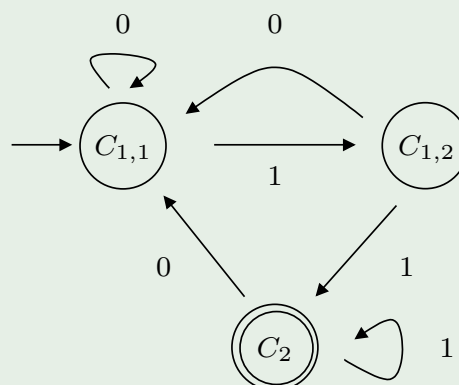
- Dividindo C_1 em $C_{1,1} = \{A, B, C, F\}$ e $C_{1,2} = \{D, E\}$ obtém-se



- Pode verificar-se que $C_{1,1}$, $C_{1,2}$ e C_2 são classes de equivalência, pelo que se chegou à versão reduzida do autômato.

Algoritmo de Redução de AFD (4)

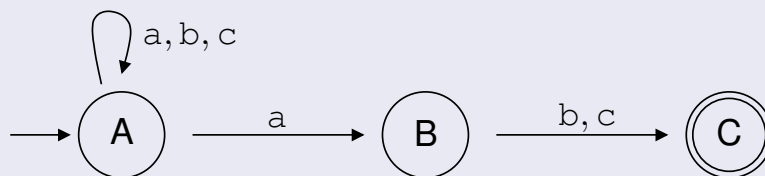
- Autômato reduzido



- Nos apontamentos encontra uma versão não gráfica do algoritmo.

Autômato finito não determinista

Um **autômato finito não determinista** é um autômato finito

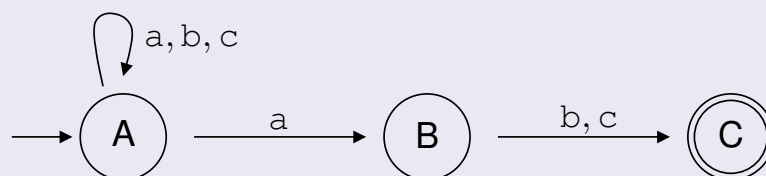


onde

- as transições estão associadas a símbolos individuais do alfabeto **ou à palavra vazia** (ϵ);
 - de cada estado saem **zero ou mais** transições por cada símbolo do **alfabeto ou** ϵ ;
 - há um estado inicial;
 - há 0 ou mais estados de aceitação, que determinam as palavras aceites;
 - os caminhos que começam no estado inicial e terminam num estado de aceitação representam as palavras aceites (reconhecidas) pelo autômato.
-
- As transições múltiplas ou com ϵ permitem alternativas de reconhecimento.
 - As transições ausentes representam quedas num estado de **morte** (estado não representado).

AFND: caminhos alternativos

- Analise o processo de reconhecimento da palavra **abab** ?



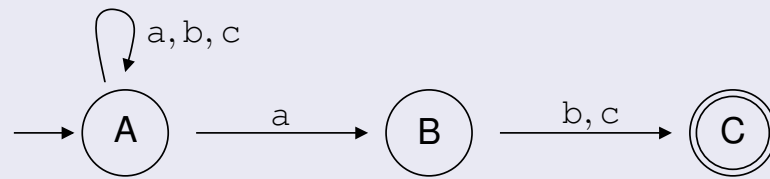
- Há 3 caminhos alternativos

- 1 $A \xrightarrow{a} B \xrightarrow{b} C \xrightarrow{a} \text{X}$
- 2 $A \xrightarrow{a} A \xrightarrow{b} A \xrightarrow{a} A \xrightarrow{b} A$
- 3 $A \xrightarrow{a} A \xrightarrow{b} A \xrightarrow{a} B \xrightarrow{b} C$

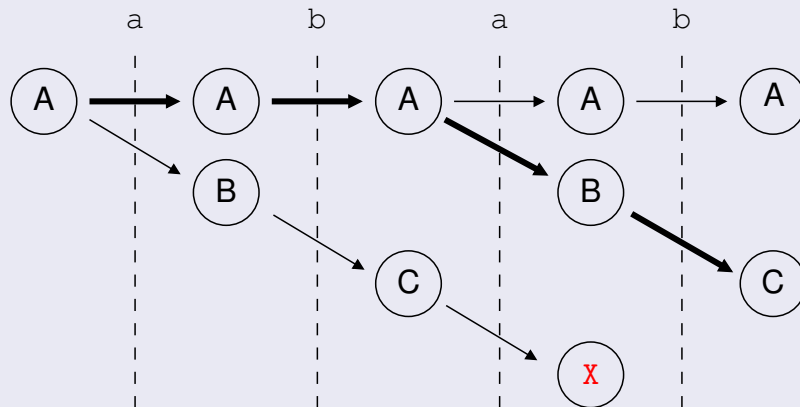
- Como há um caminho que conduz a um estado de aceitação a palavra é reconhecida pelo autômato

AFND: caminhos alternativos

- Analise o processo de reconhecimento da palavra *abab* ?

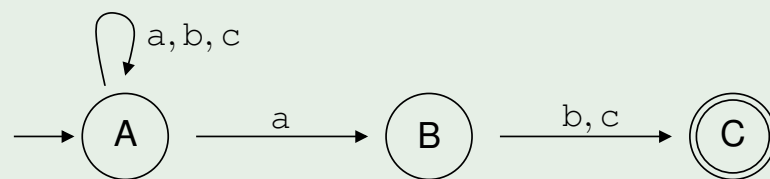


- Que se podem representar de forma arbórea



AFND: exemplo

- Q** Que palavras são reconhecidas pelo autómato seguinte?



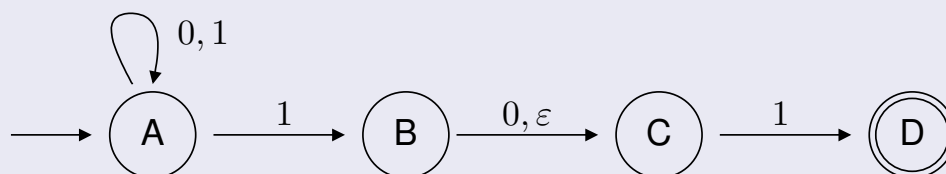
- R** Todas as palavras que terminarem em *ab* ou *ac*

$$L = \{\omega ax : \omega \in A^* \wedge x \in \{b, c\}\}.$$

- Percebe-se uma grande analogia entre este autómato e a expressão regular $(a|b|c)^*a(b|c)$

AFND com transições- ϵ

- Considere o AFND seguinte que contém uma transição- ϵ .



- A palavra 101 é reconhecida pelo autômato através do caminho
 $A \xrightarrow{1} B \xrightarrow{0} C \xrightarrow{1} D$
- A palavra 11 é reconhecida pelo autômato através do caminho
 $A \xrightarrow{1} B \xrightarrow{\epsilon} C \xrightarrow{1} D$
porque $11 = 1\epsilon 1$
- Este autômato reconhece todas as palavras terminadas em 11 ou 101
 $L = \{\omega_1\omega_2 : \omega_1 \in A^* \wedge \omega_2 \in \{11, 101\}\}.$

AFND: definição

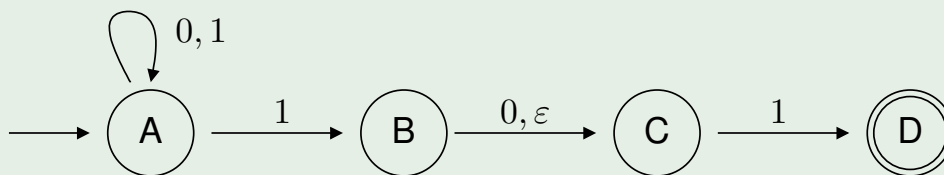
\mathcal{D} Um **autômato finito não determinista** (AFND) é um quintuplo $M = (A, Q, q_0, \delta, F)$, em que:

- A é o alfabeto de entrada;
- Q é um conjunto finito não vazio de estados;
- $q_0 \in Q$ é o estado inicial;
- $\delta \subseteq (Q \times A_\epsilon \times Q)$ é a relação de transição entre estados, com $A_\epsilon = A \cup \{\epsilon\}$;
- $F \subseteq Q$ é o conjunto dos estados de aceitação.

-
- Apenas a definição de δ difere em relação aos AFD.
 - Se se representar δ na forma de uma tabela, as células são preenchidas com elementos de $\wp(Q)$, ou seja, sub-conjuntos de Q .

AFND: Exemplo (2)

Q Represente textualmente o AFND



R $M = (A, Q, q_0, \delta, F)$ com

• $A = \{0, 1\}$

• $Q = \{A, B, C, D\}$

• $q_0 = A$

• $F = \{D\}$

• $\delta = \{$

$(A, 0, A), (A, 1, A),$

$(A, 1, B), (B, 0, C),$

$(B, \varepsilon, C), (C, 1, D)$

$\}$

• $\delta =$

	0	1	ε
A	$\{A\}$	$\{A, B\}$	$\{\}$
B	$\{C\}$	$\{\}$	$\{C\}$
C	$\{\}$	$\{D\}$	$\{\}$
D	$\{\}$	$\{\}$	$\{\}$

- O par $(A, 1, A), (A, 1, B)$ faz com que δ não seja uma função

AFND: linguagem reconhecida

- Diz-se que um AFND $M = (A, Q, q_0, \delta, F)$, **aceita** uma palavra $u \in A^*$ se u se puder escrever na forma $u = u_1 u_2 \cdots u_n$, com $u_i \in A_\varepsilon$, e existir uma sequência de estados s_0, s_1, \cdots, s_n , que satisfaça as seguintes condições:

① $s_0 = q_0$;

② qualquer que seja o $i = 1, \cdots, n$, $(s_{i-1}, u_i, s_i) \in \delta$;

③ $s_n \in F$.

- Caso contrário diz-se que M **rejeita** a entrada.

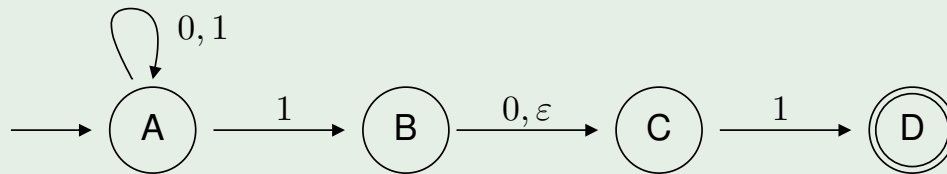
- Note que n pode ser maior que $|u|$, porque alguns dos u_i podem ser ε .

- Usar-se-á a notação $q_i \xrightarrow{u} q_j$ para indicar que a palavra u permite ir do estado q_i ao estado q_j .

- Usando esta notação tem-se $L(M) = \{u : q_0 \xrightarrow{u} q_f \wedge q_f \in F\}$.

AFND: Exemplo de aplicação

Q Sobre o alfabeto $A = \{0, 1\}$, considere o AFND M seguinte



e a linguagem $L = \{\omega \in A^* : \omega = (01)^n, n > 1\}$. Mostre que $L \subset L(M)$.

R

Equivalência entre AFD e AFND

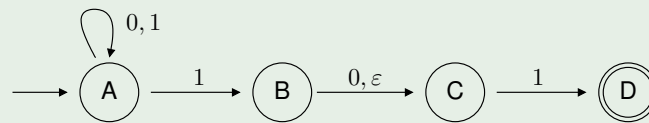
- A classe das linguagens cobertas por um AFD é a mesma que a classe das linguagens cobertas por um AFND
- Isto significa que:
 - Se M é um AFD, então $\exists_{M' \in \text{AFND}} : L(M') = L(M)$.
 - Se M é um AFND, então $\exists_{M' \in \text{AFD}} : L(M') = L(M)$.

- Como determinar um AFND equivalente a um AFD dado ?
- Pelas definições de AFD e AFND, um AFD é um AFND. Porquê?
 - Q , q_0 e F têm a mesma definição.
 - Nos AFD $\delta : Q \times A \rightarrow Q$.
 - Nos AFND $\delta \subset Q \times A_\epsilon \times Q$
 - Mas, se $\delta : Q \times A \rightarrow Q$ então $\delta \subseteq Q \times A \times Q \subset Q \times A_\epsilon \times Q$
 - Logo, um AFD é um AFND

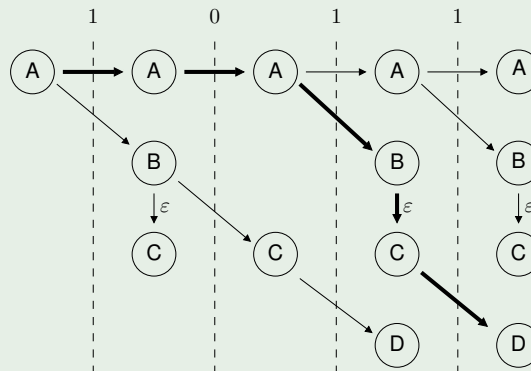
Equivalente AFD de um AFND (1)

- Como determinar um AFD equivalente a um AFND dado ?

- No AFND



a árvore de reconhecimento da palavra 1011 sugere que a evolução se faz de sub-conjunto em sub-conjunto de estados



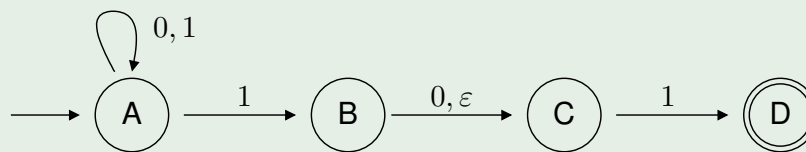
Equivalente AFD de um AFND (2)

- Dado um AFND $M = (A, Q, q_0, \delta, F)$, considere o AFD $M' = (A, Q', q'_0, \delta', F')$ onde:
 - $Q' = \wp(Q)$
 - $q'_0 = \varepsilon\text{-closure}(q_0)$
 - $F' = \{f' \in \wp(Q) : f' \cap F \neq \emptyset\}$
 - $\delta' = \wp(Q) \times A \rightarrow \wp(Q)$,
com $\delta'(q', a) = \bigcup_{q \in q'} \{s : s \in \varepsilon\text{-closure}(s') \wedge (q, a, s') \in \delta\}$
- M e M' reconhecem a mesma linguagem.

- $\varepsilon\text{-closure}(q)$ é o conjunto de estados constituído por q mais todos os direta ou indiretamente alcançáveis a partir de q apenas por transições- ε
- Note que:
 - O estado inicial (q'_0) pode conter 1 ou mais elementos de Q
 - Cada elemento do conjunto de chegada ($f' \in F'$) por conter elementos de F e $Q - F$

Equivalente AFD de um AFND: exemplo

Q Determinar um AFD equivalente ao AFND seguinte ?



R

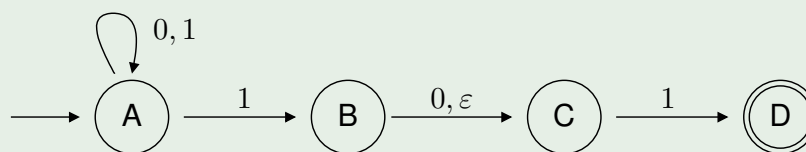
- $Q' = \{X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}\}$, com

$X_0 = \{\}$	$X_1 = \{A\}$	$X_2 = \{B\}$	$X_3 = \{A, B\}$
$X_4 = \{C\}$	$X_5 = \{A, C\}$	$X_6 = \{B, C\}$	$X_7 = \{A, B, C\}$
$X_8 = \{D\}$	$X_9 = \{A, D\}$	$X_{10} = \{B, D\}$	$X_{11} = \{A, B, D\}$
$X_{12} = \{C, D\}$	$X_{13} = \{A, C, D\}$	$X_{14} = \{B, C, D\}$	$X_{15} = \{A, B, C, D\}$

- $q'_0 = \varepsilon\text{-closure}(A) = \{A\} = X_1$
- $F' = \{X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}\}$

Equivalente AFD de um AFND: exemplo

Q Determinar um AFD equivalente ao AFND seguinte ?



R

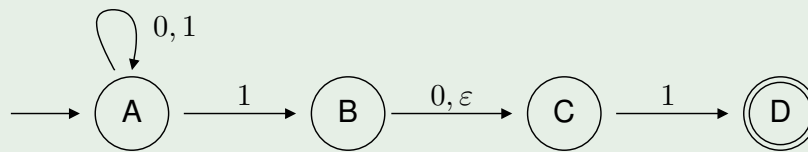
- $\delta' =$

estado	0	1		estado	0	1
$X_0 = \{\}$	X_0	X_0		$X_1 = \{A\}$	X_1	X_7
$X_2 = \{B\}$	X_4	X_0		$X_3 = \{A, B\}$	X_5	X_7
$X_4 = \{C\}$	X_0	X_8		$X_5 = \{A, C\}$	X_1	X_{15}
$X_6 = \{B, C\}$	X_4	X_8		$X_7 = \{A, B, C\}$	X_5	X_{15}
$X_8 = \{D\}$	X_0	X_0		$X_9 = \{A, D\}$	X_1	X_7
$X_{10} = \{B, D\}$	X_4	X_0		$X_{11} = \{A, B, D\}$	X_5	X_7
$X_{12} = \{C, D\}$	X_0	X_8		$X_{13} = \{A, C, D\}$	X_1	X_{15}
$X_{14} = \{B, C, D\}$	X_4	X_8		$X_{15} = \{A, B, C, D\}$	X_5	X_{15}

- Serão todos estes estados necessários?

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?

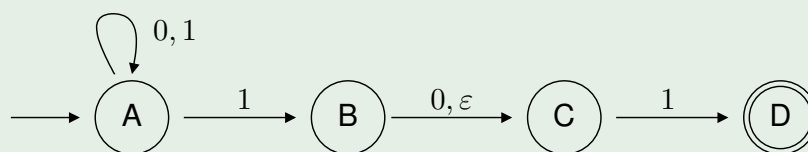


\mathcal{R}

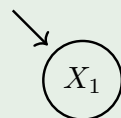
- Conseguir-se o mesmo resultado através de um processo construtivo.

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



\mathcal{R}

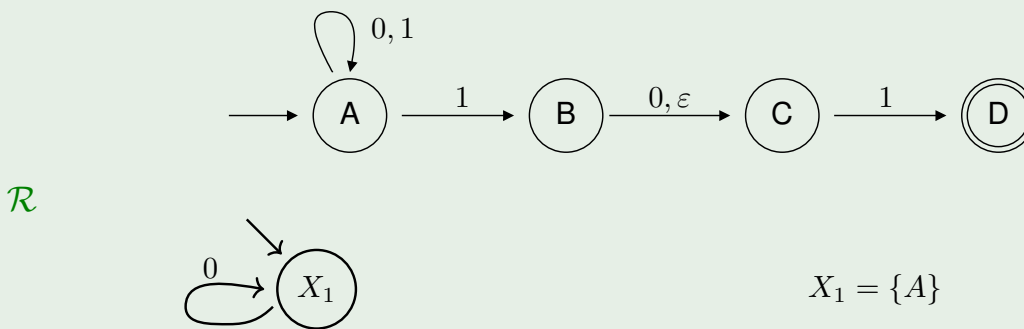


$$X_1 = \{A\}$$

- Comece-se com o estado inicial ($X_1 = \{A\}$)

Equivalente AFD de um AFND: exemplo (2)

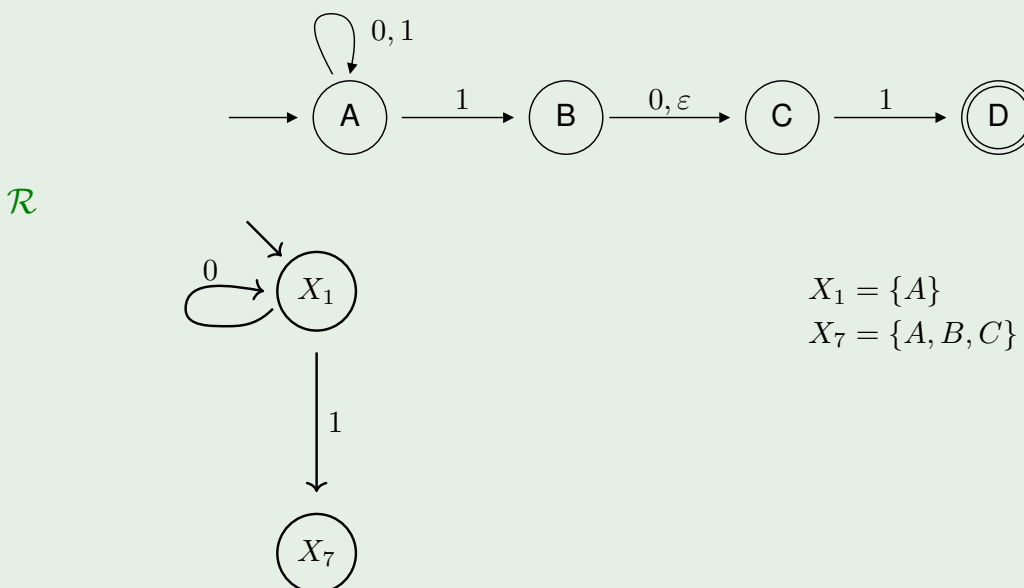
Q Determinar um AFD equivalente ao AFND seguinte ?



- $\delta'(X_1, 0) = \varepsilon\text{-closure}(A) = \{A\}$

Equivalente AFD de um AFND: exemplo (2)

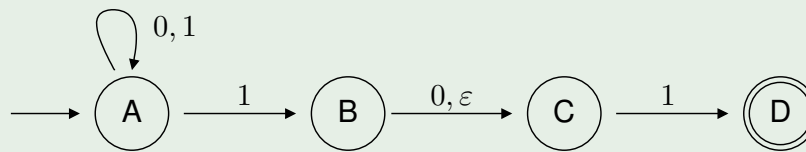
Q Determinar um AFD equivalente ao AFND seguinte ?



