

Tema 1

Compiladores, Linguagens e Gramáticas

Introdução

Compiladores, 2º semestre 2022-2023

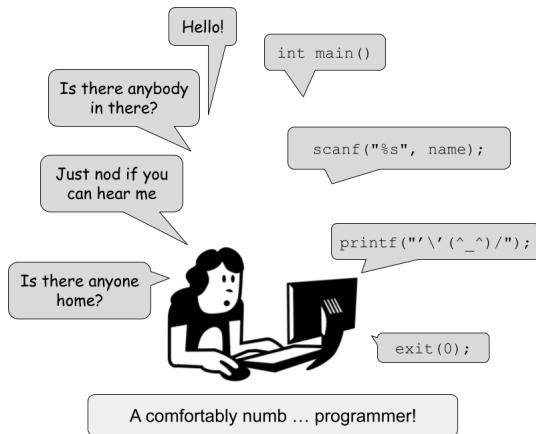
Miguel Oliveira e Silva, Artur Pereira, DETI, Universidade de Aveiro

Conteúdo

1 Enquadramento	1
1.1 Linguagens de programação	3
2 Compiladores: Introdução	3
3 Estrutura de um Compilador	5
3.1 Análise Lexical	5
3.2 Análise Sintáctica	6
3.3 Análise Semântica	6
3.4 Síntese	7
4 Implementação de um Compilador	7
4.1 Análise léxica	7
4.2 Análise sintáctica	9
4.3 Análise semântica	9
4.4 Síntese: interpretação do código	11
5 Linguagens: Definição como Conjunto	13
5.1 Conceito básicos e terminologia	13
5.2 Operações sobre palavras	15
5.3 Operações sobre linguagens	16
6 Introdução às gramáticas	19
6.1 Hierarquia de Chomsky	21
6.2 Autómatos	22
6.2.1 Máquina de Turing	22
6.2.2 Autómatos linearmente limitados	23
6.2.3 Autómatos de pilha	23
6.2.4 Autómatos finitos	24

1 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.



- Se tivesse que definir *linguagem* como é que o faria?
 - Podemos dizer que é um protocolo que nos permite *expressar, transmitir e receber ideias*.
 - Até meados do século passado, dir-se-ia que era uma forma de *comunicação* entre pessoas ou, eventualmente, entre seres vivos.
 - Com o advento dos computadores, o conceito de linguagem foi alargado para a comunicação com e entre máquinas.
 - Em comum, a necessidade de ter mais do que uma entidade comunicante, e um código e conjunto de regras que torna essa comunicação inteligível para todas as partes.
 - Uma pessoa a tentar comunicar em português com outra que desconheça essa língua é tão eficaz como tentar pôr um gato a tocar uma pauta de uma sonata de Beethoven (outra linguagem: a música).
 - Portanto, é necessário não só ter uma codificação e um conjunto de regras adequadas a cada linguagem, como também interlocutores que as conheçam.
 - Curiosamente, línguas diferentes como o português e o inglês, são compostas por *palavras* diferentes, mas partilham muitos dos símbolos utilizados para construir essas palavras.
 - Assim, “*adeus*” e “*goodbye*” são sequências diferentes de letras, muito embora tenham um significado semelhante e partilhem também o alfabeto de letras com que são construídas.
 - No entanto, dificilmente se reconhece alguma partilha com a tradução para uma das primeiras linguagens escritas – a linguagem cuneiforme babilónica (~ 3400BC) 
 - Por outro lado, também existem palavras iguais com significados diferentes (dependendo do contexto):
 - *morro, rio, caminho,*
 - Por outro lado, a nossa compreensão de um texto não se faz somente lendo a sequência de letras que o compõem, mas sim compreendendo a sua sequência de *ideias*, que são expressas por *frases*, que por sua vez são sequências gramaticalmente correctas de *palavras*, elas sim compostas por sequências de *letras* e outros símbolos do alfabeto.
 - 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 → adeus

¹Traduzido em: <https://funtranslations.com/babylonian>

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

adeus + e + até + amanhã → adeus e até amanhã

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

- Inerente às linguagens, é a necessidade de decidir se uma sequência de símbolos do alfabeto é válida.

- **correcto:**

a + d + e + u + s → adeus

adeus + e + até + amanhã → adeus e até amanhã

- **incorrecto:**

e + d + u + a + s → edues

até + adeus + amanhã + e → 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.

1.1 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!

2 Compiladores: Introdução

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

- Os *compiladores* são programas que permitem:
 1. decidir sobre a correcção de sequências de símbolos do respectivo alfabeto;
 2. 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

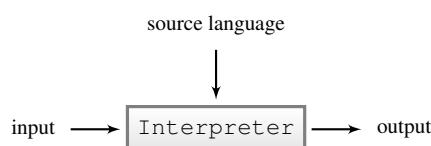
- Código fonte:

```
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);
    }
}
```

- 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

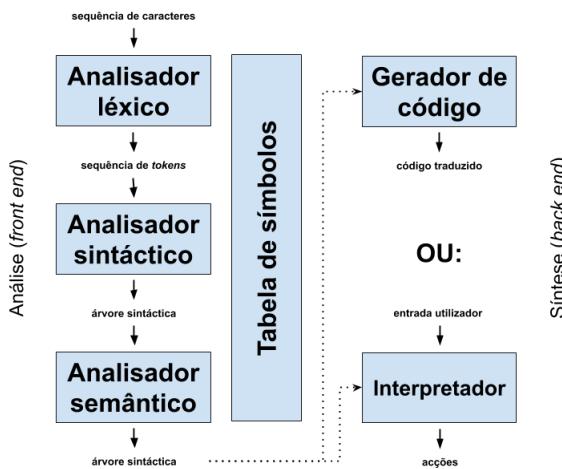
- Código fonte:

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

- Uma possível interpretação:

```
2.5
```

3 Estrutura de um Compilador



- Uma característica interessante da compilação de linguagens de alto nível, é o facto de, tal como no caso das linguagens naturais, essa compilação envolver mais do que uma linguagem:
 - **análise léxica**: composição de letras e outros caracteres em palavras (*tokens*);
 - **análise 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.

3.1 Análise Lexical

- 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 > <;>
```

- 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:

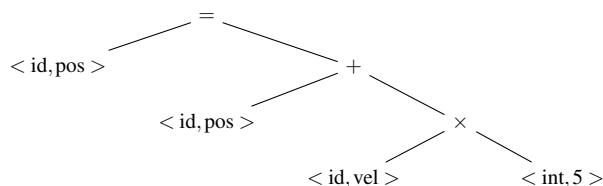
`distance (0 , 0) (4 , 3)`

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

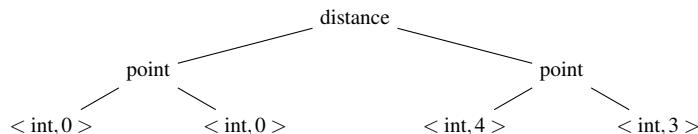
```
<distance> <(> <num,0> <,> <num,0> <)>
<(> <num,4> <,> <num,3> <)>
```

3.2 Análise Sintáctica

- 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 sintaticamente processar, podemos sempre fazer uma aproximação à sua estrutura formal através dumha 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 ;`):



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



- Chama-se a atenção para duas características das árvores sintácticas:

- 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).

3.3 Análise Semântica

- 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).
- 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).

3.4 Síntese

- Havendo garantia de que o código da linguagem fonte é válido, então podemos passar aos efeitos pretendidos com esse código.
- Os efeitos podem ser:
 1. simplesmente a indicação de validade do código fonte;
 2. a tradução do código fonte numa linguagem destino;
 3. ou a interpretação e execução imediata.
- Em todos os casos, pode haver interesse na identificação e localização precisa de eventuais erros.
- Como a maioria do código fonte assenta em texto, é usual indicar não só a instrução mas também a linha onde cada erro ocorre.

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 optimizado na fase seguinte da compilação:

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

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

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

4 Implementação de um Compilador

Para ilustrar o trabalho envolvido em processadores de linguagens vamos implementar “à mão” um interpretador completo para a linguagem sugerida pela instrução: `distance (0 , 0) (4 , 3)`.

4.1 Análise léxica

Para desenvolvermos “à mão” um analisador léxico sem complicações excessivas, vamos obrigar a que *tokens* da linguagem estejam separados por pelo menos um espaço em branco e/ou uma mudança de linha. Dessa forma, podemos utilizar a classe `Scanner` (métodos `hasNext` e `next`) da biblioteca nativa Java.

Como estratégia de base para implementar este analisador, vamos considerar que este tem sempre a si associado um *token* actual. Para o início e fim, existirão dois *tokens* especiais: `NONE` e `EOF`.

Cada *token* tem a si associado o seu tipo, sendo que *tokens* do mesmo tipo partilham as mesmas propriedades léxicas; e, quando aplicável, um atributo (textual) que complete a sua definição.

Este analisador será utilizável por um método (`nextToken`) que vai gerar o próximo *token* consumindo caracteres da entrada.

A listagem I mostra uma possível implementação desse programa.

Compilando e executando este programa com a entrada:

```
echo "distance ( 0 , 0 ) ( 1 , 4 )" | java -ea lexer/GeometryLanguageLexer
```

Listing 1: Exemplo de um analisador lexical

```

package lexer;

import static java.lang.System.*;
import java.util.Scanner;

public class GeometryLanguageLexer {
    /**
     * token types
     */
    public enum tokenId {
        NONE, NUMBER, COMMA, OPEN_PARENTHESSES, CLOSE_PARENTHESSES, DISTANCE, EOF
    }

    /**
     * Updates actual token to the next input token.
     */
    public static void nextToken() {
        assert token != tokenId.EOF;

        token = tokenId.EOF;
        attr = "";
        if (sc.hasNext()) {
            text = sc.next();
            switch(text) {
                case ",": token = tokenId.COMMA; break;
                case "(": token = tokenId.OPEN_PARENTHESSES; break;
                case ")": token = tokenId.CLOSE_PARENTHESSES; break;
                case "distance": token = tokenId.DISTANCE; break;
                default:
                    attr = text;
                    try {
                        value = Double.parseDouble(text);
                        token = tokenId.NUMBER;
                    }
                    catch(NumberFormatException e) {
                        err.println("ERROR: unknown lexeme "+text+"");
                        exit(1);
                    }
                    break;
            }
        }
    }

    /**
     * Actual token type
     */
    public static tokenId token() { return token; }

    /**
     * Actual token attribute
     */
    public static String attr() { return attr; }

    /**
     * Actual token value
     */
    public static Double value() { return value; }

    public static void main(String[] args) {
        do {
            nextToken();
            out.print("<" + token() + (attr().length() > 0 ? "," + attr() : "") + ">");
        }
        while(token() != tokenId.EOF);
        out.println();
    }

    protected static final Scanner sc = new Scanner(System.in);

    protected static tokenId token = tokenId.NONE;
    protected static String text = "";
    protected static String attr;
    protected static double value;
}

```

obtemos a seguinte saída:

```
<OP_DISTANCE> <OPEN_PARENTHESSES> <NUMBER,0> <COMMA> <NUMBER,0> <CLOSE_PARENTHESSES>
<OPEN_PARENTHESSES> <NUMBER,1> <COMMA> <NUMBER,4> <CLOSE_PARENTHESSES> <EOF>
```

4.2 Análise sintáctica

Como primeira aproximação para a construção de uma analisador sintáctico vamos fazer com que este apenas indique se o código fonte é uma sequência válida de *tokens* (ou não). Com esse objectivo, vamos seguir a seguinte estratégia:

- Identificar as estruturas importantes da linguagem (regras);
- Associar métodos booleanos ao reconhecimento de cada regra;
- Garantir que, na invocação desses métodos, o *token* actual é o que seria de esperar no inicio dessas regras;
- O processo de reconhecimento de regras terá três comportamentos possíveis:
 1. Se for bem sucedido, todos os tokens associados à regra foram consumidos (i.e. fazem parte do passado do analisador léxico);
 2. Falha por não reconhecimento do primeiro *token* da regra. Neste caso não há lugar ao consumo de nenhum token;
 3. Falha no meio do reconhecimento da regra. Nesta situação, o analisador limita-se a rejeitar a sequência de *tokens*.
- Neste processo, sempre que é reconhecido um *token*, o analisador sintáctico consumirá esse *token* (i.e. avança para o próximo).

As estruturas (regras) importantes no esboço apresentado são a instrução *distância*. Como esta instrução se aplica a dois pontos, temos também a regra *ponto* (que por sua vez se aplica a um par de números).

A listagem 2 mostra uma possível implementação desse programa.

4.3 Análise semântica

A linguagem definida até agora não permite erros semânticos que possam servir de exemplo para esta secção. Para resolver esse problema, vamos acrescentar à linguagem a possibilidade de definir e utilizar variáveis. A existência de variáveis possibilita a existência de erros semânticos resultantes da utilização de variáveis não definidas.

As variáveis, são um recurso programático que permite o armazenamento de valores recorrendo a nomes (designados *identificadores*). Nesta linguagem, vamos definir um identificador como sendo uma sequência não vazia de letras minúsculas (sem acentos).

O analisador léxico tem de ser alterado, acrescentando os *tokens* ID e EQUAL:

```
...
public enum tokenId {
    NONE, NUMBER, COMMA, OPEN_PARENTHESSES, CLOSE_PARENTHESSES, DISTANCE, ID, EQUAL, EOF
}
public static void nextToken() {
...
    case "=": token = tokenId.EQUAL; break;
    default:
        attr = text;
        if (attr.matches("[a-zA-Z]+"))
            token = tokenId.ID;
        else
        {
            try {
                value = Double.parseDouble(text);
                token = tokenId.NUMBER;
            }
            catch(NumberFormatException e) {
                err.println("ERROR: unknown lexeme \\" + text + "\\");
            }
        }
}
```

Listing 2: Exemplo de um analisador sintático

```

package parser;

import static java.lang.System.*;
import static lexer.GeometryLanguageLexer.*;

public class GeometryLanguageParser {
    /**
     * Start rule: attempts to parse the whole input.
     */
    public static boolean parse() {
        assert token() == tokenId.NONE;

        nextToken();
        return parseDistance();
    }

    /**
     * Distance rule parsing.
     */
    public static boolean parseDistance() {
        assert token() != tokenId.NONE;

        boolean result = token() == tokenId.DISTANCE;
        if (result) {
            nextToken();
            result = parsePoint();
            if (result) {
                result = parsePoint();
            }
        }
        return result;
    }

    /**
     * Point rule parsing.
     */
    public static boolean parsePoint() {
        assert token() != tokenId.NONE;

        boolean result = token() == tokenId.OPEN_PARENTHESSES;
        if (result) {
            nextToken();
            result = token() == tokenId.NUMBER;
            if (result) {
                nextToken();
                result = token() == tokenId.COMMA;
                if (result) {
                    nextToken();
                    result = token() == tokenId.NUMBER;
                    if (result) {
                        nextToken();
                        result = token() == tokenId.CLOSE_PARENTHESSES;
                        if (result) {
                            nextToken();
                        }
                    }
                }
            }
        }
        return result;
    }

    public static void main(String[] args) {
        if (parse())
            out.println("Ok");
        else
            out.println("ERROR");
    }
}

```

Listing 3: Expressões.

```

public static boolean parseExpression() {
    assert token() != tokenId.NONE;

    boolean result = true;
    switch(token()) {
        case NUMBER:
            nextToken();
            break;
        case ID:
            if (!symbolTable.containsKey(attr()))
            {
                err.println("ERROR: undefined variable \\" + attr() + "\\");
                exit(1);
            }
            nextToken();
            break;
        default:
            result = parseDistance();
            break;
    }
    return result;
}

```

```

        exit(1);
    }
    break;
...
}
...

```

A estrutura de dados adequada para lidar com variáveis é o *array* associativo. Com esta estrutura de dados, podemos associar ao nome da variável os valores que quisermos. Para já apenas queremos saber se a variável está, ou não está, definida. Pelo que basta a existência do identificador no *array* associativo.

```
protected static Map<String, Object> symbolTable = new HashMap<String, Object>();
```

Vamos também generalizar um pouco a linguagem definido o conceito de *expressão*. Uma expressão vai ser uma entidade do programa que tem a si associada um valor numérico. No caso, poderá ser um número literal, uma variável ou a instrução de distância. O código [3] apresenta um método que faz essa análise sintáctica.

Para activar esta generalização, basta substituir a análise sintáctica de número literal (NUMBER) por uma invocação deste novo método. Note que esta nova estrutura sintáctica aumenta imenso a flexibilidade da linguagem, já que agora onde se espera um valor numérico (coordenada de um ponto, atribuição de valor) pode aparecer uma qualquer expressão (em vez de somente um número literal).

Precisamos agora de acrescentar uma instrução de atribuição de valor (que define, ou redefine, o valor duma variável). O código [4] mostra esse método.

Para tornar activa a nova instrução vamos modificar o método inicial de análise sintáctica (código [5]).

4.4 Síntese: interpretação do código

Para completar este exemplo, falta apenas implementar acções ligadas à linguagem definida. Para não complicar o problema, vamos considerar que todas as instruções têm um valor numérico, e que o efeito de uma instruções é a escrita desse valor. Assim, por exemplo, a aplicação da instrução de distância deve calcular e escrever esse valor.

Para implementar este comportamento vamos inserir directamente no analisador sintáctico o código necessário para realizar estas acções. Como as instruções passam a estar associadas a valores numéricos, precisamos de arranjar forma de lhes associar esses valores. Nesse sentido, vamos substituir os resultados booleanos pelo tipo de dados não primitivo `Double`. Um resultado igual a `null` indica erro sintáctico, e

Listing 4: Atribuição de valor.

```
public static boolean parseAssignment() {
    assert token() != tokenIds.NONE;

    boolean result = token() == tokenIds.ID;
    if (result) {
        String var = attr();
        nextToken();
        result = token() == tokenIds.EQUAL;
        if (result) {
            nextToken();
            result = parseExpression();
            if (result) {
                symbolTable.put(var, null);
            }
        }
    }
    return result;
}
```

Listing 5: Novo método *parse*.

```
public static boolean parse() {
    assert token() == tokenIds.NONE;

    nextToken();
    boolean result = true;
    while(result && token() != tokenIds.EOF) {
        result = parseDistance();
        if (!result)
            result = parseAssignment();
    }
    return result;
}
```

um valor não nulo, expressa o valor associado à instrução. A única excepção a este procedimento será a análise sintáctica de pontos já que estes têm de estar associados a um par de valores (e não apenas a um).

O código [6] exemplifica o código do interpretador (em anexo é fornecido o programa completo).

5 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.
- Como definir linguagens de forma sintética e objectiva?
- Definir por *extensão* – isto é, enumerando todas as possíveis ocorrências – 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*.

5.1 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: $\{n \mid n \in \mathbb{N} \wedge n \leq 9\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Um *símbolo* (ou *letra*) é a unidade atómica (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 .

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

- 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.

Listing 6: Interpretador.

```

...
public static Double parseAssignment() {
    assert token() != tokenId.NONE;

    Double result = null;
    if (token() == tokenId.ID) {
        String var = attr();
        nextToken();
        if (token() == tokenId.EQUAL) {
            nextToken();
            result = parseExpression();
            if (result != null) {
                symbolTable.put(var, result);
            }
        }
    }
    return result;
}
...

public static Double parseDistance() {
    assert token() != tokenId.NONE;

    Double result = null;
    if (token() == tokenId.DISTANCE) {
        nextToken();
        Double[] p1 = parsePoint();
        if (p1 != null) {
            Double[] p2 = parsePoint();
            if (p2 != null) {
                result = Math.sqrt(Math.pow(p1[0]-p2[0],2)+Math.pow(p1[1]-p2[1],2));
            }
        }
    }
    return result;
}
...

public static Double[] parsePoint() {
    assert token() != tokenId.NONE;

    Double[] result = null;
    if (token() == tokenId.OPEN_PARENTHESSES) {
        nextToken();
        Double x = parseExpression();
        if (x != null) {
            if (token() == tokenId.COMMA) {
                nextToken();
                Double y = parseExpression();
                if (y != null) {
                    if (token() == tokenId.CLOSE_PARENTHESSES) {
                        nextToken();
                        result = new Double[2];
                        result[0] = x;
                        result[1] = y;
                    }
                }
            }
        }
    }
    return result;
}
...

```

2016,234523,9999999999999999,0

- $A = \{0, 1, \dots, 0, a, b, \dots, z, @, \dots\}$
- mos@ua.pt, Bom dia!

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

5.2 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

$$|\epsilon| = 0$$

- É habitual interpretar-se a palavra u como uma função de acesso aos seus símbolos (tipo *array*):

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

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

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

- 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)
 - $u.\epsilon = \epsilon.u = u$ (elemento neutro)
 - $u \neq \epsilon \wedge v \neq \epsilon \wedge u \neq v \implies u.v \neq v.u$ (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, $\underbrace{uu \dots u}_{n \times}$.
- $u^0 = \epsilon$

5.3 Operações sobre linguagens

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 $A = \{a, b\}$:

$$\begin{aligned} L_1 &= \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\} \\ L_2 &= \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\} \end{aligned}$$

- qual será o resultado da reunião destas linguagens?

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

- Resposta:

$$L = \{w_1 aw_2 \mid w_1, w_2 \in A^* \wedge (w_1 = \epsilon \vee w_2 = \epsilon)\}$$

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 $A = \{a, b\}$:

$$\begin{aligned} L_1 &= \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\} \\ L_2 &= \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\} \end{aligned}$$

- qual será o resultado da intercepção destas linguagens?

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

- Resposta:

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

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 $A = \{a, b\}$:

$$\begin{aligned} L_1 &= \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\} \\ L_2 &= \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\} \end{aligned}$$

- 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 $A = \{a, b\}$:

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

- qual será o resultado da complementação desta linguagem?

$$L = \bar{L}_1 = ?$$

- Resposta:

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

- ou:

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

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 $A = \{a,b\}$:

$$\begin{aligned} L_1 &= \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\} \\ L_2 &= \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\} \end{aligned}$$

- qual será o resultado da concatenação destas linguagens?

$$L = L_1.L_2 = ?$$

- Resposta:

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

Operações sobre linguagens: potenciação

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

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

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

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

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

$$L = L_1^2 = ?$$

- Resposta:

$$L = \{aw_1aw_2 \mid w_1, w_2 \in A^*\}$$

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:

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

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

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:
$$A_1 \cup A_2$$

6 Introdução às gramáticas

- A utilização de conjuntos para definir linguagens não é frequentemente a forma mais adequada e versátil para as descrever.
- Muitas vezes é preferível identificar estruturas intermédias, que abstraem partes ou subconjuntos importantes, da linguagem.
- Tal como em programação, muitas vezes descrições recursivas são bem mais simples, sem perda da 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*.
- Por exemplo, em português a frase “O cão ladra” pode ser gramaticalmente descrita por:

$$\begin{array}{lcl} \text{frase} & \rightarrow & \text{sujeito predicado} \\ \text{sujeito} & \rightarrow & \text{artigo substantivo} \\ \text{predicado} & \rightarrow & \text{verbo} \\ \text{artigo} & \rightarrow & \mathbf{O} \mid \mathbf{Um} \\ \text{substantivo} & \rightarrow & \mathbf{cão} \mid \mathbf{lobo} \\ \text{verbo} & \rightarrow & \mathbf{ladra} \mid \mathbf{uiva} \end{array}$$

- Esta gramática (não recursiva) descreve formalmente 8 possíveis frases, o que é ainda pouco interessante.
- No entanto, contém mais informação do que a frase original, já que classifica os vários elementos da frase (sujeito, predicado, etc.).
- Contém 6 *símbolos terminais* e 6 *símbolos não terminais*.
- Um símbolo não terminal é definido por uma *produção* descrevendo possíveis representações desse símbolo, em função de símbolos terminais e/ou não terminais.
- Formalmente, uma gramática é um quádruplo $G = (T, N, S, P)$, onde:

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

Gramáticas: exemplos

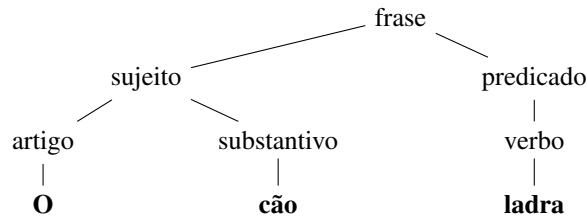
- Formalmente, a gramática anterior será:

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

- P é constituído pelas regras já apresentadas:

$$\begin{aligned} \text{frase} &\rightarrow \text{sujeito predicado} \\ \text{sujeito} &\rightarrow \text{artigo substantivo} \\ \text{predicado} &\rightarrow \text{verbo} \\ \text{artigo} &\rightarrow \mathbf{O} \mid \mathbf{Um} \\ \text{substantivo} &\rightarrow \mathbf{cão} \mid \mathbf{lobo} \\ \text{verbo} &\rightarrow \mathbf{ladra} \mid \mathbf{uiva} \end{aligned}$$

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



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

$$\begin{aligned} S &\rightarrow 0S \\ S &\rightarrow 0A \\ A &\rightarrow 0A1 \\ A &\rightarrow \epsilon \end{aligned}$$

- Qual será a linguagem definida por esta gramática?

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

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

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

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

$$\begin{aligned} S &\rightarrow aX \\ X &\rightarrow aX \\ X &\rightarrow bX \\ X &\rightarrow \epsilon \end{aligned}$$

ou:

$$\begin{aligned} S &\rightarrow aX \\ X &\rightarrow aX \mid bX \mid \epsilon \end{aligned}$$

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

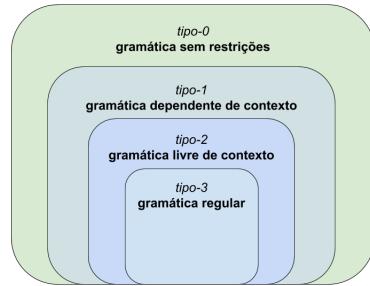
$$L_3 = \{u \mid u \in A^* \wedge u.\text{count}(1) \bmod 2 = 0\}$$

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

$$\begin{aligned} S &\rightarrow S1S1S \mid A \\ A &\rightarrow 0A \mid \epsilon \end{aligned}$$

6.1 Hierarquia de Chomsky

- 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: $S \rightarrow \epsilon$).
 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 $A \rightarrow \epsilon$, onde A e B são símbolos não terminais (A pode ser igual a B) e c um símbolo terminal. Isto é, em todas as produções, o β só pode ter no máximo um símbolo não terminal sempre à direita (ou, alternativamente, sempre à esquerda).

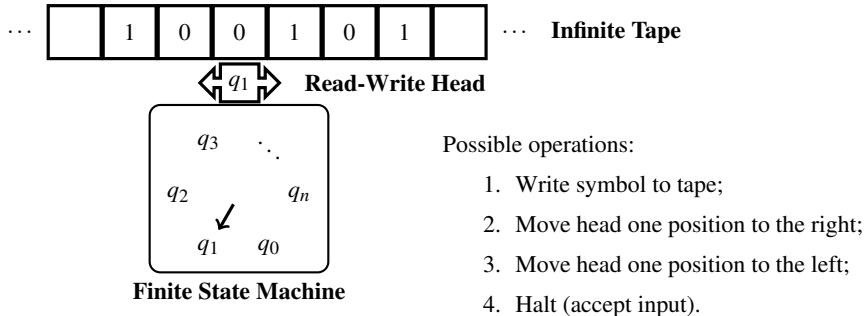


- Para cada um desses tipos podem ser definidos diferentes tipos de máquinas (algoritmos, autômatos) que as podem reconhecer.
- Quanto mais simples for a gramática, mas simples e eficiente é a máquina que reconhece essas linguagens.
- Cada classe de linguagens do *tipo-i* contém a classe de linguagens *tipo-(i+1)* ($i = 0, 1, 2$)
- Esta hierarquia não traduz apenas as características formais das linguagens, mas também expressam os requisitos de computação necessários:
 1. As *máquinas de Turing* processam gramáticas sem restrições (tipo-0);
 2. Os *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).

6.2 Autómatos

6.2.1 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.



- 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"): $\dots \mathbf{0}111101110\dots$ (o resultado pretendido será: $\dots \mathbf{0}111111000\dots$).
- Considerando que os estados são designados por $E_i, i \geq 1$ (sendo E_1 o estado inicial); e as operações:

d mover uma posição para a direita;

e mover uma posição para a esquerda;

0 escrever o símbolo 0 na fita;

1 escrever o símbolo 1 na fita;

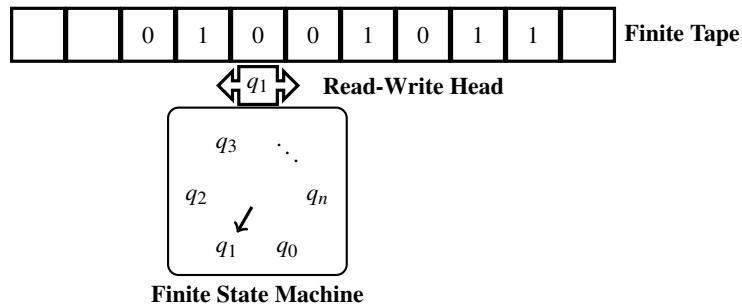
h aceitar e terminar autómato.

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

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

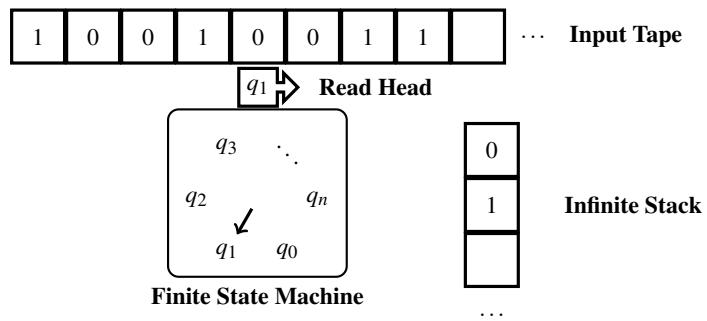
- $E_1 \cdots 0111101110 \cdots \rightarrow E_1 \cdots 0111101110 \cdots \xrightarrow{*} E_2 \cdots 0111101110 \cdots \rightarrow E_3 \cdots 0111111110 \cdots \rightarrow E_3 \cdots 0111111110 \cdots \xrightarrow{*} E_3 \cdots 0111111110 \cdots \rightarrow E_4 \cdots 0111111110 \cdots \rightarrow E_5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

6.2.2 Autómatos linearmente limitados



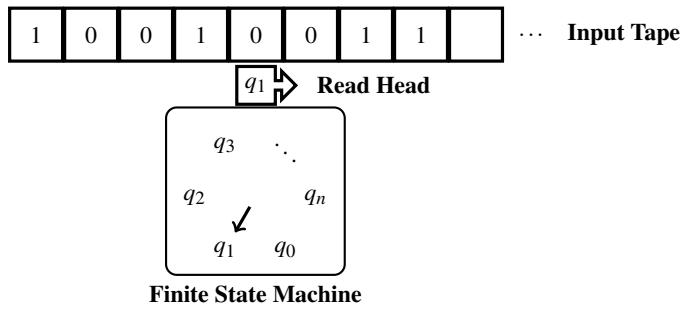
- Diferem das MT pela finitude da fita.

6.2.3 Autómatos de pilha



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

6.2.4 Autómatos finitos



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

Tema 2

ANTLR4

Introdução, Estrutura, Aplicação

Compiladores, 2º semestre 2022-2023

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

Conteúdo

1 Apresentação	3
2 Exemplos	4
2.1 <i>Hello</i>	4
2.2 <i>Expr</i>	5
2.3 Exemplo figuras	8
2.4 Exemplo <i>visitor</i>	9
2.5 Exemplo <i>listener</i>	9
3 Construção de gramáticas	10
3.1 Especificação de gramáticas	11
4 ANTLR4: Estrutura léxica	11
4.1 Comentários	11
4.2 Identificadores	12
4.3 Literais	12
4.4 Palavras reservadas	12
4.5 Acções	12
5 ANTLR4: Regras léxicas	13
5.1 Padrões léxicos típicos	14
5.2 Operador léxico “não ganancioso”	14
6 ANTLR4: Estrutura sintáctica	15
6.1 Secção de <i>tokens</i>	15
6.2 Acções no preâmbulo da gramática	15
7 ANTLR4: Regras sintácticas	16
7.1 Padrões sintácticos típicos	17
7.2 Precedência	17
7.3 Associatividade	17
7.4 Herança de gramáticas	17

8 ANTLR4: outras funcionalidades	18
8.1 Mais sobre acções	18
8.2 Exemplo: tabelas CSV	18
8.3 Gramáticas ambíguas	19
8.4 Predicados semânticos	21
8.5 Separar analisador léxico do analisador sintáctico	22
8.6 “Ilhas” lexicais	23
8.7 Enviar <i>tokens</i> para canais diferentes	23
8.8 Reescrever a entrada	24
8.9 Desacoplar código da gramática - ParseTreeProperty	25
9 ANTLR4: gestão de erros	26
9.1 ANTLR4: relatar erros	26
9.2 ANTLR4: recuperar de erros	27
9.3 ANTLR4: alterar estratégia de gestão de erros	27

1 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>

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

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

2 Exemplos

2.1 Hello

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)
  
```

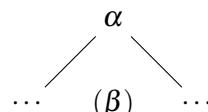
- As duas gramáticas – lexical e sintáctica – são expressas com instruções com a seguinte estrutura:

$$\alpha : \beta;$$

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 .

ANTLR4: Hello (2)

- Uma sequência de símbolos na entrada que seja reconhecido por esta regra grammatical 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.

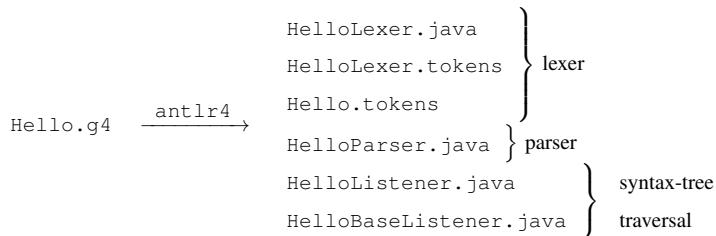
```
antlr4 Hello.g4
antlr4-javac Hello*.java
echo "hello compiladores" | antlr4-test Hello r -tokens
```

- Utilização:

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

ANTLR4: Ficheiros gerados

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



- 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 sintácticas do compilador.
- Podemos executar o ANTLR4 com a opção `-visitor` para gerar também código Java para o padrão tipo *visitor* (difere do *listener* porque a visita tem de ser explicitamente requerida).
 - `HelloVisitor.java` e `HelloBaseVisitor.java`: código Java que implementa automaticamente um padrão de execução de código tipo *visitor* todos os pontos de entrada e saída de todas as regras sintácticas do compilador.

2.2 Expr

ANTLR4: Expr

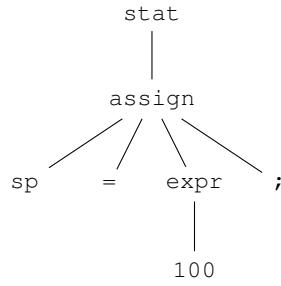
- Exemplo:

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

- Se executarmos o compilador criado com a entrada:

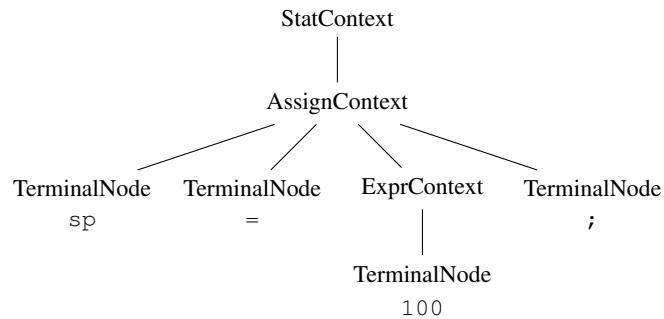
```
sp = 100;
```

- Vamos obter a seguinte árvore sintáctica:



ANTLR4: contexto automático

- 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:



ANTLR4: contexto automático (2)

(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 ;

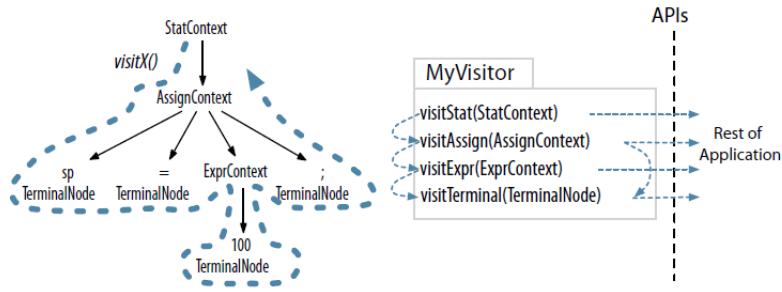
```

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

ANTLR4: visitor

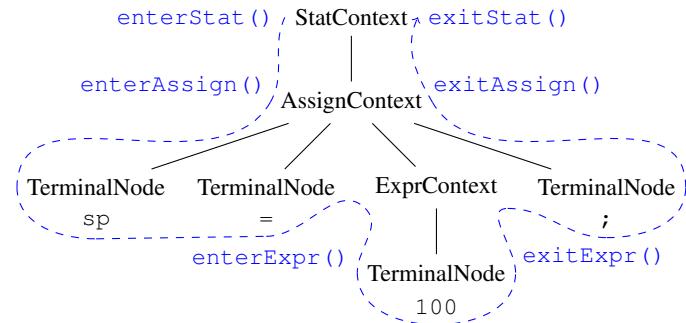
- Os objectos de contexto têm a si associada toda a informação relevante da análise sintáctica (*tokens*, referência aos nós filhos da árvore, etc.)
- Por exemplo o contexto `AssignContext` contém métodos `ID` e `expr` para aceder aos respectivos nós.

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

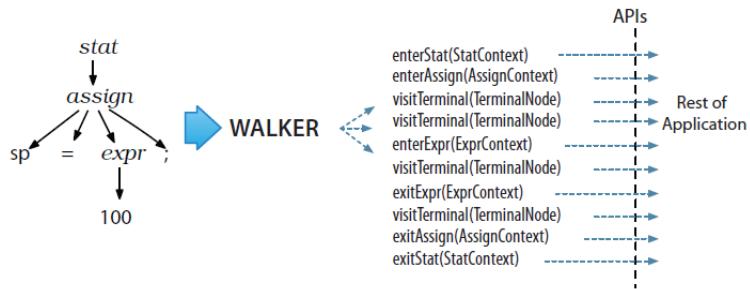


ANTLR4: listener

- 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 ação não tem de ser necessariamente Java (existem muitas outras possíveis, como C++ e python).

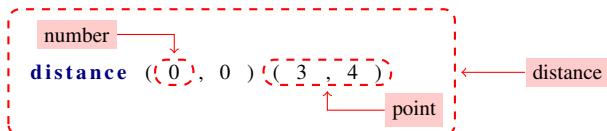
- Também podemos passar atributos para a regra (tipo passagem de argumentos para um método):

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

- É clara a semelhança com a passagem de argumentos e resultados de métodos.
- Diz que os atributos são *sintetizados* quando a informação provém de sub-regras, e *herdados* quando se envia informação para sub-regras.

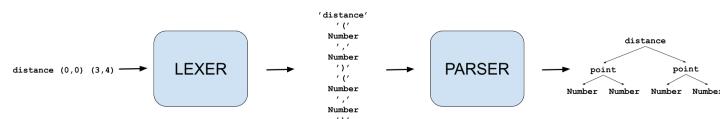
2.3 Exemplo figuras

- 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;
```



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);
    }
}

```

- O comando `antlr4-main` gera automaticamente esta classe com uma primeira implementação do método `main`.

2.4 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;
    }
}

```

- 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 específica nos argumentos do comando é mantida).

2.5 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);
}
}

```

- 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> ...
```

3 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 (tokens na gramática lexical e símbolos sintácticos na outra) são definidos por produções (regras).
- No fim, todos os símbolos não terminais, com mais ou menos transformações, devem poder ser expressos em símbolos terminais.
- Uma gramática é construída especificando as *regras* ou produções dos elementos gramaticais.

```

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

```

- Sendo a sua construção uma forma de programação, podemos beneficiar da identificação e reutilização de padrões comuns de resolução de problemas.
- Surpreendentemente, o número de padrões base é relativamente baixo:
 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.

- 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;
    }
}

```

- 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.

3.1 Especificação de gramáticas

- Uma linguagem para especificação de gramáticas precisa de suportar este conjunto de padrões.
 - Para especificar elementos léxicos (*tokens*) a notação utilizada assenta em *expressões regulares*.
 - A notação tradicionalmente utilizada para a análise sintáctica denomina-se por *BNF* (*Backus-Naur Form*).
- <symbol> ::= <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> ;

4 ANTLR4: Estrutura léxica

4.1 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
```

4.2 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, conditionalStatement // parser rule names
```

- Como em Java, podem ser utilizados caracteres Unicode.

4.3 Literais

- 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 ('\u0001'), assim como as sequências de escape habituais ('\r\t\n')

4.4 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.

4.5 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.
- As acções podem aparecer nas regras léxicas, nas regras sintácticas, na especificação de exceções 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 ;
```

5 ANTLR4: Regras léxicas

- 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*).
- 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 sintácticas, estão definidos *antes* dos definidos explicitamente por regras léxicas.
- A especificação destas regras utiliza *expressões regulares*.

Expressões regulares em ANTLR4

Syntax	Description
$R : \dots ;$	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
$XY \dots Z$	Match a sequence of rule lexer elements
(...)	Lexer subrule
$X?$	Match rule element X
X^*	Match rule element X zero or more times
X^+	Match rule element X one or more times
$\sim x$	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)
{...}	Lexer action
{ p }?	Evaluate semantic predicate p (if false, the rule is ignored)
$x \dots z$	Multiple alternatives

5.1 Padrões léxicos típicos

Token category	Possible implementation
Identifiers	<pre>ID: LETTER (LETTER DIGIT)*; fragment LETTER: 'a'..'z'/'A'..'Z'/_'; // same as: [a-zA-Z_] fragment DIGIT: '0'..'9'; // same as: [0-9]</pre>
Numbers	<pre>INT: DIGIT+; FLOAT: DIGIT+ '.' DIGIT+ '.' DIGIT+;</pre>
Strings	<pre>STRING: '"' (ESC .)*? '"'; fragment ESC: '\\"' '\\\\\' ;</pre>
Comments	<pre>LINE_COMMENT: '//' .*? '\n' -> skip; COMMENT: '/*' .*? '*/' -> skip;</pre>
Whitespace	<pre>WS: [\t\n\r]+ -> skip;</pre>

5.2 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!
```

6 ANTLR4: Estrutura sintáctica

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

6.1 Secção de *tokens*

- 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).

6.2 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 tipos 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 inicio de ficheiros, e a segunda permite que se acrescente 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`)

7 ANTLR4: Regras sintácticas

Construção de regras: síntese

Syntax	Description
$r : \dots ;$	Define rule r
x	Match rule element x
$xy \dots z$	Match a sequence of rule elements
(\dots)	Subrule
$x?$	Match rule element x
x^*	Match rule element x zero or more times
x^+	Match rule element x one or more times
$x \mid \dots \mid z$	Multiple alternatives

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).

Regras sintácticas: movendo informação

- Em ANTLR4 cada regra sintática 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*).

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

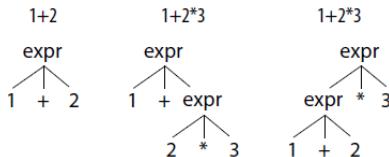
INT: [0-9]+;
WS: [ \t\n\r]+ -> skip;
```

7.1 Padrões sintáticos típicos

<i>Pattern name</i>	<i>Possible implementation</i>
<i>Sequence</i>	<code>x y ... z [' INT+ '] [' INT* ']</code>
<i>Sequence with terminator</i>	<code>(instruction ';')* // program sequence (row '\n')* // lines of data</code>
<i>Sequence with separator</i>	<code>expr (',' expr)* // function call arguments (expr (',' expr)*)? // optional arguments</code>
<i>Choice</i>	<code>type: 'int' 'float'; instruction: conditional loop ... ;</code>
<i>Token dependence</i>	<code>(' expr ') // nested expression ID '[' expr ']' // array index '{' instruction+ '}' // compound instruction '<' ID (',' ID)* '>' // generic type specifier</code>
<i>Recursivity</i>	<code>expr: '(' expr ')' ID; classDef: 'class' ID '{' (classDef/method/field)* '}';</code>

7.2 Precedência

- 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
    ;
  
```

7.3 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:
 $a \uparrow b \uparrow c = a^{b^c} = a^{(bc)}$
- Este problema é resolvido em ANTLR4 de seguinte forma:

```

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

7.4 Herança de gramáticas

- A secção de *import* implementa um mecanismo de herança entre gramáticas.

- Por exemplo as gramáticas:

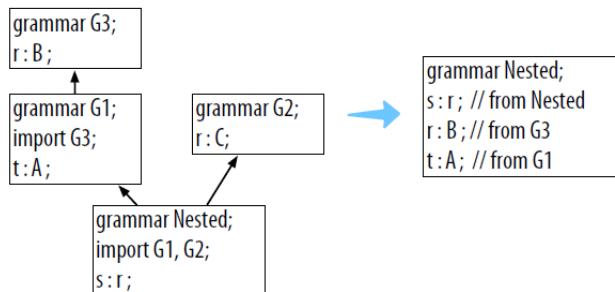
```
grammar ELang;
stat : (expr ';' )* EOF ;
expr : INT ;
INT : [0-9]+ ;
WS : [ \r\t\n]+ -> skip ;
```

```
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.
- 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.

8 ANTLR4: outras funcionalidades

8.1 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.

8.2 Exemplo: tabelas CSV

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]*;

```

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]*;

```

8.3 Gramáticas ambíguas

- A definição de gramáticas presta-se, com alguma facilidade, a gerar ambiguidades.
- Esta característica nas linguagens humanas é por vezes procurada (onde estaria a literatura e a poesia se não fosse assim), mas geralmente é um problema.

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

“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.

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]+;
...