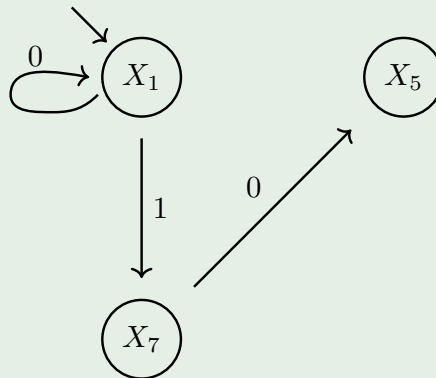
- $\delta'(X_1, 1) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(B) = \{A\} \cup \{B, C\} = \{A, B, C\}$

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



\mathcal{R}

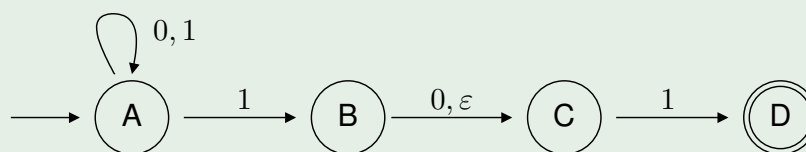


$X_1 = \{A\}$
 $X_7 = \{A, B, C\}$
 $X_5 = \{A, C\}$

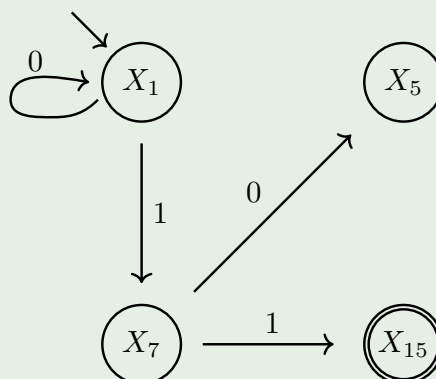
- $\delta'(X_7, 0) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(C) = \{A\} \cup \{C\} = \{A, C\}$

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



\mathcal{R}

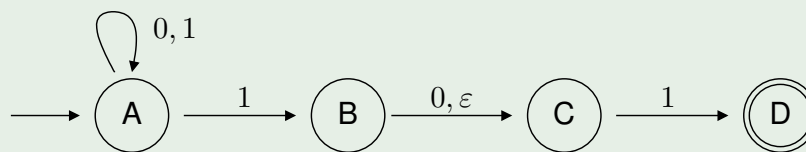


$X_1 = \{A\}$
 $X_7 = \{A, B, C\}$
 $X_5 = \{A, C\}$
 $X_{15} = \{A, B, C, D\}$

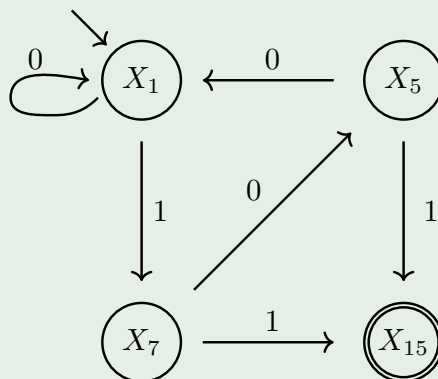
- $\delta'(X_7, 1) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(B) \cup \varepsilon\text{-closure}(D) = \{A\} \cup \{B, C\} \cup \{D\} = \{A, B, C, D\}$
- É de aceitação porque $\{A, B, C, D\} \cap \{D\} \neq \emptyset$

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



\mathcal{R}

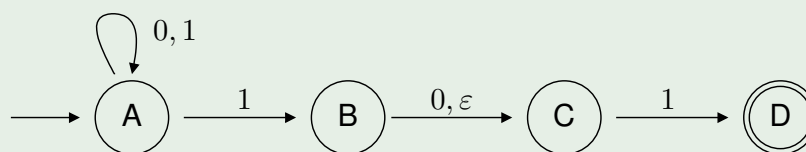


$X_1 = \{A\}$
 $X_7 = \{A, B, C\}$
 $X_5 = \{A, C\}$
 $X_{15} = \{A, B, C, D\}$

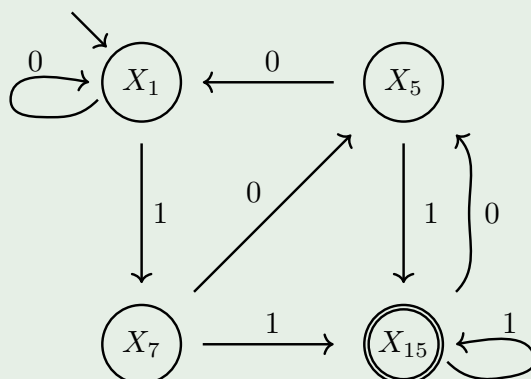
- $\delta'(X_5, 0) = \varepsilon\text{-closure}(A) = \{A\}$
- $\delta'(X_5, 1) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(B) \cup \varepsilon\text{-closure}(D) = \{A\} \cup \{B, C\} \cup \{D\} = \{A, B, C, D\}$

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



\mathcal{R}



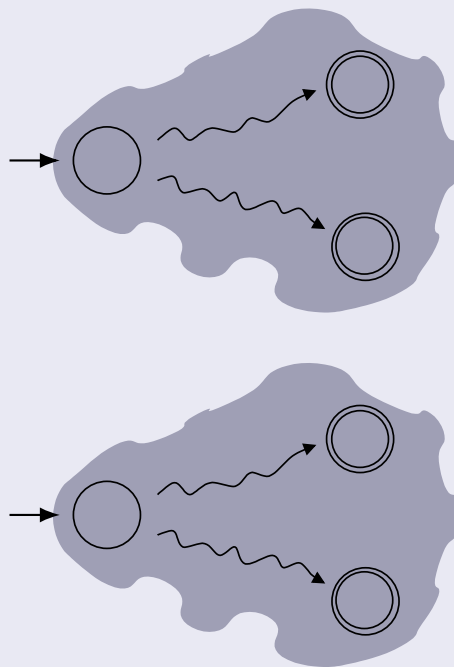
$X_1 = \{A\}$
 $X_7 = \{A, B, C\}$
 $X_5 = \{A, C\}$
 $X_{15} = \{A, B, C, D\}$

- $\delta'(X_{15}, 0) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(C) = \{A\} \cup \{C\} = \{A, C\}$
- $\delta'(X_{15}, 1) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(B) \cup \varepsilon\text{-closure}(D) = \{A\} \cup \{B, C\} \cup \{D\} = \{A, B, C, D\}$

Operações sobre AFD e AFND

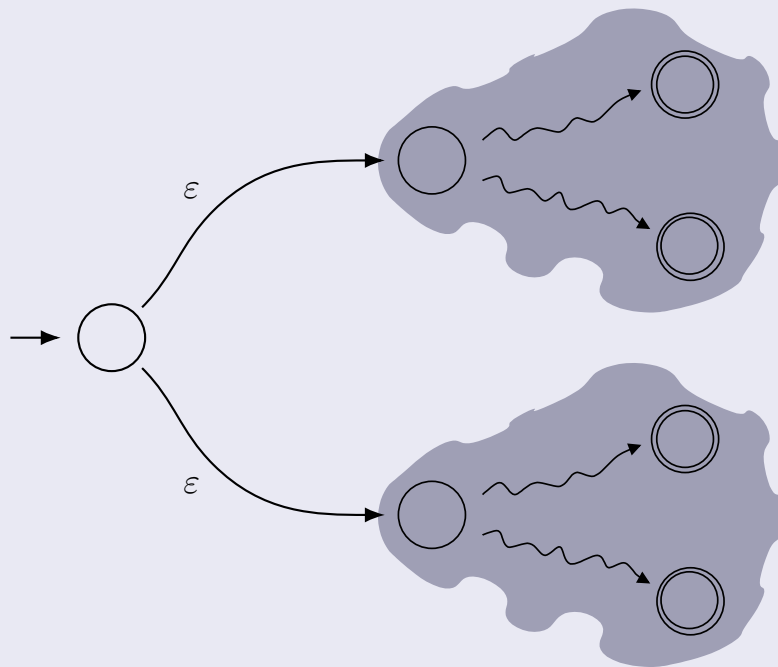
- Os automáto finitos (AF) são fechados sobre as operações de:
 - Reunião
 - Concatenação
 - Fecho
 - Intersecção
 - Complementação

Reunião de AF



- Como criar um AF que represente a reunião destes dois AF?

Reunião de AF



- acrescenta-se um novo estado que passa a ser o inicial
- e acrescentam-se transições- ε deste novo estado para os estados iniciais originais

Reunião de AF: definição

D Seja $M_1 = (A, Q_1, q_1, \delta_1, F_1)$ e $M_2 = (A, Q_2, q_2, \delta_2, F_2)$ dois autómatos (AFD ou AFND) quaisquer.

O AFND $M = (A, Q, q_0, \delta, F)$, onde

$$Q = Q_1 \cup Q_2 \cup \{q_0\}, \quad \text{com } q_0 \notin Q_1 \wedge q_0 \notin Q_2$$

$$F = F_1 \cup F_2$$

$$\delta = \delta_1 \cup \delta_2 \cup \{(q_0, \varepsilon, q_1), (q_0, \varepsilon, q_2)\}$$

implementa a reunião de M_1 e M_2 , ou seja, $L(M) = L(M_1) \cup L(M_2)$.

Reunião de AF: exemplo (1)

Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AF que reconheça $L = L_1 \cup L_2$.

R

- Como criar um AF que represente a reunião de L_1 e L_2 ?

Reunião de AF: exemplo (1)

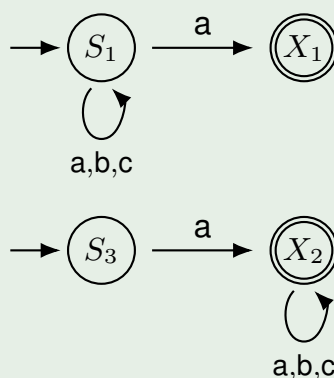
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AF que reconheça $L = L_1 \cup L_2$.

R



- Constroi-se um AF para a linguagem L_1
- Constroi-se um AF para a linguagem L_2

Reunião de AF: exemplo (1)

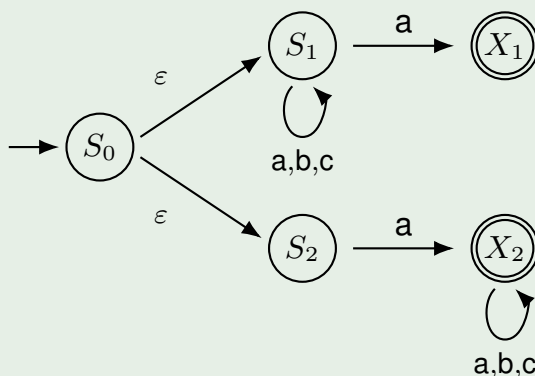
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AF que reconheça $L = L_1 \cup L_2$.

R



- Acrescenta-se um novo estado (S_0), que passa a ser o inicial
- E acrescentam-se transições- ϵ de S_0 (novo estado inicial) para S_1 e S_2 (os estados iniciais originais)

Reunião de AF: exemplo (1)

Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AF que reconheça $L = L_1 \cup L_2$.

R

$M_1 = (A, Q_1, q_1, \delta_1, F_1)$ com

$$Q_1 = \{S_1, X_1\}, \quad q_1 = S_1, \quad F_1 = \{X_1\}$$

$$\delta_1 = \{(S_1, a, S_1), (S_1, b, S_1), (S_1, c, S_1), (S_1, a, X_1)\}$$

$M_2 = (A, Q_2, q_2, \delta_2, F_2)$ com

$$Q_2 = \{S_2, X_2\}, \quad q_2 = S_2, \quad F_2 = \{X_2\}$$

$$\delta_2 = \{(S_2, a, X_2), (X_2, a, X_2), (X_2, b, X_2), (X_2, c, X_2)\}$$

$M = M_1 \cup M_2 = (A, Q, q_0, \delta, F)$ com

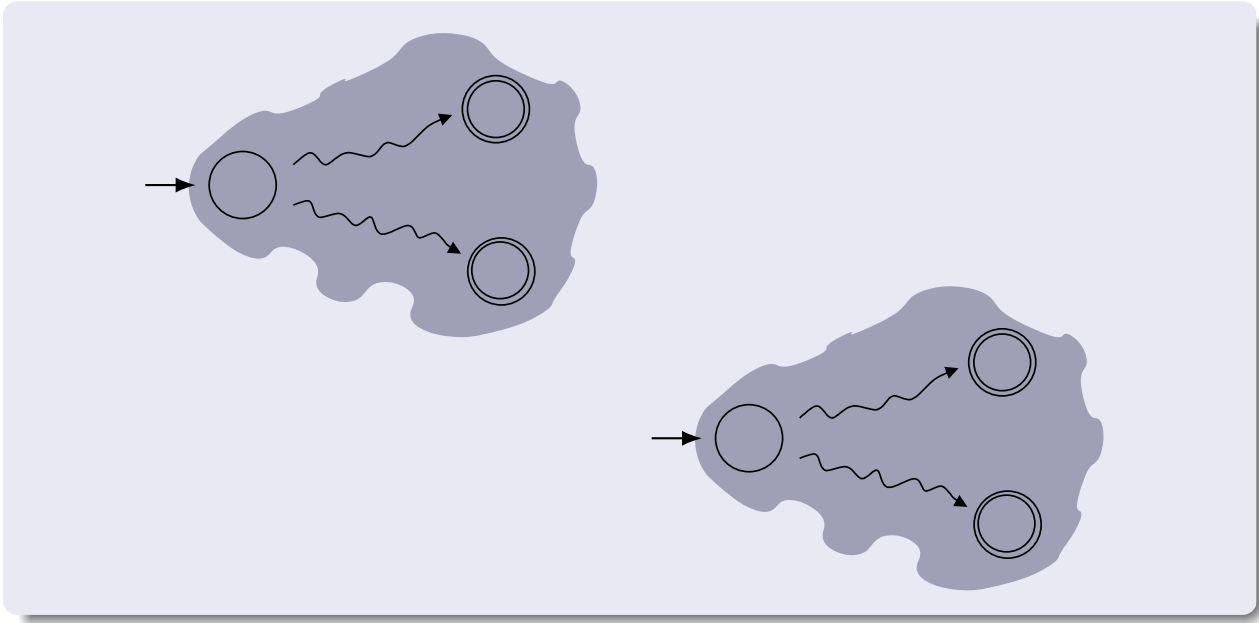
$$Q = \{S_0, S_1, X_1, S_2, X_2\}, \quad q_0 = S_0, \quad F = \{X_1, X_2\},$$

$$\delta = \{(S_0, \epsilon, S_1), (S_0, \epsilon, S_2), (S_1, a, S_1), (S_1, b, S_1), (S_1, c, S_1),$$

$$(S_1, a, X_1), (S_2, a, X_2), (X_2, a, X_2), (X_2, b, X_2), (X_2, c, X_2)\}$$

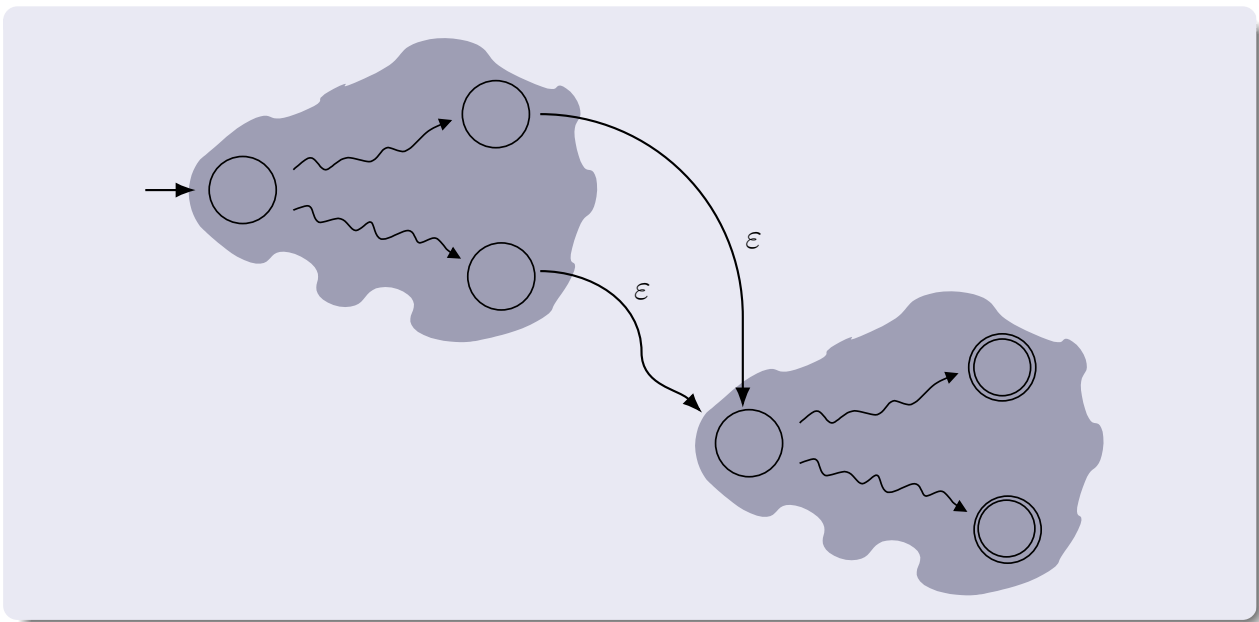
- Alternativamente, pode ser escrito de forma textual

Concatenação de AF



- Como criar um AF que represente a concatenação destes dois AF?

Concatenação de AF



- O estado inicial passa a ser o estado inicial do AF da esquerda
- Os estados de aceitação são apenas os estados de aceitação do AF da direita
- acrescentam-se transições- ϵ dos (antigos) estados de aceitação do AF da esquerda para o estado inicial do AF da direita

Concatenação de AF: definição

D Seja $M_1 = (A, Q_1, q_1, \delta_1, F_1)$ e $M_2 = (A, Q_2, q_2, \delta_2, F_2)$ dois autómatos (AFD ou AFND) quaisquer.

O AFND $M = (A, Q, q_0, \delta, F)$, onde

$$Q = Q_1 \cup Q_2$$

$$q_0 = q_1$$

$$F = F_2$$

$$\delta = \delta_1 \cup \delta_2 \cup (F_1 \times \{\varepsilon\} \times \{q_2\})$$

implementa a concatenação de M_1 e M_2 , ou seja,
 $L(M) = L(M_1) \cdot L(M_2)$.

Concatenação de AF: exemplo

Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AF que reconheça $L = L_1 \cdot L_2$.

R

-
- Como criar um AF que represente a concatenação de L_1 com L_2 ?

Concatenação de AF: exemplo

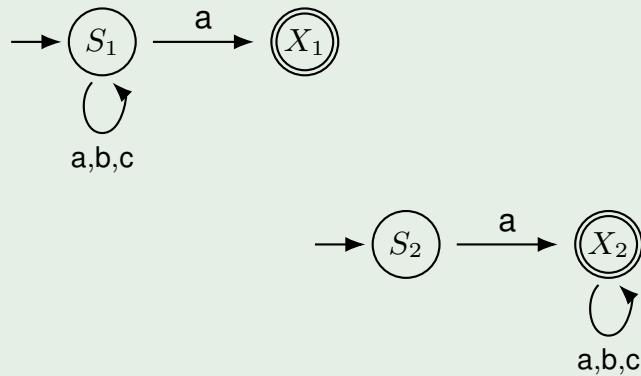
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AF que reconheça $L = L_1 \cdot L_2$.

R



- Constroi-se AF para as linguagens L_1 e L_2

Concatenação de AF: exemplo

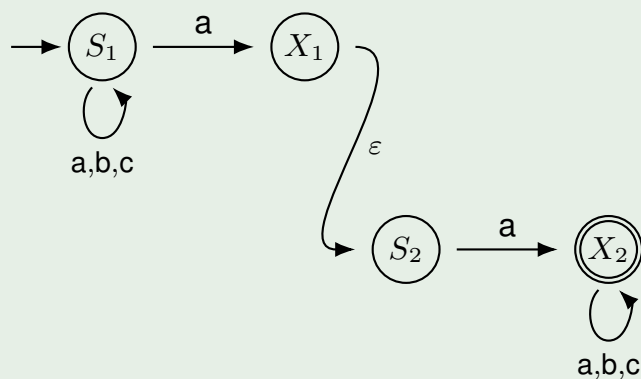
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

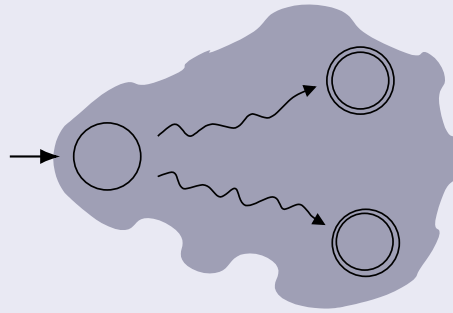
Determine um AF que reconheça $L = L_1 \cdot L_2$.

R



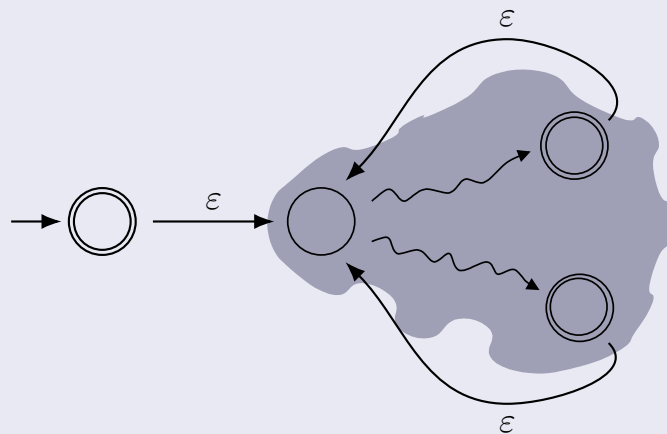
- X_1 deixa de ser de aceitação; S_2 deixa de ser de entrada
- acrescenta-se uma transição- ϵ de X_1 para S_2

Fecho de AF



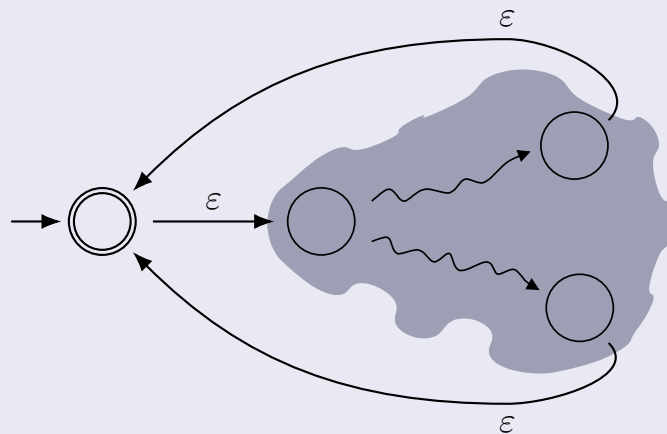
- Como criar um AF que represente o fecho deste AF?

Fecho de AF



- acrescenta-se um novo estado que passa a ser o inicial
- o novo estado inicial é de aceitação
- acrescentam-se transições- ϵ dos estados de aceitação do AF para o estado inicial original

Fecho de AF



- acrescenta-se um novo estado que passa a ser o inicial
 - o novo estado inicial é de aceitação
 - ou acrescentam-se transições- ϵ dos estados de aceitação do AF para o novo estado inicial (caso em que antigos estados de aceitação podem deixar de o ser)
- ◇ Note que em geral não se pode fundir o novo estado inicial com o antigo

Fecho de AF: definição

D Seja $M_1 = (A, Q_1, q_1, \delta_1, F_1)$ um autómato (AFD ou AFND) qualquer. O AFND $M = (A, Q, q_0, \delta, F)$, onde

$$Q = Q_1 \cup \{q_0\}$$

$$F = \{q_0\}$$

$$\delta = \delta_1 \cup (F_1 \times \{\epsilon\} \times \{q_0\}) \cup \{(q_0, \epsilon, q_1)\}$$

implementa o fecho de M_1 , ou seja, $L(M) = L(M_1)^*$.

- Em alternativa poder-se-á considerar que $F = F_1 \cup \{q_0\}$ e que de F_1 as novas transições- ϵ se dirigem a q_1

Fecho de AF: exemplo

Q Sobre o alfabeto $A = \{a, b, c\}$, seja

$$L_1 = \{a\omega \mid \omega \in A^*\}$$

Determine o AFND que reconhece a linguagem L_1^* .

R

- Como criar um AF que represente o fecho de L_1 ?

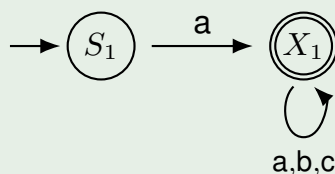
Fecho de AF: exemplo

Q Sobre o alfabeto $A = \{a, b, c\}$, seja

$$L_1 = \{a\omega \mid \omega \in A^*\}$$

Determine o AFND que reconhece a linguagem L_1^* .

R



- Constroi-se um AF para L_1

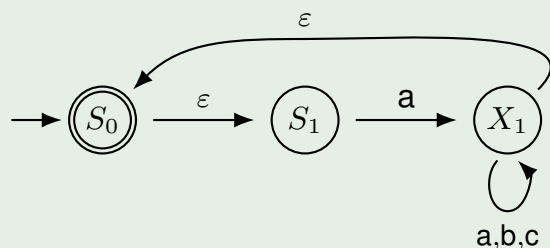
Fecho de AF: exemplo

Q Sobre o alfabeto $A = \{a, b, c\}$, seja

$$L_1 = \{a\omega \mid \omega \in A^*\}$$

Determine o AFND que reconhece a linguagem L_1^* .

R



- acrescenta-se um novo estado (S_0), que passa a ser o inicial e é de aceitação
- liga-se este estado ao S_1 (inicial anterior) por uma transição- ϵ
- liga-se o estado X_1 (aceitação anterior) ao S_0 (novo inicial)
- X_1 deixa (pode deixar) de ser de aceitação

Intersecção de AF: exemplo

Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AFD ou AFND que reconheça $L = L_1 \cap L_2$.

R

- Como criar um AF que represente a intersecção de L_1 e L_2 ?

Intersecção de AF: exemplo

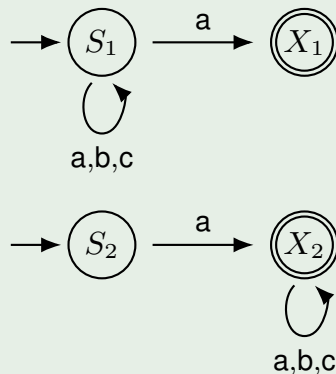
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AFD ou AFND que reconheça $L = L_1 \cap L_2$.

R



- Constroi-se AF para as linguagens L_1 e L_2

Intersecção de AF: exemplo

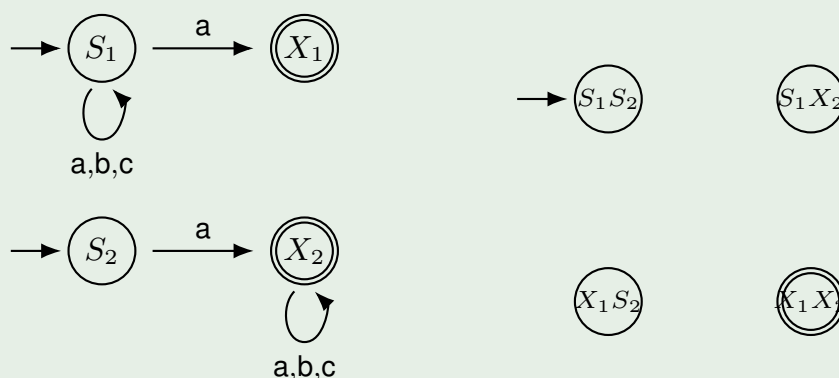
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AFD ou AFND que reconheça $L = L_1 \cap L_2$.

R



- Definem-se os estados que resultam do produto cartesiano $\{S_1, X_1\} \times \{S_2, X_2\}$
- Mas, alguns podem não ser alcançáveis

Intersecção de AF: exemplo

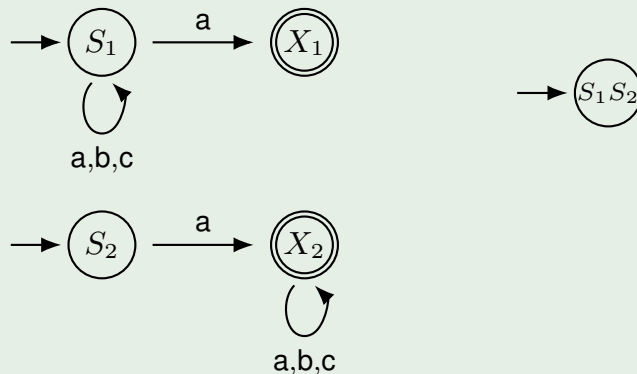
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AFD ou AFND que reconheça $L = L_1 \cap L_2$.

R



- Pelo que começemos apenas pelo S_1S_2 , que corresponde ao estado inicial

Intersecção de AF: exemplo

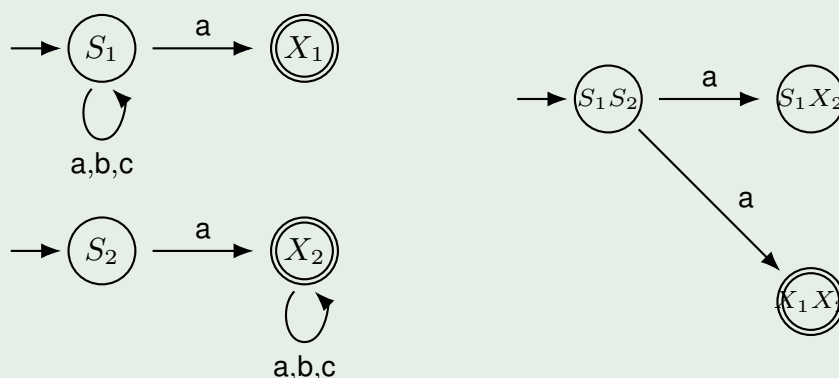
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AFD ou AFND que reconheça $L = L_1 \cap L_2$.

R



- de $S_1 \xrightarrow{a} S_1$ e $S_2 \xrightarrow{a} X_2$ aparece $S_1S_2 \xrightarrow{a} S_1X_2$
- de $S_1 \xrightarrow{a} X_1$ e $S_2 \xrightarrow{a} X_2$ aparece $S_1S_2 \xrightarrow{a} X_1X_2$

Intersecção de AF: exemplo

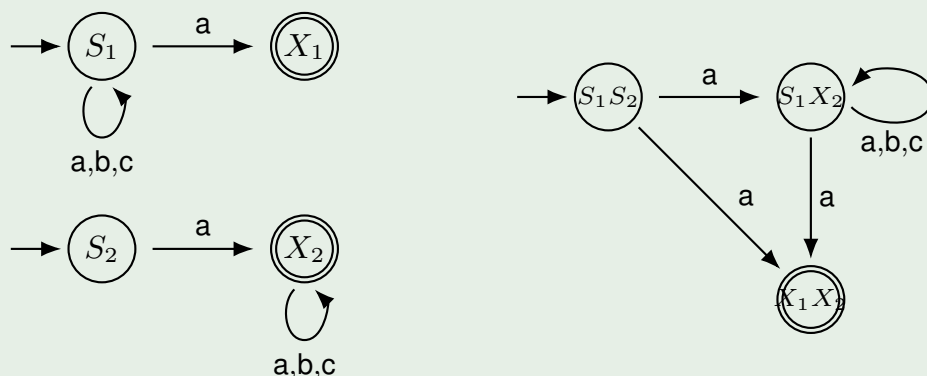
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

$$L_2 = \{a\omega \mid \omega \in A^*\}$$

Determine um AFD ou AFND que reconheça $L = L_1 \cap L_2$.

R



- de $S_1 \xrightarrow{x} S_1$ e $X_2 \xrightarrow{x} X_2$ aparece $S_1X_2 \xrightarrow{x} S_1X_2$, para $x \in \{a, b, c\}$
- de $S_1 \xrightarrow{a} X_1$ e $X_2 \xrightarrow{a} X_2$ aparece $S_1X_2 \xrightarrow{a} X_1X_2$

Intersecção de AF: definição

D Seja $M_1 = (A, Q_1, q_1, \delta_1, F_1)$ e $M_2 = (A, Q_2, q_2, \delta_2, F_2)$ dois autómatos (AFD ou AFND) quaisquer.

O AFND $M = (A, Q, q_0, \delta, F)$, onde

$$Q = Q_1 \times Q_2$$

$$q_0 = (q_1, q_2)$$

$$F = F_1 \times F_2$$

$$\delta \subseteq (Q_1 \times Q_2) \times A_\epsilon \times (Q_1 \times Q_2)$$

sendo δ definido de modo que

$((q_i, q_j), a, (q'_i, q'_j)) \in \delta$ se e só se $(q_i, a, q'_i) \in \delta_1$ e $(q_j, a, q'_j) \in \delta_2$,

implementa intersecção de M_1 e M_2 , ie., $L(M) = L(M_1) \cap L(M_2)$.

Complementação de AF

Q Sobre o alfabeto $A = \{a, b, c\}$, seja

$$L_1 = \{a\omega \mid \omega \in A^*\}$$

Determine um AF que reconheça a linguagem $\overline{L_1}$.

R

- Para se obter o complementar de um autômato finito determinista (em sentido estrito, ie. com todos os estados representados) basta complementar o conjunto de aceitação
- Para o caso de um autômato finito não determinista **é preciso** calcular o determinista equivalente e complementá-lo.

Complementação de AF: exemplo

Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

Determine um AFD ou AFND que reconheça $L = \overline{L_1}$.

R

-
- Como criar um AF que represente a intersecção de L_1 e L_2 ?

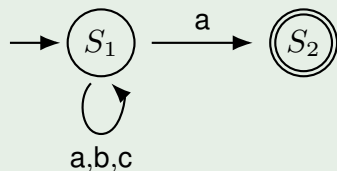
Complementação de AF: exemplo

Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

Determine um AFD ou AFND que reconheça $L = \overline{L_1}$.

R



- Considere-se um AFND para a linguagem L_1

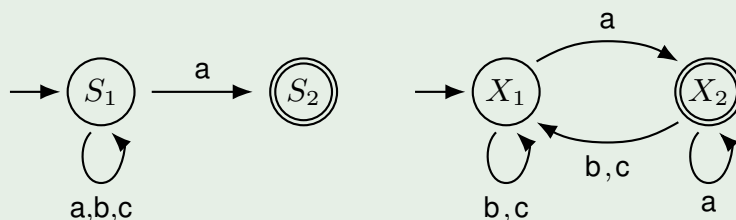
Complementação de AF: exemplo

Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

Determine um AFD ou AFND que reconheça $L = \overline{L_1}$.

R



- Obtenha-se um determinista equivalente

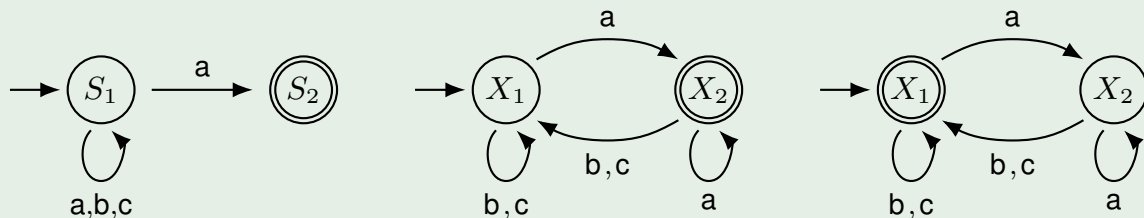
Complementação de AF: exemplo

Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{\omega a \mid \omega \in A^*\}$$

Determine um AFD ou AFND que reconheça $L = \overline{L_1}$.

R



- Complemente-se os estados de aceitação

Operações sobre AF: exercício

Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

$$L_1 = \{v\omega \mid v \in \{a, b\} \wedge \omega \in A^*\} \quad (\text{palavras começadas por } a \text{ ou } b)$$

$$L_2 = \{\omega \in A^* \mid \#(a, \omega) \bmod 2 = 0\} \quad (\text{palavras com um número par de } a)$$

Determine AF que reconheça a linguagem

- L_1
- L_2
- $L_3 = L_1 \cup L_2$
- $L_4 = L_1 \cdot L_2$
- $L_6 = \overline{L_1} \cap L_2$
- $L_7 = \overline{L_2}$
- $L_8 = (L_4 \cup L_3)^*$

Equivalência entre ER e AF

- A classe das linguagens cobertas por expressões regulares (ER) é a mesma que a classe das linguagens cobertas por autómatos finitos (AF)
- Logo:
 - Se e é uma ER, então $\exists M \in AF : L(M) = L(e)$
 - Se M é um AF, então $\exists e \in ER : L(e) = L(M)$
- Isto introduz duas operações:
 - Como converter uma ER num AF equivalente
 - Como converter um AF numa ER equivalente

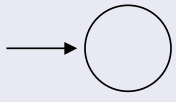
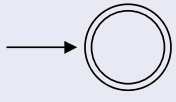
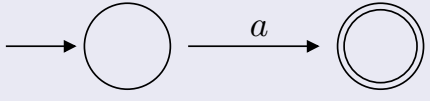
Conversão de uma ER num AF

Abordagem

- Já se viu anteriormente que uma expressão regular qualquer é:
 - ou um elemento primitivo;
 - ou uma expressão do tipo $e_1|e_2$, sendo e_1 e e_2 duas expressões regulares quaisquer
 - ou uma expressão do tipo e_1e_2 , sendo e_1 e e_2 duas expressões regulares quaisquer
 - ou uma expressão do tipo e^* , sendo e uma expressão regular qualquer
- Já se viu anteriormente como realizar a **reunião**, a **concatenação** e o **fecho** de autómatos finitos
- Então, se se identificar autómatos finitos equivalentes às expressões regulares primitivas, tem-se o problema da conversão de uma expressão regular para um autómato finito resolvido

Conversão de uma ER num AF

Autômatos dos elementos primitivos

expressão regular	autômato finito
\emptyset	
ε	
a	

- Na realidade, o autômato referente a ε pode ser obtido aplicando o fecho ao autômato de \emptyset

Conversão de uma ER num AF

Algoritmo de conversão

- Se a expressão regular é do tipo primitivo, o autômato correspondente pode ser obtido da tabela anterior
- Se é do tipo e^* , aplica-se este mesmo algoritmo na obtenção de um autômato equivalente à expressão regular e e, de seguida, aplica-se o fecho de autômatos
- Se é do tipo e_1e_2 , aplica-se este mesmo algoritmo na obtenção de autômatos para as expressões e_1 e e_2 e, de seguida, aplica-se a concatenação de autômatos
- Finalmente, se é do tipo $e_1|e_2$, aplica-se este mesmo algoritmo na obtenção de autômatos para as expressões e_1 e e_2 e, de seguida, aplica-se a reunião de autômatos

- Na realidade, o algoritmo corresponde a um processo de decomposição arbórea a partir da raiz seguido de um processo de construção arbórea a partir das folhas

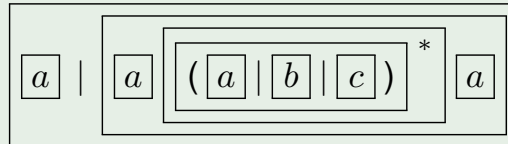
Conversão de uma ER num AF

Exemplo

Q Construa um autômato equivalente à expressão regular $e = a|a(a|b|c)^*a$

R

1 Decomposição



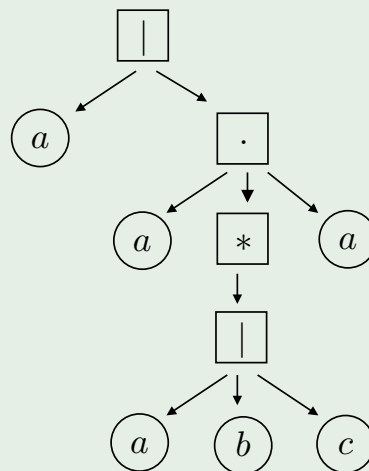
Conversão de uma ER num AF

Exemplo

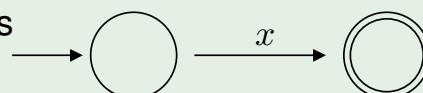
Q Construa um autômato equivalente à expressão regular $e = a|a(a|b|c)^*a$

R

1 Decomposição



2 Autômatos primitivos

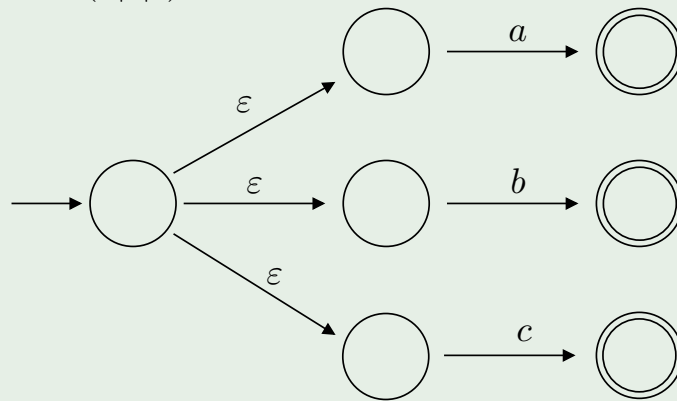


com $x = \{a, b, c\}$

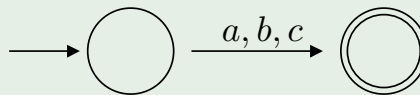
Conversão de uma ER num AF

Exemplo

3 Reunião para obter $(a|b|c)$



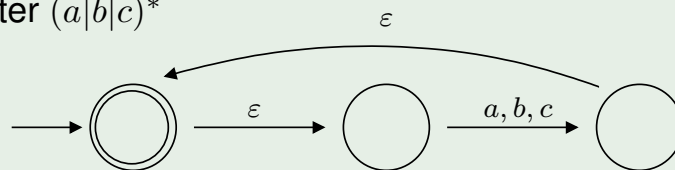
4 Simplificando



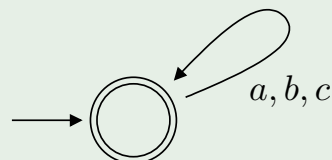
Conversão de uma ER num AF

Exemplo

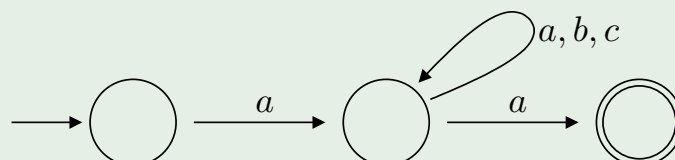
5 Fecho para obter $(a|b|c)^*$



6 Simplificando



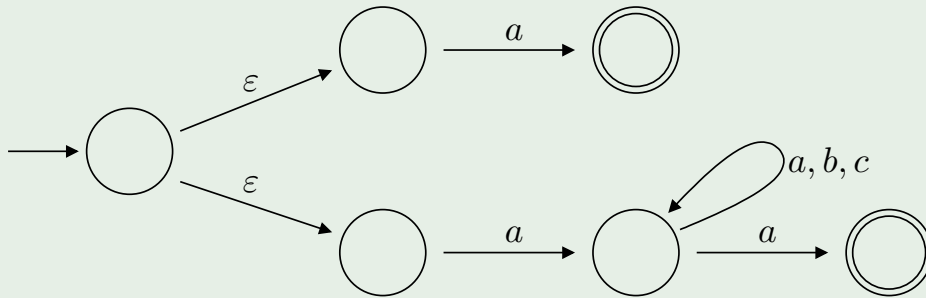
7 Concatenação (já com simplificação) para obter $a(a|b|c)^*a$



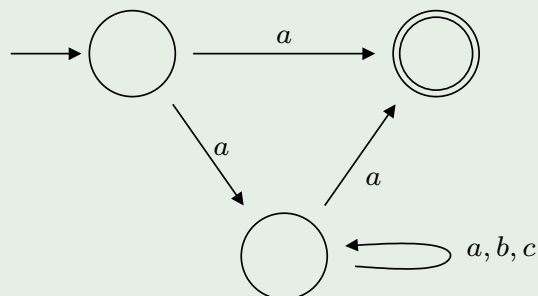
Conversão de uma ER num AF

Exemplo

8 Finalmente obtenção de $a|a(a|b|c)^*a$



9 Simplificando



Autômato finito generalizado (AFG)

Definição

\mathcal{D} Um **autômato finito generalizado** (AFG) é um quintuplo

$M = (A, Q, q_0, \delta, F)$, em que:

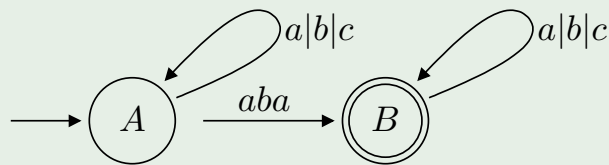
- A é o alfabeto de entrada
- Q é um conjunto finito não vazio de estados
- $q_0 \in Q$ é o estado inicial
- $\delta \subseteq (Q \times E \times Q)$ é a relação de transição entre estados, sendo E o conjunto das expressões regulares definidas sobre A
- $F \subseteq Q$ é o conjunto dos estados de aceitação

- A diferença em relação ao AFD e AFND está na definição da relação δ . Neste caso as etiquetas são *expressões regulares*
- Com base nesta definição os AFD e os AFND são autômatos finitos generalizados

Autômato finito generalizado (AFG)

Exemplo

- O AFG seguinte representa o conjunto das palavras, definidas sobre o alfabeto $A = \{a, b, c\}$, que contêm a sub-palavra aba

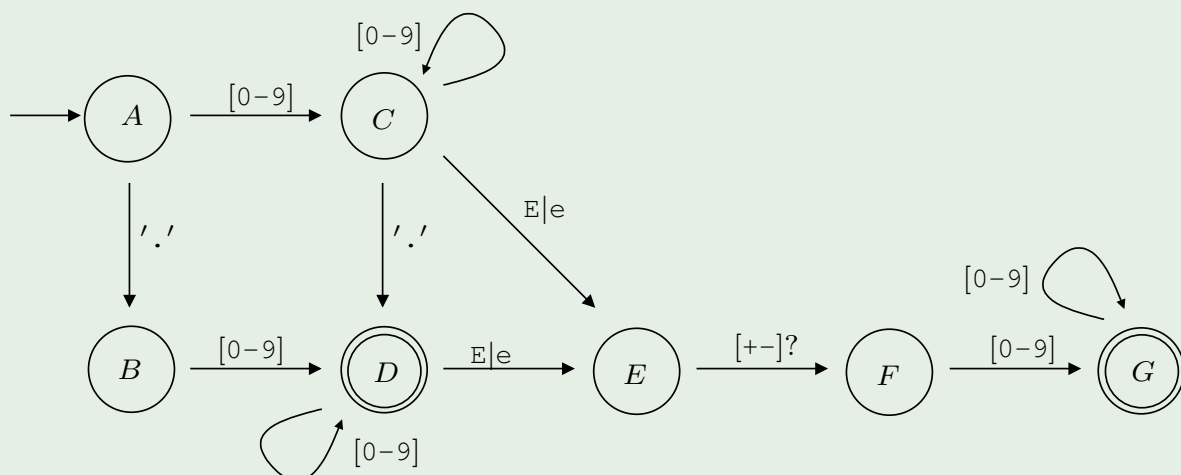


- Note que a etiqueta das transições $A \rightarrow A$ e $B \rightarrow B$ é $a|b|c$ (uma expressão regular) e não a, b, c (que representa 3 transições, uma em a , uma em b e uma em c)

Autômato finito generalizado (AFG)

Exemplo

- O AFG seguinte representa as constantes reais em \mathbb{C}

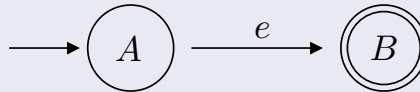


- Note que se usou $.,'$ e não $.,$, porque o último é uma expressão regular que representa qualquer letra do alfabeto

Conversão de um AFG numa ER

Abordagem

\mathcal{D} UM AFG com a forma



designa-se por **autômato finito generalizado reduzido**

- Note que:
 - O estado A não é de aceitação e não tem transições a chegar
 - O estado B é de aceitação e não tem transições a sair
- Se se reduzir um AFG à forma anterior, e é uma expressão regular equivalente ao autômato
- O processo de conversão resume-se assim à conversão de AFG à forma reduzida

Conversão de um AFG numa ER

Algoritmo de conversão

- 1 transformação de um AFG noutro cujo estado inicial **não tenha transições a chegar**
 - Se necessário, acrescenta-se um novo estado inicial com uma transição em ε para o antigo
- 2 transformação de um AFG noutro com **um único estado de aceitação, sem transições de saída**
 - Se necessário, acrescenta-se um novo estado, que passa a ser o único de aceitação, que recebe transições em ε dos anteriores estados de aceitação, que deixam de o ser
- 3 Eliminação dos estados intermédios
 - Os estados são eliminados um a um, em processos de transformação que mantêm a equivalência

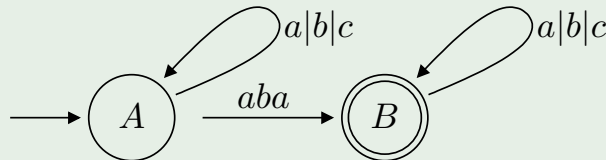
Conversão de um AFG numa ER

Ilustração com um exemplo

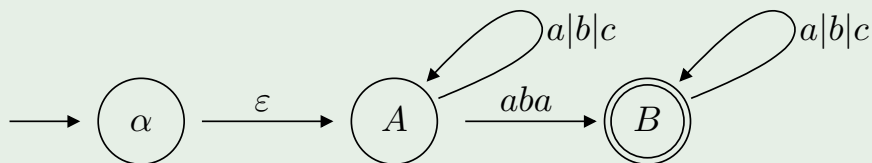
- 1 transformação de um AFG noutro cujo estado inicial **não tenha transições a chegar**

- Se necessário, acrescenta-se um novo estado inicial com uma transição em ε para o antigo

antes



depois



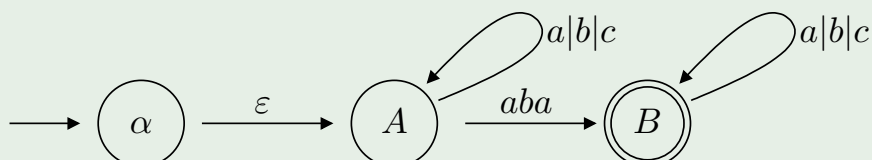
Conversão de um AFG numa ER

Ilustração com um exemplo

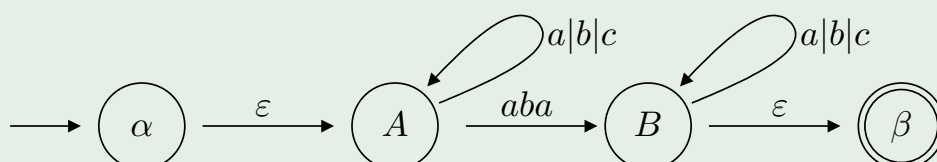
- 2 transformação de um AFG noutro com **um único estado de aceitação e sem transições de saída**