```

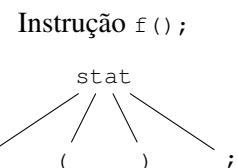
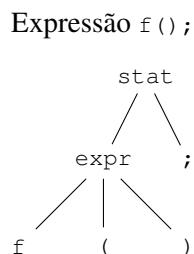
- Neste caso a sequência de caracteres **if** tanto pode dar um identificador como uma palavra reservada.
- O ANTLR4 utiliza duas regras fora das expressões regulares para lidar com ambiguidade:
 1. Por omissão, escolhe o *token* que consume o máximo número de caracteres da entrada;
 2. Dá prioridade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

Gramáticas ambíguas: analisador sintáctico

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

<pre> stat: ID '=' expr ID '=' expr ; expr: NUM ; </pre>	<pre> stat: expr ';' ID '(' ')' ';' ; expr: ID '(' ')' NUM ; </pre>
--	---

- Em ambos os casos a ambiguidade resulta de ser ter uma sub-regra repetida, directamente, no primeiro caso, e indirectamente, no segundo caso.
- A gramática diz-se ambígua porque, para a mesma entrada, poderíamos ter duas árvores sintácticas diferentes.



- Outros exemplos de ambiguidade são os da precedência e associatividade de operadores (secções 7.2 e 7.3).
- O ANTLR4 tem regras adicionais para eliminar ambiguidades sintácticas.
- Tal como no analisador léxico, regras *Ad hoc* fora da notação das gramáticas 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 sintácticas apresentadas no exemplo anterior, a gramática definida impõe a primeira alternativa.
- 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*.

8.4 Predicados semânticos

- Em ANTLR4 é possível utilizar informação semântica (expressa na linguagem destino e injetada na gramática), para orientar o analisador sintáctico.
- Essa funcionalidade chama-se *predicados semânticos*: { . . . } ?
- 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)

Exemplo

```
grammar Seq;

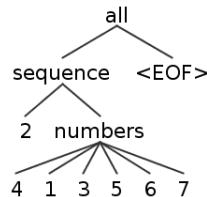
all: sequence* EOF;

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:



Exemplo

```
grammar Seq;

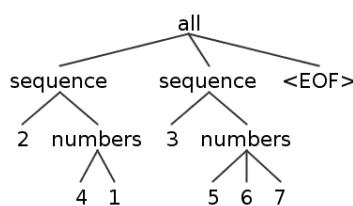
all: sequence* EOF;

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:



8.5 Separar analisador léxico do analisador sintáctico

- Muito embora se possa definir a gramática completa, juntando a análise léxica e a sintáctica 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 sintáctico 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 sintáctica.
 5. A gramática sintáctica tem de incluir uma opção (`tokenVocab`) a indicar o analisador léxico.

```
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

```
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
```

8.6 “Ilhas” lexicais

- 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 dumha “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`

Exemplo

```
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;
}

a11: ( ACTION_START | OUTSIDE_TOKEN | ACTION_END |
INSIDE_TOKEN)* EOF;

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;
}

a11: ( ACTION_START | OUTSIDE_TOKEN | ACTION_END |
INSIDE_ACTION_START | INSIDE_TOKEN)* EOF;
```

8.7 Enviar *tokens* para canais diferentes

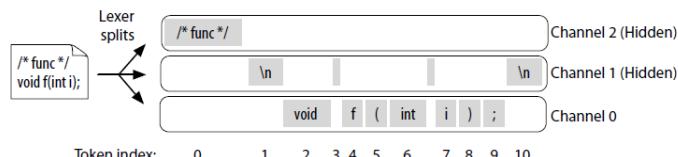
- Nos exemplos de gramáticas que temos vindo a apresentar, tem-se optado pela acção `skip` quando na presença dos chamados espaços em branco ou de comentários.
- Esta acção faz desaparecer esses *tokens* simplificando a análise sintáctica.
- O preço a pagar (geralmente irrelevante) é perder o texto completo 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

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);
```

8.8 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, redirecionando-os para um canal escondido.
- Agora podemos criar um *listener* para resolver este problema.

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");
    }
}
```

8.9 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`.
- 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:

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;
```

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());
    }
}
```

9 ANTLR4: gestão de erros

9.1 ANTLR4: relatar erros

- 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.

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);
    }
}
```

- 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: .;
```

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);
    }
}
```

9.2 ANTLR4: recuperar de erros

- 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.)

9.3 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.

Tema 3

Análise Semântica

Gramáticas de atributos, tabela de símbolos

Compiladores, 2º semestre 2022-2023

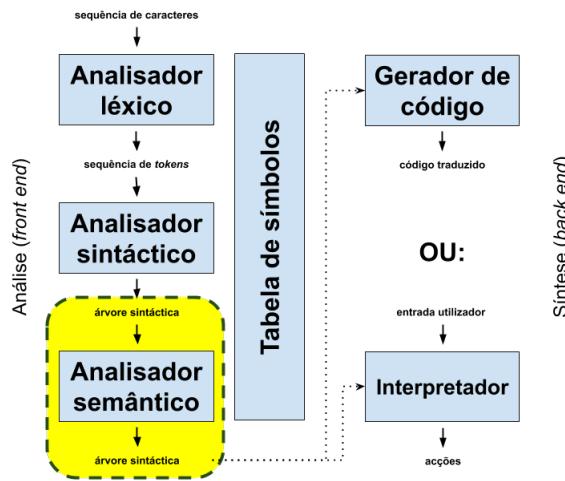
Miguel Oliveira e Silva, Artur Pereira, DETI, Universidade de Aveiro

Conteúdo

1 Análise semântica: Estrutura de um Compilador	2
1.1 Avaliação dirigida pela sintaxe	2
1.2 Detecção estática ou dinâmica	2
2 Sistema de tipos	3
3 Gramáticas de atributos	3
3.1 Dependência local: classificação de atributos	4
3.2 ANTLR4: Declaração de atributos associados à árvore sintáctica	5
4 Tabela de símbolos	6
4.1 Agrupando símbolos em contextos	8
5 Instruções restringidas por contexto	8

1 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 sintática; noutras casos, só é possível fazer essa verificação após uma análise sintática bem sucedida, fazendo uso da informação retirada dessa análise.

1.1 Avaliação dirigida pela sintaxe

- No processamento de linguagens, a avaliação semântica pode ser feita associando informação e acções às regras sintáticas 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 sintática, pode ser feita durante a própria análise sintática, e/ou posteriormente recorrendo a *visitors* e/ou *listeners*.

1.2 Detecção estática ou dinâmica

- 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).

2 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 $data$.

3 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 sintática, 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 ao 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.

- Entre os diferentes tipos de atributos, existem alguns cujo valor depende apenas da sua vizinhança sintáctica.
 - 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.

3.1 Dependência local: classificação de atributos

- Os atributos podem ser classificados duas formas, consoante as dependências que lhes são aplicáveis:
 - 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).
 - Dizem-se *herdados*, se depende de nós "irmãos" ou de nós ascendentes.



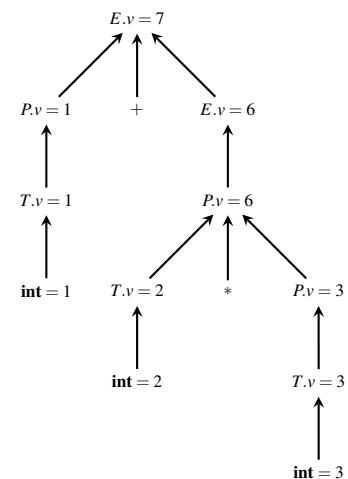
- Formalmente podem-se designar os atributos anotando com uma seta no sentido da dependência (para cima, nos atributos sintetizados; e para baixo nos herdados).

Exemplo dependência local: expressão aritmética

- Considere a seguinte gramática:

$$\begin{aligned} E &\rightarrow P + E \mid P \\ P &\rightarrow T * P \mid T \\ T &\rightarrow (E) \mid \text{int} \end{aligned}$$

- 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 sintáctica anotada:



$$\begin{aligned} E &\rightarrow P + E \mid P \\ P &\rightarrow T * P \mid T \\ T &\rightarrow (E) \mid \text{int} \end{aligned}$$

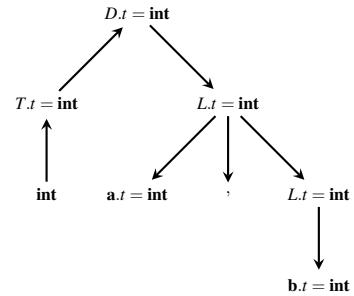
Produção	Regra semântica
$E_1 \rightarrow P + E_2$	$E_1.v = P.v + E_2.v$
$E \rightarrow P$	$E.v = P.v$
$P_1 \rightarrow T * P_2$	$P_1.v = T.v * P_2.v$
$P \rightarrow T$	$P.v = T.v$
$T \rightarrow (E)$	$T.v = E.v$
$T \rightarrow \text{int}$	$T.v = \text{int.value}$

Exemplo dependência local: declaração

- Considere a seguinte gramática:

$$\begin{aligned} D &\rightarrow TL \\ T &\rightarrow \text{int} \mid \text{real} \\ L &\rightarrow \text{id}, L \mid \text{id} \end{aligned}$$

- Se quisermos definir um atributo t 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 sintáctica anotada:



$$\begin{aligned} D &\rightarrow TL \\ T &\rightarrow \text{int} \mid \text{real} \\ L &\rightarrow \text{id}, L \mid \text{id} \end{aligned}$$

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

3.2 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 sintáticas;

```

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`.

ANTLR4: Declaração de atributos `ParseTreeProperty`

- 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*).

Gramáticas de atributos em ANTLR4: síntese

- Podemos associar três tipos de informação a regras sintácticas:
 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.

4 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 noutro 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).
- 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);
    }
}
```

- 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"); }

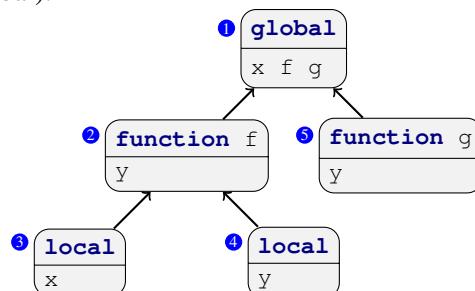
    public boolean subtype(Type other) {
        return super.subtype(other) ||
               other.name().equals("real");
    }
}
```

4.1 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.
- 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
    { int x; }       // define variable x in nested local scope
    { int y; }       // define variable y in another nested local scope
}
❸ 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 ❶ o símbolo x está definido em ❶.
- 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).



- 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.

5 Instruções restringidas por contexto

- Algumas linguagens de programação restringem a utilização de certas instruções a determinados contextos.
- 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 sintática recorrendo a predicados semânticos e um contador (ou uma pilha) que registe 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

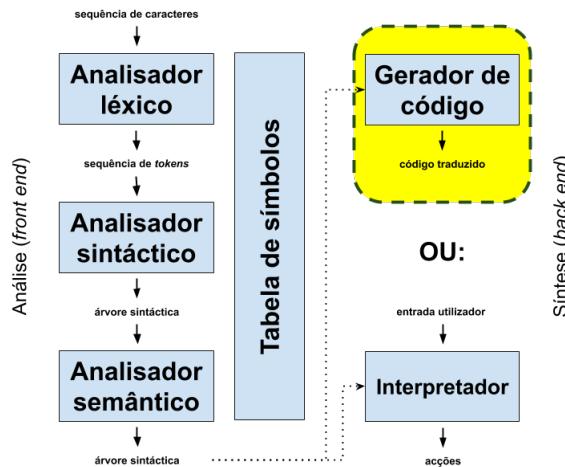
Compiladores, 2º semestre 2022-2023

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

Conteúdo

1 Síntese: geração de código	2
1.1 Geração de código máquina	2
1.2 Geração de código	3
2 String Template	3
2.1 Geração de código: padrões comuns	6
2.2 Geração de código para expressões	6
3 Síntese: geração de código intermédio	7
3.1 Código de triplo endereço	7
3.2 TAC: Exemplo de expressões binárias	7
3.3 TAC: Endereços e instruções	7
3.4 Controlo de fluxo	8
3.5 Funções	8

1 Síntese: geração de código



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

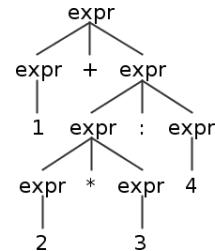
source language → **Compiler** → target language

- 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 sintática (mas já com a garantia muito importante de inexistência de erros sintáticos e semânticos) gerando o código destino nos pontos apropriados.

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);
    }
}
```

1.1 Geração de código máquina

- 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 registo do processador.
- (Consultar a bibliografia recomendada para estudar este tipo de geração de código.)

1.2 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.



- 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 um 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.

2 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.
- 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);
```

String Template Group

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

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

- 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");
...

```

String Template: dicionários e condicionais

- Neste 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).

String Template: 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>
>>

```

String Template: listas

- 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>

2.1 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.
- 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.

2.2 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.
- 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"
]
```

3 Síntese: geração de código intermédio

3.1 Código de triplo endereço

- O padrão para expressões é um exemplo dumha 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).

3.2 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 registo (o que, num compilador gerador de código máquina, é resolvido numa fase posterior de optimização) – é codificável em linguagens de baixo nível.

3.3 TAC: Endereços e instruções

- 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 **iffFalse 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** $x_1 \dots$ **param** x_n ; **call** p,n ; $y =$ **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)

3.4 Controlo de fluxo

- 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 11
A
goto 12
label 11:
B
label 12:
```

- De forma similar podemos gerar código para ciclos:

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

```
label 11:
iffalse cond goto 12
A
goto 11
label 12:
```

3.5 Funções

- A geração de código para funções pode ser feita recorrendo a uma estratégia tipo “macro” (i.e. na invocação da funções é colocado o código que implementa a função), ou implementando módulos algorítmicos separados.
- Neste último caso (que, entre outras coisas, permite a recursividade), 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).
- Geralmente as arquitecturas de linguagens de baixo nível (CPU’s) têm instruções específicas para lidar com esta estrutura de dados.
- Vamos exemplificar esse procedimento: Este código apenas ilustra a ideia. Para uma análise mais detalhada devem consultar a temática de arquitectura de computadores *frame-pointer*.

```
// 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
```


Compiladores:

Apontamentos sobre linguagens livres de contexto

(ano letivo de 2022-2023)

Artur Pereira, Miguel Oliveira e Silva

Maio de 2023

Nota prévia

Este documento representa apenas um compilar de notas sobre a matéria teórico-prática lecionada na unidade curricular “Compiladores”, sem pretensões, por isso, de ser um texto exaustivo. No caso do último capítulo, na realidade, é apenas um enumerar das secções do *Dragon Book* usadas na produção dos *slides*. A sua leitura deve apenas ser encarada como ponto de partida e nunca como único elemento de estudo. Os *slides*, em muitos aspectos, estão mais completos que estes apontamentos.

Conteúdo

1 Gramática livre de contexto	1
1.1 Definição de gramática livre de contexto	2
1.2 Derivação	3
1.3 Operações sobre gramáticas livres de contexto	4
1.3.1 Reunião	4
1.3.2 Concatenação	5
1.3.3 Fecho de Kleene	5
1.3.4 Intersecção e complementação	5
1.4 Árvore de derivação	5
1.4.1 Ambiguidade	6
1.5 Limpeza de gramáticas	8
1.5.1 Símbolos produtivos e não produtivos	8
1.5.2 Símbolos acessíveis e não acessíveis	9
1.5.3 Gramáticas limpas	10
1.6 Transformações em GIC	10
1.6.1 Eliminação de produções- ε	10
1.6.2 Eliminação de recursividade à esquerda	11
1.6.3 Fatorização à esquerda	12
1.7 Os conjuntos first , follow e predict	13
1.7.1 O conjunto first	13
1.7.2 O conjunto follow	13
1.7.3 O conjunto predict	14

2 Análise sintática descendente	15
2.1 Reconhecimento preditivo	16
2.2 Reconhecedores recursivo-descendentes	17
2.3 Reconhecedores descendentes não recursivos	18
2.4 Implementação de gramáticas de atributos	20
3 Análise sintáctica ascendente	21
3.1 Conflitos	23
3.2 Construção de um reconhecedor ascendente	25
3.2.1 Construção da coleção de conjuntos de itens	25
3.2.2 Tabela de decisão para um reconhecimento ascendente	27
3.2.3 Algoritmo de reconhecimento	27
4 Gramática de atributos	29
4.1 Definição de gramática de atributos	29
4.1.1 Atributos herdados e atributos sintetizados	29
4.1.2 Construção de gramáticas de atributos	29
4.2 Ordem de avaliação dos atributos	30
4.2.1 Grafo de dependências	30
4.2.2 Tipos de gramáticas de atributos	30

Capítulo 1

Gramática livre de contexto

Considere uma estrutura G , definido sobre o alfabeto $T = \{a, b, c\}$, com as regras de rescrita

$$\begin{array}{lcl} S & \rightarrow & \varepsilon \\ & | & X \ S \\ X & \rightarrow & a \\ & | & c \\ & | & a \ b \ c \end{array}$$

Que palavras se podem gerar a partir de S ? Podem gerar-se 0 ou mais concatenações de X , sendo cada X substituível por a , c ou abc . Ou seja, pode gerar-se a mesma linguagem que a descrita pela expressão regular $e = (a \mid c \mid abc)^*$. Embora a linguagem seja regular, a gramática não o é. A produção $S \rightarrow X \ S$ viola a definição de gramática regular, porque tem dois símbolos não terminais.

Considere agora a gramática G

$$\begin{array}{lcl} S & \rightarrow & \varepsilon \\ & | & a \ S \ b \end{array}$$

definida sobre o alfabeto $T = \{a, b\}$. Qual é a linguagem L descrita pela gramática G ? A partir de S podem gerar-se as palavras ε , ab , $aabb$, \dots , ou seja $L = \{a^n b^n \mid n \geq 0\}$. Pode provar-se que esta linguagem não é regular. Por conseguinte, não é possível definir uma expressão regular, gramática regular ou autómato finito que a represente.

Exemplo 1.1

Considere sobre o alfabeto $T = \{a, b\}$, a gramática

$$\begin{array}{lcl} S & \rightarrow & a \ S \\ & | & a \ X \\ X & \rightarrow & a \ b \\ & | & a \ X \ b \end{array}$$

Esta gramática gera a linguagem $L = \{a^m b^n \mid n > 0 \wedge m > n\}$. O símbolo X gera a linguagem $a^n b^n$, com $n > 0$. O símbolo S gera a^k , com $k > 0$, seguido de X . A conjugação resulta nas palavras $a^k a^n b^n$, com $k, n > 0$.

Exemplo 1.2

Considere sobre o alfabeto $T = \{a, b, c\}$, a gramática

$$\begin{array}{lcl} S & \rightarrow & c \\ & | & a \ S \ a \\ & | & b \ S \ b \end{array}$$

Esta gramática gera a linguagem $L = \{w c w^R \mid w \in \{a, b\}^*\}$, onde w^R representa a palavra w com os símbolos colocados por ordem inversa.

As linguagens dos dois últimos exemplos também não são regulares, não podendo, por isso, ser descritas por expressões regulares. Correspondem a linguagens designadas **livres de contexto**, ou **independentes do contexto**. As gramáticas adequadas para descrever estas linguagens têm todas as suas produções da forma $A \rightarrow \beta$, em que A é um símbolo não terminal e β é uma sequência constituída por zero ou mais símbolos terminais ou não terminais.

No exemplo seguinte apresenta-se uma gramática **dependente do contexto**.

Exemplo 1.3

Considere sobre o alfabeto $T = \{a, b, c\}$, a gramática

$$\begin{array}{lcl} S & \rightarrow & a \ S \ B \ C \\ & | & a \ b \ C \\ C \ B & \rightarrow & B \ C \\ b \ B & \rightarrow & b \ b \\ b \ C & \rightarrow & b \ c \\ c \ C & \rightarrow & c \ c \end{array}$$

Esta gramática gera a linguagem $L = \{a^n b^n c^n \mid n > 0\}$. Note que, por exemplo, na produção $b \ B \rightarrow b \ b$ o símbolo não terminal B apenas se pode transformar em b se tiver um b imediatamente à sua esquerda.

1.1 Definição de gramática livre de contexto

Formalmente, uma gramática livre de contexto 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, cada uma da forma $\alpha \rightarrow \beta$, onde
 - $\alpha \in N$
 - $\beta \in (T \cup N)^*$
- $S \in N$ é o símbolo inicial.

Nas produções, α e β são designados por **cabeça da produção** e **corpo da produção**, respectivamente. Note que relativamente à definição de gramática regular a diferença está na definição de β . No caso das gramáticas dependentes do contexto, como é o caso da do exemplo 1.3, a cabeça das produções (α) deixa de ter de ser necessariamente um único símbolo não terminal.

1.2 Derivação

D Dada uma produção $u \rightarrow v$ e uma palavra $\alpha u \beta$, chama-se **derivação direta** à rescrita de $\alpha u \beta$ em $\alpha v \beta$, denotando-se

$$\alpha u \beta \Rightarrow \alpha v \beta$$

Em algumas situações, será usada o termo **passo de derivação** como sinónimo de derivação direta.

D Chama-se **derivação** a uma sucessão de zero ou mais derivações diretas, denotando-se

$$\alpha \Rightarrow^* \beta$$

ou, equivalentemente,

$$\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_n = \beta$$

onde n é o comprimento da derivação.

- A notação $\alpha \Rightarrow^n \beta$ é usada para representar uma derivação de comprimento n
- A notação $\alpha \Rightarrow^+ \beta$ é usada para representar uma derivação de comprimento não nulo

D Dada uma gramática $G = (T, N, P, S)$ e uma palavra $u \in (T \cup N)^+$, o conjunto das **palavras derivadas** a partir de u é representado por

$$D(u) = \{v \in T^* : u \Rightarrow^* v\}$$

D A linguagem gerada pela gramática $G = (T, N, P, S)$ é representada por

$$L(G) = D(S) = \{v \in T^* : S \Rightarrow^* v\}$$

Nas gramáticas livres de contexto, em geral, em cada passo de uma derivação estão envolvidos vários símbolos não terminais. Como as produções têm a forma $\alpha \rightarrow \beta$, com $\alpha \in N$ e $\beta \in (T \cup N)^*$, β pode conter vários símbolos não terminais, que serão introduzidos na derivação se a produção for utilizada.

A gramática seguinte, definida sobre o alfabeto $T = \{a, b\}$, é livre de contexto e gera a linguagem $L = \{w \in T^* \mid \#(a, w) = \#(b, w)\}$

$$\begin{aligned} S &\rightarrow aB \mid bA \\ A &\rightarrow a \mid aS \mid bAA \\ B &\rightarrow b \mid bS \mid aBB \end{aligned}$$

A palavra $aabbab$ tem 4 derivações possíveis, das quais são apresentadas duas.

1. $\underline{S} \Rightarrow a\underline{B} \Rightarrow aa\underline{B}B \Rightarrow aab\underline{B} \Rightarrow aabb\underline{S} \Rightarrow aabba\underline{B} \Rightarrow aabbab$
2. $\underline{S} \Rightarrow a\underline{B} \Rightarrow aa\underline{B}B \Rightarrow aaBb\underline{S} \Rightarrow aaBba\underline{B} \Rightarrow aaBbab \Rightarrow aabbab$

Usou-se o sublinhado para destacar o símbolo não terminal expandido em cada derivação direta.

A derivação 1 designa-se por **derivação à esquerda**, porque em cada passo se expande o símbolo não-terminal mais à esquerda. A derivação 2 designa-se por **derivação à direita**, porque em cada passo se expande o símbolo não-terminal mais à direita.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.2.3, "Derivations".

1.3 Operações sobre gramáticas livres de contexto

A classe das gramáticas livres de contexto é fechada sobre as operações de reunião, concatenação e fecho, mas não o é sobre as operações de intersecção e complementação.

1.3.1 Reunião

Sejam $G_1 = (T, N_1, P_1, S_1)$ e $G_2 = (T, N_2, P_2, S_2)$ duas gramáticas livres de contexto que geram as linguagens L_1 e L_2 , respectivamente. A gramática $G = (T, N, P, S)$, onde

$$\begin{aligned} S &\notin (T \cup N_1 \cup N_2); \\ N &= N_1 \cup N_2 \cup \{S\} \\ P &= P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\} \end{aligned}$$

gera a linguagem $L = L_1 \cup L_2$.

1.3.2 Concatenação

Sejam $G_1 = (T, N_1, P_1, S_1)$ e $G_2 = (T, N_2, P_2, S_2)$ duas gramáticas livres de contexto que geram as linguagens L_1 e L_2 , respectivamente. A gramática $G = (T, N, P, S)$, onde

$$\begin{aligned}S &\notin (T \cup N_1 \cup N_2); \\N &= N_1 \cup N_2 \cup \{S\} \\P &= P_1 \cup P_2 \cup \{S \rightarrow S_1 \ S_2\}\end{aligned}$$

gera a linguagem $L = L_1 \cdot L_2$.

1.3.3 Fecho de Kleene

Seja $G_1 = (T, N_1, P_1, S_1)$ uma gramática livre de contexto que gera a linguagem L_1 . A gramática $G = (T, N, P, S)$, onde

$$\begin{aligned}S &\notin (T \cup N_1); \\N &= N_1 \cup \{S\} \\P &= P_1 \cup \{S \rightarrow \varepsilon, S \rightarrow S_1 \ S\}\end{aligned}$$

gera a linguagem $L = L_1^*$.

1.3.4 Intersecção e complementação

Se L_1 e L_2 são duas linguagens livres de contexto, e, por conseguinte, descritíveis por gramáticas livres de contexto, a sua intersecção pode não o ser. Já se disse atrás que a linguagem

$$L = \{a^i b^i c^i \mid i \geq 0\}$$

não é livre de contexto. No entanto, ela pode ser obtida por intersecção das linguagens $L_1 = \{a^i b^i c^j \mid i, j \geq 0\}$ e $L_2 = \{a^i b^j c^j \mid i, j \geq 0\}$ que o são.

Sabe-se que $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$. Então, se a intersecção de linguagens livres de contexto pode resultar numa linguagem que não o é, o mesmo pode acontecer com a complementação.

1.4 Árvore de derivação

Será que as várias derivações de uma mesma palavra são diferentes? Poderão ter interpretações diferentes? Veja-se com um exemplo que de facto podem. Considere a gramática

$$S \rightarrow S + S \mid S \cdot S \mid \neg S \mid (S) \mid 0 \mid 1$$

e compare as duas derivações esquerdas seguintes da palavra $1+1 \cdot 0$.

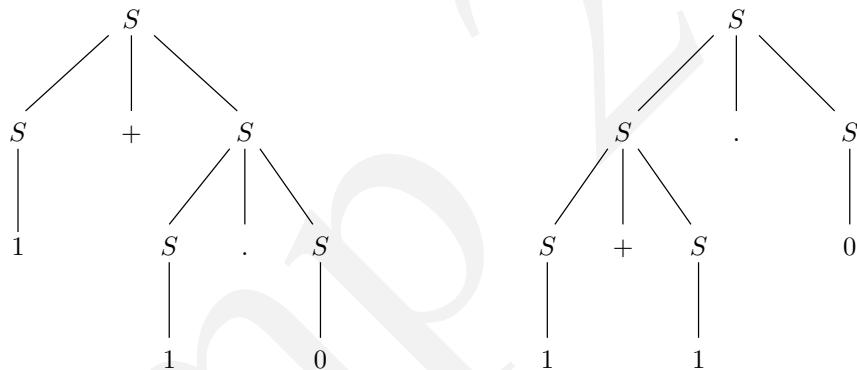
Derivação 1:

$$\underline{S} \Rightarrow \underline{S} + S \Rightarrow 1 + \underline{S} \Rightarrow 1 + \underline{S} \cdot S \Rightarrow 1 + 1 \cdot \underline{S} \Rightarrow 1 + 1 \cdot 0$$

Derivação 2:

$$\underline{S} \Rightarrow \underline{S} \cdot S \Rightarrow \underline{S} + S \cdot S \Rightarrow 1 + \underline{S} \cdot S \Rightarrow 1 + 1 \cdot \underline{S} \Rightarrow 1 + 1 \cdot 0$$

Serão estas derivações equivalentes? Para responder a esta pergunta vão-se representar as derivações usando um outro formalismo. A **árvore de derivação** (*parse tree*) é um mecanismo de representação de uma derivação que capta a interpretação dada nessa derivação. Veja-se as duas derivações anteriores representadas de forma arbórea.



A árvore da esquerda corresponde à derivação 1 e a da direita à 2. Vê-se claramente que as duas árvores têm interpretações distintas: a da esquerda é equivalente a ter-se $1 + (1 \cdot 0)$, enquanto que a direita é equivalente a $(1 + 1) \cdot 0$. Em termos de álgebra booleana as duas expressões são bastante diferentes.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.2.4, "Parse trees and derivations".

1.4.1 Ambiguidade

Diz-se que uma palavra é derivada **ambiguamente** se possuir duas ou mais árvores de derivação distintas. Na gramática anterior a palavra $1+1 \cdot 0$ é gerada ambiguamente. Diz-se que uma gramática é **ambígua** se possuir pelo menos uma palavra gerada ambiguamente. A gramática anterior é ambígua.

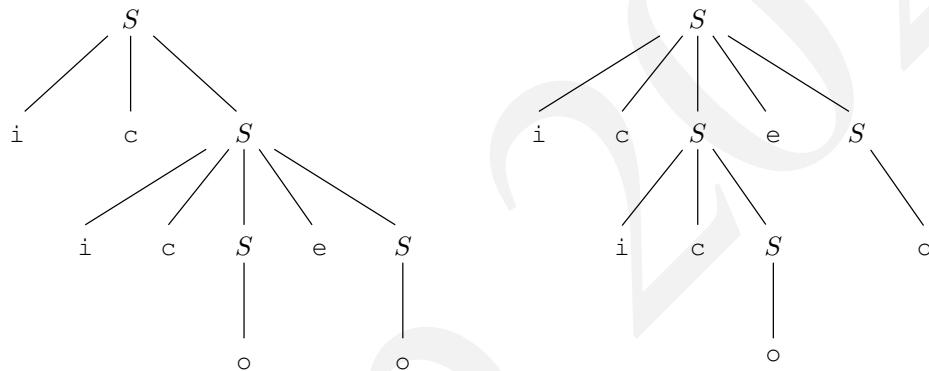
Frequentemente é possível definir-se uma gramática não ambígua que gere a mesma linguagem que uma ambígua. A gramática anterior pode ser rescrita por

$$\begin{aligned} S &\rightarrow T \mid S + T \\ T &\rightarrow F \mid T . F \\ F &\rightarrow K \mid \neg K \\ K &\rightarrow 0 \mid 1 \mid (S) \end{aligned}$$

que não é ambígua e gera exatamente a mesma linguagem. A gramática

$$S \rightarrow o \mid i \ c \ S \mid i \ c \ S \ e \ S$$

é uma abstração da instrução `if-then-else` e possui ambiguidade. A palavra `icicoeo` tem duas árvores de derivação possíveis, representadas abaixo.



A árvore da esquerda corresponde à interpretação dada na linguagem C, em que `o` (`else`) está associado ao `i` (`if`) mais à direita. A árvore da direita associa `o` ao `i` mais à esquerda.

É possível por manipulação gramatical obter-se uma gramática equivalente sem ambiguidade. A gramática seguinte não é ambígua e descreve a mesma linguagem que a anterior.

$$\begin{aligned} S &\rightarrow o \mid i \ c \ S \mid i \ c \ S' \ e \ S \\ S' &\rightarrow o \mid i \ c \ S' \ e \ S' \end{aligned}$$

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.2.5, "Ambiguity".

Linguagens inherentemente ambíguas

Há linguagens **inherentemente ambíguas**, no sentido em que é impossível definir-se uma gramática não ambígua que gere essa linguagem. É, por exemplo, o caso da linguagem

$$L = \{a^i b^j c^k \mid i = j \vee j = k\}$$

1.5 Limpeza de gramáticas

1.5.1 Símbolos produtivos e não produtivos

Seja $G = (T, N, P, S)$ uma gramática qualquer. Um símbolo não terminal A diz-se **produtivo** se for possível transformá-lo num 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.

Sobre o alfabeto $T = \{a, b, c\}$, considere a gramática

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

S é produtivo, porque

$$S \Rightarrow ab \quad \wedge \quad ab \in T^*$$

Em contrapartida, X é improdutivo.

$$X \Rightarrow cX \Rightarrow ccX \Rightarrow^* c \cdots cX$$

Por mais que se tente é impossível transformar X numa sequência de símbolos terminais.

Seja $G = (T, N, P, S)$ uma gramática qualquer. O conjunto dos seus símbolos produtivos, N_p , pode ser obtido por aplicação 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  $A \in N_p$ 
```