- Se necessário, acrescenta-se um novo estado, que passa a ser o único de aceitação, que recebe transições em ε dos anteriores estados de aceitação, que deixam de o ser

antes



depois



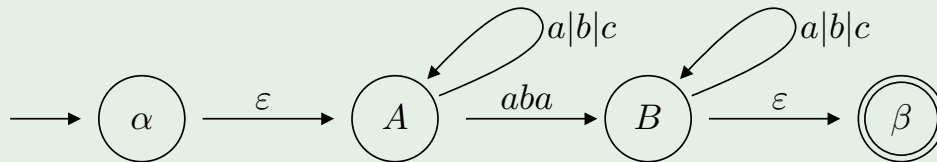
Conversão de um AFG numa ER

Ilustração com um exemplo

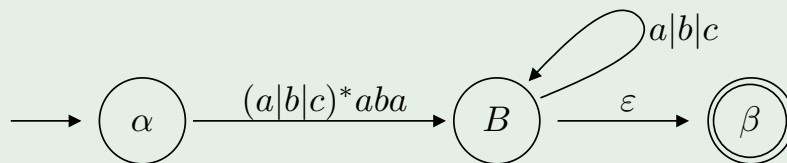
3 Eliminação dos restantes estados

- Os estados são eliminados um a um, em processos de transformação que mantêm a equivalência
- Comece-se pelo estado A

antes



depois da eliminação de A



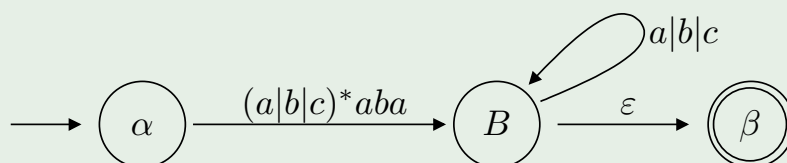
Conversão de um AFG numa ER

Ilustração com um exemplo

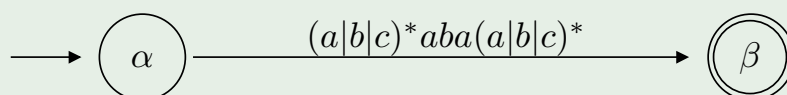
3 Eliminação dos restantes estados

- Os estados são eliminados um a um, em processos de transformação que mantêm a equivalência
- Remova-se agora o estado B

depois da eliminação de A



depois da eliminação de A, seguido da eliminação de B

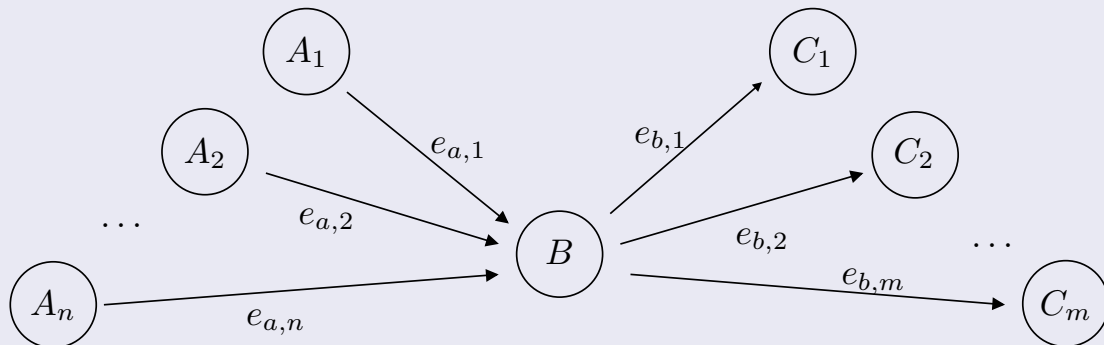


- Sendo $(a|b|c)^*aba(a|b|c)^*$ a expressão regular pretendida

Conversão de um AFG numa ER

Algoritmo de eliminação de um estado

- Caso em que o estado a eliminar (B) **não tem** transições de si para si

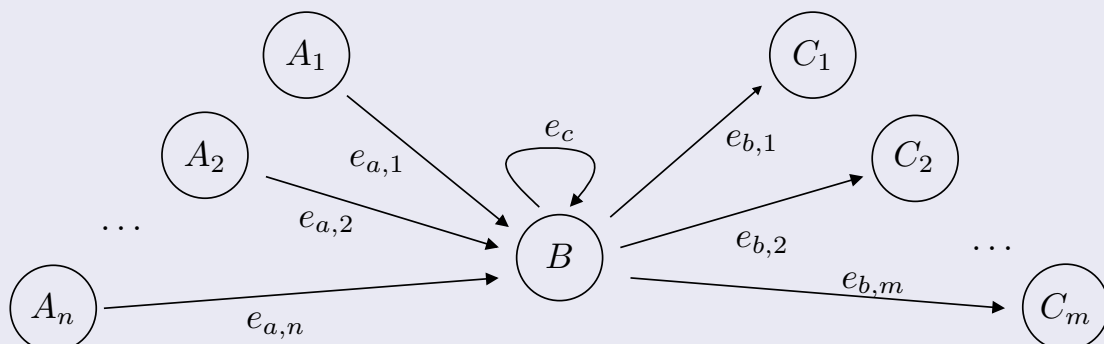


- Pode acontecer que haja $A_i = C_j$
- Para ir de A_i para C_j através de B , para $i = 1, 2, \dots, n$ e $j = 1, 2, \dots, m$, é preciso uma palavra que encaixe na expressão regular $(e_{a,i})(e_{b,j})$
- Então, se se retirar B , é preciso acrescentar uma transição de A_i para C_j que contemple essas palavras, ou seja, com a etiqueta $(e_{a,i})(e_{b,j})$
- Esta transição fica em paralelo com uma que já exista

Conversão de um AFG numa ER

Algoritmo de eliminação de um estado

- Caso em que o estado a eliminar (B) **tem** transições de si para si

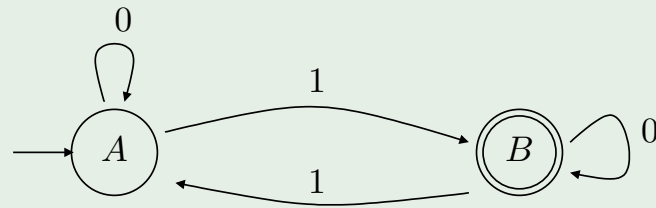


- Pode acontecer que haja $A_i = C_j$
- Para ir de A_i para C_j através de B , para $i = 1, 2, \dots, n$ e $j = 1, 2, \dots, m$, é preciso uma palavra que encaixe na expressão regular $(e_{a,i})(e_c)^*(e_{b,j})$
- Então, se se retirar B , é preciso acrescentar uma transição de A_i para C_j que contemple essas palavras, ou seja com etiqueta $(e_{a,i})(e_c)^*(e_{b,j})$
- Esta transição fica em paralelo com uma que já exista

Conversão de um AFG numa ER

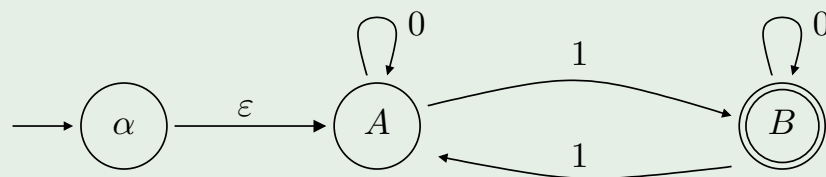
Exercício

Q Obtenha uma ER equivalente ao AF seguinte



R Aplique-se passo a passo o algoritmo de conversão

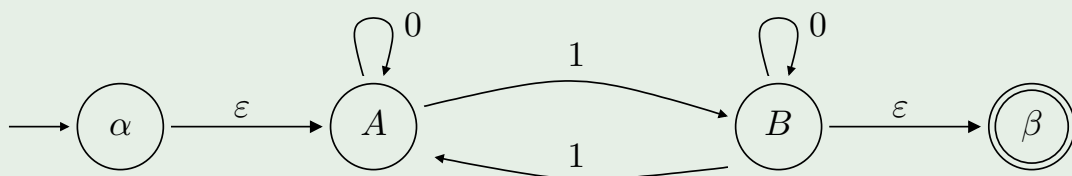
- Porque o estado inicial possui uma transição a entrar, deve substituir-se o estado inicial, de acordo com o passo 1 do algoritmo



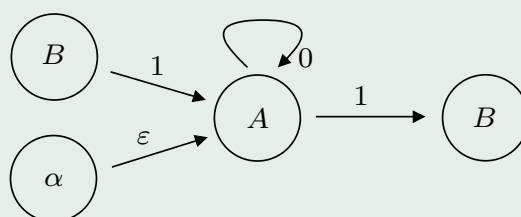
Exemplo de conversão de um AFG numa ER

Exercício

- Porque o estado de aceitação possui uma transição a sair, deve-se aplicar o passo 2 do algoritmo de conversão



- Elimine-se o estado A. Para isso é preciso ver os segmentos de caminhos que passam por A.

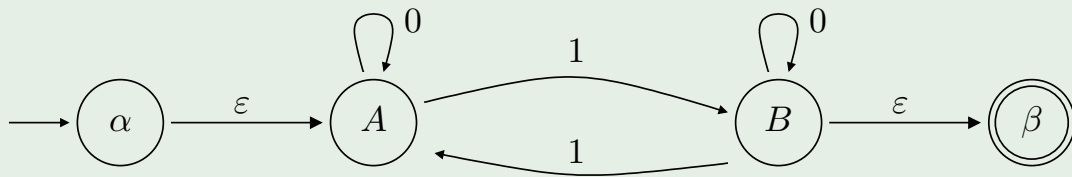


- Note que B aparece à esquerda e à direita

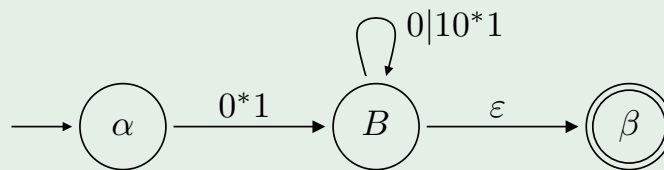
Exemplo de conversão de um AFG numa ER

Exercício

- Porque o estado de aceitação possui uma transição a sair, deve-se aplicar o passo 2 do algoritmo de conversão



- Eliminando o estado A obtém-se



- Finalmente, eliminando o estado B obtém-se a ER $0^*1(0|10^*1)^*$

Equivalência entre GR e AF

- A classe das linguagens cobertas por gramáticas regulares (ER) é a mesma que a classe das linguagens cobertas por autómatos finitos (AF)
- Logo:
 - Se G é uma ER, então $\exists_{M \in AF} : L(M) = L(G)$
 - Se M é um AF, então $\exists_{G \in ER} : L(G) = L(M)$
- Isto introduz duas operações:
 - Como converter um AF numa GR equivalente
 - Como converter uma GR num AF equivalente

Conversão de um AF numa GR

Procedimento de conversão

\mathcal{A} Seja $M = (A, Q, q_0, \delta, F)$ um autômato finito qualquer.

A GR $G = (T, N, P, S)$, onde

$$T = A$$

$$N = Q$$

$$S = q_0$$

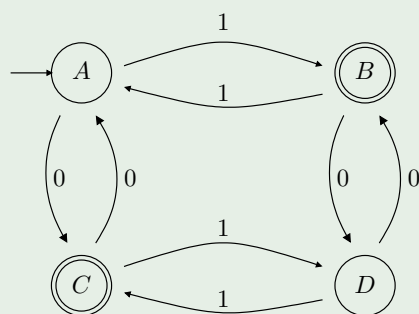
$$P = \{p \rightarrow a q : p, q \in Q \wedge a \in T \wedge (p, a, q) \in \delta\} \\ \cup \{p \rightarrow \varepsilon : p \in F\}$$

representa a mesma linguagem que M , isto é, $L(G) = L(M)$.

Conversão de um AF numa GR

Exemplo

\mathcal{Q} Determine uma GR equivalente ao AF



\mathcal{R}

$$A \rightarrow 0 C \mid 1 B$$

$$B \rightarrow 0 D \mid 1 A \mid \varepsilon$$

$$C \rightarrow 0 A \mid 1 D \mid \varepsilon$$

$$D \rightarrow 0 B \mid 1 C$$

Conversão de uma GR num AFG

Procedimento de conversão

Seja $G = (T, N, P, S)$ uma gramática regular qualquer.

O AF $M = (A, Q, q_0, \delta, F)$, onde

$$A = T$$

$$Q = N \cup \{q_f\}, \quad \text{com } q_f \notin N$$

$$q_0 = S$$

$$F = \{q_f\}$$

$$\delta = \{(q_i, e, q_j) : q_i, q_j \in N \wedge e \in T^* \wedge q_i \rightarrow e q_j \in P\} \\ \cup \{(q, e, q_f) : q \in N \wedge e \in T^* \wedge q \rightarrow e \in P\}$$

representa a mesma linguagem que G , isto é, $L(M) = L(G)$.

Conversão de uma GR num AFG

Exemplo

Determine um AFG equivalente à GR

$$S \rightarrow a S \mid b S \mid c S \mid a b a X$$

$$X \rightarrow a X \mid b X \mid c X \mid \varepsilon$$

Se

Sendo $M = (A, Q, q_0, \delta, F)$ o AFG equivalente, tem-se

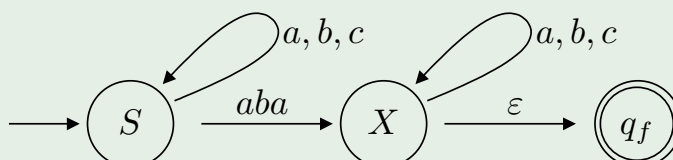
$$A = \{a, b, c\}$$

$$Q = \{S, X, q_f\}$$

$$q_0 = S$$

$$\delta = \{(S, a, S), (S, b, S), (S, c, S), (S, aba, X), \\ (X, a, X), (X, b, X), (X, c, X), (X, \varepsilon, q_f)\}$$

$$F = \{q_f\}$$





Compiladores

Gramáticas livres de contexto

Artur Pereira <artur@ua.pt>,
Miguel Oliveira e Silva <mos@ua.pt>

DETI, Universidade de Aveiro

Ano letivo de 2022-2023

Sumário

- ① Gramáticas livres de contexto (GLC)
- ② Derivação e árvore de derivação
- ③ Ambiguidade
- ④ Projeto de gramáticas
- ⑤ Operações sobre GLC
- ⑥ Limpeza de gramáticas

Gramáticas

Definição

Uma gramática é um quádruplo $G = (T, N, P, S)$, onde

- T é um conjunto finito não vazio de símbolos **terminais**;
- N , com $N \cap T = \emptyset$, é um conjunto finito não vazio de símbolos **não terminais**;
- P é um conjunto de **produções** (ou regras de rescrita), cada uma da forma $\alpha \rightarrow \beta$;
- $S \in N$ é o símbolo inicial.

-
- α e β são designados por **cabeça da produção** e **corpo da produção**, respetivamente.
 - No caso geral $\alpha \in (N \cup T)^* \times N \times (N \cup T)^*$ e $\beta \in (N \cup T)^*$.
 - Em ANTLR:
 - os terminais são representados por ids começados por letra maiúscula
 - os não terminais são representados por ids começados por letra minúscula

Gramáticas livres de contexto – GLC

Definição

\mathcal{D} Uma gramática $G = (T, N, P, S)$ diz-se **livre de contexto** (ou **independente do contexto**) se, para qualquer produção $(\alpha \rightarrow \beta) \in P$, as duas condições seguintes são satisfeitas

$$\begin{aligned}\alpha &\in N \\ \beta &\in (T \cup N)^*\end{aligned}$$

- A linguagem gerada por uma gramática livre de contexto diz-se livre de contexto
- As gramáticas regulares são livres de contexto
- As gramáticas livres de contexto são fechadas sob as operações de reunião, concatenação e fecho
 - **mas não o são** sob as operações de intersecção e complementação.

-
- Note que: se $\beta \in T^* \cup T^* N$, então $\beta \in (T \cup N)^*$

Derivação

Exemplo

Q Considere, sobre o alfabeto $T = \{a, b, c\}$, a gramática

$$S \rightarrow \varepsilon \mid a B \mid b A \mid c S$$

$$A \rightarrow a S \mid b A A \mid c A$$

$$B \rightarrow a B B \mid b S \mid c B$$

e transforme o símbolo inicial S na palavra $aabcbcb$ por aplicação sucessiva de produções da gramática

R

$$\begin{aligned} S &\Rightarrow aB \Rightarrow aaBB \Rightarrow aabSB \Rightarrow aabcSB \Rightarrow aabcB \Rightarrow aabcbS \\ &\Rightarrow aabcbcbS \Rightarrow aabcbcb \end{aligned}$$

- Acabou de se obter uma **derivação à esquerda** da palavra $aabcbcb$
- Cada passo dessa derivação é uma **derivação direta à esquerda**

- Quando há dois ou mais símbolos não terminais, opta-se por expandir primeiro o mais à esquerda

Derivação

Definições

D Dada uma palavra $\alpha A \beta$, com $A \in N$ e $\alpha, \beta \in (N \cup T)^*$, e uma produção $(A \rightarrow \gamma) \in P$, com $\gamma \in (N \cup T)^*$, chama-se **derivação direta** à rescrita de $\alpha A \beta$ em $\alpha \gamma \beta$, denotando-se

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

D Dada uma palavra $\alpha A \beta$, com $A \in N$, $\alpha \in T^*$ e $\beta \in (N \cup T)^*$, e uma produção $(A \rightarrow \gamma) \in P$, com $\gamma \in (N \cup T)^*$, chama-se **derivação direta à esquerda** à rescrita de $\alpha A \beta$ em $\alpha \gamma \beta$, denotando-se

$$\alpha A \beta \xRightarrow{E} \alpha \gamma \beta$$

D Dada uma palavra $\alpha A \beta$, com $A \in N$, $\alpha \in (N \cup T)^*$ e $\beta \in T^*$, e uma produção $(A \rightarrow \gamma) \in P$, com $\gamma \in (N \cup T)^*$, chama-se **derivação direta à direita** à rescrita de $\alpha A \beta$ em $\alpha \gamma \beta$, denotando-se

$$\alpha A \beta \xRightarrow{D} \alpha \gamma \beta$$

Derivação

Definições

\mathcal{D} Chama-se **derivação** a uma sucessão de zero ou mais derivações diretas, denotando-se

$$\alpha \Rightarrow^* \beta \quad \equiv \quad \alpha = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \cdots \Rightarrow \gamma_n = \beta$$

onde n é o comprimento da derivação.

\mathcal{D} Chama-se **derivação à esquerda** a uma sucessão de zero ou mais derivações diretas à esquerda, denotando-se

$$\alpha \Rightarrow^E * \beta \quad \equiv \quad \alpha = \alpha_0 \xRightarrow{E} \alpha_1 \xRightarrow{E} \cdots \xRightarrow{E} \alpha_n = \beta$$

onde n é o comprimento da derivação.

\mathcal{D} Chama-se **derivação à direita** a uma sucessão de zero ou mais derivações diretas à direita, denotando-se

$$\alpha \xRightarrow{D} * \beta \quad \equiv \quad \alpha = \gamma_0 \xRightarrow{D} \gamma_1 \xRightarrow{D} \cdots \xRightarrow{D} \gamma_n = \beta$$

onde n é o comprimento da derivação.

Derivação

Exemplo

\mathcal{Q} Considere, sobre o alfabeto $T = \{a, b, c\}$, a gramática seguinte

$$S \rightarrow \varepsilon \mid a B \mid b A \mid c S$$

$$A \rightarrow a S \mid b A A \mid c A$$

$$B \rightarrow a B B \mid b S \mid c B$$

Determine as derivações à esquerda e à direita da palavra $aabcbcb$

\mathcal{R}

à esquerda

$$\begin{aligned} S &\Rightarrow aB \Rightarrow aaBB \Rightarrow aabSB \Rightarrow aabcSB \\ &\Rightarrow aabcB \Rightarrow aabcbS \Rightarrow aabcbcbS \Rightarrow aabcbcb \end{aligned}$$

à direita

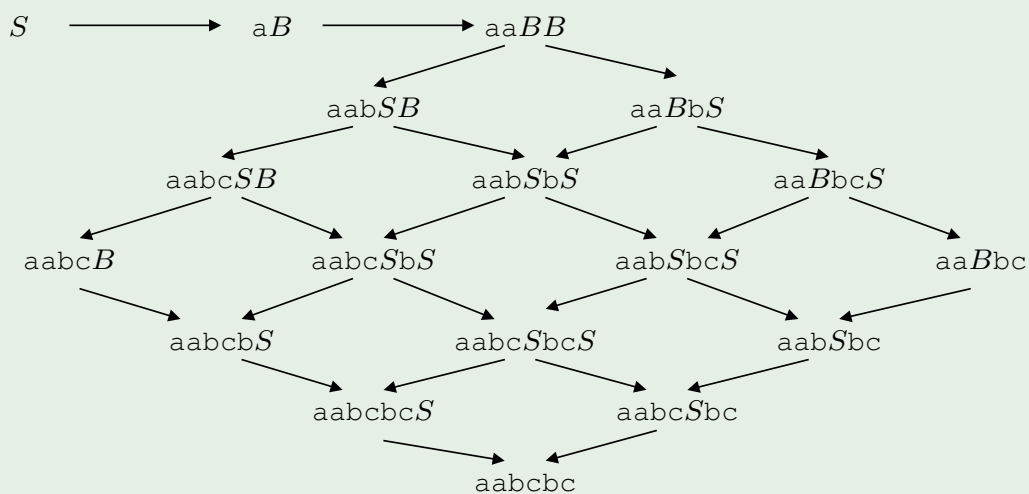
$$\begin{aligned} S &\Rightarrow aB \Rightarrow aaBB \Rightarrow aaBbS \Rightarrow aaBbcbS \\ &\Rightarrow aaBbcb \Rightarrow aabSbcb \Rightarrow aabcbSbcb \Rightarrow aabcbcb \end{aligned}$$

• Note que se usou \Rightarrow em vez de \xRightarrow{D} e \xRightarrow{E}

Derivação

Alternativas de derivação

- O grafo seguinte capta as alternativas de derivação. Considera-se novamente a palavra `aabcbcb` e a gramática anterior



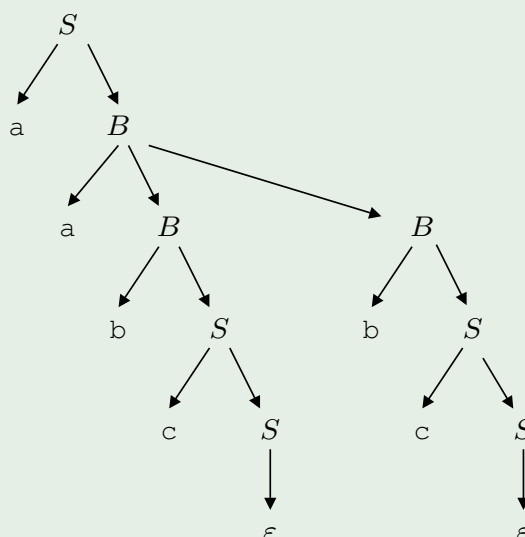
- Identifique os caminhos que correspondem às derivações à direita e à esquerda

Derivação

Árvore de derivação

- \mathcal{D} Uma **árvore de derivação** (*parse tree*) é uma representação de uma derivação onde os nós-ramos são símbolos não terminais e os nós-folhas são símbolos terminais

- A árvore de derivação da palavra `aabcbcb` na gramática anterior é

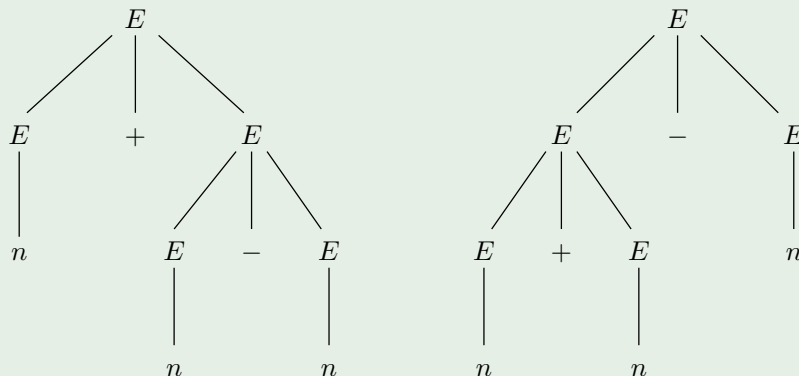


Ambiguidade

Ilustração através de um exemplo

- Considere a gramática $S \rightarrow S + S \mid S - S \mid (S) \mid n$ e desenhe a árvore de derivação da palavra $n+n-n$

\mathcal{R} Podem obter-se duas árvores de derivação diferentes



- Pode haver duas interpretações diferentes para a palavra; há **ambiguidade**

Ambiguidade

Definição

- \mathcal{D} Diz-se que uma palavra é derivada **ambiguamente** se possuir duas ou mais árvores de derivação distintas
- \mathcal{D} Diz-se que uma gramática é **ambígua** se possuir pelo menos uma palavra gerada ambiguamente
- Frequentemente é possível definir-se uma gramática não ambígua que gera a mesma linguagem que uma ambígua
- No entanto, há gramáticas **inerentemente ambíguas**

Por exemplo, a linguagem

$$L = \{a^i b^j c^k \mid i = j \vee j = k\}$$

não possui uma gramática não ambígua que a represente.

Ambiguidade

Remoção da ambiguidade

\mathcal{R} Considere-se novamente a gramática

$$S \rightarrow S + S \mid S - S \mid (S) \mid n$$

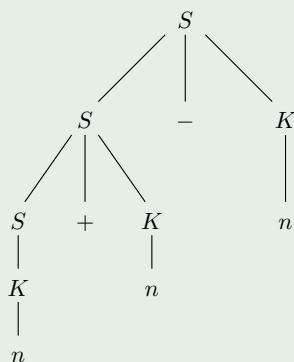
e obtenha-se uma gramática não ambígua equivalente

\mathcal{R}

$$S \rightarrow K \mid S + K \mid S - K$$

$$K \rightarrow n \mid (S)$$

\mathcal{Q} Desenhe a árvore de derivação da palavra $n+n-n$ na nova gramática



Projeto de gramáticas

Exemplo #1, solução #1

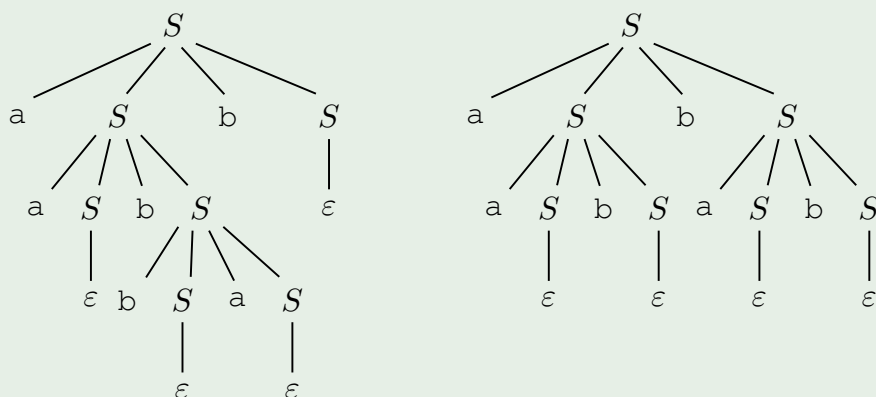
\mathcal{Q} Sobre o conjunto de terminais $T = \{a, b\}$, determine uma gramática livre de contexto que represente a linguagem

$$L_1 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$$

\mathcal{R}_1

$$S \rightarrow \varepsilon \mid a S b S \mid b S a S$$

\mathcal{Q} A gramática é ambígua? Analise a palavra $aabbab$



Projeto de gramáticas

Exemplo #1, solução #2

Q Sobre o conjunto de terminais $T = \{a, b\}$, determine uma gramática livre de contexto que represente a linguagem

$$L_1 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$$

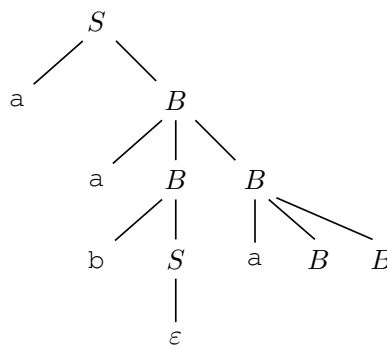
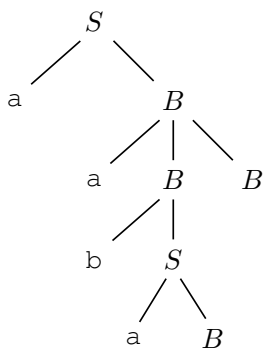
\mathcal{R}_2

$$S \rightarrow \varepsilon \mid a B \mid b A$$

$$A \rightarrow a S \mid b A A$$

$$B \rightarrow a B B \mid b S$$

Q A gramática é ambígua?
Analise a palavra aababbb.



- Falta expandir alguns nós

Projeto de gramáticas

Exemplo #1, solução #3

Q Sobre o conjunto de terminais $T = \{a, b\}$, determine uma gramática livre de contexto que represente a linguagem

$$L_1 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$$

\mathcal{R}_3

$$S \rightarrow \varepsilon \mid a B S \mid b A S$$

$$A \rightarrow a \mid b A A$$

$$B \rightarrow a B B \mid b$$

Q A gramática é ambígua? Analise a palavra aababbb

Projeto de gramáticas

Exemplo #2

Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática livre de contexto que represente a linguagem

$$L_2 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$$

R

$$S \rightarrow \varepsilon \mid a B S \mid b A S \mid c S$$

$$A \rightarrow a \mid b A A \mid c A$$

$$B \rightarrow a B B \mid b \mid c B$$

Q A gramática é ambígua?

Projeto de gramáticas

Exemplo #3, solução #1

Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática livre de contexto que represente a linguagem

$$L_3 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega) \wedge \\ \forall i \leq |\omega| \#(a, \text{prefix}(i, \omega)) \geq \#(b, \text{prefix}(i, \omega))\}$$

R₁

$$S \rightarrow \varepsilon \mid a S b S \mid c S$$

Q A gramática é ambígua? Analise a palavra aababb

- O número de ocorrências das letras a e b é igual, mas em qualquer prefixo das palavras da linguagem não pode haver mais bs que as, ou seja o a aparece antes
- Solução inspirada na do exemplo 1.1, removendo a produção $S \rightarrow b S a S$

Projeto de gramáticas

Exemplo #3: solução #2

Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática livre de contexto que represente a linguagem

$$L_3 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega) \wedge \forall_{i \leq |\omega|} \#(a, \text{prefix}(i, \omega)) \geq \#(b, \text{prefix}(i, \omega))\}$$

\mathcal{R}_2

$$S \rightarrow \varepsilon \mid a B \mid c S$$

$$B \rightarrow a B B \mid b S \mid c B$$

Q A gramática é ambígua? Analise a palavra aababb

- Solução inspirada na do exemplo 1.2, removendo a produção $S \rightarrow b A$ e as começadas por A

Projeto de gramáticas

Exemplo #3: solução #3

Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática livre de contexto que represente a linguagem

$$L_3 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega) \wedge \forall_{i \leq |\omega|} \#(a, \text{prefix}(i, \omega)) \geq \#(b, \text{prefix}(i, \omega))\}$$

\mathcal{R}_3

$$S \rightarrow \varepsilon \mid a B S \mid c S$$

$$B \rightarrow a B B \mid b \mid c B$$

Q A gramática é ambígua? Analise a palavra aababb

- Solução inspirada na do exemplo 1.3, removendo a produção $S \rightarrow b A S$ e as começadas por A

Projeto de gramáticas

Exercício

- Q Sobre o conjunto de terminais $T = \{a, b, c, (,), +, *\}$, determine uma gramática independente do contexto que represente a linguagem

$$L = \{ \omega \in T^* : \\ \omega \text{ representa uma expressão regular sobre o alfabeto } \{a, b, c\} \}$$

- R Em ANTLR, poder-se-ia fazer

```
S → E
E → E '*'
   | E E
   | E '+' E
   | '(' E ')'
   | 'a' | 'b' | 'c'
```

mas em geral não, porque, em geral, as alternativas estão todas ao mesmo nível

- Como escrever a gramática de modo à precedência ser imposta por construção?

-
- Está a usar-se o operador + em vez do |

Projeto de gramáticas

Exercício (cont.)

- R Em geral

```
S → E
E → E '+' T
   | T
T → T F
   | F
F → F '*'
   | O
O → '(' E ')'
   | 'a' | 'b' | 'c'
```

- Uma expressão é vista como uma 'soma' de termos
- Um termo é visto como um 'produto' (concatenação) de fatores
- Um fator é visto como um 'fecho' de operandos
- Um operando ou é um elemento base ou uma expressão entre parêntesis

-
- Está a usar-se o operador + em vez do |

Reunião de GLC

Exemplo

- Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática livre de contexto que represente a linguagem

$$L = \{ \omega \in T^* : \#(a, \omega) = \#(b, \omega) \vee \#(a, \omega) = \#(c, \omega) \}$$

R

$L_1 = \{ \omega \in T^* : \#(a, \omega) = \#(b, \omega) \}$	$S_1 \rightarrow \varepsilon \mid a S_1 b S_1$ $\mid b S_1 a S_1 \mid c S_1$
$L_2 = \{ \omega \in T^* : \#(a, \omega) = \#(c, \omega) \}$	$S_2 \rightarrow \varepsilon \mid a S_2 c S_2$ $\mid b S_2 \mid c S_2 a S_2$
$L = L_1 \cup L_2$	$S \rightarrow S_1 \mid S_2$ $S_1 \rightarrow \varepsilon \mid a S_1 b S_1$ $\mid b S_1 a S_1 \mid c S_1$ $S_2 \rightarrow \varepsilon \mid a S_2 c S_2$ $\mid b S_2 \mid c S_2 a S_2$

- Para esta linguagem, mesmo que as gramáticas de L_1 e L_2 não sejam ambíguas, a de L será ambígua. Porquê?

Operações sobre GLCs

Reunião

- D Sejam $G_1 = (T_1, N_1, P_1, S_1)$ e $G_2 = (T_2, N_2, P_2, S_2)$ duas gramáticas livres de contexto quaisquer, com $N_1 \cap N_2 = \emptyset$.

A gramática $G = (T, N, P, S)$ onde

$$T = T_1 \cup T_2$$

$$N = N_1 \cup N_2 \cup \{S\} \quad \text{com} \quad S \notin (N_1 \cup N_2)$$

$$P = \{S \rightarrow S_1, S \rightarrow S_2\} \cup P_1 \cup P_2$$

é livre de contexto e gera a linguagem $L = L(G_1) \cup L(G_2)$

- As novas produções $S \rightarrow S_i$, com $i = 1, 2$, permitem que G gere a linguagem $L(G_i)$
- Esta definição é idêntica à que foi dada para a operação de reunião nas gramáticas regulares

Concatenação de GLC

Exemplo

Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática livre de contexto que represente a linguagem

$$L = \{ \omega_1 \omega_2 : \omega_1, \omega_2 \in T^* \}$$

$$\wedge \#(a, \omega_1) = \#(b, \omega_1) \wedge \#(a, \omega_2) = \#(c, \omega_2) \}$$

R

$L_1 = \{ \omega \in T^* : \#(a, \omega) = \#(b, \omega) \}$	$S_1 \rightarrow \varepsilon \mid a S_1 b S_1$ $\mid b S_1 a S_1 \mid c S_1$
$L_2 = \{ \omega \in T^* : \#(a, \omega) = \#(c, \omega) \}$	$S_2 \rightarrow \varepsilon \mid a S_2 c S_2$ $\mid b S_2 \mid c S_2 a S_2$
$L = L_1 \cdot L_2$	$S \rightarrow S_1 S_2$ $S_1 \rightarrow \varepsilon \mid a S_1 b S_1$ $\mid b S_1 a S_1 \mid c S_1$ $S_2 \rightarrow \varepsilon \mid a S_2 c S_2$ $\mid b S_2 \mid c S_2 a S_2$

Operações sobre gramáticas:

Concatenação

D Sejam $G_1 = (T_1, N_1, P_1, S_1)$ e $G_2 = (T_2, N_2, P_2, S_2)$ duas gramáticas livres de contexto quaisquer, com $N_1 \cap N_2 = \emptyset$.

A gramática $G = (T, N, P, S)$ onde

$$T = T_1 \cup T_2$$

$$N = N_1 \cup N_2 \cup \{S\} \text{ com } S \notin (N_1 \cup N_2)$$

$$P = \{S \rightarrow S_1 S_2\} \cup P_1 \cup P_2$$

é livre de contexto e gera a linguagem $L = L(G_1) \cdot L(G_2)$

- A nova produção $S \rightarrow S_1 S_2$ justapõe palavras de $L(G_2)$ às de $L(G_1)$
- Esta definição é **diferente** da que foi dada para a operação de concatenação nas gramáticas regulares

Fecho de Kleene de GLC

Exemplo

- Q Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática livre de contexto que represente a linguagem

$$L = \{\omega \in T^* : \#(a, \omega) \geq \#(b, \omega)\}$$

R

$X = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$	$X \rightarrow \varepsilon \mid a B \mid b A \mid c X$ $A \rightarrow a X \mid b A A \mid c A$ $B \rightarrow a B B \mid b X \mid c B$
$A = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega) + 1\}$	Basta usar o A anterior como símbolo inicial
$L = X \cup A^*$	$S \rightarrow \varepsilon \mid A S \mid X$ $X \rightarrow \varepsilon \mid a B \mid b A \mid c X$ $A \rightarrow a X \mid b A A \mid c A$ $B \rightarrow a B B \mid b X \mid c B$

- O fecho de A inclui a palavra vazia mas não as outras palavras com $\#_a = \#_b$

Operações sobre gramáticas

Fecho de Kleene

Seja $G_1 = (T_1, N_1, P_1, S_1)$ uma gramática livre de contexto qualquer. A gramática $G = (T, N, P, S)$ onde

$$\begin{aligned} T &= T_1 \\ N &= N_1 \cup \{S\} \quad \text{com } S \notin N_1 \\ P &= \{S \rightarrow \varepsilon, S \rightarrow S_1 S\} \cup P_1 \end{aligned}$$

é livre de contexto e gera a linguagem $L = (L(G_1))^*$

- A produção $S \rightarrow \varepsilon$, per si, garante que $L^0(G_1) \subseteq L(G)$
- As produções $S \rightarrow S_1 S$ e $S \rightarrow \varepsilon$ garantem que $L^i(G_1) \subseteq L(G)$, para qualquer $i > 0$
- Esta definição é **diferente** da que foi dada para a operação de fecho nas gramáticas regulares

Símbolos produtivos e improdutivos

Exemplo de ilustração

Q Sobre o conjunto de terminais $T = \{a, b, c, d\}$, considere a gramática

$$\begin{aligned} S &\rightarrow a A b \mid b B \\ A &\rightarrow c C \mid b B \mid d \\ B &\rightarrow d D \mid b \\ C &\rightarrow A C \mid B D \mid S D \\ D &\rightarrow A D \mid B C \mid C S \\ E &\rightarrow a A \mid b B \mid \varepsilon \end{aligned}$$

- Tente expandir (através de uma derivação) o símbolo não terminal A para uma sequência apenas com símbolos terminais ($S \Rightarrow^* u$, com $u \in T^*$)
 - $A \Rightarrow d$
- Faça o mesmo com o símbolo C
 - Não consegue
- A é um símbolo **produtivo**; C é um símbolo **improdutivo**

Símbolos produtivos e improdutivos

Definição de símbolo produtivo

- Seja $G = (T, N, P, S)$ uma gramática qualquer
- Um símbolo não terminal A diz-se **produtivo** se for possível expandi-lo para uma expressão contendo apenas símbolos terminais
- Ou seja, A é produtivo se

$$A \Rightarrow^+ u \quad \wedge \quad u \in T^*$$

- Caso contrário, diz-se que A é **improdutivo**
- Uma gramática é improdutiva se o seu símbolo inicial for improdutivo

- Na gramática

$$\begin{aligned} S &\rightarrow a b \mid a S b \mid X \\ X &\rightarrow c X \end{aligned}$$

- S é produtivo, porque $S \Rightarrow ab \quad \wedge \quad ab \in T^*$
- X é improdutivo, porque $X \Rightarrow cX \Rightarrow ccX \Rightarrow^* c \cdots cX$

Símbolos produtivos

Algoritmo de cálculo

- O conjunto dos símbolos produtivos, N_p , pode ser obtido por aplicação sucessiva das seguintes regras construtivas

```
if  $(A \rightarrow \alpha) \in P$  and  $\alpha \in T^*$  then  $A \in N_p$   
if  $(A \rightarrow \alpha) \in P$  and  $\alpha \in (T \cup N_p)^*$  then  $A \in N_p$ 
```

- Algoritmo de cálculo:

```
let  $N_p \leftarrow \emptyset$ ,  $P_p \leftarrow P$       #  $N_p$  – símbolos produtivos  
repeat  
  nothingAdded  $\leftarrow$  true  
  foreach  $(A \rightarrow \alpha) \in P_p$  do  
    if  $\alpha \in (T \cup N_p)^*$  then          # se todos são terminais ou produtivos,  $A$  é produtivo  
      if  $A \notin N_p$  then                # se ainda não pertence aos produtivos  
         $N_p \leftarrow N_p \cup \{A\}$       # é lá colocado  
        nothingAdded  $\leftarrow$  false      # e é preciso repetir o processo  
         $P_p \leftarrow P_p - \{A \rightarrow \alpha\}$  # a produção já não precisa de ser processada mais  
  until nothingAdded or  $N_p = N$ 
```

Símbolos acessíveis e inacessíveis

Exemplo de ilustração

Q Sobre o conjunto de terminais $T = \{a, b, c, d\}$, considere a gramática

```
 $S \rightarrow a A b \mid b B$   
 $A \rightarrow c C \mid b B \mid d$   
 $B \rightarrow d D \mid b$   
 $C \rightarrow A C \mid B D \mid S D$   
 $D \rightarrow A D \mid B C \mid C S$   
 $E \rightarrow a A \mid b B \mid \varepsilon$ 
```

- Tente alcançar (através de uma derivação) o símbolo não terminal C a partir do símbolo inicial (S) ($S \Rightarrow^* \alpha C \beta$, com $\alpha, \beta \in (T \cup N)^*$)
 - $S \Rightarrow b B \Rightarrow b d D \Rightarrow b d B C$
- Faça o mesmo com o símbolo E
 - Não consegue
- C é um símbolo **acessível**; E é um símbolo **inacessível**

Símbolos acessíveis e inacessíveis

Definição de símbolo acessível

- Seja $G = (T, N, P, S)$ uma gramática qualquer
- Um símbolo terminal ou não terminal x diz-se **acessível** se for possível expandir S (o símbolo inicial) para uma expressão que contenha x
- Ou seja, x é acessível se

$$S \Rightarrow^* \alpha x \beta$$

- Caso contrário, diz-se que x é **inacessível**

- Na gramática

$$S \rightarrow \varepsilon \mid a S b \mid c C c$$

$$C \rightarrow c S c$$

$$D \rightarrow d X d$$

$$X \rightarrow C C$$

- D , d , e X são inacessíveis
- Os restantes são acessíveis

Símbolos acessíveis

Algoritmo de cálculo

- O conjunto dos seus símbolos acessíveis, V_A , pode ser obtido por aplicação das seguintes regras construtivas

$$S \in V_A$$

$$\text{if } A \rightarrow \alpha B \beta \in P \text{ and } A \in V_A \text{ then } B \in V_A$$

- Algoritmo de cálculo:

$$V_A \leftarrow \{S\}$$

no fim, ficará com todos os símbolos acessíveis

$$N_A \leftarrow \{S\}$$

conjunto de símbolos não terminais acessíveis a processar

repeat

$$X \leftarrow \text{elementOf}(N_A)$$

retira um elemento qualquer de N_A

foreach $(X \rightarrow \alpha) \in P$ **do**

foreach x **in** α **do**

if $x \notin V_A$ **then**

se ainda não está marcado como acessível

$$V_A \leftarrow V_A \cup \{x\}$$

passa a estar

if $x \in N$ **then**

se adicionalmente é não terminal

$$N_A \leftarrow N_A \cup \{x\}$$

terá de ser processado

until $N_A = \emptyset$

Gramáticas limpas

Algoritmo de limpeza

- Numa gramática, os símbolos inacessíveis e os símbolos improdutivos são **símbolos inúteis**
- Se tais símbolos forem removidos obtém-se uma gramática equivalente
- Diz-se que uma gramática é **limpa** se não possuir símbolos inúteis
- Para limpar uma gramática deve-se:
 - começar por a expurgar dos símbolos improdutivos
 - só depois remover os inacessíveis

Gramáticas limpas

Exemplo #1

Q Sobre o conjunto de terminais $T = \{a, b, c, d\}$, determine uma gramática limpa equivalente à gramática seguinte

$$\begin{aligned} S &\rightarrow a A b \mid b B \\ A &\rightarrow c C \mid b B \mid d \\ B &\rightarrow d D \mid b \\ C &\rightarrow A C \mid B D \mid S D \\ D &\rightarrow A D \mid B C \mid C S \\ E &\rightarrow a A \mid b B \mid \varepsilon \end{aligned}$$

- Cálculo dos símbolos produtivos

- 1 Inicialmente $N_p \leftarrow \emptyset$
- 2 $A \rightarrow d \wedge d \in T^* \implies N_p \leftarrow N_p \cup \{A\}$
- 3 $B \rightarrow b \wedge b \in T^* \implies N_p \leftarrow N_p \cup \{B\}$
- 4 $E \rightarrow \varepsilon \wedge \varepsilon \in T^* \implies N_p \leftarrow N_p \cup \{E\}$
- 5 $S \rightarrow a A b \wedge a, A, b \in (T \cup N_p)^* \implies N_p \leftarrow N_p \cup \{S\}$
- 6 Nada mais se consegue acrescentar a $N_p \implies C$ e D são improdutivos

Gramáticas limpas

Exemplo #1, cont.

- Gramática após a remoção dos símbolos improdutivos

$$S \rightarrow a A b \mid b B$$

$$A \rightarrow b B \mid d$$

$$B \rightarrow b$$

$$E \rightarrow a A \mid b B \mid \varepsilon$$

- Cálculo dos símbolos não terminais acessíveis sobre a nova gramática

1 S é acessível, porque é o inicial

2 sendo S acessível, de $S \rightarrow a A b$, tem-se que A é acessível

3 sendo S acessível, de $S \rightarrow b B$, tem-se que B é acessível

4 de A só se chega a B , que já foi marcado como acessível

5 de B não se chega a nenhum não terminal

6 Logo E não é acessível, pelo que a gramática limpa é

$$S \rightarrow a A b \mid b B$$

$$A \rightarrow b B \mid d$$

$$B \rightarrow b$$



Compiladores

Análise sintática descendente

Artur Pereira <artur@ua.pt>,
Miguel Oliveira e Silva <mos@ua.pt>

DETI, Universidade de Aveiro

Ano letivo de 2022-2023

Sumário

- ① Análise sintática descendente
- ② Analisador (*parser*) recursivo-descendente preditivo
- ③ Fatorização à esquerda
- ④ Remoção de recursividade à esquerda
- ⑤ Conjuntos *first*, *follow* e *predict*
- ⑥ Tabela de decisão de um reconhecedor descendente LL(1)

Análise sintática

Introdução

- Dada uma gramática $G = (T, N, P, S)$ e uma palavra $u \in T^*$, o papel da análise sintática é:
 - descobrir uma derivação que a partir de S produza u
 - gerar uma árvore de derivação (*parse tree*) que transforme S (a raiz) em u (as folhas)
- Se nenhuma derivação/árvore existir, então $u \notin L(G)$
- A análise sintática pode ser **descendente** ou **ascendente**
- Na análise sintática descendente:
 - a derivação pretendida é **à esquerda**
 - a árvore é gerada **a partir da raiz**, descendo para as folhas
- Na análise sintática ascendente:
 - a derivação pretendida é **à direita**
 - a árvore é gerada **a partir das folhas**, subindo para a raiz
- O objetivo final é a transformação da gramática num programa (reconhecedor sintático) que produza tais derivações/árvores
 - Para as gramáticas independentes do contexto, estes reconhecedores são os **autómatos de pilha**

Análise sintática descendente

Exemplo

- Considere a gramática

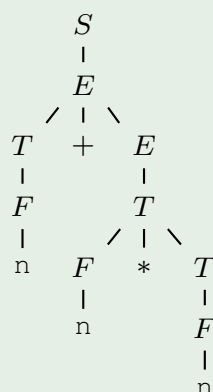
$$S \rightarrow E$$

$$E \rightarrow T + E \mid T$$

$$T \rightarrow F * T \mid F$$

$$F \rightarrow n \mid (E)$$

- Desenhe-se a árvore de derivação da palavra $n+n*n$ a partir de S



Análise sintática descendente

Conceitos

- Existem diferentes abordagens à análise sintática descendente
- Análise sintática descendente **recursiva**
 - Os símbolos não terminais transformam-se em funções recursivas
 - Abordagem genérica
 - Pode requerer um algoritmo de *backtracking* (tentativa e erro) para descobrir a produção a aplicar a cada momento
- Análise sintática descendente **preditiva**
 - Abordagem recursiva ou através de uma tabela de decisão
 - No caso da tabela, os símbolos não terminais transformam-se no alfabeto da pilha
 - Não requer *backtracking*
 - A produção a aplicar a cada momento é escolhida com base no primeiro(s) *token(s)* da entrada que ainda não foram consumidos (**lookahead**)
 - São designados $LL(k)$
 - k é o número (máximo) de *tokens* usados na tomada de decisão
 - O primeiro L significa que a entrada é analisada da esquerda para a direita
 - O segundo L significa que se faz uma derivação à esquerda
 - Assenta em 3 elementos de análise
 - os conjuntos **first**, **follow** e **predict**

Analizador (*parser*) recursivo-descendente preditivo

Exemplo #1

- Sobre o alfabeto $\{a, b\}$, considere linguagem
$$L = \{a^n b^n : n \geq 0\}$$
descrita pela gramática
$$S \rightarrow a S b \mid \epsilon$$
- Construa-se um programa com *lookahead* de 1, em que o símbolo não terminal S seja uma função recursiva, que reconheça a linguagem L .

```
int lookahead;

int main()
{
    while (1)
    {
        printf(">> ");
        adv();
        S();
        eat('\n');
        printf("\n");
    }
    return 0;
}

void S(void)
{
    switch(lookahead)
    {
        case 'a':
            eat('a'); S(); eat('b');
            break;
        default:
            epsilon();
            break;
    }
}

void adv()
{
    lookahead = getchar();
}

void eat(int c)
{
    if (lookahead != c) error();
    adv();
}

void epsilon()
{
}

void error()
{
    printf("Unexpected symbol\n");
    exit(1);
}
```

Analizador (*parser*) recursivo-descendente

Análise do exemplo #1

No programa anterior:

- `lookahead` é uma variável global que representa o próximo símbolo à entrada
- `adv()` é uma função que avança na entrada, colocando em `lookahead` o próximo símbolo
- `eat(c)` é uma função que verifica se no `lookahead` está o símbolo `c`, gerando erro se não estiver, e avança para o próximo
- Há duas produções da gramática com cabeça S , sendo a decisão central do programa a escolha de qual usar face ao valor do `lookahead`.
 - deve escolher-se $S \rightarrow a S b$ se o `lookahead` for `a`
 - e $S \rightarrow \varepsilon$ se o `lookahead` for `$` ou `b`

No programa anterior, o símbolo `$`, marcador de fim de entrada, corresponde ao `\n`

- Uma palavra é aceite pelo programa se e só se
`S(); eat($)`
não der erro.

Analizador (*parser*) recursivo-descendente preditivo

Exemplo #2

- Sobre o alfabeto $\{a, b\}$, considere linguagem

$$L = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$$

descrita pela gramática

$$S \rightarrow \varepsilon \mid a B S \mid b A S$$

$$A \rightarrow a \mid b A A$$

$$B \rightarrow a B B \mid b$$

- Construa um programa em que os símbolos não terminais sejam funções recursivas que reconheça a linguagem L .
- O programa terá 3 funções recursivas, A , B e S , semelhantes à função S do exemplo anterior
- Em A , deve escolher-se $A \rightarrow a$ se `lookahead` for `a` e $A \rightarrow b A A$ se for `b`
- Em B , deve escolher-se $B \rightarrow b$ se `lookahead` for `b` e $B \rightarrow a B B$ se for `a`
- Em S , deve escolher-se $S \rightarrow a B S$ se `lookahead` for `a`, $S \rightarrow b A S$ se for `b` e $S \rightarrow \varepsilon$ se for `$` (este último, mais tarde saber-se-á porquê)

Analizador (*parser*) recursivo-descendente preditivo

Exemplo #2a

- Sobre o alfabeto $\{a, b\}$, considere linguagem

$$L = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$$

descrita pela gramática

$$S \rightarrow \varepsilon \mid a B \mid b A$$

$$A \rightarrow a S \mid b A A$$

$$B \rightarrow a B B \mid b S$$

- Construa um programa em que os símbolos não terminais sejam funções recursivas que reconheça a linguagem L .