A 1^a regra é um caso particular da 2^a, pelo que poderia ser retirada. Optou-se por a incluir porque torna a leitura mais fácil.

Começando com um N_p igual ao resultado da aplicação da 1^a regra a todas as produções da gramática e extendendo depois esse conjunto por aplicação sucessiva da 2^a regra obtem-se o conjunto de todos os símbolos produtivos de G . O algoritmo seguinte executa esse procedimento.

Algoritmo 1.1 (Cálculo dos símbolos produtivos)

```
let  $N_p = \emptyset$ ,  $P_p = P$ 
repeat
    nothingAdded = TRUE
    foreach  $(A \rightarrow \alpha) \in P_p$  do
        if  $\alpha \in T^*$  then
            let  $N_p' = N_p \cup \{A\}$ 
            if  $N_p' \neq N_p$  then
                let  $N_p = N_p'$ 
                nothingAdded = FALSE
        else
            let  $N_p' = N_p \cup \{A\}$ 
            if  $N_p' \neq N_p$  then
                let  $N_p = N_p'$ 
                nothingAdded = FALSE
    if nothingAdded then
        break
```

```

if  $\alpha \in (T \cup N_p)^*$  then
  if  $A \notin N_p$  then
     $N_p = N_p \cup \{A\}$ 
    nothingAdded = FALSE
     $P_p = P_p \setminus \{A \rightarrow \alpha\}$ 
  until nothingAdded or  $N_p = N$ 

```

Nele, N_p representa o conjunto dos símbolos produtivos já identificados e P_p o conjunto das produções contendo símbolos ainda não identificados como produtivos. Se numa iteração nenhum símbolo for marcado como produtivo o algoritmo pára, sendo o conjunto dos símbolos produtivos o conjunto N_p tido nesse momento. Obviamente que o algoritmo também pára, se no fim de uma iteração, $N_p = N$, isto é, se todos os símbolos foram marcados como produtivos.

1.5.2 Símbolos acessíveis e não acessíveis

Seja $G = (T, N, P, S)$ uma gramática qualquer. Um símbolo terminal ou não terminal x diz-se **acessível** se for possível transformar S (o símbolo inicial) numa expressão que contenha x . Ou seja,

$$S \Rightarrow^* \alpha x \beta$$

Caso contrário, diz-se que x é **inacessível**.

Considere a gramática

$$\begin{array}{l} S \rightarrow \varepsilon \mid a \ S \ b \mid c \ C \ c \\ C \rightarrow c \ S \ c \\ D \rightarrow d \ X \ d \\ X \rightarrow C \ C \end{array}$$

É impossível transformar S numa expressão que contenha D , d , ou X , pelo que estes símbolos são inacessíveis. Os restantes são acessíveis.

Seja $G = (T, N, P, S)$ uma gramática qualquer. O conjunto dos seus símbolos acessíveis, V_A , pode ser obtido por aplicação das seguintes regras construtivas

```

 $S \in V_A$ 
if  $A \rightarrow \alpha B \beta \in P$  and  $A \in V_A$  then  $B \in V_A$ 

```

Começando com $V_A = \{S\}$ e aplicando sucessivamente a 2º regra até que ela não acrescente nada a V_A obtém-se o conjunto dos símbolos acessíveis. O algoritmo seguinte executa esse procedimento. Nele, V_A representa o conjunto dos símbolos acessíveis já identificados e N_X o conjunto dos símbolos não terminais acessíveis já identificados mas ainda não processados. No fim, quando N_X for o conjunto vazio, V_A contém todos os símbolos acessíveis.

Algoritmo 1.2 (Cálculo dos símbolos acessíveis)

```

let  $V_A = \{S\}$ ,  $N_X = V_A$ 
repeat
    let  $A =$  um elemento de  $N_X$ 
     $N_X = N_X \setminus \{A\}$ 
    foreach  $(A \rightarrow \alpha) \in P$  do
        foreach  $x$  in  $\alpha$  do
            if  $x \notin V_A$  then
                 $V_A = V_A \cup \{A\}$ 
            if  $x \in N$  then
                 $N_X = N_X \cup \{A\}$ 
    until  $N_X = \emptyset$ 

```

1.5.3 Gramáticas limpas

Numa gramática os símbolos inacessíveis e os símbolos improdutivos são **símbolos inúteis**, porque não contribuem para as palavras que a gramática pode gerar. Se tais símbolos forem removidos obtém-se uma gramática equivalente, em termos da linguagem que descreve. 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 se devem remover os inacessíveis.

1.6 Transformações em gramáticas livres de contexto

Em muitas situações práticas — algumas serão abordadas nos capítulos seguintes — é necessário transformar uma gramática numa outra que seja equivalente e goze de determinada propriedade. Apresentam-se a seguir algumas dessas transformações.

1.6.1 Eliminação de produções- ε

Uma **produção- ε** é uma produção do tipo $A \rightarrow \varepsilon$, para um qualquer símbolo não terminal A . Se L é uma linguagem livre de contexto tal que $\varepsilon \notin L$, é possível descrever L por uma gramática livre de contexto sem produções- ε . Se assim é então tem de ser possível transformar uma gramática que descreva uma linguagem L e possua produções- ε numa outra equivalente que as não possua.

Considere a gramática

$$\begin{array}{l} I \rightarrow 0 \ I \mid 1 \ P \\ P \rightarrow \varepsilon \mid 0 \ P \mid 1 \ I \end{array}$$

que descreve a linguagem L formada pelas palavras definidas sobre o alfabeto $\{0, 1\}$, com número ímpar de $1s$. Claramente, a palavra vazia não pertence a L porque não tem número ímpar de uns. Mas, a gramática contém a produção $P \rightarrow \varepsilon$. Então, de acordo com o dito anteriormente, existe uma gramática equivalente que não tem produções- ε .

A existência de tal produção na gramática anterior significa que as produções $I \rightarrow 1P$ e $P \rightarrow 0P$ podem produzir as derivações $I \Rightarrow 1$ e $P \Rightarrow 0$, respectivamente. Mas, estas derivações podem ser contempladas acrescentando as produções $I \rightarrow 1$ e $P \rightarrow 0$ à gramática, tornando desnecessária a produção- ε . Na realidade a gramática

$$\begin{array}{l} I \rightarrow 0I \mid 1P \mid 1 \\ P \rightarrow 0P \mid 0 \mid 1I \end{array}$$

é equivalente à anterior e não possui produções- ε .

Em geral, o papel da produção $A \rightarrow \varepsilon$ sobre uma produção $B \rightarrow \alpha A \beta$ pode ser representado pela inclusão da produção $B \rightarrow \alpha \beta$. Assim a eliminação das produções- ε de uma gramática pode ser obtido por aplicação do algoritmo seguinte

Algoritmo 1.3 (Eliminação de produções- ε , 1^a versão)

```
foreach  $A \rightarrow \varepsilon$  do
    foreach  $B \rightarrow \alpha A \beta$  do
        add  $B \rightarrow \alpha \beta$  to  $P$ .
    remove  $A \rightarrow \varepsilon$  from  $P$ .
```

O algoritmo anterior pode introduzir novas produções- ε na gramática. Se $B \rightarrow A$ for uma produção da gramática, a eliminação da produção $A \rightarrow \varepsilon$ introduz a produção $B \rightarrow \varepsilon$. O algoritmo deve ser alterado de modo a acautelar estas situações.

...

1.6.2 Eliminação de recursividade à esquerda

Diz-se que uma gramática é **recursiva à esquerda** se possuir um símbolo não terminal A que admita uma derivação do tipo $A \Rightarrow^+ A\gamma$, ou seja, que seja possível, em um ou mais passos de derivação, transformar A numa expressão que tem A no início.

A gramática seguinte é recursiva à esquerda

$$\begin{array}{l} E \rightarrow X T \\ X \rightarrow \varepsilon \mid E + \\ T \rightarrow a \mid b \mid (E) \end{array}$$

A derivação $E \Rightarrow X T \Rightarrow E + T$ mostra que é possível transformar E numa expressão com E à esquerda. Logo, esta gramática tem recursividade à esquerda associada ao símbolo não terminal E .

Se a obtenção da expressão começada por A se faz em apenas um passo de derivação, então diz-se que a recursividade é **imediata**. Esta última situação só ocorre se a gramática possuir uma ou mais produções do tipo $A \rightarrow A \alpha$.

No gramática seguinte a recursividade à esquerda é imediata.

$$\begin{aligned} E &\rightarrow T \mid E + T \\ T &\rightarrow a \mid b \mid (E) \end{aligned}$$

A eliminação de recursividade imediata à esquerda fazer-se com um algoritmo bastante simples. Considere que $A \rightarrow \beta$ e $A \rightarrow A \alpha$, onde A é um símbolo não terminal e α e β sequências de zero ou mais símbolos terminais ou não terminais, são duas produções de uma gramática qualquer. Será possível substituir as duas produções por outras que não possuam recursividade à esquerda e produzam uma gramática equivalente? Para responder a esta pergunta observem-se as palavras que se podem obter a partir de A . Numa derivação de um passo obtem-se $A \Rightarrow \beta$. Numa de dois passos obtem-se $A \Rightarrow A\alpha \Rightarrow \beta\alpha$. Numa de n passos, com $n > 0$, obtem-se $A \Rightarrow \beta\alpha^{n-1}$. Mas estas palavras também podem ser obtidas com as produções

$$\begin{aligned} A &\rightarrow \beta X \\ X &\rightarrow \varepsilon \mid \alpha X \end{aligned}$$

que não possui recursividade à esquerda. A nova gramática continua a ser recursiva. Na realidade, não pode deixar de o ser, a recursividade passou para à direita.

O algoritmo anterior pode ser facilmente adaptado a situações em que possa haver mais do que uma produção a introduzir a recursividade imediata à esquerda. Considere que

$$A \rightarrow \beta_1 \mid \beta_2 \mid \dots \beta_m \mid A \alpha_1 \mid A \alpha_2 \mid \dots \mid A \alpha_n$$

são as produções de uma gramática com A à cabeça. As palavras que se podem gerar com estas produções são as mesmas que se podem gerar com as produções seguintes e que não possuem recursividade à esquerda.

$$\begin{aligned} A &\rightarrow \beta_1 X \mid \beta_2 X \mid \dots \beta_m X \\ X &\rightarrow \varepsilon \mid \alpha_1 X \mid \alpha_2 X \mid \dots \mid \alpha_n X \end{aligned}$$

Se a recursividade à esquerda não é imediata o algoritmo de eliminação é um pouco mais complexo.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.3.3, "Elimination of left recursion".

1.6.3 Fatorização à esquerda

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.3.4, "Left factoring".

1.7 Os conjuntos **first**, **follow** e **predict**

A construção de reconhecedores sintáticos (*parsers*) — apresentados nos capítulos seguintes — é auxiliada por três funções associadas às gramáticas. Estas funções são os conjuntos **first**, **follow** e **predict**.

1.7.1 O conjunto **first**

Seja $G = (T, N, P, S)$ uma gramática qualquer e α uma sequência de símbolos terminais e não terminais, isto é, $\alpha \in (T \cup N)^*$. O conjunto **first**(α) contém todos os símbolos terminais que podem aparecer como primeira letra de sequências obtidas a partir de α por aplicação de produções da gramática. Ou seja, um símbolo terminal x pertence a **first**(α) se e só se $\alpha \Rightarrow^* x\beta$, com β qualquer. Adicionalmente, considera-se que ε pertence ao conjunto **first**(α) se $\alpha \Rightarrow^* \varepsilon$.

Algoritmo 1.4 (Conjunto **first**)

```
first( $\alpha$ ) {
    if ( $\alpha = \varepsilon$ ) then
        return  $\{\varepsilon\}$ 
     $h = \text{head } (\alpha)$       # com  $|h| = 1$ 
     $\omega = \text{tail } (\alpha)$     # tal que  $\alpha = h\omega$ 
    if ( $h \in T$ ) then
        return  $\{h\}$ 
    else
        return  $\bigcup_{(h \rightarrow \beta_i) \in P} \text{first}(\beta_i \omega)$ 
}
```

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 (que é o **tail** do α).

1.7.2 O conjunto **follow**

O conjunto **follow** está relacionado com os símbolos não terminais de uma gramática. Seja $G = (T, N, P, S)$ uma gramática e A um elemento de N ($A \in N$). O conjunto **follow**(A) contém todos os símbolos terminais que podem aparecer imediatamente à direita de A num processo derivativo qualquer. Formalmente, $\text{follow}(A) = \{a \in T \mid S \Rightarrow^* \gamma A a \psi\}$, com α e β quaisquer.

O cálculo dos conjuntos **follow** dos símbolos não terminais da gramática $G = (T, N, P, S)$ baseia-se na aplicação das 4 regras seguintes, onde \supseteq significa “contém”.

1. $\$ \in \text{follow}(S)$.
2. se $(A \rightarrow \alpha B) \in P$, então $\text{follow}(B) \supseteq \text{follow}(A)$.
3. se $(A \rightarrow \alpha B \beta) \in P$ e $\varepsilon \notin \text{first}(\beta)$, então $\text{follow}(B) \supseteq \text{first}(\beta)$.
4. se $(A \rightarrow \alpha B \beta) \in P$ e $\varepsilon \in \text{first}(\beta)$, então $\text{follow}(B) \supseteq ((\text{first}(\beta) - \{\varepsilon\}) \cup \text{follow}(A))$.

A primeira regra é óbvia. Sendo o símbolo inicial da gramática, S representa as palavras da linguagem. Logo $\$$ vem a seguir.

A segunda regra diz que se $A \rightarrow \alpha B$ é uma produção da gramática e $x \in \text{follow}(A)$, então $x \in \text{follow}(B)$. Considere, por hipótese, que $x \in \text{follow}(A)$. Então, pela definição de conjunto **follow**, $S \Rightarrow^* \gamma A x \psi$. Logo, sendo $A \rightarrow \alpha B$ uma produção da gramática, tem-se que $S \Rightarrow^* \gamma \alpha B x \psi$, ou seja, $x \in \text{follow}(B)$.