- O programa terá 3 funções recursivas, A , B e S , semelhantes à função S do exemplo anterior, exceto no critério de escolha da produção $S \rightarrow \varepsilon$
- Escolher $S \rightarrow \varepsilon$ quando lookahead for $\$$ pode não resolver
- Por exemplo, com o lookahead igual a a , há situações em que se tem de escolher $S \rightarrow a B$ e outras $S \rightarrow \varepsilon$
- É o que acontece com a entrada $bbaa$
 $S \Rightarrow b A \Rightarrow bb A A \Rightarrow bba S A \Rightarrow \dots$
momento em que o S tem de ser expandido para ε e o lookahead é a

Analizador (*parser*) recursivo-descendente preditivo

Exemplo #2b

- Sobre o alfabeto $\{a, b\}$, considere linguagem

$$L = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$$

descrita pela gramática

$$\begin{array}{l} S \rightarrow \varepsilon \\ \quad \mid a S b S \\ \quad \mid b S a S \end{array}$$

- Construa um programa em que os símbolos não terminais sejam funções recursivas que reconheça a linguagem L
- Tal como no caso anterior, escolher $S \rightarrow \varepsilon$ quando lookahead for $\$$ pode não resolver

Analizador (*parser*) recursivo-descendente preditivo

Exemplo #3

- Sobre o alfabeto $\{a, b\}$, considere linguagem
$$L = \{a^n b^n : n \geq 1\}$$
descrita pela gramática
$$\begin{array}{l} S \rightarrow a S b \\ \quad | a b \end{array}$$
- Construa um programa em que o símbolo não terminal S seja uma função recursiva que reconheça a linguagem L .
- Como escolher entre as duas produções se ambas começam pelo mesmo símbolo?
- Há duas abordagens:
 - Pôr em evidência o a à esquerda, transformando a gramática para
$$\begin{array}{l} S \rightarrow a X \\ X \rightarrow S b \mid b \end{array}$$
 - Aumentar o número de símbolos de *lookahead* para 2
 - se for aa , escolhe-se $S \rightarrow a S b$
 - se for ab , escolhe-se $S \rightarrow a b$

Analizador (*parser*) recursivo-descendente preditivo

Exemplo #4

- Sobre o alfabeto $\{a, b\}$, considere linguagem
$$L = \{(ab)^n : n \geq 1\}$$
descrita pela gramática
$$\begin{array}{l} S \rightarrow S a b \\ \quad | a b \end{array}$$
- Construa um programa em que o símbolo não terminal S seja uma função recursiva que reconheça a linguagem L .
- Escolher a primeira produção cria um ciclo infinito, por causa da recursividade à esquerda
 - O ANTLR consegue lidar com este tipo (simples) de recursividade à esquerda, mas falha com outros tipos
 - Mas, em geral os reconhecedores descendentes não lidam bem com recursividade à esquerda
- Solução geral: eliminar a recursividade à esquerda

Questões a resolver

Q Que fazer quando há prefixos comuns?

R Pô-los em evidência (fatorização à esquerda)

Q Como lidar com a recursividade à esquerda?

R Transformá-la em recursividade à direita

Q Para que valores do *lookahead* usar uma regra $A \rightarrow \alpha$?

R **predict** ($A \rightarrow \alpha$)

Fatorização à esquerda

Exemplo de ilustração

- Sobre o alfabeto $\{a, b\}$, considere linguagem

$$L = \{a^n b^n : n \geq 1\}$$

descrita pela gramática

$$\begin{array}{l} S \rightarrow a S b \\ \quad | \quad a b \end{array}$$

- Obtenha uma gramática equivalente, pondo em evidência o a
- Relaxando a definição *standard* de gramática que se tem usado, pode obter-se

$$S \rightarrow a (S b \mid b)$$

- e criando um símbolo não terminal que represente o que está entre parêntesis, obtém-se a gramática

$$\begin{array}{l} S \rightarrow a X \\ X \rightarrow b \\ \quad | \quad S b \end{array}$$

- Esta gramática permite a construção de um programa preditivo com *lookahead* de

Eliminação de recursividade à esquerda

Recursividade direta simples

- A gramática seguinte, onde α e β representam sequências de símbolos terminais e/ou não terminais, com β não começando por A , representa genericamente a recursividade direta simples à esquerda

$$\begin{array}{l} A \rightarrow A \alpha \\ | \quad \beta \end{array}$$

- Aplicando a primeira produção n vezes e a seguir a segunda, obtém-se

$$A \Rightarrow A \alpha \Rightarrow A \alpha \alpha \Rightarrow A \alpha \cdots \alpha \alpha \Rightarrow \beta \underbrace{\alpha \cdots \alpha \alpha}_{n \geq 0}$$

- Ou seja

$$A = \beta \alpha^n \quad n \geq 0$$

- Que corresponde ao β seguido do fecho de α , podendo ser representada pela gramática

$$\begin{array}{l} A \rightarrow \beta X \\ X \rightarrow \varepsilon \\ | \quad \alpha X \end{array}$$

- Em ANTLR seria possível fazer-se $A \rightarrow \beta (\alpha) ^*$

Eliminação de recursividade à esquerda

Exemplo com recursividade direta simples

- Para a gramática

$$\begin{array}{l} S \rightarrow S a b \\ | \quad c b a \end{array}$$

obtenha-se uma gramática equivalente sem recursividade à esquerda

- Aplicando a estratégia anterior, tem-se

$$S \Rightarrow S \underbrace{a b}_{\alpha} \Rightarrow S \underbrace{a b}_{\alpha} \cdots \underbrace{a b}_{\alpha} \Rightarrow \underbrace{c b a}_{\beta} \underbrace{a b}_{\alpha} \cdots \underbrace{a b}_{\alpha}$$

- Ou seja

$$S = \underbrace{(c b a)}_{\beta} \underbrace{(a b)}_{\alpha}^n, \quad n \geq 0$$

- Que corresponde à gramática

$$\begin{array}{l} S \rightarrow c b a X \\ X \rightarrow \varepsilon \\ | \quad a b X \end{array}$$

Eliminação de recursividade à esquerda

Recursividade direta múltipla

- A gramática seguinte, onde α_i e β_j representam seqüências de símbolos terminais e/ou não terminais, com os β_j não começando por A , representa genericamente a recursividade direta múltipla à esquerda

$$A \rightarrow A \alpha_1 \mid A \alpha_2 \mid \cdots \mid A \alpha_n \\ \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$$

- Aplicando a estratégia anterior, tem-se

$$A = (\beta_1 \mid \beta_2 \mid \cdots \mid \beta_m)(\alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n)^k \quad k \geq 0$$

- Que corresponde à gramática

$$A \rightarrow \beta_1 X \mid \beta_2 X \mid \cdots \mid \beta_m X \\ X \rightarrow \varepsilon \\ \mid \alpha_1 X \mid \alpha_2 X \mid \cdots \mid \alpha_n X$$

- Em ANTLR seria possível fazer-se $(\beta_1 \mid \beta_2 \mid \cdots \mid \beta_m)(\alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n)^*$

Eliminação de recursividade à esquerda

Exemplo com recursividade direta múltipla

- Obtenha-se uma gramática equivalente à seguinte sem recursividade à esquerda

$$S \rightarrow S a b \mid S c \\ \mid b b \mid c c$$

- As palavras da linguagem são da forma

$$S = (b b \mid c c)(a b \mid c)^k, \quad k \geq 0$$

- Obtendo-se a gramática

$$S \rightarrow b b X \mid c c X \\ X \rightarrow \varepsilon \\ \mid a b X \mid c X$$

Eliminação de recursividade à esquerda

Ilustração de recursividade indireta

- Aplique-se o procedimento anterior à gramática seguinte, assumindo que a recursividade à esquerda está no A

$$S \rightarrow A a \mid b$$

$$A \rightarrow A c \mid S d \mid \varepsilon$$

- O resultado seria

$$S \rightarrow A a \mid b$$

$$A \rightarrow S d X \mid X$$

$$X \rightarrow \varepsilon \mid c X$$

- A recursividade não foi eliminada

$$S \Rightarrow A a \Rightarrow S d X a \Rightarrow A a d X a$$

- Porque a recursividade existe de forma indireta
- Como resolver a recursividade à esquerda (direta e indireta)?

-
- S pode transformar-se em algo começado por A que, por sua vez, se pode transformar em algo que começa por S

Eliminação de recursividade à esquerda

Recursividade indireta

- Considere a gramática (genérica) seguinte, em que alguns dos $\alpha_i, \beta_i, \dots, \Omega_i$ podem começar por A_j , com $i, j = 1, 2, \dots, n$

$$A_1 \rightarrow \alpha_1 \mid \beta_1 \mid \dots \mid \Omega_1$$

$$A_2 \rightarrow \alpha_2 \mid \beta_2 \mid \dots \mid \Omega_2$$

...

$$A_n \rightarrow \alpha_n \mid \beta_n \mid \dots \mid \Omega_n$$

- Algoritmo:

- Define-se uma ordem para os símbolos não terminais, por exemplo

$$A_1, A_2, \dots, A_n$$

- Para cada A_i :

- fazem-se transformações de equivalência de modo a garantir que nenhuma produção com cabeça A_i se expande em algo começado por A_j , com $j < i$
- elimina-se a recursividade à esquerda direta que as produções começadas por A_i possam ter

Eliminação de recursividade à esquerda

Exemplo com recursividade indireta

- Aplique-se este procedimento à gramática seguinte, estabelecendo-se a ordem S, A

$$S \rightarrow A a \mid b$$

$$A \rightarrow A c \mid S d \mid \varepsilon$$

- As produções começadas por S satisfazem a condição, pelo que não é necessária qualquer transformação
- A produção $A \rightarrow S d$ viola a regra definida, pelo que, nela, S é substituído por $(A a \mid b)$, obtendo-se

$$S \rightarrow A a \mid b$$

$$A \rightarrow A c \mid A a d \mid b d \mid \varepsilon$$

- Elimina-se a recursividade à esquerda direta das produções começadas por A , obtendo-se

$$S \rightarrow A a \mid b$$

$$A \rightarrow b d X \mid X$$

$$X \rightarrow \varepsilon \mid c X \mid a d X$$

Conjuntos **predict**, **first** e **follow**

Definições

- Considere uma gramática $G = (T, N, P, S)$ e uma produção $(A \rightarrow \alpha) \in P$
- O conjunto **predict** $(A \rightarrow \alpha)$ representa os valores de *lookahead* para os quais A deve ser expandido para α . Define-se por:

$$\mathbf{predict}(A \rightarrow \alpha) =$$

$$\begin{cases} \mathbf{first}(\alpha) & \varepsilon \notin \mathbf{first}(\alpha) \\ (\mathbf{first}(\alpha) - \{\varepsilon\}) \cup \mathbf{follow}(A) & \varepsilon \in \mathbf{first}(\alpha) \end{cases}$$

- O conjunto **first** (α) representa as letras (símbolos terminais) pelas quais as palavras geradas por α podem começar mais ε se for possível transformar todo o α em ε . Define-se por:

$$\mathbf{first}(\alpha) = \{t \in T : \alpha \Rightarrow^* t\omega \wedge \omega \in T^*\} \cup \{\varepsilon : \alpha \Rightarrow^* \varepsilon\}$$

- O conjunto **follow** (A) representa as letras (símbolos terminais) que podem aparecer imediatamente à frente de A numa derivação. Define-se por:

$$\mathbf{follow}(A) = \{t \in T_{\$} : S\$ \Rightarrow^* \gamma A t\omega\} \quad \text{com} \quad T_{\$} = \{T \cup \$\}$$

Conjunto **first**

Algoritmo de cálculo

- Trata-se de um algoritmo recursivo

```
first( $\alpha$ ) {  
  if ( $\alpha = \varepsilon$ ) then  
    return  $\{\varepsilon\}$   
   $h = \mathbf{head}(\alpha)$       # com  $|h| = 1$   
   $\omega = \mathbf{tail}(\alpha)$      # tal que  $\alpha = h\omega$   
  if ( $h \in T$ ) then  
    return  $\{h\}$   
  else  
    return  $\bigcup_{(h \rightarrow \beta_i) \in P} \mathbf{first}(\beta_i \omega)$     # concatenação de  $\beta_i$  com  $\omega$   
}
```

- Note que no último **return** o argumento do **first** é $\beta_i \omega$, concatenação dos β_i (que vêm dos corpos das produções começadas por h) com o ω (**tail** do α)
- Este algoritmo pode não convergir se a gramática tiver recursividade à esquerda

Conjunto **first**

Exemplo #1

- Considere a gramática

$$S \rightarrow a S \mid B C$$

$$B \rightarrow \varepsilon \mid b S$$

$$C \rightarrow c \mid c S$$

- Determine o conjunto **first** ($a S$)

- Porque $a S$ começa pelo símbolo terminal a

$$\mathbf{first}(a S) = \{a\}.$$

- Determine o conjunto **first** ($B C$)

- Porque $B C$ começa pelo símbolo não terminal B

$$\mathbf{first}(B C) = \mathbf{first}(C) \cup \mathbf{first}(b S C)$$

- Porque C começa pelo símbolo não terminal C

$$\mathbf{first}(C) = \mathbf{first}(c) \cup \mathbf{first}(c S)$$

$$\therefore \mathbf{first}(B C) = \mathbf{first}(c) \cup \mathbf{first}(c S) \cup \mathbf{first}(b S C) = \{b, c\}$$

- Note que, embora B se possa transformar em ε , $\varepsilon \notin \mathbf{first}(B C)$
- Por essa razão, $\mathbf{first}(B C) \neq \mathbf{first}(B) \cup \mathbf{first}(C)$

Conjunto **first**

Exemplo #2

- Considere a gramática

$$S \rightarrow a S \mid B C$$

$$B \rightarrow \varepsilon \mid b S$$

$$C \rightarrow \varepsilon \mid c S$$

- Determine o conjunto **first** (BC)

- Porque BC começa pelo símbolo não terminal B

$$\mathbf{first}(BC) = \mathbf{first}(C) \cup \mathbf{first}(bSC)$$

- Porque C começa pelo símbolo não terminal C

$$\mathbf{first}(C) = \mathbf{first}(\varepsilon) \cup \mathbf{first}(cS)$$

$$\mathbf{first}(BC) = \mathbf{first}(\varepsilon) \cup \mathbf{first}(cS) \cup \mathbf{first}(bSC)$$

$$= \{\varepsilon, b, c\}$$

- Note que a gramática não é a mesma
- Note que $\varepsilon \in \mathbf{first}(BC)$ apenas porque todo o BC se pode transformar em ε

Conjunto **follow**

Algoritmo de cálculo

- Os conjuntos **follow** podem ser calculados através de um algoritmo iterativo envolvendo todos os símbolos não terminais
- Aplicam-se as seguintes regras:

① $\$ \in \mathbf{follow}(S)$

② **if** ($A \rightarrow \alpha B \in P$) **then**
 $\mathbf{follow}(B) \supseteq \mathbf{follow}(A)$

③ **if** ($A \rightarrow \alpha B \beta \in P \wedge (\varepsilon \notin \mathbf{first}(\beta))$) **then**
 $\mathbf{follow}(B) \supseteq \mathbf{first}(\beta)$

④ **if** ($A \rightarrow \alpha B \beta \in P \wedge (\varepsilon \in \mathbf{first}(\beta))$) **then**
 $\mathbf{follow}(B) \supseteq ((\mathbf{first}(\beta) - \{\varepsilon\}) \cup \mathbf{follow}(A))$

- Partindo de conjuntos vazios, aplicam-se sucessivamente estas regras até que nada seja acrescentado

- Note que \supseteq significa **contém** e não está contido

Conjunto follow

Exemplo #1

- Considere a gramática

$$S \rightarrow a S \mid B C$$

$$B \rightarrow \varepsilon \mid b S$$

$$C \rightarrow c \mid c S$$

- Determine o conjunto **follow**(B)
 - Procuram-se ocorrências de B no lado direito das produções. Há uma: $B C$
 - A produção $S \rightarrow B C$ encaixa nas regras 3 ou 4, dependendo de o ε pertencer ou não ao **first**(C)
 - **first**(C) = $\{c\}$
 - $\therefore \text{follow}(B) \supseteq \text{first}(C)$ [regra 3]
 - Não havendo mais contribuições, tem-se
 $\text{follow}(B) = \{c\}$

Conjunto follow

Exemplo #2

- Considere a gramática

$$S \rightarrow a S \mid B C$$

$$B \rightarrow \varepsilon \mid b S$$

$$C \rightarrow \varepsilon \mid c S$$

- Determine o conjunto **follow**(B)
 - A produção $S \rightarrow B C$ encaixa nas regras 3 ou 4, dependendo de o ε pertencer ou não ao **first**(C)
 - **first**(C) = $\{\varepsilon, c\}$
 - $\therefore \text{follow}(B) \supseteq ((\text{first}(C) - \{\varepsilon\}) \cup \text{follow}(S))$ [regra 4]
 - Porque S é o símbolo inicial, $\$ \in \text{follow}(S)$ [regra 1]
 - A produção $S \rightarrow a S$ é irrelevante, porque diz que $\text{follow}(S) \supseteq \text{follow}(S)$
 - A produção $B \rightarrow b S$ diz que $\text{follow}(S) \supseteq \text{follow}(B)$
 - A produção $C \rightarrow c S$ diz que $\text{follow}(S) \supseteq \text{follow}(C)$
 - A produção $S \rightarrow B C$ diz que $\text{follow}(C) \supseteq \text{follow}(S)$
 - Pelas contribuições tem-se que
 $\text{follow}(B) = \{c, \$\}$
 - Também se ficou a saber que $\text{follow}(S) = \text{follow}(B) = \text{follow}(C)$

Conjunto follow

Exemplo #3

- Considere a gramática

$$S \rightarrow a S \mid B C$$

$$B \rightarrow b \mid b S$$

$$C \rightarrow \varepsilon \mid S c$$

- Determine o conjunto **follow**(B)
 - A produção $S \rightarrow B C$ encaixa nas regras 3 ou 4, dependendo de ε pertencer ou não ao **first**(C)
 - **first**(C) = $\{\varepsilon, a, b\}$
 - $\therefore \text{follow}(B) \supseteq (\text{first}(C) - \{\varepsilon\}) \cup \text{follow}(S)$
 - Porque S é o símbolo inicial, $\$ \in \text{follow}(S)$
 - A produção $S \rightarrow a S$ é irrelevante, porque diz que $\text{follow}(S) \supseteq \text{follow}(S)$
 - A produção $B \rightarrow b S$ diz que $\text{follow}(S) \supseteq \text{follow}(B)$
 - A produção $C \rightarrow S c$ diz que $\text{follow}(S) \supseteq \{c\}$
 - Pelas contribuições tem-se que
$$\text{follow}(B) = \{a, b, c, \$\}$$
 - Note que o ε **nunca pertence** a um **follow**

Reconhecedor descendente preditivo

Tabela de decisão (*parsing table*)

- Para uma gramática $G = (T, N, P, S)$ e um *lookahead* de 1, o reconhecedor descendente pode basear-se numa tabela de decisão
- Corresponde a uma função $\tau : N \times T_{\$} \rightarrow \wp(P)$, onde $T_{\$} = T \cup \{\$\}$ e $\wp(P)$ representa o conjunto dos subconjuntos de P
- Pode ser representada por uma tabela, onde os elementos de N indexam as linhas, os elementos de $T_{\$}$ indexam as colunas, e as células são subconjuntos de P
- Pode ser obtida (ou a tabela preenchida) usando o seguinte algoritmo:

Algoritmo:

foreach $(n, t) \in (N \times T_{\$})$

$\tau(n, t) = \emptyset$ *# começa-se com as células vazias*

foreach $(A \rightarrow \alpha) \in P$

foreach $t \in \text{predict}(A \rightarrow \alpha)$

add $(A \rightarrow \alpha)$ to $\tau(A, t)$

Tabela de decisão

Exemplo #1

- Considere a gramática

$$S \rightarrow a S b \mid \varepsilon$$

- Preencha a tabela de decisão de um reconhecedor descendente desta linguagem com *lookahead* de 1

$$\mathbf{first}(a S b) = \{a\}$$

$$\therefore \mathbf{predict}(S \rightarrow a S b) = \{a\}$$

$$\mathbf{first}(\varepsilon) = \{\varepsilon\}$$

$$\mathbf{follow}(S) = \{\$, b\}$$

$$\therefore \mathbf{predict}(S \rightarrow \varepsilon) = \{\$, b\}$$

Tabela de decisão

	a	b	\$
S	a S b	ε	ε

- Não havendo células com 2 ou mais produções, a gramática é LL(1)

- Para simplificação, optou-se por pôr nas células apenas o corpo da produção, uma vez que a cabeça é definida pela linha da tabela

Tabela de decisão

Exemplo #2

- Considere a gramática

$$S \rightarrow \varepsilon \mid a B S \mid b A S$$

$$A \rightarrow a \mid b A A$$

$$B \rightarrow a B B \mid b$$

- Preencha a tabela de decisão de um reconhecedor descendente desta linguagem com *lookahead* de 1

$$\mathbf{predict}(S \rightarrow a B S) = \{a\}$$

$$\mathbf{predict}(S \rightarrow b A S) = \{b\}$$

$$\mathbf{predict}(S \rightarrow \varepsilon) = \{\$\}$$

$$\mathbf{predict}(A \rightarrow a) = \{a\}$$

$$\mathbf{predict}(A \rightarrow b A A) = \{b\}$$

$$\mathbf{predict}(B \rightarrow b) = \{b\}$$

$$\mathbf{predict}(B \rightarrow a B B) = \{a\}$$

Tabela de decisão

	a	b	\$
S	a B S	b A S	ε
A	a	b A A	
B	a B B	b	

- As células vazias correspondem a situações de erro

Tabela de decisão

Exemplo #2a

- Considere a gramática

$$S \rightarrow \varepsilon \mid a B \mid b A$$

$$A \rightarrow a S \mid b A A$$

$$B \rightarrow a B B \mid b S$$

- Preencha a tabela de decisão de um reconhecedor descendente desta linguagem com *lookahead* de 1

$$\text{predict}(S \rightarrow a B) = \{a\}$$

$$\text{predict}(S \rightarrow b A) = \{b\}$$

$$\text{predict}(S \rightarrow \varepsilon) = \{a, b, \$\}$$

$$\text{predict}(A \rightarrow a S) = \{a\}$$

$$\text{predict}(A \rightarrow b A A) = \{b\}$$

$$\text{predict}(B \rightarrow b S) = \{b\}$$

$$\text{predict}(B \rightarrow a B B) = \{a\}$$

Tabela de decisão

	a	b	\$
S	a B, ε	b A, ε	ε
A	a S	b A A	
B	a B B	b S	

- As células (S, a) e (S, b) têm duas produções cada, o que torna o reconhecimento inviável para um *lookahead* de 1, pelo que a linguagem não é LL(1)

Tabela de decisão

Exemplo #2b

- Considere a gramática

$$S \rightarrow \varepsilon$$

$$\mid a S b S$$

$$\mid b S a S$$

- Preencha a tabela de decisão de um reconhecedor descendente desta linguagem com *lookahead* de 1

$$\text{predict}(S \rightarrow a S b S) = \{a\}$$

$$\text{predict}(S \rightarrow b S a S) = \{b\}$$

$$\text{predict}(S \rightarrow \varepsilon) = \{a, b, \$\}$$

Tabela de decisão

	a	b	\$
S	a A b S, ε	b S a S, ε	ε

- As células (S, a) e (S, b) têm duas produções cada, o que torna o reconhecimento inviável para um *lookahead* de 1, pelo que a linguagem não é LL(1)

Tabela de decisão

Exemplo #3

- Considere, sobre o alfabeto $\{i, f, v, , ;\}$, a linguagem L_4 descrita pela gramática

$$D \rightarrow T L ;$$
$$T \rightarrow i$$
$$| f$$
$$L \rightarrow v$$
$$| v , L$$

- Obtenha-se uma tabela de decisão de um reconhecedor descendente, com *lookahead* de 1, que reconheça a linguagem L_4 .
 - Pretende-se que, se necessário, se transforme a gramática numa equivalente que seja LL(1)
 - Neste caso, existem produções com prefixos comuns (os conjuntos **predict** não são disjuntos)
- Antes de calcular os conjuntos **predict** é necessário começar por fatorizar à esquerda, por causa das produções com cabeça L

Tabela de decisão

Exemplo #3 (cont.)

$$D \rightarrow T L ;$$
$$T \rightarrow i$$
$$| f$$
$$L \rightarrow v X$$
$$X \rightarrow$$
$$| , L$$
$$\text{predict}(D \rightarrow T L ;) = ?$$
$$\text{first}(T L ;) = ?$$
$$\text{first}(T) = \text{first}(i) \cup \text{first}(f) = \{i\} \cup \{f\}$$
$$\therefore \text{first}(T L ;) = \{i, f\}$$
$$\therefore \text{predict}(D \rightarrow T L ;) = \{i, f\}$$
$$\text{predict}(T \rightarrow i) = ?$$
$$\text{first}(i) = \{i\}$$
$$\therefore \text{predict}(T \rightarrow i) = \{i\}$$
$$\text{predict}(T \rightarrow f) = \{f\}$$
$$\text{predict}(L \rightarrow v X) = ?$$
$$\text{first}(v X) = \{v\}$$
$$\therefore \text{predict}(L \rightarrow v X) = \{v\}$$
$$\text{predict}(X \rightarrow \epsilon) = ?$$
$$\text{first}(\epsilon) = \{\epsilon\}$$
$$\therefore \text{predict}(X \rightarrow \epsilon) = \text{follow}(X)$$
$$\text{follow}(X) = \text{follow}(L) = \{;\}$$
$$\therefore \text{predict}(X \rightarrow \epsilon) = \{;\}$$
$$\text{predict}(X \rightarrow , L) = \{, \}$$

Tabela de decisão

Exemplo #3 (cont.)