A terceira regra diz que se $A \rightarrow \alpha B \beta$ é uma produção da gramática, com $\varepsilon \notin \text{first}(\beta)$, e $x \in \text{first}(\beta)$, então $x \in \text{follow}(B)$. Considere, por hipótese, que $x \in \text{first}(\beta)$. Então, pela definição de conjunto **first**, $\beta \Rightarrow^* x \gamma$ e, consequentemente, $A \Rightarrow^* \alpha B x \gamma$, ou seja, $x \in \text{follow}(B)$.

Finalmente, a quarta e última regra diz que se $A \rightarrow \alpha B \beta$ é uma produção da gramática, com $\varepsilon \in \text{first}(\beta)$, e $x \in (\text{first}(\beta) - \{\varepsilon\}) \cup \text{follow}(A)$, então $x \in \text{follow}(B)$. Esta regra pode ser entendida cruzando as duas regras anteriores. Considere-se os elementos de $\text{first}(\beta)$ diferentes de ε . Pela regra 3, pertencem ao **follow**(B). Se β se transforma em ε , então $A \Rightarrow^* \alpha B$ e, pela regra 2, se $x \in \text{follow}(A)$, então $x \in \text{follow}(B)$.

1.7.3 O conjunto **predict**

O conjunto **predict** aplica-se às produções de uma gramática e envolve os conjuntos **first** e **follow**. É dado pela seguinte equação.

$$\text{predict}(A \rightarrow \alpha) = \begin{cases} \text{first}(\alpha) & \varepsilon \notin \text{first}(\alpha) \\ (\text{first}(\alpha) - \{\varepsilon\}) \cup \text{follow}(A) & \varepsilon \in \text{first}(\alpha) \end{cases}$$

Capítulo 2

Análise sintática descendente

NOTA PRÉVIA: Este capítulo não está completo e não foi devidamente revisto, pelo que, por um lado, há partes omissas e, por outro lado, pode conter falhas.

Dada uma gramática $G = (T, N, P, S)$ e dada uma palavra $u \in T^*$, $u \in L(G)$ se e só se existir uma derivação que produza u a partir de S , isto é, se $S \Rightarrow^* u$. Um **reconhecedor sintático** da gramática G é um mecanismo que responde à pergunta “ $u \in L(G)??$ ”, tentando produzir a derivação anterior. Para o fazer, o reconhecedor pode partir de S e tentar chegar a u ou partir de u e tentar chegar a S . No primeiro caso diz-se que o reconhecedor é **descendente**, porque o seu procedimento corresponde à geração da árvore de derivação da palavra u de cima (raiz) para baixo (folhas).

O papel da análise sintática é definir procedimentos que permitam construir reconhecedores sintáticos a partir da gramática. Por exemplo, considere a linguagem L descrita pela gramática G seguinte.

$$S \rightarrow a \ S \ b \mid c \ S \mid \varepsilon$$

Será que a palavra $acacb \in L$? Pertencerá se $S \Rightarrow^* acacb$. Na verdade pertence porque

$$S \Rightarrow aSb \Rightarrow acSb \Rightarrow acaSbb \Rightarrow acacSbb \Rightarrow acacb$$

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4, "Top-down parsing".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4.3, "LL(1) grammars".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4.1, "Predictive parsing".

2.1 Reconhecimento preditivo

Mas como deterministicamente produzir a derivação anterior se houver duas ou mais produções com o mesmo símbolo à cabeça? Por exemplo, considerando o caso anterior, como saber que se deve optar por expandir o S usando $S \rightarrow a S b$ e não $S \rightarrow c S$ ou $S \rightarrow \varepsilon$? A solução baseia-se na observação antecipada dos próximos símbolos à entrada (*lookahead*). No exemplo, se o próximo símbolo à entrada for um a expande-se usando a produção $S \rightarrow a S b$; se for um c usa-se $S \rightarrow c S$; se for b usa-se $S \rightarrow \varepsilon$. Se a entrada se esgotou, o que é representado considerando que a próxima entrada é um $\$$, também se expande usando a produção $S \rightarrow \varepsilon$.

Pode-se então definir uma tabela que para cada símbolo não terminal da gramática e para cada valor do *lookahead* indica qual a produção da gramática que deve ser usada na expansão. O profundidade da observação antecipada (número de símbolos de *lookahead*) pode ser qualquer, embora apenas a profundidade 1 será usada neste documento. Para o exemplo anterior a tabela assume a forma

symbol	<i>lookahead</i>			
	a	b	c	\$
S	$S \rightarrow a S b$	$S \rightarrow \varepsilon$	$S \rightarrow c S$	$S \rightarrow \varepsilon$

Na tabela anterior, o preenchimento das colunas a e c são óbvias, visto que as produções associadas começam pelo próprio símbolo do *lookahead*. Nas colunas b e $\$$ tal não acontece. O preenchimento da tabela de decisão baseia-se nos conjuntos **predict**, apresentados na secção 1.7, e faz-se usando o algoritmo seguinte:

Algoritmo 2.1 (Preenchimento da tabela de decisão para *lookahead* 1)

```

foreach  $(A \rightarrow \alpha) \in P$ 
    foreach  $a \in \text{predict}(A \rightarrow \alpha)$ 
        add  $(A \rightarrow \alpha)$  to  $T[A, a]$ 

```

As células da tabela que fiquem vazias representam situações de erro sintático. As células da tabela que fiquem com dois ou mais produções representam situações de não determinismo: com base no *lookahead* usado não é possível escolher que produção usar na expansão. Uma gramática diz-se **LL(1)** se na tabela de decisão, para um *lookahead* de profundidade 1, não houver células com mais que uma produção. Equivalentemente, uma gramática diz-se **LL(1)** se para todas as produções com o mesmo símbolo à cabeça os seus conjuntos **predict** são disjuntos entre si.

Exemplo 2.1

Calcule a tabela de decisão de um reconhecedor preditivo para a gramática seguinte.

$$\begin{array}{l} S \rightarrow A B \\ A \rightarrow \varepsilon \mid a A \\ B \rightarrow \varepsilon \mid b B \end{array}$$

Resposta:

(Deixo ao cuidado do leitor o cálculo dos conjuntos ***predict***.)

$$\begin{aligned} \mathbf{predict}(S \rightarrow A B) &= \{a, b, \$\} \\ \mathbf{predict}(A \rightarrow \varepsilon) &= \{b, \$\} \\ \mathbf{predict}(A \rightarrow a A) &= \{a\} \quad l \\ \mathbf{predict}(B \rightarrow \varepsilon) &= \{\$\} \\ \mathbf{predict}(B \rightarrow b B) &= \{b\} \end{aligned}$$

symbol	lookahead		
	a	b	\$
S	$S \rightarrow A B$	$S \rightarrow A B$	$S \rightarrow A B$
A	$A \rightarrow a A$	$A \rightarrow \varepsilon$	$A \rightarrow \varepsilon$
B		$B \rightarrow b B$	$B \rightarrow \varepsilon$

2.2 Reconhecedores recursivo-descendentes

O reconhecimento preditivo, sintetizado na tabela de decisão apresentada acima, permite construir programas de reconhecimento, reconhecedores (ou *parsers* na terminologia em inglês). Uma solução para a construção do reconhecedor é baseada numa estrutura na qual cada símbolo não terminal da gramática dá origem a uma função, possivelmente recursiva.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 2.4.2, "Recursive-descent parsing".

Exemplo 2.2

Considere que com base na gramática e na tabela de decisão do exemplo anterior se contrói o programa seguinte.

```

function eat(token)
    if lookahead = token then
        lookahead := nextToken().
    else
        REJECT.

function S()
    A(), B().

function A()
    case lookahead in
        a : eat(a), A(), return .
        b, $ : return .

function B()
    case lookahead in
        a : REJECT.
        b : eat(b), B(), return .
        $ : return .

program Parser()
    lookahead := nextToken(). S().
    if lookahead = $ then
        ACCEPT.
    else
        REJECT.

```

Se executar o programa *Parser* quando a entrada é *aab* verificará que após o retorno da função *S* o *lookahead* é igual a $\$$, indicando que a palavra é válida. Mas a palavra *aba* é rejeitada, porque durante a execução a função *B* vai ser invocada numa altura em que o *lookahead* é igual a *a*. (Confirme.)

2.3 Reconhecedores descendentes não recursivos

Uma solução alternativa para implementar o reconhecedor preditivo usa uma pilha (*stack*) para reter o estado no processo de reconhecimento e usa a tabela de decisão para decidir como evoluir no processo de reconhecimento.

Seja $G = (T, N, P, S)$ uma gramática independente do contexto, que se assume seja LL(1). Seja M a tabela de decisão de G para um *lookahead* de profundidade 1. Finalmente, considere que dispõe de uma

pilha onde pode armazenar elementos do conjunto $Z = T \cup N \cup \{\$\}$ e que pode ser manipulada com as funções **push**, **pop** e **top**, que, respectivamente, coloca uma sequência de símbolos na pilha, retira o símbolo no topo da pilha e mostra qual o símbolo no topo sem o retirar. O programa seguinte é um reconhecedor das palavras da gramática G

```
program Parser()
  push(S $).
  lookahead = getToken().
  forever
    z := top().
    if z ∈ T then
      if z ≠ lookahead then
        REJECT.
      elseif z = $ then
        ACCEPT.
      else (* z = lookahead ∧ z ≠ $ *)
        pop() , lookahead = getToken().
    else (* z ∈ N *)
      α := M(z, lookahead).
      if α = ∅ then
        REJECT.
      else
        pop() , push(α).
```

A evolução do programa anterior no processo de reconhecimento pode ser captado por uma tabela onde se mostre a cada passo os estados da pilha e da entrada e a ação tomada. Se se considerar a gramática e a tabela de decisão do exemplo 2.1, a execução do programa anterior sobre a palavra *aab* resulta na seguinte tabela. Na coluna da pilha, o símbolo mais à direita é o que está no topo e, na coluna entrada, o símbolo mais à esquerda é o *lookahead*.

Pilha	Entrada	Ação
\$ S	a a b \$	pop() , push(A B)
\$ B A	a a b \$	pop() , push(a A)
\$ B A a	a a b \$	pop() , lookahead = getToken()
\$ B A	a b \$	pop() , push(a A)
\$ B A a	a b \$	pop() , lookahead = getToken()
\$ B A	b \$	pop()
\$ B	b \$	pop() , push(b B)
\$ B b	b \$	pop() , lookahead = getToken()
\$ B	\$	pop()
\$	\$	ACCEPT

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.4.4, "Nonrecursive predictive parsing".

2.4 Implementação de gramáticas de atributos

Os reconhecedores recursivo-descendentes (ver secção 2.2) permitem implementar facilmente gramáticas de atributos do tipo L (ver secção 4.2.2). As funções recursivas podem ter parâmetros de entrada e de saída. O valor de retorno pode também funcionar como parâmetro de saída. Os primeiros permitem passar informação da função chamadora para a função chamada, o que corresponde, na árvore de derivação, a definir atributos herdados dos nós filhos com base em atributos do nó pai. Os segundos permitem passar informação da função chamada para a função chamadora, o que corresponde ao suporte de atributos sintetizados e de atributos herdados de nós filhos com base em atributos de nós filhos à esquerda na árvore de derivação.

Capítulo 3

Análise sintáctica ascendente

NOTA PRÉVIA: Este capítulo não está completo e não foi devidamente revisto, pelo que, por um lado, há partes omissas e, por outro lado, pode/deve conter falhas.

Considere a gramática

$$\begin{array}{l} D \rightarrow T L ; \\ T \rightarrow i \mid r \\ L \rightarrow v \mid L , v \end{array}$$

que representa uma declaração de variáveis *a la C*. Como reconhecer a palavra “ $u = i v , v ;$ ” como pertencente à linguagem gerada pela gramática dada? Se u pertence à linguagem gerada pela gramática, então $D \Rightarrow^* u$. Tente-se chegar lá andando no sentido contrário ao de uma derivação, ie. de u para D .

$$\begin{array}{ll} & i v , v ; \\ \Leftarrow & T v , v ; \quad (\text{por aplicação da regra } T \rightarrow i) \\ \Leftarrow & T L , v ; \quad (\text{por aplicação da regra } L \rightarrow v) \\ \Leftarrow & T L ; \quad (\text{por aplicação da regra } L \rightarrow L , v) \\ \Leftarrow & D \quad (\text{por aplicação da regra } D \rightarrow T L ;) \end{array}$$

Colocando ao contrário

$$D \Rightarrow T L ; \Rightarrow T L , v ; \Rightarrow T v , v ; \Rightarrow i v , v ;$$

vê-se que corresponde a uma derivação à direita. A tabela seguinte mostra como, na prática, se realiza esta (retro)derivação.

pilha	entrada	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	\$	aceitação

Inicialmente, o topo da pilha apenas possui um símbolo especial, representado por um \$, que indica, quando no topo, que a pilha está vazia. A entrada possui a palavra a reconhecer seguida também de um símbolo especial, aqui também representado por um \$, que indica fim da entrada. Em cada ciclo realiza-se, normalmente, uma operação de deslocamento ou de redução. A operação de **deslocamento** (no inglês, *shift*) transfere o símbolo não terminal da entrada para o topo da pilha. A operação de **redução** (no inglês, *reduce*) substitui os símbolos do topo da pilha que correspondem ao corpo de uma produção da gramática pela cabeça dessa regra.

Se se atingir uma situação em que a entrada apenas possui o símbolo \$ e a pilha apenas possui o símbolo \$ e o símbolo inicial da gramática, tal como acontece na tabela anterior, a palavra é reconhecida como pertencendo à linguagem descrita pela gramática. Caso contrário, a palavra é rejeitada.

Veja-se a reação deste procedimento a uma entrada errada, por exemplo a palavra i v v ; .

pilha	entrada	ação
\$	i v v ; \$	deslocamento
\$ i	v v ; \$	redução por $T \rightarrow i$
\$ T	v v ; \$	deslocamento
\$ T v	v ; \$	rejeição

Com $T v$ na pilha e v na entrada é impossível chegar-se à aceitação. Porque se se reduzir v para L ficar-se-ia com um $T L$ na pilha e v na entrada, que não pertence ao conjunto $\text{follow}(L)$. Mais à frente voltaremos a este assunto.

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.1, "Reductions".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.2, "Handle pruning".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.3, "Shift-reduce parsing".

3.1 Conflitos

O procedimento acabado de descrever pode acarretar situações ambíguas chamadas **conflitos**. Considere a gramática

$$\begin{array}{lcl} S & \rightarrow & i \ c \ S \\ & | & i \ c \ S \in S \\ & | & a \end{array}$$

e execute o procedimento de reconhecimento com a palavra `i c i c a e a`. Obtém-se

pilha	entrada	ação
\$	i c i c a e a \$	deslocamento
\$ i	c i c a e a \$	deslocamento
\$ i c	i c a e a \$	deslocamento
\$ i c i	c a e a \$	deslocamento
\$ i c i c	a e a \$	deslocamento
\$ i c i c a	e a \$	redução por $S \rightarrow a$
\$ i c i c S	e a \$	conflito: redução usando $S \rightarrow i c S$ ou deslocamento para tentar $S \rightarrow i c S \in S$?

Na última linha é possível reduzir-se por aplicação da regra $S \rightarrow i c S$ ou deslocar-se o `e` para tentar posteriormente a redução pela regra $S \rightarrow i c S \in S$. Trata-se de um conflito deslocamento-redução (*shift-reduce conflict*). Perante este tipo de conflitos, ferramentas como o *bison* optam pelo deslocamento, mas pode não ser a mais adequada.

Também pode haver conflitos entre reduções (*reduce-reduce conflict*). Considere a gramática

$$\begin{array}{lcl} S & \rightarrow & A \\ & | & B \\ A & \rightarrow & c \\ & | & A \ a \\ B & \rightarrow & c \\ & | & B \ b \end{array}$$

e a palavra `c`. O procedimento de reconhecimento produz

pilha	entrada	ação
\$	c \$	deslocamento
\$ c	\$	conflito: redução usando $A \rightarrow c$ ou $B \rightarrow c$?

Na última linha é possível reduzir-se por aplicação das regras $A \rightarrow c$ ou $B \rightarrow c$. Perante este tipo de conflitos, ferramentas como o *bison* optam pela produção que aparece primeiro. Neste caso é irrelevante, mas pode não ser o adequado.

Veja-se agora a situação de um falso conflito. Considere a gramática

$$\begin{aligned} S &\rightarrow a \mid (S) \mid aP \mid (S)S \\ P &\rightarrow (S) \mid (S)S \end{aligned}$$

e reconheça-se a palavra “a (a) a”.

pilha	entrada	ação
\$	a (a) a \$	deslocamento
\$ a	(a) a \$	redução usando $S \rightarrow a$ deslocamento para tentar $S \rightarrow aP$?

Considerar a redução corresponde a realizar a retro-derivação

$$a (a) a \Leftarrow S (a) a$$

que não faz sentido porque $(\notin \text{follow}(S)$. Não há, portanto, conflito, sendo realizado o deslocamento.

pilha	entrada	ação
\$	a (a) a \$	deslocamento
\$ a	(a) a \$	deslocamento
\$ a (a) a \$	deslocamento
\$ a (a) a \$	redução por $S \rightarrow a$
\$ a (S) a \$	deslocamento
\$ a (S)	a \$	deslocamento, porque $a \notin \text{follow}(S), \text{follow}(P)$
\$ a (S) a	\$	redução por $S \rightarrow a$
\$ a (S) S	\$	redução por $P \rightarrow (S)S$
\$ a P	\$	redução por $S \rightarrow aP$
\$ S	\$	aceitação

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.5.4, "Conflicts during shift-reduce parsing".

Pode-se alterar uma gramática de modo a eliminar a fonte de conflito. Considerando que se pretendia optar pelo deslocamento, a gramática seguinte gera a mesma linguagem e está isenta de conflitos.

$$\begin{aligned} S &\rightarrow a \\ &\mid i \in S \\ &\mid i \in S' \in S \\ S' &\rightarrow a \\ &\mid i \in S' \in S' \end{aligned}$$

3.2 Construção de um reconhecedor ascendente

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.6, "Introduction to LR parsing: Simple LR".

O procedimento de reconhecimento apresentado atrás é um mecanismo iterativo. Em cada passo, a operação a realizar — deslocamento, redução, aceitação ou rejeição — depende da configuração nesse 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 —, mas a entrada é desconhecida, conhecendo-se apenas o próximo símbolo (*lookahead*). Então a decisão a tomar só pode basear-se no conteúdo da pilha e no *lookahead*.

Mas, se se quiser construir um reconhecedor apenas com capacidade de observar o topo da pilha, uma pilha onde se guardam os símbolos terminais e não terminais tem pouco interesse. Deve-se guardar um símbolo que represente tudo o que está para trás.

Como definir esses símbolos?

A associação de um símbolo diferente por cada configuração da pilha não serve porque a pilha pode, em geral, crescer de forma não limitada. Os símbolos a colocar na pilha devem representar estados no processo de deslocamento-redução. O alfabeto da pilha representa assim o conjunto de estados nesse processo de reconhecimento.

Cada estado representa um conjunto de itens. O item de uma produção representa uma fase no processo de obtenção dessa produção. É representado por uma produção com um ponto (.) numa posição do seu corpo. Por exemplo, a produção $A \rightarrow B_1 B_2 B_3$, produz 4 itens, a saber

$$\begin{aligned} A &\rightarrow \cdot B_1 B_2 B_3 \\ A &\rightarrow B_1 \cdot B_2 B_3 \\ A &\rightarrow B_1 B_2 \cdot B_3 \\ A &\rightarrow B_1 B_2 B_3 \cdot \end{aligned}$$

A produção $A \rightarrow \varepsilon$ produz um único item

$$A \rightarrow \cdot$$

Um item representa o quanto de uma produção já foi obtido e, simultaneamente, o quanto falta obter. Por exemplo, $A \rightarrow B_1 \cdot B_2 B_3$, significa que já foi obtido algo correspondente a B_1 , faltando obter o correspondente a $B_2 B_3$. Se o ponto(.) se encontra à direita, então poder-se-á reduzir $B_1 B_2 B_3$ a A .

3.2.1 Construção da coleção de conjuntos de itens

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 4.6.2, "Items and the LR(0) automaton".

Considere a gramática

$$\begin{array}{l} S \rightarrow E \\ E \rightarrow a \mid (E) \end{array}$$

O estado inicial (primeiro elemento da coleção de conjunto de itens) contém o item

$$Z_0 = \{S \rightarrow \cdot E \$\}$$

Este conjunto tem de ser fechado. O facto de o ponto (\cdot) se encontrar imediatamente à esquerda de um símbolo não terminal, significa que para se avançar no processo de reconhecimento é preciso obter esse símbolo. Isso é considerado juntando ao conjunto Z_0 os itens iniciais das produções cuja cabeça é E . Fazendo-o, Z_0 passa a

$$Z_0 = \{S \rightarrow \cdot E \$\} \cup \{E \rightarrow \cdot a, E \rightarrow \cdot (E)\}$$

Se nos novos elementos adicionados ao conjunto voltasse a acontecer de o ponto (\cdot) ficar imediatamente à esquerda de outros símbolos não terminais o processo deve ser repetido para esses símbolos.

O estado Z_0 pode evoluir por ocorrência de um E , um a ou um $($. Correspondem aos símbolos que aparecem imediatamente à direita do ponto (\cdot), e produzem 3 novos estados

$$\begin{array}{l} Z_1 = \delta(Z_0, E) = \{S \rightarrow E \cdot \$\} \\ Z_2 = \delta(Z_0, a) = \{E \rightarrow a \cdot\} \\ Z_3 = \delta(Z_0, ()) = \{E \rightarrow (\cdot E)\} \cup \{E \rightarrow \cdot a, E \rightarrow \cdot (E)\} \end{array}$$

Note que Z_3 foi estendido pela função de fecho, uma vez que o ponto ficou imediatamente à esquerda de um símbolo não terminal (E). Z_1 representa um situação de aceitação se o símbolo à entrada (*lookahead*) for igual a $\$$ e de erro caso contrário. Z_2 representa uma possível situação de redução pela regra $E \rightarrow a$. Esta redução só faz sentido se o símbolo à entrada (*lookahead*) for um elemento do conjunto **follow**(E). Caso contrário corresponde a uma situação de erro. Finalmente, Z_3 pode evoluir por ocorrência de um E , um a ou um $($, que correspondem aos símbolos que aparecem imediatamente à direita do ponto (\cdot). Estas evoluções são indicadas a seguir

$$\begin{array}{l} Z_4 = \delta(Z_3, E) = \{E \rightarrow (E \cdot)\} \\ \delta(Z_3, a) = Z_2 \\ \delta(Z_3, ()) = Z_3 \end{array}$$

Apenas um novo estado foi gerado (Z_4). Este estado apenas evolui por ocorrência de $)$.

$$Z_5 = \delta(Z_4, ()) = \{E \rightarrow (E) \cdot\}$$

Pondo tudo agrupado, a coleção de conjunto de itens é

$$\begin{array}{l} Z_0 = \{S \rightarrow \cdot E \$\} \cup \{E \rightarrow \cdot a, E \rightarrow \cdot (E)\} \\ Z_1 = \delta(Z_0, E) = \{S \rightarrow E \cdot \$\} \\ Z_2 = \delta(Z_0, a) = \{E \rightarrow a \cdot\} \\ Z_3 = \delta(Z_0, ()) = \{E \rightarrow (\cdot E)\} \cup \{E \rightarrow \cdot a, E \rightarrow \cdot (E)\} \\ Z_4 = \delta(Z_3, E) = \{E \rightarrow (E \cdot)\} \\ Z_5 = \delta(Z_4, ()) = \{E \rightarrow (E) \cdot\} \end{array}$$

3.2.2 Tabela de decisão para um reconhecimento ascendente

A coleção de conjuntos de itens (conjunto de estados) fornece a base para a construção de uma tabela usada no algoritmo de reconhecimento. A tabela de decisão é uma matriz dupla, em que as linhas são indexadas pelo alfabeto da pilha (coleção de conjuntos de itens) e as colunas são indexadas pelos símbolos terminais e não terminais da gramática. Representa simultaneamente duas funções, designadas ACTION e GOTO. A função ACTION tem como argumentos um estado (símbolo da pilha) e um símbolo terminal (incluindo o \$) e define a ação a realizar. Pode ser uma de *shift*, *reduce*, *accept* ou *error*. A função GOTO mapeia um estado e um símbolo não terminal num estado. É usada após uma operação de redução.

Veja-se um exemplo. Considerando a gramática e a coleção de conjunto de itens anteriores, obtém-se a seguinte tabela de decisão.

	a	()	\$	E
Z ₀	shift Z ₂	shift Z ₃			Z ₁
Z ₁				accept	
Z ₂			reduce E → a	reduce E → a	
Z ₃	shift Z ₂	shift Z ₃			Z ₄
Z ₄			shift Z ₅		
Z ₅			reduce E → (E)	reduce E → (E)	

- $Z = \{Z_0, Z_1, Z_2, Z_3, Z_4, Z_5\}$ é o alfabeto da pilha e foi obtida calculando a coleção de conjuntos de itens.
- **shift** Z_i , com $Z_i \in Z$, representa um deslocamento, no qual é consumido o símbolo à entrada e é feito o empilhamento (*push*) do símbolo Z_i .
- **reduce** $A \rightarrow \alpha$, onde $A \rightarrow \alpha$ é uma produção da gramática, representa uma redução, na qual são retirados da pilha tantos símbolo quantos os símbolos do corpo da regra.
- Os símbolos $Z_i \in Z$ na última coluna, representam os símbolos a empilhar após uma redução.
- **accept** representa a aceitação.
- As células vazias representam situações de erro.

3.2.3 Algoritmo de reconhecimento

O algoritmo seguinte mostra como se usa a tabela anterior. Nele, `top`, `push` e `pop` são funções de manipulação da pilha, com os significados habituais, e `lookahead` e `adv` são funções de manipulação da entrada que, respetivamente, devolve o próximo símbolo terminal à entrada e consome um símbolo.

Algoritmo 3.1

```

push( $Z_0$ )
forever
  if (top() ==  $Z_1$  && lookahead() == $)
    aceita a entrada como pertencendo à linguagem.
    acc = ACTION[top(), lookahead()]
  if (acc is shift  $Z_i$ )
    adv(); push( $Z_i$ );
  else if (acc is reduce  $A \rightarrow \alpha$ )
    pop | $\alpha$ | símbolos; push(GOTO[top(),  $A$ ] );
  else
    rejeita a entrada

```

A aplicação deste algoritmo à palavra “((a))” resulta na tabela seguinte. No preenchimento dessa tabela, optou-se por separar em duas linhas as operações de *pop* e *push* das ações de redução. Desta forma fica mais claro que o símbolo a empilhar resulta do símbolo no topo da pilha após os *pops*.

pilha	entrada	ação
Z_0	((a)) \$	shift Z_3
$Z_0 Z_3$	(a)) \$	shift Z_3
$Z_0 Z_3 Z_3$	a)) \$	shift Z_2
$Z_0 Z_3 Z_3 Z_2$)) \$	reduce $E \rightarrow a$
$Z_0 Z_3 Z_3$)) \$	push Z_4
$Z_0 Z_3 Z_3 Z_4$)) \$	shift Z_5
$Z_0 Z_3 Z_3 Z_4 Z_5$) \$	reduce $E \rightarrow (E)$
$Z_0 Z_3$) \$	push Z_4
$Z_0 Z_3 Z_4$) \$	shift Z_5
$Z_0 Z_3 Z_4 Z_5$	\$	reduce $E \rightarrow (E)$
Z_0	\$	push Z_1
$Z_0 Z_1$	\$	accept

Na redução com a produção $E \rightarrow a$ foi feito o *pop* de 1 símbolo (número de símbolos no corpo da produção), ficando, em consequência, um Z_3 no topo da pilha. O Z_4 que foi empilhado logo a seguir corresponde a $\text{table}[Z_3, E]$. Nas duas reduções com a produção $E \rightarrow (E)$ são feitos o *pop* de 3 símbolos, ficando, em consequência, um Z_3 no topo da pilha, no primeiro caso, e um Z_0 no segundo.

Capítulo 4

Gramática de atributos

NOTA PRÉVIA: Este capítulo é apenas um enumerado das secções do livro de referência ([Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition;](#)) que cobrem a matéria sobre gramáticas de atributos.

4.1 Definição de gramática de atributos

No contexto destes apontamentos considera-se *gramáticas de atributos* o que no livro de referência se designa por *syntax-directed definitions*.

[Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.1,](#) "Syntax-directed definitions".

4.1.1 Atributos herdados e atributos sintetizados

[Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.1.1,](#) "Inherited and synthesized attributes".

4.1.2 Construção de gramáticas de atributos

[Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.1.2,](#) "Evaluating an SDD at the nodes of a parse tree".

4.2 Ordem de avaliação dos atributos

4.2.1 Grafo de dependências

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.1, "Dependency graphs".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.2, "Ordering the evaluation of attributes".

4.2.2 Tipos de gramáticas de atributos

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.3, "S-attributed definitions".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.4, "L-attributed definitions".

Compilers: principles, techniques, and tools; Aho, Lam, Sethi and Ullman; 2nd edition; sec. 5.2.5, "Semantic rules with controlled side effects".

Compiladores:

Apontamentos sobre linguagens regulares

(ano letivo de 2022-2023)

Artur Pereira, Miguel Oliveira e Silva

Maio de 2023

Nota prévia

Este documento representa apenas um compilar de notas sobre matéria teórico-prática lecionada na unidade curricular “Compiladores”, sem pretensões, por isso, de ser um texto exaustivo. A sua leitura deve apenas ser encarada como ponto de partida e nunca como único elemento de estudo.

Conteúdo

1 Linguagens	1
1.1 Exemplos de linguagens	1
1.2 Elementos básicos sobre linguagens	2
1.3 Operações sobre palavras	3
1.4 Operações sobre linguagens	4
1.4.1 Reunião	4
1.4.2 Intersecção	4
1.4.3 Diferença e complementação	5
1.4.4 Concatenação, potência e fecho de Kleene	5
1.4.5 Propriedades das várias operações	6
2 Linguagens Regulares (LR) e Expressões Regulares (ER)	7
2.1 Definição de linguagem regular	7
2.2 Expressões regulares	8
2.3 Propriedades das expressões regulares	9
2.4 Simplificação notacional	10
2.5 Extensão notacional	11
2.6 Aplicação das expressão regulares	13
2.7 Exercícios	13

3 Gramáticas regulares (GR)	14
3.1 Definição de gramática regular	15
3.2 Operações sobre gramáticas regulares	15
3.2.1 Reunião de gramáticas regulares	15
3.2.2 Concatenação de gramáticas regulares	16
3.2.3 Fecho de Kleene de gramáticas regulares	17
3.3 Definição de gramática regular generalizada	18
4 Equivalência entre ER e GR	19
4.1 Conversão de ER em GR	19
4.1.1 Gramáticas regulares dos elementos primitivos	20
4.1.2 Algoritmo de conversão	20
4.2 Conversão de GR em ER	22
4.2.1 Algoritmo de conversão	22
5 Autómatos Finitos Deterministas (AFD)	24
5.1 Definição de autômato finito determinista	25
5.2 Linguagem reconhecida por um AFD	27
5.3 Projeto de um AFD	28
5.4 AFD reduzido	30
5.4.1 Algoritmo de redução de AFD	31
6 Autómatos Finitos Não Deterministas (AFND)	34
6.1 Definição de autômato finito não determinista	35
6.2 Árvore de caminhos	36
6.3 Linguagem reconhecida por um AFND	37
6.4 Fecho- ϵ	38
7 Equivalência entre AFD e AFND	39
7.1 Conversão de AFND em AFD	39

8 Operações sobre AFD e AFND	43
8.1 Reunião de autómatos finitos	43
8.2 Concatenação de autómatos finitos	46
8.3 Fecho de Kleene de autómatos finitos	48
8.4 Complementação de autómatos finitos	50
8.5 Intersecção de autómatos finitos	51
8.6 Diferença de autómatos	53
9 Equivalência entre ER e AF	54
9.1 Conversão de ER em AF	54
9.1.1 Autómatos dos elementos primitivos	55
9.1.2 Algoritmo de conversão	55
9.2 Conversão de AF em ER	57
9.2.1 Autómato finito generalizado	57
9.2.2 Algoritmo de conversão	58
10 Equivalência entre GR e AF	62
10.1 Conversão de AF em GR	63
10.2 Conversão de GR em AF	64

Capítulo 1

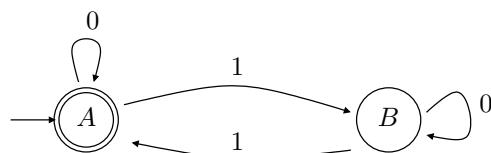
Linguagens

Uma linguagem é um sistema de símbolos usado para comunicar informação. Uma mensagem nessa linguagem é uma sequência de símbolos, mas nem todas as sequências são válidas. Logo, uma linguagem é caracterizada por um conjunto de símbolos e uma forma de descrever as sequências de símbolos válidas, ou seja, uma linguagem é um conjunto de sequências definidas sobre um conjunto de símbolos.

1.1 Exemplos de linguagens

- O **conjunto dos vocábulos em português** é uma linguagem, cujos símbolos são as letras do alfabeto (incluindo as letras acentuadas e cedilhadas) e cujas sequências válidas são os vocábulos do português. É uma linguagem finita, pelo que as sequências válidas podem ser descritas por extenso. Poderão também ser descritas através de regras de produção. Por exemplo, as formas verbais de todos os verbos regulares terminados em ar são iguais, a menos de um radical. Estas palavras podem ser definidas pelo produto cartesiano dos possíveis radicais com as possíveis terminações.
- O **conjunto das sequências binárias com um número par de uns** é uma linguagem, cujos símbolos são os dígitos binários (0 e 1) e cujas sequências válidas são aquelas cujo número de uns é par.

Como representar as sequências válidas? O grafo seguinte é uma forma de o fazer.



Considerando que representa um caminho sobre o grafo, uma sequência pertence à linguagem se, começando no nó A, percorrer um caminho sobre o grafo e terminar no nó A.

- O **conjunto dos políndromos definidos com as letras a e b** é uma linguagem, cujos símbolos são as letras a e b e cujas sequências válidas são as que se leem da mesma maneira se lidas da esquerda para a direita ou vice-versa. Esta linguagem pode ser definida por indução:
 1. a, b, aa e bb são políndromos;
 2. se s é um políndromo, então também o são aSa e bSb .
- O **conjunto das sequências binárias começadas por '1' e terminadas em '0'** é uma linguagem que pode ser representada pela expressão (regular) $1(0|1)^*0$. (As expressões regulares são tratadas mais à frente.)
- O **conjunto das sequências reconhecidas por uma máquina de calcular** é uma linguagem, cujos símbolos são os dígitos, o separador decimal (ponto ou vírgula), os operadores, etc., e cujas sequências válidas são aquelas que representam expressões aritméticas válidas. Por exemplo: $10+1$ é uma expressão válida, mas $10++1$ não o é.
- A **língua portuguesa** é uma linguagem, cujos símbolos são os vocábulos do português (mais os sinais de pontuação), e cujas sequências são os textos em português. É um conjunto infinito, pelo que as sequências válidas apenas podem ser descritas através de produções gramaticais, a gramática do português.

1.2 Elementos básicos sobre linguagens

- O **símbolo**, também designado por **letra**, é o átomo do mundo das linguagens.
- O **alfabeto** é o conjunto de símbolos de suporte a uma dada linguagem. Deve ser um conjunto finito, não vazio, de símbolos. Exemplos: $A_1 = \{0, 1, \dots, 9\}$ é o alfabeto do conjunto dos números inteiros positivos; $A_2 = \{0, 1\}$ é o alfabeto do conjunto dos números binários.
- A **palavra**, também designada por **string**, é uma sequência de símbolos sobre um dado alfabeto,

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

Exemplos:

- 00011 é uma palavra sobre o alfabeto $\{0, 1\}$;
- $abbbbcc$ é uma palavra sobre o alfabeto $\{a, b, c\}$.
- O **comprimento** de uma palavra u denota-se por $|u|$ e representa o seu número de símbolos.

- É habitual interpretar-se a palavra u como uma função

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

Se $u = abc$, então $u_1 = a$, $u_2 = b$ e $u_3 = c$.

- A **palavra vazia** é uma sequência de 0 (zero) símbolos e denota-se por ε .¹

Note que ε não pertence ao alfabeto.

- A^* representa o conjunto de todas as palavras sobre o alfabeto A , incluindo a palavra vazia. Por exemplo, se $A = \{0, 1\}$, então $A^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$.

Note que, dada um alfabeto A qualquer, qualquer **linguagem** sobre A mais não é do que um subconjunto de A^* . Note ainda que $\{\}$, $\{\varepsilon\}$ e A^* são subconjuntos de A , qualquer que seja o A , e por conseguinte são linguagens sobre o alfabeto A .

1.3 Operações sobre palavras

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 dos símbolos de v . Note que $|u.v| = |u| + |v|$.

- A concatenação goza da propriedade **associativa**: $u.(v.w) = (u.v).w = u.v.w$.
- A concatenação goza da propriedade **existência de elemento neutro**: $u.\varepsilon = \varepsilon.u = u$.

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, $\underbrace{uu \cdots u}_{n \times}$. Note que u^0 é igual a ε , qualquer que seja o u .

Prefixo de uma palavra u é uma sequência com parte dos símbolos iniciais de u ; **sufixo** de uma palavra u é uma sequência com parte dos símbolos finais de u ; **sub-palavra** de uma palavra u é uma sequência de parte dos símbolos intermédios de u . Note que ε e u são prefixos, sufixos e subpalavras de u , qualquer que seja o u .

O **reverso** de uma palavra u é a palavra, denotada por u^R , que se obtém invertendo a ordem dos símbolos de u , i.e., se $u = u_1u_2 \cdots u_n$ então $u^R = u_n \cdots u_2u_1$.

Denota-se por $\#(x, u)$ a função que devolve o número de ocorrências do símbolo x na palavra u .

¹Nas linguagens de programação, é habitual representar-se as *strings* entre aspas. Nestes casos uma *string* vazia é representada por " "

1.4 Operações sobre linguagens

O conjunto das linguagens definíveis sobre um alfabeto A é fechado sobre as seguintes operações: reunião, intersecção, diferença, complementação, concatenação, potenciação e fecho de Kleene. Apresentam-se a seguir as definições destas operações assim como algumas propriedades de que gozam. As operações serão ilustradas com exemplos, tomando como ponto de partida as linguagens L_a e L_b definidas sobre o alfabeto $A = \{a, b, c\}$ da seguinte maneira:

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

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

1.4.1 Reunião

A **reunião** de duas linguagens L_1 e L_2 denota-se por $L_1 \cup L_2$ e é definida da seguinte forma:

$$L_1 \cup L_2 = \{u \mid u \in L_1 \vee u \in L_2\} \quad (1.1)$$

Exemplo 1.1

Sendo $L_a = \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\}$ e $L_b = \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\}$

$$\begin{aligned} L_a \cup L_b &= \{u \mid u \text{ começa por } a \vee u \text{ termina com } a\} \\ &= \{xwy \mid w \in A^* \wedge (x = a \vee y = a)\} \cup \{a\} \\ &= \{w_1aw_2 \mid w_1, w_2 \in A^* \wedge (w_1 = \varepsilon \vee w_2 = \varepsilon)\} \end{aligned}$$

1.4.2 Intersecção

A **intersecção** de duas linguagens L_1 e L_2 denota-se por $L_1 \cap L_2$ e é definida da seguinte forma:

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

Exemplo 1.2

Sendo $L_a = \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\}$ e $L_b = \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\}$

$$\begin{aligned} L_a \cap L_b &= \{u \mid u \text{ começa por } a \wedge u \text{ termina com } a\} \\ &= \{awa \mid w \in A^*\} \cup \{a\} \end{aligned}$$

1.4.3 Diferença e complementação

A **diferença** entre duas linguagens L_1 e L_2 denota-se por $L_1 \setminus L_2$ e é definida da seguinte forma²:

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

Exemplo 1.3

Sendo $L_a = \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\}$ e $L_b = \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\}$

$$\begin{aligned} L_a \setminus L_b &= \{u \mid u \text{ começa por } a \wedge u \text{ não termina com } a\} \\ &= \{awx \mid w \in A^* \wedge x \in A \wedge x \neq a\} \end{aligned}$$

A linguagem **complementar** da linguagem L denota-se por \bar{L} e é definida da seguinte forma:

Exemplo 1.4

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

$$\begin{aligned} \bar{L} &= A^* \setminus L = \{u \mid u \notin L\} \\ &= \{u \mid u \text{ não começa por } a\} = \{xw \mid w \in A^* \wedge x \in A \wedge x \neq a\} \cup \{\varepsilon\} \end{aligned}$$

1.4.4 Concatenação, potência e fecho de Kleene

A **concatenação** de duas linguagens L_1 e L_2 denota-se por $L_1.L_2$ e é definida da seguinte forma:

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

Exemplo 1.5

Sendo $L_a = \{u \mid u \text{ começa por } a\} = \{aw \mid w \in A^*\}$ e $L_b = \{u \mid u \text{ termina com } a\} = \{wa \mid w \in A^*\}$

$$\begin{aligned} L_a.L_b &= \{uv \mid u \text{ começa por } a \wedge v \text{ termina com } a\} \\ &= \{awa \mid w \in A^*\} \end{aligned}$$

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

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

²Frequentemente, também é usada a notação $L_1 - L_2$

Exemplo 1.6

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

$$L^0 = \{\varepsilon\}$$

$$L^1 = L^0 \cdot L = \{\varepsilon\} \cdot L = L = \{aw \mid w \in A^*\}$$

$$L^2 = L^1 \cdot L = \{auav \mid u, v \in A^*\}$$

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

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

Exemplo 1.7

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

$$L^* = L^0 \cup L^1 \cup L^2 \cup \dots$$

$$= L^0 \cup L^1$$

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

Para perceber a simplificação anterior note que $L^{n+1} \subset L^n, n > 0$.

1.4.5 Propriedades das várias operações

Sendo as linguagens conjuntos, todas as propriedades aplicáveis a conjunto são naturalmente aplicáveis a linguagens. Assim, as operações de reunião e intersecção gozam das propriedades comutativa, associativa e distributiva.

As operações de reunião, intersecção e complementação gozam das **leis de De Morgan**:

$$L_1 - (L_2 \cup L_3) = (L_1 - L_2) \cap (L_1 - L_3)$$

$$L_1 - (L_2 \cap L_3) = (L_1 - L_2) \cup (L_1 - L_3)$$

A concatenação goza das propriedades:

associativa: $L_1 \cdot (L_2 \cdot L_3) = (L_1 \cdot L_2) \cdot L_3 = L_1 \cdot L_2 \cdot L_3$

existência de elemento neutro: $L \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L = L$

existência de elemento absorvente: $L \cdot \emptyset = \emptyset \cdot L = \emptyset$

distributiva em relação à reunião: $L_1 \cdot (L_2 \cup L_3) = L_1 \cdot L_2 \cup L_1 \cdot L_3$

distributiva em relação à intersecção: $L_1 \cdot (L_2 \cap L_3) = L_1 \cdot L_2 \cap L_1 \cdot L_3$

Capítulo 2

Linguagens Regulares (LR) e Expressões Regulares (ER)

2.1 Definição de linguagem regular

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

1. O conjunto vazio, \emptyset , é uma linguagem regular.
2. Qualquer que seja o $a \in A$, o conjunto $\{a\}$ ¹ é uma linguagem regular.
3. Se L_1 e L_2 são linguagens regulares, então $L_1 \cup L_2$ é uma linguagem regular.
4. Se L_1 e L_2 são linguagens regulares, então $L_1 \cdot L_2$ é uma linguagem regular.
5. Se L é uma linguagem regular, então L^* é uma linguagem regular.
6. Nada mais é linguagem regular.

Note que $\{\varepsilon\}$ é uma linguagem regular, uma vez que $\emptyset^* = \{\varepsilon\}$ e \emptyset é uma linguagem regular.

Exemplo 2.1

Mostre que o conjunto dos números binários começados em 1 e terminados em 0 é uma linguagem regular sobre o alfabeto $A = \{0, 1\}$.

Resposta: O conjunto pretendido pode ser representado por $L = \{1\} \cdot A^* \cdot \{0\}$. Este conjunto pode ser obtido indutivamente da seguinte forma:

1. $\{0\}$ e $\{1\}$ são regulares pela regra 2.
2. $A = \{0, 1\} = \{0\} \cup \{1\}$ é regular por aplicação da regra 3.

¹Note que o elemento do conjunto é uma palavra com uma letra e não uma letra. Se se fizesse uma analogia com linguagens de programação como o C/C++ ou o Java, o elemento do conjunto corresponde à palavra "a" e não à letra 'a'.

- 3. Se A é regular, A^* também o é por aplicação da regra 5.
- 4. Finalmente, $\{1\} \cdot A^* \cdot \{0\}$ é regular por aplicação 2 vezes da regra 4.

Exemplo 2.2

Mostre que o conjunto $N = \{0, 1, 2, 3, \dots, 10, \dots\}$, conjunto dos números inteiros positivos, é uma linguagem regular sobre o alfabeto $A = \{0, 1, 2, \dots, 9\}$.

Resposta: O conjunto pretendido corresponde ao conjunto formado pela palavra 0 mais todas as sequências de 1 ou mais dígitos decimais, começadas por um dígito diferente de 0. Ou seja, $N = \{0\} \cup A' \cdot A^*$, onde $A' = \{1, 2, \dots, 9\}$. N pode ser obtido indutivamente da seguinte forma:

- 1. Qualquer que seja o $d \in A$, $\{d\}$ é uma linguagem regular pela regra 2.
- 2. $A = \{0, 1, 2, \dots, 9\}$ é uma linguagem regular, por aplicação sucessiva da regra 3.
- 3. $A' = \{1, 2, \dots, 9\}$ é uma linguagem regular, por aplicação sucessiva da regra 3.
- 4. Se A é uma linguagem regular, A^* também o é, por aplicação da regra 5.
- 5. Se A' e A^* são linguagens regulares, $A' \cdot A^*$, é uma linguagem regular por aplicação da regra 4.
- 6. Finalmente, $N = \{0\} \cup A' \cdot A^*$ é uma linguagem regular por aplicação da regra 3.