$D \rightarrow T L ;$

$T \rightarrow i$

$\mid f$

$L \rightarrow v X$

$X \rightarrow$

\mid , L

predict ($D \rightarrow T L ;$) = $\{i, f\}$

predict ($T \rightarrow i$) = $\{i\}$

predict ($T \rightarrow f$) = $\{f\}$

predict ($L \rightarrow v X$) = $\{v\}$

predict ($X \rightarrow \epsilon$) = $\{;\}$

predict ($X \rightarrow , L$) = $\{, \}$

Tabela de decisão

	i	f	v	,	;	\$
D	$T L ;$	$T L ;$				
T	i	f				
L			$v X$			
X				$, L$	ϵ	

- As células vazias são situações de erro



Compiladores

Análise sintática ascendente

Artur Pereira <artur@ua.pt>,
Miguel Oliveira e Silva <mos@ua.pt>

DETI, Universidade de Aveiro

Ano letivo de 2022-2023

Sumário

- ① Introdução
- ② Conflitos
- ③ Construção de um reconhecedor
- ④ Conjunto de itens
- ⑤ Tabela de decisão de um reconhecedor ascendente

Análise sintática ascendente

Ilustração por um exemplo

- Considere a gramática

$$\begin{aligned} D &\rightarrow T L ; \\ T &\rightarrow i \mid r \\ L &\rightarrow v \mid L , v \end{aligned}$$

que representa uma declaração de variáveis *a la C*

- Como reconhecer a palavra “ $u = i v , v ;$ ” como pertencente à linguagem definida pela gramática dada?
- Se u pertence à linguagem definida pela gramática, então $D \Rightarrow^+ u$
- Gerando uma derivação à direita, tem-se
 $D \Rightarrow T L ; \Rightarrow T L , v ; \Rightarrow T v , v ; \Rightarrow i v , v ;$
- Tente-se agora fazer a derivação no sentido contrário, isto é, indo de u para D

Análise sintática ascendente

Ilustração por um exemplo (cont.)

- Considere a gramática

$$\begin{aligned} D &\rightarrow T L ; \\ T &\rightarrow i \mid r \\ L &\rightarrow v \mid L , v \end{aligned}$$

e reduza-se a palavra “ $u = i v , v ;$ ” ao símbolo inicial D

- $i v , v ;$
 $\Leftarrow T v , v ;$ (por aplicação da produção $T \rightarrow i$)
 $\Leftarrow T L , v ;$ (por aplicação da produção $L \rightarrow v$)
 $\Leftarrow T L ;$ (por aplicação da produção $L \rightarrow L , v$)
 $\Leftarrow D$ (por aplicação da produção $D \rightarrow T L ;$)

- Colocando ao contrário, tem-se

$$D \Rightarrow T L ; \Rightarrow T L , v ; \Rightarrow T v , v ; \Rightarrow i v , v ;$$

que corresponde à derivação à direita da palavra “ $u = i v , v ;$ ”

Análise sintática ascendente

Ilustração por um exemplo (cont.)

- A tabela seguinte mostra como, na prática, se realiza esta (retro)derivação

$$\begin{aligned} D &\rightarrow T L ; \\ T &\rightarrow i \mid r \\ L &\rightarrow v \mid L , v \end{aligned}$$

pilha	entrada	próxima ação
	$i \ v \ , \ v ; \$$	deslocamento
i	$v \ , \ v ; \$$	redução por $T \rightarrow i$
T	$v \ , \ v ; \$$	deslocamento
$T v$	$\ , \ v ; \$$	redução por $L \rightarrow v$
$T L$	$\ , \ v ; \$$	deslocamento
$T L ,$	$v ; \$$	deslocamento
$T L , v$	$; \$$	redução por $L \rightarrow L , v$
$T L$	$; \$$	deslocamento
$T L ;$	$\$$	redução por $D \rightarrow T L ;$
D	$\$$	deslocamento / aceitação
$D \$$		aceitação

- A palavra à entrada foi reduzida ao símbolo inicial pelo que é aceite como pertencendo à linguagem

- A aceitação pode ser feita antes de consumir o $\$$ ou depois

Análise sintática ascendente

Ilustração de um erro sintático

- Veja-se a reação deste procedimento a uma entrada errada, por exemplo a palavra $i \ v \ v ;$.

$$\begin{aligned} D &\rightarrow T L ; \\ T &\rightarrow i \mid r \\ L &\rightarrow v \mid L , v \end{aligned}$$

pilha	entrada	próxima ação
	$i \ v \ v ; \$$	deslocamento
i	$v \ v ; \$$	redução por $T \rightarrow i$
T	$v \ v ; \$$	deslocamento
$T v$	$v ; \$$	redução por $L \rightarrow v$
$T L$	$v ; \$$	deslocamento
$T L v$	$; \$$	rejeição

- Rejeita porque $L v$ não corresponde ao prefixo de uma produção da gramática
- Na realidade, o erro poderia ter sido detetado dois passos antes, aquando da segunda redução, porque $v \notin \text{follow}(L)$
 - v corresponde ao símbolo à entrada
 - L é o símbolo que iria aparecer no topo da pilha se se fizesse a redução por $L \rightarrow v$

Análise sintática ascendente

Ilustração de conflito entre deslocamento e redução

- Considere a gramática

$$\begin{array}{l} S \rightarrow i \ c \ S \\ \quad | \ i \ c \ S \ e \ S \\ \quad | \ a \end{array}$$

e aplique-se o procedimento anterior à palavra `icicaea`

pilha	entrada	próxima ação
	icicaea\$	deslocamento
i	cicaea\$	deslocamento
ic	icaea\$	deslocamento
ici	caea\$	deslocamento
icic	aea\$	deslocamento
icica	ea\$	redução por $S \rightarrow a$
icicS	ea\$	conflito:
		– redução por $S \rightarrow icS$
		– deslocamento para tentar $S \rightarrow icSeS$

- Esta gramática representa uma estrutura típica em linguagens de programação. Qual?

Análise sintática ascendente

Ilustração de conflito entre reduções

- Considere a gramática

$$\begin{array}{l} S \rightarrow A \\ \quad | \ B \\ A \rightarrow c \\ \quad | \ A \ a \\ B \rightarrow c \\ \quad | \ B \ b \end{array}$$

e aplique-se o procedimento anterior à palavra `c`

pilha	entrada	próxima ação
	c\$	deslocamento
c	\$	conflito:
		– redução usando $A \rightarrow c$
		– redução usando $B \rightarrow c$

Análise sintática ascendente

Ilustração de falso conflito

- Considere a gramática

$$\begin{array}{l} S \rightarrow a \\ \quad | \quad < S > \\ \quad | \quad a P \\ \quad | \quad < S > S \\ P \rightarrow < S > \\ \quad | \quad < S > S \end{array}$$

e aplique-se o procedimento de reconhecimento à palavra “a < a > a”

pilha	entrada	próxima ação
	a < a > a \$	deslocamento
a	< a > a \$	falso conflito: <ul style="list-style-type: none"> – redução usando $S \rightarrow a$ – deslocamento para tentar $S \rightarrow a P$

- Deslocamento, porque se se optasse pela redução no topo da pilha ficaria um S e $< \notin \text{follow}(S)$

Análise sintática ascendente

Ilustração de falso conflito (cont.)

- Optando pelo deslocamento e continuando...

pilha	entrada	próxima ação
	a < a > a \$	deslocamento
a	< a > a \$	deslocamento, porque $< \notin \text{follow}(S)$
a <	a > a \$	deslocamento
a < a	> a \$	redução por $S \rightarrow a$
a < S	> a \$	deslocamento
a < S >	a \$	deslocamento, porque $a \notin \text{follow}(P)$
a < S > a	\$	redução por $S \rightarrow a$
a < S > S	\$	redução por $P \rightarrow < S > S$
a P	\$	redução por $S \rightarrow a P$
S	\$	deslocamento
S \$		aceitação

Análise sintática ascendente

Eliminação de conflito

- Pode ser possível alterar uma gramática de modo a eliminar a fonte de conflito
- Considerando que se pretendia optar pelo deslocamento, a gramática da esquerda gera a mesma linguagem que a da direita e está isenta de conflitos.

$$\begin{array}{l} S \rightarrow a \\ \quad | \quad i \ c \ S \\ \quad | \quad i \ c \ S' \ e \ S \\ S' \rightarrow a \\ \quad | \quad i \ c \ S' \ e \ S' \end{array}$$

$$\begin{array}{l} S \rightarrow a \\ \quad | \quad i \ c \ S \\ \quad | \quad i \ c \ S \ e \ S \end{array}$$

Análise sintática ascendente

if..then..else sem conflitos

- Considere a gramática seguinte e processe-se a palavra "icicaea"

$$\begin{array}{l} S \rightarrow a \mid i \ c \ S \mid i \ c \ S' \ e \ S \\ S' \rightarrow a \mid i \ c \ S' \ e \ S' \end{array}$$

pilha	entrada	próxima ação
	icicaea\$	deslocamento
i	cicaea\$	deslocamento
ic	icaea\$	deslocamento
ici	caea\$	deslocamento
icic	aea\$	deslocamento
icica	ea\$	redução por $S' \rightarrow a$ // $e \in \text{follow}(S')$, $e \notin \text{follow}(S)$
icicS'	ea\$	deslocamento
icicS'e	a\$	deslocamento
icicS'ea	\$	redução por $S \rightarrow a$ // $\$ \in \text{follow}(S)$, $\$ \notin \text{follow}(S')$
icicS'eS	\$	redução por $S \rightarrow i \ c \ S' \ e \ S$
icS	\$	redução por $S \rightarrow i \ c \ S$
S	\$	deslocamento e aceitação

Construção de um reconhecedor ascendente

Abordagem

- Como determinar de forma sistemática a ação a realizar (deslocamento, redução, aceitação, rejeição)?

pilha	entrada	próxima ação
	$i \ v \ v ; \$$	deslocamento
i	$v \ v ; \$$	redução por $T \rightarrow i$
T	$v \ v ; \$$	deslocamento
$T \ v$	$v ; \$$	rejeição

- A ação a realizar em cada passo do procedimento de reconhecimento – deslocamento, redução, aceitação ou rejeição – depende da configuração em cada momento
- Uma **configuração** é formada pelo conteúdo da pilha mais a parte da entrada ainda não processada
- A pilha é conhecida – na realidade, é preenchida pelo procedimento de reconhecimento
- Da entrada, em cada momento, apenas se conhece o *lookahead*

Construção de um reconhecedor ascendente

Abordagem (cont.)

pilha	entrada	próxima ação
	$i \ v \ v ; \$$	deslocamento
i	$v \ v ; \$$	redução por $T \rightarrow i$
T	$v \ v ; \$$	deslocamento
$T \ v$	$v ; \$$	rejeição

- Quantos símbolos da pilha usar?
- Poder-se-á usar apenas um?
- Se se quiser e puder construir um reconhecedor que apenas use o símbolo no topo, uma pilha onde se guardam os símbolos terminais e não terminais tem pouco interesse
- Mas pode definir-se um alfabeto adequado para a pilha
- Os símbolos a colocar na pilha devem representar estados no processo de deslocamento/redução/aceitação
- Por exemplo, um dado símbolo pode significar que, na produção " $D \rightarrow T L ;$ ", já se processou algo que corresponde ao " $T L$ ", faltando o " $;$ "

Construção de um reconhecedor ascendente

Itens de uma gramática

- O alfabeto da pilha representa assim o conjunto de possíveis estados nesse processo de reconhecimento
- Cada estado representa um conjunto de itens
- Cada item representa o quanto de uma produção já foi processado e o quanto ainda falta processar
 - Usa-se um ponto (·) ao longo dos símbolos de uma produção para o representar
- A produção $A \rightarrow B_1 B_2 B_3$ produz 4 itens:
$$\begin{aligned}A &\rightarrow \cdot B_1 B_2 B_3 \\A &\rightarrow B_1 \cdot B_2 B_3 \\A &\rightarrow B_1 B_2 \cdot B_3 \\A &\rightarrow B_1 B_2 B_3 \cdot\end{aligned}$$
- A produção $A \rightarrow \varepsilon$ produz um único item:
$$A \rightarrow \cdot$$
- Um item com um ponto (·) à direita representa uma **ação de redução**

Conjunto dos conjuntos de itens

Ilustração com um exemplo

- Considere a gramática
$$\begin{aligned}S &\rightarrow E \\E &\rightarrow a \mid (E)\end{aligned}$$
- Reconhecer a palavra $u = u_1 u_2 \cdots u_n$, significa reduzir $u\$$ a $S\$$, então, o estado inicial no processo de reconhecimento pode ser definido por
$$Z_0 = \{S \rightarrow \cdot E \$\}$$
- O facto de o ponto (·) se encontrar imediatamente à esquerda de um símbolo significa que para se avançar no processo de reconhecimento é preciso obter esse símbolo
 - Se o símbolo é terminal, isso corresponde a uma ação de deslocamento
 - Se o símbolo é não terminal, é preciso dar-se a redução de uma produção que o produza
 - Isso é considerado juntando ao conjunto os itens iniciais das produções cuja cabeça é o símbolo pretendido
$$Z_0 = \{S \rightarrow \cdot E \$\} \cup \{E \rightarrow \cdot a, E \rightarrow \cdot (E)\}$$
- Se aparecerem novos símbolos não terminais imediatamente à direita de um ponto (·), repete-se o processo. Faz-se o **fecho (closure)**

Conjunto dos conjuntos de itens

Ilustração com um exemplo (cont.)

- Evolução de Z_0 :

$$Z_0 = \{ S \rightarrow \cdot E \$ \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

- O estado Z_0 pode evoluir por ocorrência de um E , um a ou um $($, que correspondem aos símbolos que aparecem imediatamente à direita do ponto (\cdot)

$$\delta(Z_0, E) = \{ S \rightarrow E \cdot \$ \} = Z_1 \quad \text{um estado novo}$$

$$\delta(Z_0, a) = \{ E \rightarrow a \cdot \} = Z_2 \quad \text{um estado novo}$$

$$\delta(Z_0, () = \{ E \rightarrow (\cdot E) \} = Z_3 \quad \text{um estado novo}$$

- Z_3 tem de ser estendido pela função de fecho, uma vez que o ponto (\cdot) ficou imediatamente à esquerda de um símbolo não terminal (E)

$$Z_3 = \delta(Z_0, () = \{ E \rightarrow (\cdot E) \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

- Z_2 , apenas possui um item terminal (com o ponto (\cdot) à direita), que representa uma situação passível de redução, neste caso pela produção $E \rightarrow a$

Conjunto dos conjuntos de itens

Ilustração com um exemplo (cont.)

- Evolução de Z_1 :

$$Z_1 = \{ S \rightarrow E \cdot \$ \}$$

- Apenas evolui por ocorrência de um $\$$

$$\delta(Z_1, \$) = \{ S \rightarrow E \$ \cdot \} \implies \text{ACCEPT}$$

que corresponde à situação de aceitação

- Se o símbolo inicial da gramática não aparecer no corpo de qualquer produção (como acontece aqui), Pode-se considerar Z_1 como uma situação de aceitação se o *lookahead* for $\$$

- Evolução de Z_3 :

$$Z_3 = \{ E \rightarrow (\cdot E) \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

- Pode evoluir por ocorrência de um E , um a ou um $($

$$\delta(Z_3, E) = \{ E \rightarrow (E \cdot) \} = Z_4 \quad \text{um estado novo}$$

$$\delta(Z_3, a) = \{ E \rightarrow a \cdot \} = Z_2 \quad \text{um estado repetido}$$

$$\delta(Z_3, () = \{ E \rightarrow (\cdot E) \} = Z_3 \quad \text{um estado repetido}$$

Conjunto dos conjuntos de itens

Ilustração com um exemplo (cont.)

- Evolução de Z_4

$$Z_4 = \{ E \rightarrow (E \cdot) \}$$

- Apenas evolui por ocorrência de $)$

$$\delta(Z_4,) = \{ E \rightarrow (E) \cdot \} = Z_5 \quad \text{um estado novo}$$

- Z_5 apenas possui um item terminal, que representa uma situação passível de redução pela regra $E \rightarrow (E)$

- Pode acontecer que um dado elemento (conjunto de itens) possua itens terminais (associados a reduções) e não terminais

Conjunto dos conjuntos de itens

Ilustração com um exemplo (cont.)

- Pondo tudo junto

$$Z_0 = \{ S \rightarrow \cdot E \$ \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

$$Z_1 = \delta(Z_0, E) = \{ S \rightarrow E \cdot \$ \}$$

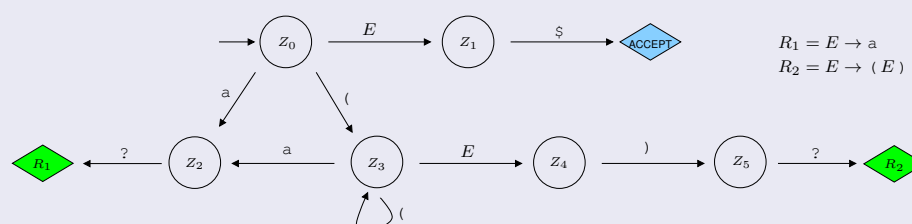
$$Z_2 = \delta(Z_0, a) = \{ E \rightarrow a \cdot \}$$

$$Z_3 = \delta(Z_0, () = \{ E \rightarrow (\cdot E) \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

$$Z_4 = \delta(Z_3, E) = \{ E \rightarrow (E \cdot) \}$$

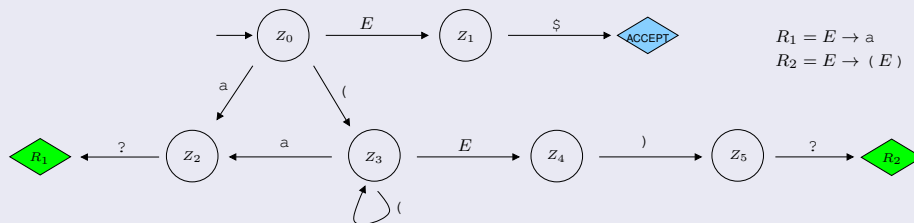
$$Z_5 = \delta(Z_4,) = \{ E \rightarrow (E) \cdot \}$$

- Representando na forma de um autômato, tem-se



Conjunto dos conjuntos de itens

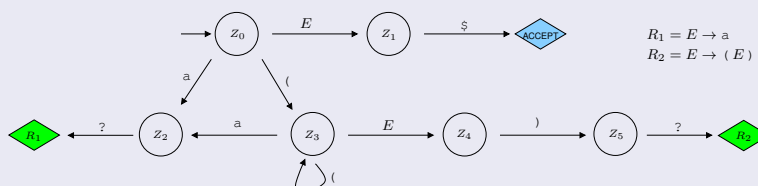
Ilustração com um exemplo (cont.)



- Neste autômato, os estados representam o alfabeto da pilha
- As transições representam operações de *push*
- As transições etiquetadas com símbolos terminais representam adicionalmente ações de deslocamento (*shift*)
- As ações de redução provocam operações de *pop*, em número igual ao número de elementos do corpo da produção
- As transições etiquetadas com símbolos não terminais ocorrem após as ações de redução
- Tudo isto representa o funcionamento de um autômato de pilha que permite fazer o reconhecimento da linguagem

Tabela de decisão de um reconhecedor ascendente

Introdução



- O autômato de pilha pode ser implementado usando uma tabela de decisão
- Esta tabela contém duas matrizes, ACTION e GOTO
 - as linhas de ambas são indexadas pelo alfabeto da pilha (conjunto de conjuntos de itens)
- A matriz ACTION representa ações
 - as colunas são indexadas pelos símbolos terminais da gramática, incluindo o marcador de fim de entrada ($\$$)
 - As células contêm as ações *shift*, *reduce*, *accept* ou *error*
 - No caso de *shift*, também inclui o próximo símbolo a colocar na pilha
- A matriz GOTO representa a operação após uma redução
 - as colunas são indexadas pelos símbolos não terminais da gramática
 - As células indicam que valor colocar na *stack* após uma ação de redução

Tabela de decisão de um reconhecedor ascendente

Exemplo

- Considere-se o conjunto de conjunto de itens obtido anteriormente

$$Z_0 = \{ S \rightarrow \cdot E \$ \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

$$Z_1 = \delta(Z_0, E) = \{ S \rightarrow E \cdot \$ \}$$

$$Z_2 = \delta(Z_0, a) = \{ E \rightarrow a \cdot \}$$

$$Z_3 = \delta(Z_0, () = \{ E \rightarrow (\cdot E) \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

$$Z_4 = \delta(Z_3, E) = \{ E \rightarrow (E \cdot) \}$$

$$Z_5 = \delta(Z_4,) = \{ E \rightarrow (E) \cdot \}$$

- E o correspondente autômato de pilha

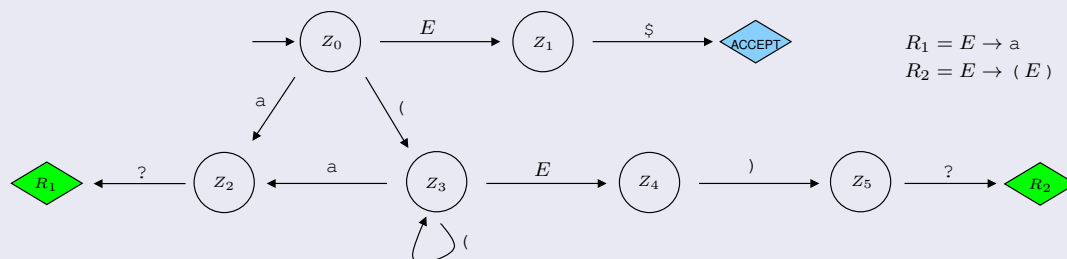
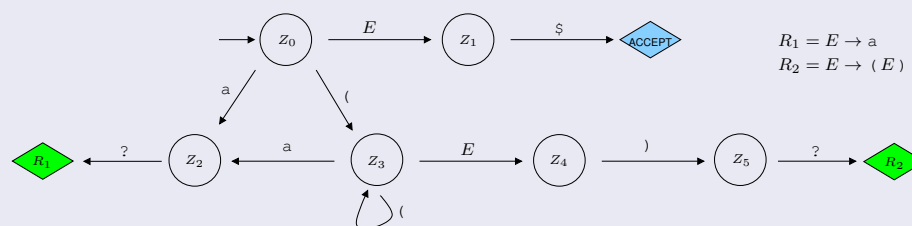


Tabela de decisão de um reconhecedor ascendente

Exemplo

- A este autômato de pilha



- Corresponde a tabela de decisão

	ACTION				GOTO
	a	()	\$	E
Z_0	shift, Z_2	shift, Z_3			Z_1
Z_1				ACCEPT	
Z_2			reduce, $E \rightarrow a$	reduce, $E \rightarrow a$	
Z_3	shift, Z_2	shift, Z_3			Z_4
Z_4			shift, Z_5		
Z_5			reduce, $E \rightarrow (E)$	reduce, $E \rightarrow (E)$	

- Com *lookahead* de 1, as reduções apenas são colocadas nas colunas correspondentes aos **follow**.