Existem diversos formalismos para descrever/representar linguagens regulares. Iremos tratar os seguintes: **expressões regulares**, **gramáticas regulares**, **gramáticas regulares generalizadas**, **autômatos finitos deterministas**, **autômatos finitos não deterministas** e **autômatos finitos generalizados**, assim como os procedimentos de conversão de uns nos outros.

2.2 Expressões regulares

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

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

Note que é habitual representar-se por ε a expressão regular \emptyset^* . Esta expressão representa a linguagem regular formada apenas pela palavra vazia, ($\{\varepsilon\}$).

Exemplo 2.3

Obtenha uma expressão regular que represente o conjunto N do exemplo anterior (exemplo 2.2).

Resposta: Uma expressão regular que representa N é

$$e = 0|((((((1|2)|3)|4)|5)|6)|7)|8)|9)((((((0|1)|2)|3)|4)|5)|6)|7)|8)|9)*$$

Esta expressão tem a forma $e_1|(e_2.e_3*)$, com $e_1 = 0$, $e_2 = ((((((1|2)|3)|4)|5)|6)|7)|8)|9)$ e $e_3 = (((((0|1)|2)|3)|4)|5)|6)|7)|8)|9)$. A expressão e_1 representa o elemento 0. A expressão $e_2.e_3*$ representa as sequências começadas por um dígito diferente de 0. A expressão

$$e = 0|(1|2|\dots|9)(0|1|2|\dots|9)*$$

representa o mesmo conjunto e é claramente mais legível que a anterior. No entanto, a sua utilização só é válida por causa de propriedades de que gozam os operadores das expressões regulares e que serão apresentados a seguir.

2.3 Propriedades das expressões regulares

A operação de escolha ($|$) goza das propriedades **associativa**, **comutativa**, **existência de elemento neutro** e **idempotência**.

- As propriedades **associativa** e **comutativa** estabelecem respetivamente que

$$e_1|(e_2|e_3) = (e_1|e_2)|e_3 = e_1|e_2|e_3$$

e que

$$e_1|e_2 = e_2|e_1$$

- A expressão vazia, representando o conjunto vazio, é o **elemento neutro** da operação de escolha

$$e_1|\emptyset = \emptyset|e_1 = e_1$$

- A operação de escolha goza ainda de **idempotência**

$$e_1|e_1 = e_1$$

A operação de concatenação goza das propriedades **associativa**, **existência de elemento neutro** e **existência de elemento absorvente**.

- A propriedade **associativa** estabelece que

$$e_1(e_2e_3) = (e_1e_2)e_3 = e_1e_2e_3$$

2. A palavra vazia, representando o conjunto formado simplesmente pela palavra vazia ($\{\varepsilon\}$) é o **elemento neutro** da concatenação

$$e_1\varepsilon = \varepsilon e_1 = e_1$$

3. A expressão vazia, representando o conjunto vazio, é o **elemento absorvente** da concatenação

$$e_1\emptyset = \emptyset e_1 = \emptyset$$

Finalmente, os operadores de escolha e concatenação gozam das propriedades **distributiva à esquerda da concatenação em relação à escolha**

$$e_1(e_2|e_3) = e_1e_2|e_1e_3$$

e **distributiva à direita da concatenação em relação à escolha**

$$(e_1|e_2)e_3 = e_1e_3|e_2e_3$$

Note que o fecho de Kleene não goza da propriedade distributiva. Quer isto dizer que em geral

$$(e_1|e_2)^* \neq e_1^*|e_2^*$$

e

$$(e_1e_2)^* \neq e_1^*e_2^*$$

2.4 Simplificação notacional

Na escrita de expressões regulares assume-se que a operação de fecho (*) tem precedência em relação à operação de concatenação e esta tem precedência em relação à operação de escolha (|). O uso destas precedências, aliado às propriedades apresentadas, permite a queda de alguns parêntesis e consequentemente uma notação simplificada. A expressão final do exemplo 2.3 é um exemplo desta notação simplificada.

Exemplo 2.4

Determine uma expressão regular que represente o conjunto das sequências binárias em que o número de 0 (zeros) é igual a 2.

Resposta: Não fazendo uso da notação simplificada apresentada atrás, obter-se-ia, por exemplo, a expressão regular

$$e = (((((1^*)0)(1^*))0)(1^*))$$

A aplicação das simplificações resulta em

$$1^*01^*01^*$$

Exemplo 2.5

Sobre o alfabeto $A = \{a, b, c\}$ construa uma expressão regular que reconheça a linguagem L , onde $\#(a, w)$ é uma função que devolve o número de ocorrências da letra a na palavra w ,

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

Resposta: A expressão regular pretendida é

$$e = (b|c)^* a (b|c)^* a (b|c)^* a (b|c)^*$$

2.5 Extensão notacional

Mesmo com as simplificações notacionais apresentadas, por vezes, uma expressão regular pode ser difícil de entender. Isto é particularmente verdade quando os alfabetos de entrada são extensos.

Exemplo 2.6

Repita o exemplo 2.5 considerando que o alfabeto de entrada é o conjunto das letras minúsculas.

Resposta: Para construir a expressão regular pretendida basta substituir na resposta do exemplo 2.5 cada ocorrência de $(b|c)$ por $(b|c|d| \dots |y|y|z)$, obtendo-se

$$e = e_1^* = ((b|c|d| \dots |y|z)^* a (b|c|d| \dots |y|z)^* a (b|c|d| \dots |y|z)^* a (b|c|d| \dots |y|z)^*)^*$$

Embora com um grau de complexidade igual ao do exemplo 2.5 esta expressão regular é mais difícil de ler.

Considere-se um exemplo onde esta dificuldade de leitura é ainda mais patente.

Exemplo 2.7

Considerando que o alfabeto de entrada é o conjunto das letras maiúsculas e minúsculas, construa uma expressão regular que represente o conjunto das palavras com um número par de vogais.

Resposta: A expressão regular pretendida é dada por

$$\begin{aligned} &((b|c|d|f| \dots |y|z|B|C|D|F| \dots |Y|Z) \\ &\quad |(a|e|i|o|u|A|E|I|O|U)(b|c|d|f| \dots |y|z|B|C|D|F| \dots |Y|Z) * (a|e|i|o|u|A|E|I|O|U)) * \end{aligned}$$

Torna-se, por isso, necessário definir métodos de representação das expressões regulares que possuam maior poder expressivo. Um dos métodos usados faz uso de várias extensões notacionais. Ir-se-ão considerar as seguintes extensões notacionais:

1. O operador sufixo $+$ é usado para representar 1 ou mais ocorrências da expressão regular a que se aplica. Assim,

$$e^+ \equiv ee^*$$

2. O operador sufixo $?$ é usado para representar zero ou uma ocorrência da expressão a que se aplica. Assim,

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

3. O símbolo $.$ é usado para representar um símbolo qualquer do alfabeto². Por exemplo, sobre o alfabeto das letras minúsculas,

$$\cdot \equiv (a|b|c|\cdots|y|z)$$

4. A expressão $[x_1x_2x_3\cdots x_n]$ é usada para representar um (note bem, **um e um só**) símbolo do conjunto $\{x_1, x_2, x_3, \dots, x_n\}$. Por exemplo, a expressão regular $[aeiouAEIOU]$ representa uma vogal, maiúscula ou minúscula.

A construção anterior permite ainda especificar gamas de símbolos usando um intervalo de valores. A expressão $[x_i - x_f]$ representa um símbolo do alfabeto entre x_i e x_f . Por exemplo, $[a-z]$ representa uma letra minúscula.

É possível construir expressões juntando gamas e símbolos considerados individualmente. Por exemplo, a expressão $[a-zA-Z0-9_]$ representa uma letra, maiúscula ou minúscula, um algarismo decimal ou o símbolo ‘ $_$ ’, ou seja, representa um símbolo do conjunto $\{a, b, \dots, y, z, A, B, \dots, Y, Z, 0, 1, \dots, 8, 9, _\}$.

5. A expressão $[^x_1x_2x_3\cdots x_n]$ pode ser usada para representar um símbolo do conjunto complementar do conjunto $\{x_1, x_2, x_3, \dots, x_n\}$. Por exemplo, sobre o alfabeto das letras minúsculas e maiúsculas a expressão $[^aeiouAEIOU]$ representa uma consoante³.

6. A expressão $e\{n\}$ representa n concatenações da expressão e , ou seja, e^n .
7. A expressão $e\{n_1, n_2\}$ representa entre n_1 e n_2 concatenações da expressão e , ou seja, a expressão $e^{n_1}|e^{n_1+1}|\cdots|e^{n_2}$.
8. A expressão $e\{n,\}$ representa n ou mais concatenações da expressão e a expressão $e^n|e^{n+1}|\cdots$.

Note, no entanto, que as extensões notacionais disponíveis podem variar dependendo da ferramenta usada⁴.

²Algumas ferramentas, como a linguagem lexical `flex`, excluem do $.$ a mudança de linha.

³Note que o que é dito apenas é verdade sobre o alfabeto das letras. A mesma expressão terá um significado diferente se o alfabeto for outro. Por exemplo, nas linguagens que aceitam expressões regulares, o alfabeto é todo o código ASCII. Nesses casos, a expressão representaria qualquer símbolo à exceção dos das vogais.

⁴Por exemplo, em *ANTLR*, as extensões 5 a 8 não existem nessa forma. A 5 aparece na forma $\sim[\cdots]$. As outras aparecem na forma de predicados semânticos.

2.6 Aplicação das expressão regulares

- **Validação** de entrada, por exemplo em formulários.
- **Procura e seleção** de texto. Ferramentas como o word e o libre office têm suporte de expressões regulares.
- **Tokenization**, que corresponde à transformação de uma sequência de caracteres numa sequência de *tokens*, parte do processo de compilação.

2.7 Exercícios

Exercício 2.1

Determine uma expressão regular que represente as sequências binárias com um número par de uns.

Exercício 2.2

Determine uma expressão regular que represente as constantes numéricas na linguagem C. Eis alguns exemplos de constantes numéricas em C: “10”, “10.1”, “10.”, “.12”, “1e5”, “10.1E+4”.

Capítulo 3

Gramáticas regulares (GR)

Considere uma estrutura G , definido sobre o alfabeto $T = \{a, b\}$, com as regras de rescrita

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

significando que em qualquer sequência onde apareça o símbolo S ele pode ser substituído (rescrito) por b ou por $a \ S$. É habitual usar-se o símbolo ‘|’ para denotar as várias alternativas de rescrita associadas ao mesmo símbolo. Assim, a estrutura anterior é comumente representada por

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

Que palavras apenas constituídas por símbolos do alfabeto T se podem gerar a partir de S ? Substituindo S por b obtém-se b que pertence a T^* . Isto pode representar-se por

$$S \Rightarrow b$$

Substituindo S por aS e o novo S por b , obtém-se ab , escrevendo-se

$$S \Rightarrow a \ S \Rightarrow a \ b$$

Na realidade, começando em S podem gerar-se as palavras b , ab , aab , $aa \cdots ab$, ou seja, todas as palavras da linguagem L dada por

$$L = \{a^n b \mid n \geq 0\}$$

ou, equivalentemente, as palavras descritas pela expressão regular a^*b .

A estrutura G é uma **gramática** que permite gerar a linguagem L . T é o alfabeto de entrada, também designado por conjunto de símbolos terminais, ou simplesmente conjunto de terminais. S é um símbolo não-terminal, ponto de partida de todas as palavras. É designado **símbolo inicial da gramática**. As regras de rescrita são designadas **produções**. O conjunto das palavras que se podem gerar a partir do símbolo inicial da gramática por aplicação sucessiva de produções é a linguagem descrita por G .

As gramáticas podem pertencer a diferentes categorias, dependendo da definição das suas produções. Para já, apenas nos interessa abordar uma dessas categorias, as **gramáticas regulares**.

3.1 Definição de gramática regular

Formalmente, 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, cada uma da forma $\alpha \rightarrow \beta$, onde
 - $\alpha \in N$
 - $\beta \in T^* \cup (T^*N)$
- $S \in N$ é o símbolo inicial.

Nas produções, α e β são designados por **cabeça da produção** e **corpo da produção**, respectivamente. Note que os corpos das produções numa gramática regular ou apenas têm símbolos terminais (zero ou mais) ou têm apenas um símbolo não terminal, necessariamente no fim.

3.2 Operações sobre gramáticas regulares

As gramáticas regulares são fechadas sob as operações de reunião, concatenação, fecho, intersecção e complementação.

3.2.1 Reunião de gramáticas regulares

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

$$\begin{aligned} T &= T_1 \cup T_2 \\ N &= N_1 \cup N_2 \cup \{S\} \quad \text{com} \quad S \notin (N_1 \cup N_2) \\ P &= \{S \rightarrow S_1, S \rightarrow S_2\} \cup P_1 \cup P_2 \end{aligned}$$

é regular e gera a linguagem $L = L(G_1) \cup L(G_2)$. As novas produções, $S \rightarrow S_1$ e $S \rightarrow S_2$, permitem gerar, respectivamente, as palavras da primeira e segunda gramáticas, contribuindo dessa forma para a reunião.

Exemplo 3.1

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

$$L_1 = \{\omega a : \omega \in T^*\} \quad (\text{palavras terminadas em } a)$$

e

$$L_2 = \{a\omega : \omega \in T^*\} \quad (\text{palavras começadas por } a)$$

Resposta:

Comece-se por determinar as gramáticas para as linguagens L_1 e L_2

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

A aplicação do algoritmo da reunião resulta em

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

3.2.2 Concatenação de gramáticas regulares

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

$$\begin{aligned} T &= T_1 \cup T_2 \\ N &= N_1 \cup N_2 \cup \{S\} \quad \text{com } S \notin (N_1 \cup N_2) \\ P &= \{A \rightarrow \omega S_2 : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^*\} \\ &\quad \cup \{A \rightarrow \omega : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^* N_1\} \\ &\quad \cup P_2 \\ S &= S_1 \end{aligned}$$

é regular e gera a linguagem $L = L(G_1).L(G_2)$, concatenação de $L(G_1)$ com $L(G_2)$. As produções de G_1 apenas constituídas por símbolos terminais transitam para G sendo-lhes acrescentado no fim o símbolo inicial de G_2 (S_2). As restantes produções de G_1 (aqueles que terminal num símbolo não terminal) e todas as produções de G_2 transitam inalteradas. O símbolo inicial da concatenação corresponde ao (ou transforma-se diretamente no) símbolo inicial de G_1 .

Exemplo 3.2

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

$$L_1 = \{\omega a : \omega \in T^*\} \quad (\text{palavras terminadas em } a)$$

e

$$L_2 = \{a\omega : \omega \in T^*\} \quad (\text{palavras começadas por } a)$$

Resposta:

As linguagens L_1 e L_2 são as mesmas do exemplo anterior. Apresentam-se novamente aqui as suas gramáticas por conveniência.

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

A aplicação do algoritmo da concatenação resulta em

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

A única produção que transitou alterada foi a $S_1 \rightarrow a$, que passou a $S_1 \rightarrow a S_2$.

3.2.3 Fecho de Kleene de gramáticas regulares

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

$$\begin{aligned} T &= T_1 \\ N &= N_1 \cup \{S\} \text{ com } S \notin N_1 \\ P &= \{S \rightarrow \varepsilon, S \rightarrow S_1\} \\ &\cup \{A \rightarrow \omega S : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^*\} \\ &\cup \{A \rightarrow \omega : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^* N_1\} \end{aligned}$$

é regular e gera a linguagem $L = (L(G_1))^*$. O fecho corresponde a concatenações da linguagem com ela própria. Assim, as produções de G_1 apenas constituídas por símbolos terminais ganham o novo símbolo inicial (S) no fim. As restantes produções de G_1 (aqueles que terminam num símbolo não terminal) mantêm-se inalteradas. O novo símbolo inicial define o ponto de fuga do processo recursivo.

Exemplo 3.3

Sobre o conjunto de terminais $T = \{a, b, c\}$, determine uma gramática regular que represente a linguagem $L = L_1^*$, sabendo que

$$L_1 = \{\omega a : \omega \in T^*\} \quad (\text{palavras terminadas em } a)$$

Resposta: A linguagem L_1 já foi abordada anteriormente. Apresenta-se novamente aqui a sua gramática por conveniência.

$$\begin{array}{lcl} S_1 & \rightarrow & a \ S_1 \\ & | & b \ S_1 \\ & | & c \ S_1 \\ & | & a \end{array}$$

A aplicação do algoritmo de fecho resulta em

$$\begin{array}{ll} S & \rightarrow \varepsilon \\ & | \quad S_1 \end{array} \qquad \begin{array}{lcl} S_1 & \rightarrow & a \ S_1 \\ & | & b \ S_1 \\ & | & c \ S_1 \\ & | & a \ S \end{array}$$

3.3 Definição de gramática regular generalizada

Para suporte aos procedimentos de conversão de gramáticas regulares em expressões regulares, apresentados noutro capítulo, mas não só, é possível generalizar a definição de gramática regular.

Formalmente, uma gramática regular generalizada 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, cada uma da forma $\alpha \rightarrow \beta$, onde
 - $\alpha \in N$
 - $\beta \in E(T) \cup E(T)N$
- $S \in N$ é o símbolo inicial.

onde $E(T)$ representa o conjunto das expressões regulares definidas sobre o alfabeto T .

Note que esta definição abrange a dada anteriormente para gramática regular, uma vez que T^* representa um subconjunto das expressões regular sobre o alfabeto T .

Capítulo 4

Equivalência entre Expressões Regulares e Gramáticas Regulares

As expressões regulares e as gramáticas regulares são equivalentes, no sentido em que descrevem a mesma classe de linguagens, as linguagens regulares. Assim sendo, é possível converter uma expressão regular dada numa gramática regular que represente a mesma linguagem. Inversamente, é também possível converter uma gramática regular dada numa expressão regular descrevendo a mesma linguagem.

4.1 Conversão de uma expressão regular numa gramática regular

Como vimos na sua definição, uma expressão regular é construída à custa de elementos primitivos e dos operadores escolha ($|$), concatenação ($.$) e fecho ($*$). Os elementos primitivos correspondem à expressão \emptyset , representando o conjunto vazio, e às expressões do tipo a com $a \in A$, sendo A o alfabeto. É habitual considerar a expressão ε como um elemento primitivo, embora se saiba que $\varepsilon = \emptyset^*$.

É possível decompor-se uma expressão regular num conjunto de elementos primitivos interligados pelos operadores referidos acima. Dada uma expressão regular qualquer ela é:

1. um elemento primitivo;
2. ou uma expressão do tipo e^* , sendo e uma expressão regular qualquer;
3. ou uma expressão do tipo $e_1.e_2$, sendo e_1 e e_2 duas expressões regulares quaisquer;
4. ou uma expressão do tipo $e_1|e_2$, sendo e_1 e e_2 duas expressões regulares quaisquer;

Se se identificar as gramáticas regulares equivalentes das expressões primitivas, tem-se o problema da conversão de uma expressão regular para uma gramática regular resolvido, visto que se sabe como fazer a união, concatenação e fecho de gramáticas regulares (ver secção 3.2).

4.1.1 Gramáticas regulares dos elementos primitivos

A tabela seguinte mostra as gramáticas regulares correspondentes às expressões regulares ε e a , sendo a um símbolo qualquer do alfabeto.

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

4.1.2 Algoritmo de conversão

A transformação de uma expressão regular numa gramática regular pode ser feita aplicando os seguintes passos:

1. Se a expressão regular é o do tipo primitivo, a gramática regular correspondente pode ser obtida da tabela anterior.
2. Finalmente, se é do tipo $e_1|e_2$, aplica-se este mesmo algoritmo na obtenção de gramáticas regulares para as expressões e_1 e e_2 e, de seguida, aplica-se a união de gramáticas regulares descrita na secção 3.2.1.
3. Se é do tipo $e_1.e_2$, aplica-se este mesmo algoritmo na obtenção de gramáticas regulares para as expressões e_1 e e_2 e, de seguida, aplica-se a concatenação de gramáticas regulares descrita na secção 3.2.2.
4. Se é do tipo e^* , aplica-se este mesmo algoritmo na obtenção de uma gramática regular equivalente à expressão regular e e, de seguida, aplica-se o fecho de gramáticas regulares descrito na secção 3.2.3.

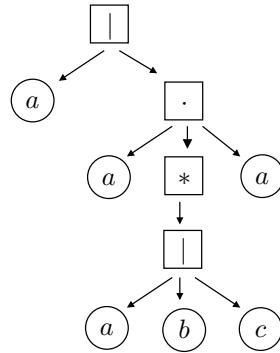
Exemplo 4.1

Construa uma gramática regular equivalente à expressão regular $e = a|a(a|b|c)^*a$.

Resposta:

A expressão regular e é do tipo $e_1|e_2$, com $e_1 = a$ e $e_2 = a(a|b|c)^*a$, pelo que se deve aplicar a regra 2 do algoritmo. A expressão e_1 é do tipo primitivo, pelo que se pode aplicar a tabela para obter a gramática regular correspondente. A expressão e_2 resulta da concatenação de 3 expressões regulares, para as quais temos de obter gramáticas regulares. Apenas a segunda expressão da concatenação

tem de ser considerada, visto já se ter as gramáticas regulares das outras. Tem-se então que converter a expressão regular $e_3 = (a|b|c)^*$, que se obtém do fecho sobre a gramática regular que represente a expressão $e_4 = a|b|c$. Esta, por sua vez, resulta da reunião das gramáticas regulares das expressões primitivas a , b e c . Esta decomposição pode ser visualizada na figura seguinte



Uma vez definida a decomposição, a construção faz-se dos elementos primitivos para a expressão global. As gramáticas regulares para as expressões a , b e c são

$$S_a \rightarrow a \quad S_b \rightarrow b \quad S_c \rightarrow c$$

Fazendo a sua reunião obtém-se a gramática regular para e_4

$$S_4 \rightarrow S_a \mid S_b \mid S_c$$

$$S_a \rightarrow a$$

$$S_b \rightarrow b$$

$$S_c \rightarrow c$$

que pode ser simplificada para

$$S_4 \rightarrow a \mid b \mid c$$

Aplicando o fecho a esta gramática obtém-se a gramática regular para a expressão e_3

$$S_3 \rightarrow \varepsilon \mid S_4$$

$$S_4 \rightarrow a \ S_3 \mid b \ S_3 \mid c \ S_3$$

que pode ser simplificada para

$$S_3 \rightarrow \varepsilon \mid a \ S_3 \mid b \ S_3 \mid c \ S_3$$

A gramática regular para e_2 resulta da concatenação das gramáticas regulares para a , e_3 e a , obtendo-se, após alguma simplificação, a gramática regular seguinte

$$S_2 \rightarrow a \ S_3$$

$$S_3 \rightarrow a \mid a \ S_3 \mid b \ S_3 \mid c \ S_3$$

Finalmente, a gramática regular para e obtém-se da reunião das gramáticas regulares para e_2 e a , o que resulta na gramática regular seguinte

$$S \rightarrow a \mid S_2$$

$$S_2 \rightarrow a \ S_3$$

$$S_3 \rightarrow a \mid a \ S_3 \mid b \ S_3 \mid c \ S_3$$

que pode ser transformada para

$$S \rightarrow a \mid a \ S_3$$

$$S_3 \rightarrow a \mid a \ S_3 \mid b \ S_3 \mid c \ S_3$$

4.2 Conversão de uma gramática regular numa expressão regular

A conversão de gramáticas regulares em expressões regulares é feita através de gramáticas regulares generalizadas, com uma pequena alteração na forma como são usadas. Cada produção da forma $X \rightarrow \beta$, com $\beta \in T^*$, é representada pelo triplo (X, β, ε) . Por outro lado, cada produção da forma $X \rightarrow \beta Y$, com $\beta \in T^*$ e $Y \in N$, é representada pelo triplo (X, β, Y) .