Reconhecedor ascendente

Algoritmo de reconhecimento

	ACTION				GOTO
	a	()	\$	E
Z_0	shift, Z_2	shift, Z_3			Z_1
Z_1				ACCEPT	
Z_2			reduce, $E \rightarrow a$	reduce, $E \rightarrow a$	
Z_3	shift, Z_2	shift, Z_3			Z_4
Z_4			shift, Z_5		
Z_5			reduce, $E \rightarrow (E)$	reduce, $E \rightarrow (E)$	

- Com base na tabela de decisão, o procedimento de reconhecimento pode ser implementado pelo seguinte algoritmo

```

push( $Z_0$ )
forever
    if top() ==  $Z_1$  and lookahead == $
        ACCEPT
    action = ACTION[top, lookahead]
    if action is (shift,  $Z_i$ )
        adv(); push( $Z_i$ );
    else if action is (reduce  $A \rightarrow \alpha$ )
        pop  $|\alpha|$  símbolos; push(GOTO[top(),  $A$ ]);
    else
        REJECT

```

- Note que após os *pops* o símbolo no *top* pode mudar e é o novo símbolo que é usado no GOTO

Reconhecedor ascendente

Ilustração com o exemplo anterior

	ACTION				GOTO
	a	()	\$	E
Z_0	shift, Z_2	shift, Z_3			Z_1
Z_1				ACCEPT	
Z_2			reduce, $E \rightarrow a$	reduce, $E \rightarrow a$	
Z_3	shift, Z_2	shift, Z_3			Z_4
Z_4			shift, Z_5		
Z_5			reduce, $E \rightarrow (E)$	reduce, $E \rightarrow (E)$	

- Aplique-se este algoritmo à palavra $((a))$

pilha	entrada	próxima ação
Z_0	$((a))\$$	$\text{ACTION}(Z_0, () = (\text{shift}, Z_3)$
$Z_0 Z_3$	$(a))\$$	$\text{ACTION}(Z_3, () = (\text{shift}, Z_3)$
$Z_0 Z_3 Z_3$	$a))\$$	$\text{ACTION}(Z_3, a) = (\text{shift}, Z_2)$
$Z_0 Z_3 Z_3 Z_2$	$)) \$$	$\text{ACTION}(Z_2,) = (\text{reduce } E \rightarrow a) \quad (1 \text{ pop})$
$Z_0 Z_3 Z_3$	$[E]$	$\text{GOTO}(Z_3, E) = Z_4 \quad (\text{push } Z_4)$
$Z_0 Z_3 Z_3 Z_4$	$)) \$$	$\text{ACTION}(Z_4,) = (\text{shift}, Z_5)$
$Z_0 Z_3 Z_3 Z_4 Z_5$	$) \$$	$\text{ACTION}(Z_5,) = (\text{reduce } E \rightarrow (E)) \quad (3 \text{ pops})$
$Z_0 Z_3 Z_3$	$[E]$	$\text{GOTO}(Z_3, E) = Z_4 \quad (\text{push } Z_4)$
$Z_0 Z_3 Z_4$	$) \$$	$\text{ACTION}(Z_4,) = (\text{shift}, Z_5)$
$Z_0 Z_3 Z_4 Z_5$	$\$$	$\text{ACTION}(Z_5, \$) = (\text{reduce } E \rightarrow (E)) \quad (3 \text{ pops})$
Z_0	$[E]$	$\text{GOTO}(Z_0, E) = Z_1 \quad (\text{push } Z_1)$
$Z_0 Z_1$	$\$$	$\text{ACTION}(Z_1, \$) = \text{ACCEPT}$

Tabela de decisão de um reconhecedor ascendente

Exemplo #3

- Q Determine-se a tabela de decisão para um reconhecedor ascendente com *lookahead* 1 da gramática seguinte

$$S \rightarrow a \mid (S) \mid aP \mid (S)S$$

$$P \rightarrow (S) \mid (S)S$$

- O primeiro passo corresponde a alterar a gramática de modo ao símbolo inicial não aparecer do lado direito

$$S_0 \rightarrow S$$

$$S \rightarrow a \mid (S) \mid aP \mid (S)S$$

$$P \rightarrow (S) \mid (S)S$$

Tabela de decisão de um reconhecedor ascendente

Exemplo #3 (cont.)

- O passo seguinte corresponde a calcular o conjunto de conjunto de itens

$$Z_0 = \{S_0 \rightarrow \cdot S \$\}$$

$$\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot aP, S \rightarrow \cdot (S)S\}$$

fecho

$$\delta(Z_0, S) = \{S_0 \rightarrow S \cdot \$\} = Z_1$$

um estado novo

$$\delta(Z_0, a) = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\}$$

um estado novo

$$\cup \{P \rightarrow \cdot (S), P \rightarrow \cdot (S)S\} = Z_2$$

fecho

$$\delta(Z_0, () = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S)S\}$$

um estado novo

$$\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot aP, S \rightarrow \cdot (S)S\} = Z_3$$

fecho

$$\delta(Z_2, P) = \{S \rightarrow aP \cdot\} = Z_4$$

um estado novo

$$\delta(Z_2, () = \{P \rightarrow (\cdot S), P \rightarrow (\cdot S)S\}$$

um estado novo

$$\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot aP, S \rightarrow \cdot (S)S\} = Z_5$$

fecho

$$\delta(Z_3, S) = \{S \rightarrow (S \cdot), S \rightarrow (S \cdot)S\} = Z_6$$

um estado novo

$$\delta(Z_3, a) = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} = Z_2$$

um estado repetido

$$\delta(Z_3, () = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S)S\} = Z_3$$

um estado repetido

$$S_0 \rightarrow S$$

$$S \rightarrow a \mid (S) \mid aP \mid (S)S$$

$$P \rightarrow (S) \mid (S)S$$

Tabela de decisão de um reconhecedor ascendente

Exemplo #3 (cont.)

- continuando, apenas mostrando os elementos envolvidos em processamento

$Z_2 = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} \cup \dots$	
$Z_3 = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} \cup \dots$	
$Z_5 = \{P \rightarrow (\cdot S), P \rightarrow (\cdot S) S\}$ $\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot a P, S \rightarrow \cdot (S) S\}$	
$Z_6 = \{S \rightarrow (S \cdot), S \rightarrow (S \cdot) S\}$	
$\delta(Z_5, S) = \{P \rightarrow (S \cdot), P \rightarrow (S \cdot) S\} = Z_7$	um estado novo
$\delta(Z_5, a) = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} = Z_2$	um estado repetido
$\delta(Z_5, () = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} = Z_3$	um estado repetido
$\delta(Z_6, () = \{S \rightarrow (S) \cdot, S \rightarrow (S) \cdot S\}$ $\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot a P, S \rightarrow \cdot (S) S\} = Z_8$	um estado novo
$\delta(Z_7, () = \{P \rightarrow (S) \cdot, P \rightarrow (S) \cdot S\}$ $\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot a P, S \rightarrow \cdot (S) S\} = Z_9$	um estado novo

$S_0 \rightarrow S$
 $S \rightarrow a \mid (S) \mid aP \mid (S)S$
 $P \rightarrow (S) \mid (S)S$

Tabela de decisão de um reconhecedor ascendente

Exemplo #3 (cont.)

- continuando...

$Z_2 = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} \cup \dots$	
$Z_3 = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} \cup \dots$	
$Z_8 = \{S \rightarrow (S) \cdot, S \rightarrow (S) \cdot S\}$ $\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot a P, S \rightarrow \cdot (S) S\}$	
$Z_9 = \{P \rightarrow (S) \cdot, P \rightarrow (S) \cdot S\}$ $\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot a P, S \rightarrow \cdot (S) S\}$	
$\delta(Z_8, S) = \{S \rightarrow (S) S \cdot\} = Z_{10}$	um estado novo
$\delta(Z_8, a) = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} = Z_2$	um estado repetido
$\delta(Z_8, () = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} = Z_3$	um estado repetido
$\delta(Z_9, S) = \{P \rightarrow (S) S \cdot\} = Z_{11}$	um estado novo
$\delta(Z_9, a) = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} = Z_2$	um estado repetido
$\delta(Z_9, () = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} = Z_3$	um estado repetido

$S_0 \rightarrow S$
 $S \rightarrow a \mid (S) \mid aP \mid (S)S$
 $P \rightarrow (S) \mid (S)S$

Tabela de decisão de um reconhecedor ascendente

Exemplo #3 (cont.)

- O que resulta em

$Z_0 = \{S_0 \rightarrow \cdot S \$\} \cup \dots$	$\delta(Z_0, S) = Z_1$	$\delta(Z_0, a) = Z_2$	$\delta(Z_0, () = Z_3$
$Z_1 = \{S_0 \rightarrow S \cdot \$\}$			
$Z_2 = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} \cup \dots$	$\delta(Z_2, P) = Z_4$	$\delta(Z_2, () = Z_5$	
$Z_3 = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} \cup \dots$	$\delta(Z_3, S) = Z_6$	$\delta(Z_3, a) = Z_2$	$\delta(Z_3, () = Z_3$
$Z_4 = \{S \rightarrow a P \cdot\}$			
$Z_5 = \{P \rightarrow (\cdot S), P \rightarrow (\cdot S) S\} \cup \dots$	$\delta(Z_5, S) = Z_7$	$\delta(Z_5, a) = Z_2$	$\delta(Z_5, () = Z_3$
$Z_6 = \{S \rightarrow (S \cdot), S \rightarrow (S \cdot) S\}$		$\delta(Z_6, () = Z_8$	
$Z_7 = \{P \rightarrow (S \cdot), P \rightarrow (S \cdot) S\}$		$\delta(Z_7, () = Z_9$	
$Z_8 = \{S \rightarrow (S) \cdot, S \rightarrow (S) \cdot S\} \cup \dots$	$\delta(Z_8, S) = Z_{10}$	$\delta(Z_8, a) = Z_2$	$\delta(Z_8, () = Z_3$
$Z_9 = \{P \rightarrow (S) \cdot, P \rightarrow (S) \cdot S\} \cup \dots$	$\delta(Z_9, S) = Z_{11}$	$\delta(Z_9, a) = Z_2$	$\delta(Z_9, () = Z_3$
$Z_{10} = \{S \rightarrow (S) S \cdot\}$			
$Z_{11} = \{P \rightarrow (S) S \cdot\}$			

$S_0 \rightarrow S$

$S \rightarrow a \mid (S) \mid aP \mid (S)S$

$P \rightarrow (S) \mid (S)S$

Tabela de decisão de um reconhecedor ascendente

Exemplo #3 (cont.)

- O que resulta em

$Z_0 = \{S_0 \rightarrow \cdot S \$\} \cup \dots$	$\delta(Z_0, S) = Z_1$	$\delta(Z_0, a) = Z_2$	$\delta(Z_0, () = Z_3$
$Z_1 = \{S_0 \rightarrow S \cdot \$\}$			
$Z_2 = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} \cup \dots$	$\delta(Z_2, P) = Z_4$	$\delta(Z_2, () = Z_5$	
$Z_3 = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} \cup \dots$	$\delta(Z_3, S) = Z_6$	$\delta(Z_3, a) = Z_2$	$\delta(Z_3, () = Z_3$
$Z_4 = \{S \rightarrow a P \cdot\}$			
$Z_5 = \{P \rightarrow (\cdot S), P \rightarrow (\cdot S) S\} \cup \dots$	$\delta(Z_5, S) = Z_7$	$\delta(Z_5, a) = Z_2$	$\delta(Z_5, () = Z_3$
$Z_6 = \{S \rightarrow (S \cdot), S \rightarrow (S \cdot) S\}$		$\delta(Z_6, () = Z_8$	
$Z_7 = \{P \rightarrow (S \cdot), P \rightarrow (S \cdot) S\}$		$\delta(Z_7, () = Z_9$	
$Z_8 = \{S \rightarrow (S) \cdot, S \rightarrow (S) \cdot S\} \cup \dots$	$\delta(Z_8, S) = Z_{10}$	$\delta(Z_8, a) = Z_2$	$\delta(Z_8, () = Z_3$
$Z_9 = \{P \rightarrow (S) \cdot, P \rightarrow (S) \cdot S\} \cup \dots$	$\delta(Z_9, S) = Z_{11}$	$\delta(Z_9, a) = Z_2$	$\delta(Z_9, () = Z_3$
$Z_{10} = \{S \rightarrow (S) S \cdot\}$			
$Z_{11} = \{P \rightarrow (S) S \cdot\}$			

$R_1 = S \rightarrow a$

$R_2 = S \rightarrow (S)$

$R_3 = S \rightarrow aP$

$R_4 = S \rightarrow (S)S$

$R_5 = P \rightarrow (S)$

$R_6 = P \rightarrow (S)S$

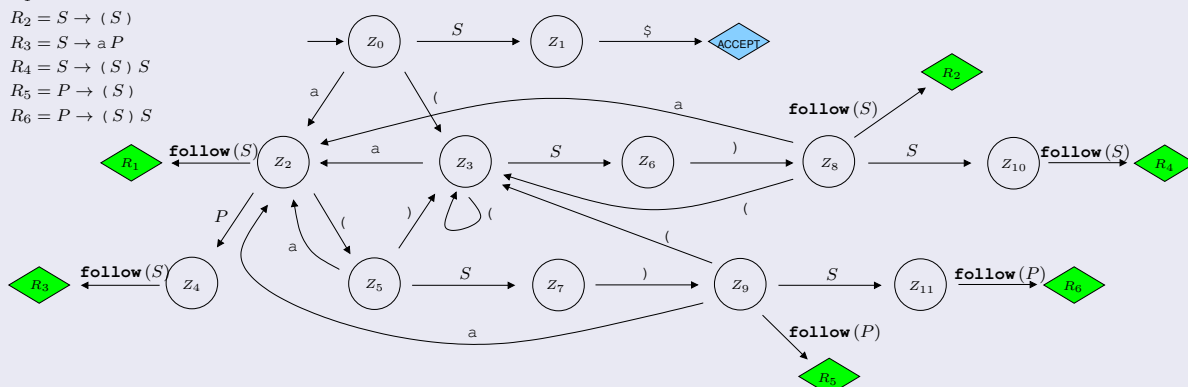


Tabela de decisão de um reconhecedor ascendente

Exemplo #3 (cont.)

- E finalmente a tabela de decisão

	a	()	\$	S	P
Z_0	shift, Z_2	shift, Z_3			Z_1	
Z_1				ACCEPT		
Z_2		shift, Z_5	reduce $S \rightarrow a$	reduce $S \rightarrow a$		Z_4
Z_3	shift, Z_2	shift, Z_3			Z_6	
Z_4			reduce $S \rightarrow a P$	reduce $S \rightarrow a P$		
Z_5	shift, Z_2	shift, Z_3			Z_7	
Z_6			shift, Z_8			
Z_7			shift, Z_9			
Z_8	shift, Z_2	shift, Z_3	reduce $S \rightarrow (S)$	reduce $S \rightarrow (S)$	Z_{10}	
Z_9	shift, Z_2	shift, Z_3	reduce $P \rightarrow (S)$	reduce $P \rightarrow (S)$	Z_{11}	
Z_{10}			reduce $S \rightarrow (S) S$	reduce $S \rightarrow (S) S$		
Z_{11}			reduce $P \rightarrow (S) S$	reduce $P \rightarrow (S) S$		

Tabela de decisão de um reconhecedor ascendente

Exercício

- Q Determine-se a tabela de decisão para um reconhecedor ascendente com *lookahead* 1 da gramática seguinte

$$S \rightarrow \varepsilon \mid S B a \mid S A b$$

$$A \rightarrow a \mid A A b$$

$$B \rightarrow B B a \mid b$$



Compiladores

Gramáticas de atributos

Artur Pereira <artur@ua.pt>,
Miguel Oliveira e Silva <mos@ua.pt>

DETI, Universidade de Aveiro

Ano letivo de 2022-2023

Sumário

- 1 Conteúdo semântico
- 2 Gramática de atributos
- 3 Avaliação dirigida pela sintaxe

Conteúdo semântico

Ilustração com uma expressão aritmética

- Considere a gramática seguinte, onde `num` é um *token* que representa

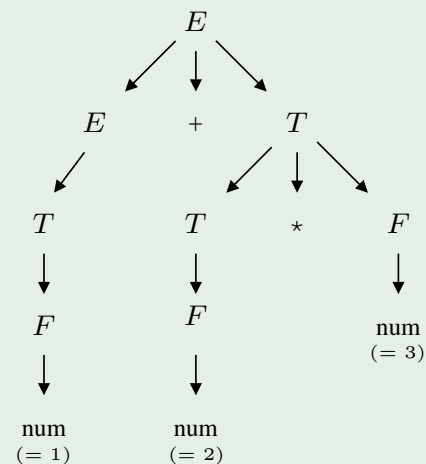
$$E \rightarrow E + T \mid T$$

$$\text{um número} \quad T \rightarrow T * F \mid F$$

$$F \rightarrow \text{num} \mid (E)$$

- Desenhe-se a árvore de derivação da palavra "`1+2*3`"
- Como dar significado a esta árvore?

- 1 Associando a cada símbolo um atributo que armazene o valor que a sub-árvore de que é raiz representa
- 2 Relacionando os atributos associados aos símbolos de cada produção através de regras de cálculo



Conteúdo semântico

Ilustração com uma expressão aritmética

- Considere a gramática seguinte, onde `num` é um *token* que representa

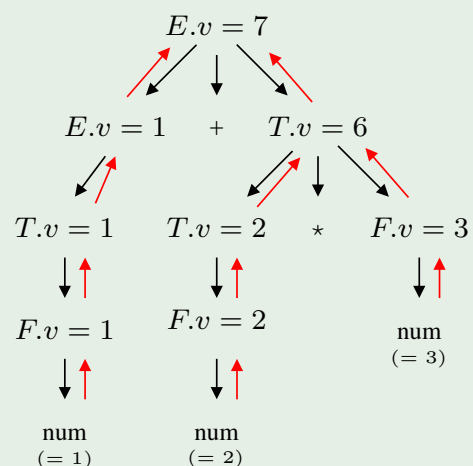
$$E \rightarrow E + T \mid T$$

$$\text{um número} \quad T \rightarrow T * F \mid F$$

$$F \rightarrow \text{num} \mid (E)$$

- Desenhe-se a árvore de derivação da palavra "`1+2*3`"
- Como dar significado a esta árvore?

- 1 Associando a cada símbolo um atributo que armazene o valor que a sub-árvore de que é raiz representa
- 2 Relacionando os atributos associados aos símbolos de cada produção através de regras de cálculo



- As setas vermelhas representam dependência entre atributos
 - o sentido indica qual influencia qual

Conteúdo semântico

Ilustração com uma declaração de variáveis

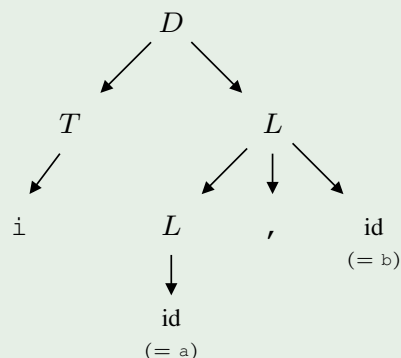
- Considere a gramática seguinte, onde *id* é um *token* que representa o nome de uma variável

$$D \rightarrow T L$$

$$T \rightarrow i \mid f$$

$$L \rightarrow id \mid L , id$$

- desenhe-se a árvore de derivação da palavra *i a, b*
- Associe-se
 - a *T* e *L* um atributo *t* que armazene o tipo



Conteúdo semântico

Ilustração com uma declaração de variáveis

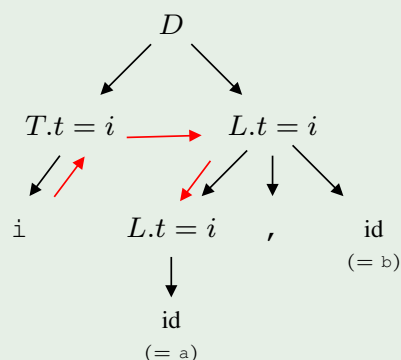
- Considere a gramática seguinte, onde *id* é um *token* que representa o nome de uma variável

$$D \rightarrow T L$$

$$T \rightarrow i \mid f$$

$$L \rightarrow id \mid L , id$$

- desenhe-se a árvore de derivação da palavra *i a, b*
- Associe-se
 - a *T* e *L* um atributo *t* que armazene o tipo



- As setas vermelhas representam dependência entre atributos
 - o sentido indica qual influencia qual

Gramática de atributos

Atributos, regras semânticas e definição semântica

- A análise sintática *per se* não atribui um significado às produções de uma gramática
 - É esse o papel da *gramática de atributos*
 - Isso é feito através de **atributos** e de **regras semânticas**
- Os atributos estão associados aos símbolos da gramática (terminais ou não terminais)
 - Cada símbolo terminal ou não terminal pode ter associado um conjunto de zero ou mais atributos
 - Um atributo pode ser uma palavra, um número, um tipo, uma posição de memória, ...
- As regras semânticas estão associadas às produções da gramática
 - Determinam os valores de atributos de símbolos não terminais em função de outros atributos
 - Podem ter efeitos laterais (alteração de uma estrutura de dados, ...)
- Uma **definição semântica** é composta por
 - uma gramática independente de contexto
 - um conjunto de atributos associados aos seus símbolos
 - um conjunto de regras semânticas associadas às suas produções
- Usar-se-á com o mesmo significado o termo **gramática de atributos**

Gramática de atributos

Regras semânticas

Seja $G = (T, N, S, P)$ uma gramática independente do contexto

- A cada produção $A \rightarrow B_1 B_2 \cdots B_n \in P$, com $B_i \in (T \cup N)^*$, podem associar-se regras semânticas para o cálculo dos valores dos atributos de símbolos não terminais

$$b = f(c_1, c_2, \cdots, c_n)$$

onde

- b é um atributo do símbolo A ou de um dos símbolos não terminais presentes em $B_1 B_2 \cdots B_n$
 - c_1, c_2, \cdots, c_n são atributos dos símbolos que ocorrem na produção
- Podem ainda associar-se regras semânticas com efeitos colaterais

$$g(c_1, c_2, \cdots, c_n)$$

- Embora este caso possa considerar-se o anterior atuando sobre um atributo fictício

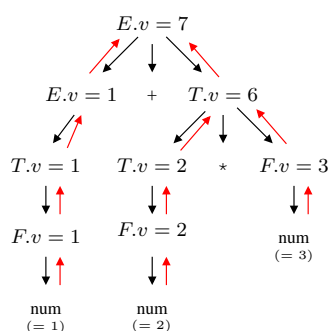
Gramática de atributos

Tipos de atributos

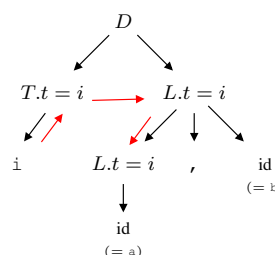
- Os atributos podem ser classificados como **sintetizados** ou **herdados**
- Considere-se uma produção $A \rightarrow B_1 B_2 \cdots B_n \in P$, com $B_i \in (T \cup N)^*$, e uma função de cálculo de um atributo associada a essa produção

$$b = f(c_1, c_2, \dots, c_n)$$

- O atributo b diz-se **sintetizado** se b está associado a A e todos os c_j , com $j = 1, 2, \dots, n$, estão associados a símbolos do corpo da produção
- O atributo b diz-se **herdado** se b está associado a um dos símbolos não terminais do corpo da produção



- Todos os atributos são sintetizados



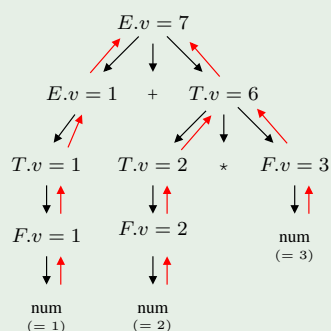
- $T.t$ é sintetizado
- $L.t$ é herdado

Gramática de atributos

Representação

- Uma gramática de atributos pode ser representada por uma tabela em que se associam as regras semânticas às produções da gramática

- Para o exemplo das expressões aritméticas, tem-se



Produções

Regras semânticas

$F \rightarrow \text{num}$	$F.v = \text{num}.v$
$F \rightarrow (E)$	$F.v = E.v$
$T \rightarrow F$	$T.v = F.v$
$T_1 \rightarrow T_2 * F$	$T_1.v = T_2.v * F.v$
$E \rightarrow T$	$E.v = T.v$
$E_1 \rightarrow E_2 + T$	$E_1.v = E_2.v + T.v$

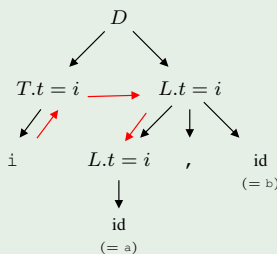
- Note que se assume que o símbolo terminal `num` tem um atributo chamado v com o valor correspondente.
- O ANTLR não suporta atributos nos terminais (*tokens*)

Gramática de atributos

Representação

- Uma gramática de atributos pode ser representada por uma tabela em que se associam as regras semânticas às produções da gramática

- Para o exemplo da declaração de variáveis, tem-se



Produções	Regras semânticas
$T \rightarrow i$	$T.t = \text{int}$
$T \rightarrow f$	$T.t = \text{float}$
$D \rightarrow T L$	$L.t = T.t$
$L_1 \rightarrow L_2 , \text{id}$	$L_2.t = L_1.t$ $\text{addsym}(\text{id}.n, L_1.t)$
$L \rightarrow \text{id}$	$\text{addsym}(\text{id}.n, L.t)$

- Assume-se que o símbolo terminal `id` tem um atributo chamado `n` com o valor correspondente
- Neste caso, para além do cálculo de atributos, faz-se a inserção numa tabela de símbolos (`addsym`)

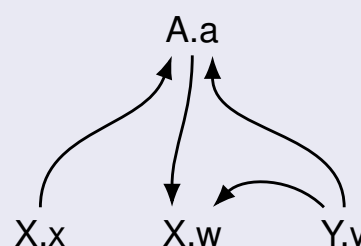
Avaliação dirigida pela sintaxe

- Numa **avaliação dirigida pela sintaxe** o cálculo dos atributos é feito à medida que é feita a análise sintática.
- Num analisador sintático ascendente (caso do bison) todos os atributos têm de ser sintetizados
- Num analisador sintático descendente (caso do Antlr) além de sintetizados os atributos podem ser herdados, desde que de símbolos à esquerda ou do símbolo pai
- para definir a ordem de cálculo dos atributos, usa-se o **grafo de dependências**

$$A \rightarrow X Y$$

$$A.a = f(X.x, Y.y)$$

$$X.w = g(A.a, Y.y)$$



- Aqui as setas apontam no sentido das dependências