Convertendo inicialmente uma gramática regular para este formato e aplicando sucessivamente transformações de equivalência, o conjunto de triplos pode ser reduzido à forma (E, e, ε) , sendo e a expressão regular equivalente à gramática ponto de partida.

4.2.1 Algoritmo de conversão

Assim sendo, a obtenção de uma expressão regular equivalente a uma gramática regular qualquer dada, pode ser feita através dos seguintes passos:

1. Conversão de uma gramática (T, N, P, S) no conjunto de triplo seguintes

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

O triplo (E, ε, S) corresponde ao acrescento à gramática de uma nova produção $E \rightarrow \varepsilon S$, de forma a garantir que o (novo) símbolo inicial não aparece no corpo de nenhuma produção. Por conseguinte, $E \notin N$.

2. Eliminação, um a um, por transformações de equivalência, de todos os símbolos de N , até se obter um único triplo da forma (E, e, ε) .

A eliminação dos estados intermédios pode ser realizada pelo procedimento seguinte:

1. Substituição de todos os triplos da forma (A, α_i, A) , com $A \in N$, por um único (A, ω_2, A) , onde $\omega_2 = \alpha_1 | \alpha_2 | \dots | \alpha_m$
2. Substituição de todos os triplos da forma (A, β_i, B) , com $A, B \in N$, por um único (A, ω_1, B) , onde $\omega_1 = \beta_1 | \beta_2 | \dots | \beta_n$
3. Substituição de cada triplo 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. Repetição dos passos anteriores enquanto houver símbolos intermédios

Exemplo 4.2

Obtenha uma expressão regular equivalente à gramática regular seguinte

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

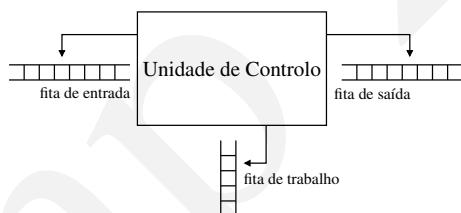
Resposta:

- Começa-se por converter a gramática num conjunto de triplos, acrescentando o triplo inicial
 $\{(E, \varepsilon, S), (S, a, S), (S, c, S), (S, aba, X), (X, a, X), (X, c, X), (X, \varepsilon, \varepsilon)\}$
- Substitui-se (S, a, S) e (S, c, S) por $(S, (a|c), S)$
 $\{(E, \varepsilon, S), (S, (a|c), S), (S, aba, X), (X, a, X), (X, c, X), (X, \varepsilon, \varepsilon)\}$
- Substitui-se (X, a, X) e (X, c, X) por $(X, (a|c), X)$
 $\{(E, \varepsilon, S), (S, (a|c), S), (S, aba, X), (X, (a|c), X), (X, \varepsilon, \varepsilon)\}$
- Substitui-se (E, ε, S) , $(S, (a|c), S)$ e (S, aba, X) por $(E, (a|c)^*aba, X)$
 $\{(E, (a|c)^*aba, X), (X, (a|c), X), (X, \varepsilon, \varepsilon)\}$
- Finalmente, substitui-se $(E, (a|c)^*aba, X)$, $(X, (a|c), X)$ e $(X, \varepsilon, \varepsilon)$ por $(a|c)^*aba(a|c)^*$
 $\{(E, ((a|c)^*aba(a|c)^*), \varepsilon)\}$
- Obtendo-se a expressão regular pretendida
 $(a|c)^*aba(a|c)^*$

Capítulo 5

Autómatos Finitos Deterministas (AFD)

Um *autómato finito* é um mecanismo reconhecedor das palavras de uma linguagem. Dada uma linguagem L , definida sobre um alfabeto A , um autómato finito reconhecedor de L é um mecanismo que reconhece as palavras de A^* que pertencem a L . Genericamente um autómato finito tem a configuração representada na figura seguinte.



Uma unidade de controlo, com capacidade finita de memorização, manipula os símbolos recolhidos de uma fita de entrada, que armazena uma palavra, e produz uma resposta. Definem-se vários tipos de autómatos, dependendo da forma como a fita de entrada é acedida e da existência ou inexistência de fitas de trabalho e de fita de saída.

A fita de entrada é uma unidade só de leitura com acesso sequencial ou aleatório aos símbolos da palavra. No acesso sequencial um símbolo da palavra de entrada lido e processado não pode ser processado novamente. Se assumirmos que a fita de entrada possui uma cabeça de leitura, esta apenas pode avançar posição a posição. No acesso aleatório o mesmo símbolo pode ser lido mais que uma vez, ou seja, assume-se que a cabeça de leitura pode ser deslocada para a frente ou para trás livremente.

Em geral, a resposta dos autómatos é do tipo sim/não ou aceito/rejeito. Quer isto dizer que um autómato não possui propriamente uma fita de saída, limitando-se a produzir um **aceito** se $u \in L$ e um **rejeito** caso contrário. Neste sentido funcionam efetivamente como *reconhecedores* das palavras de uma linguagem. No entanto, há autómatos que produzem um símbolo de saída por cada símbolo de entrada processado. Nestes casos existe uma fita de saída que representa uma unidade só de escrita com acesso sequencial. Um símbolo de saída uma vez escrito não poderá ser apagado ou rescrito.

Há autómatos em que a unidade de controlo recorre a fitas de trabalho para armazenar informação que auxilie no processamento. Estas fitas têm um funcionamento tipo pilha, ou seja, a ordem de colocação de elementos na fita é contrária à de retirada.

No resto deste capítulo ir-se-ão estudar categorias de autómatos caracterizadas pelo facto de possuírem uma fita de entrada com acesso sequencial e por não possuirem fitas de trabalho. Mais especificamente, ir-se-ão cobrir os **autómatos finitos deterministas**, os **autómatos finitos não-deterministas** e os **autómatos finitos generalizados**, que apenas produzem uma saída do tipo *aceito/rejeito*. Todos eles correspondem a modelos computacionais que representam linguagens regulares.

5.1 Definição de autómato finito determinista

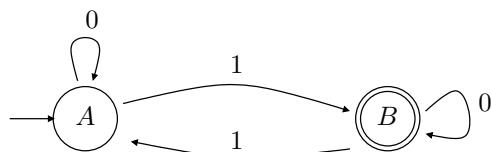
Um **autómato finito determinista** (AFD) é um quíntuplo $M = (A, Q, q_0, \delta, F)$, em que:

- 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 um conjunto dos estados de aceitação.

Graficamente um AFD pode ser representado por um grafo dirigido, onde os nós correspondem aos estados e as setas às transições. Visualmente, é habitual usarem-se círculos para representar os estados. Os círculos correspondentes aos estados de aceitação são diferenciados usando-se riscos duplos. O estado inicial é marcado com uma pequena seta sem origem. As setas são etiquetadas com elementos do alfabeto A . De cada estado tem de sair uma e uma só seta etiquetada com cada elemento do alfabeto. Duas ou mais transições entre o mesmo par de estados são representadas por uma única seta, tendo como etiqueta uma lista de elementos do alfabeto.

Exemplo 5.1

A figura seguinte



representa um autómato finito determinista $M = (A, Q, q_0, \delta, F)$, com

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

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

Este AFD reconhece o conjunto das sequências binárias com um número ímpar de uns. Os estados A e B representam respetivamente *número par de uns* e *número ímpar de uns*. Inicialmente, o AFD encontra-se em A – a sequência vazia tem um número par de uns. A seguir, por cada 1 que entra, o AFD transita entre os estados A e B , de modo a refletir o número de uns lidos até esse momento. A chegada de zeros não altera o estado. Quando a palavra de entrada se esgotar, se o AFD se encontrar em A é porque tinha um número par de uns e é por isso rejeitada; se se encontrar em B é aceite.

Por exemplo:

- A palavra 1011 faz M_1 evoluir de A para B ($A \xrightarrow{1} B \xrightarrow{0} B \xrightarrow{1} A \xrightarrow{1} B$), aceitando-a como sendo uma palavra pertencente à linguagem reconhecida por M_1 .
- A palavra 1100 faz M_1 evoluir de A para A ($A \xrightarrow{1} B \xrightarrow{1} A \xrightarrow{0} A \xrightarrow{0} A$), levando à sua rejeição.
- A palavra vazia, ε , deixa M_1 em A , levando à sua rejeição.

O exemplo anterior mostra que um AFD também pode ser representado de forma textual. Sendo o alfabeto e o conjunto de estados conjuntos finitos, a função de transição é também um conjunto finito, que pode ser representada por um conjunto de triplos (s_1, a, s_2) , sendo $s_2 = \delta(s_1, a)$. A função de transição pode também ser representada por uma matriz, em que as linhas correspondem ao estado atual, as colunas aos símbolos do alfabeto e as células contêm o estado seguinte. Isto é ilustrado pelo exemplo seguinte.

Exemplo 5.2

O AFD do exemplo anterior pode ser textualmente representado por

$$\begin{aligned} A &= \{0, 1\} \\ Q &= \{A, B\} \\ q_0 &= A \\ F &= \{B\} \\ \delta &= \{(A, 0, A), (A, 1, B), (B, 0, B), (B, 1, A)\} \end{aligned}$$

A função de transição δ pode ser representada matricialmente por

	0	1
A	A	B
B	B	A

5.2 Linguagem reconhecida por um AFD

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.

Exemplo 5.3

Usando o autómato do exemplo 5.1, mostre que a palavra 1101 é reconhecida por esse autómato.

Resposta:

$\delta(A, 1) = B, \delta(B, 1) = A, \delta(A, 0) = A$ e $\delta(A, 1) = B$, logo 1101 é reconhecida por M_1 .

Seja $\delta^* : Q \times A^* \rightarrow Q$ a extensão de δ definida indutivamente por:

$$\begin{aligned}\delta^*(q, \varepsilon) &= q \\ \delta^*(q, av) &= \delta^*(\delta(q, a), v), \quad \text{com } a \in A \wedge v \in A^*\end{aligned}$$

M aceita a palavra $u \in A^*$ se e só se $\delta^*(q_0, u) \in F$.

A linguagem reconhecida por $M = (A, Q, q_0, \delta, F)$ denota-se por $L(M)$ e é definida por

$$L(M) = \{u \in A^* \mid M \text{ aceita } u\} = \{u \in A^* \mid \delta^*(q_0, u) \in F\}$$

Exemplo 5.4

Usando δ^* verifique se a palavra $u = abab$ é aceite pelo autómato $M = (A, Q, q_0, \delta, F)$, definido por:

$$\begin{aligned}A &= \{a, b, c\} \\ Q &= \{s_1, s_2\} \\ q_0 &= s_1 \\ F &= \{s_1\} \\ \delta &= \{(s_1, a, s_2), (s_1, b, s_1), (s_1, c, s_1), (s_2, a, s_1), (s_2, b, s_2), (s_2, c, s_2)\}\end{aligned}$$

Resposta: Pretende-se verificar se $\delta^*(s_1, abab) \in F$.

$$\begin{aligned}\delta^*(s_1, abab) &= \delta^*(\delta(s_1, a)bab) = \delta^*(s_2, bab) \\ &= \delta^*(\delta(s_2, b)ab) = \delta^*(s_2, ab) \\ &= \delta^*(\delta(s_2, a)b) = \delta^*(s_1, b) \\ &= \delta^*(\delta(s_1, b)\varepsilon) = \delta^*(s_1, \varepsilon) \\ &= s_1.\end{aligned}$$

Como $s_1 \in F$, M aceita a palavra $abab$.

5.3 Projeto de um AFD

O projeto de um autómato finito é um processo criativo; como tal não pode ser reduzido a uma simples receita ou fórmula. Pode, no entanto, ajudar se se seguir os seguintes passos:

1. Identificar os estados necessários.
2. Acrescentar as transições.
3. Identificar e marcar o estado inicial.
4. Identificar e marcar os estados de aceitação.

Frequentemente, é preciso iterar diversas vezes sobre estes passos antes de se chegar à solução para o problema.

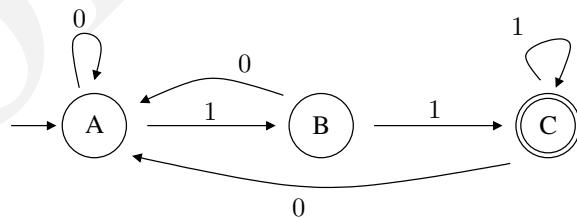
Exemplo 5.5

Projete um AFD que reconheça as sequências binárias terminadas em 11.

Resposta: Depois de alguma reflexão chega-se à conclusão de que bastam 3 estados, A , B e C :

- A significando que o último dígito que entrou não é 1;
- B significando que o último dígito que entrou é 1 mas o anterior não o é;
- C significando que os dois últimos dígitos entrados são 1.

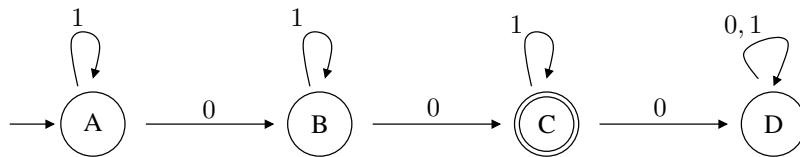
A partir daqui chega-se facilmente ao seguinte autómato



Exemplo 5.6

Projete um AFD que reconheça as sequências binárias com exatamente dois zeros e qualquer número de uns.

Resposta: Na figura seguinte apresenta-se graficamente o AFD pretendido.

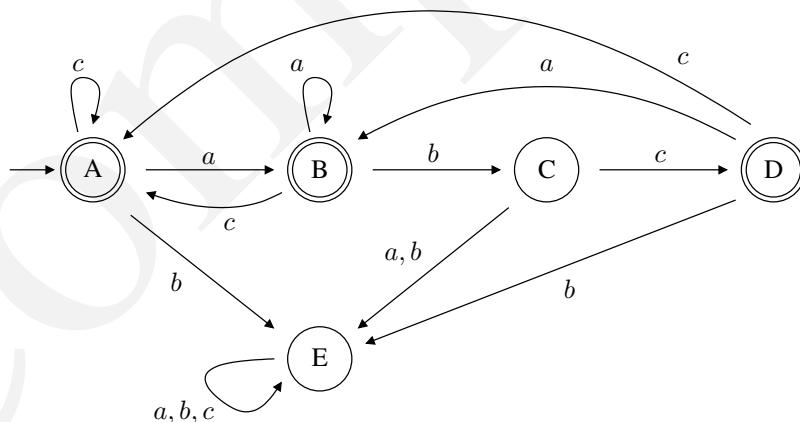


São necessários 4 estados para representar a chegada de 0, 1, 2 e mais de 2 zeros. Note que do estado D saem duas setas dirigidas ao próprio estado D , um com a etiqueta 0 e outro com a etiqueta 1. Por uma questão de simplificação notacional, é costume fundir-se as duas setas numa só, com as etiquetas separadas por vírgulas. Note ainda que num autómato finito determinista de cada estado deve sair uma seta etiquetada com cada um dos símbolos do alfabeto, mesmo que essas setas já não possam conduzir a situações de aceitação. É o caso das setas saídas do estado D .

Exemplo 5.7

Projete um AFD que reconheça 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.

Resposta: À partida, podemos considerar a existência de 4 estados para detetar a sequência abc . Isto corresponde aos estados A, B, C e D da figura abaixo e as setas entre eles indo da esquerda para a direita. Basta que um b não satisfaça o requisito imposto para que a sequência seja rejeitada. O estado E captura essas situações.



Note que os estados A, B e D são de aceitação. O estado C não o é porque se o autómato termina aí, a sequência de entrada termina em b e, por conseguinte, existe um b , esse último, que não tem um c imediatamente à sua direita. Veremos adiante que os estados A e D são equivalentes podendo ser fundidos. No entanto, isso não invalida que o AFD dado esteja correto.

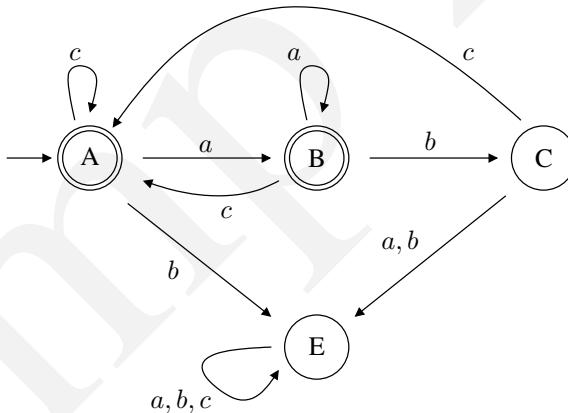
Exercício 5.1

Projete uma AFD que reconheça sequências binárias de comprimento ímpar terminadas em 0 ou de comprimento par terminadas em 1.

Dica: é possível definir-se um autómato com 5 estados.

5.4 AFD reduzido

Considere o autómato do exemplo anterior (exemplo 5.7). Os estados A e D são equivalentes. Por um lado, a sequência de entrada é aceite se o autómato termina nos estados A ou D . Por outro lado, se o autómato se encontra nos estados A ou D evolui, em ambos os casos, para o estado B se entra um a , para o E se entra um b e para o A se entra um c . Ou seja, enviar o autómato para o estado D é equivalente a enviá-lo para o estado A , podendo, por isso, um ser substituído pelo outro. Se desviarmos para A a única seta dirigida para D — vindo de C com etiqueta c —, o estado D deixa de ser alcançável a partir do estado inicial, podendo ser eliminado. O autómato transforma-se então no autómato da figura abaixo, que lhe é equivalente.



Em geral, dois estados, s_i e s_j , de um autómato $M = (A, Q, q_0, \delta, F)$ são equivalentes se e só se

$$\forall u \in A^* \quad \delta^*(s_i, u) \in F \Leftrightarrow \delta^*(s_j, u) \in F$$

A notação $s_i \equiv s_j$ é usada para representar o facto de s_i e s_j serem equivalentes. O conjunto de todos os estados equivalentes a s é denotado por $[s]$ e representa uma classe de equivalência. A relação de equivalência (\equiv) entre todos os estados de um autómato determinista permite definir o seu equivalente **reduzido**.

Seja $M = (A, Q, q_0, \delta, F)$ uma autómato determinista qualquer. O equivalente reduzido de M é o AFD $M' = (A', Q', q'_0, \delta', F')$ definido da seguinte maneira:

- $Q' = \{[q] \mid q \in Q\}$

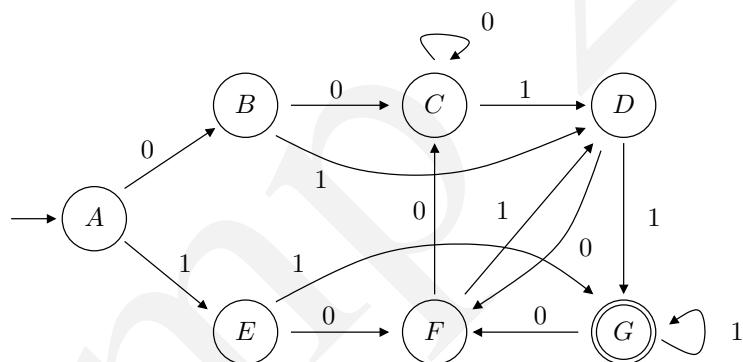
- $q'_0 = [q_0]$
- $F' = \{[q] \mid q \in F\}$
- $\delta' : Q' \times A \rightarrow Q'$ tal que $\delta'([q], a) = [\delta(q, a)]$.

5.4.1 Algoritmo de redução de AFD

O algoritmo de redução usa um método de aproximações sucessivas a partir de uma partição inicial dos estados em dois conjuntos – candidatos a classes de equivalência –, um com os estados de aceitação e outro com os restantes. Um dado conjunto C de estados é uma classe de equivalência se e só se para qualquer elemento a do alfabeto e qualquer par de estados q_1 e q_2 pertencentes a C , $[\delta(q_1, a)] = [\delta(q_2, a)]$. Sempre que um conjunto viole a condição anterior, é desdobrado em dois ou mais conjuntos e repete-se o processo, até atingir uma solução, que existe sempre.

Exemplo 5.8

Considere o autómato



que aceita sequências binárias terminadas em 11, sendo, por isso, equivalente ao autómato apresentado no exemplo 5.5. Construa-se o seu equivalente reduzido.

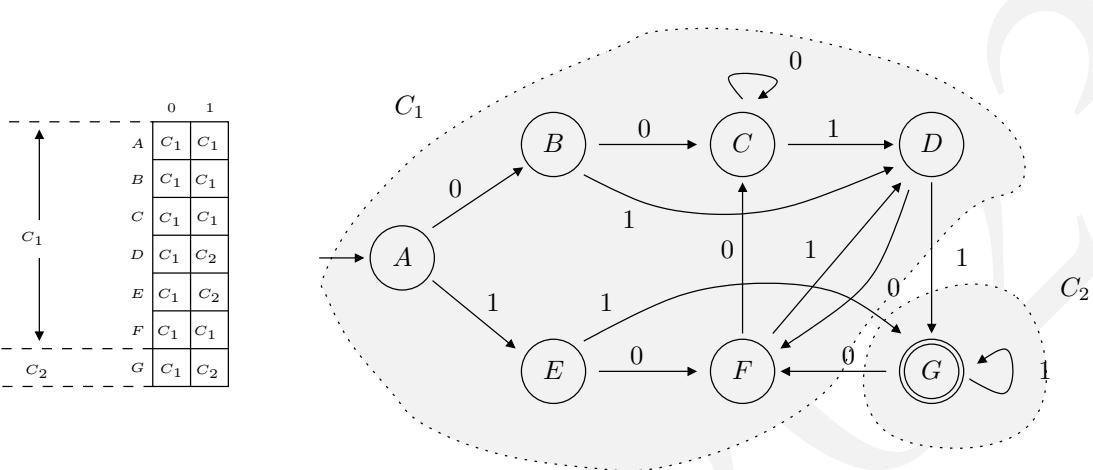
Resposta: Primeiro, definem-se dois conjuntos de estados, um com os estados de aceitação e outro com os restantes

$$\begin{aligned}C_1 &= Q - F = \{A, B, C, D, E, F\} \\C_2 &= F = \{G\}\end{aligned}$$

Seja Q' o conjunto desses candidatos a classes de equivalência, i.e., $Q' = \{C_1, C_2\}$.

A seguir, para cada elemento de Q' , avalia-se a sua evolução em termos dos elementos de Q' alcançados em consequência das possíveis transições. Isso está ilustrado na tabela à esquerda, na figura abaixo. Como se pode constatar, os estados de C_1 transitam para estados de C_1 , por ocorrência de um 0. Mas, por ocorrência de um 1, há estados que transitam para estados de C_1 e outros que transitam para estados de C_2 . Isto significa que o conjunto de estados C_1 não é uma classe de equivalência. A mesma constatação pode ser feita através da representação em grafo,

ilustrado à direita na mesma figura. Nos estados de C_1 , há transições em 1 que se mantém em C_1 e outras que se dirigem a C_2 . C_2 , porque tem apenas um elemento, é obviamente uma classe de equivalência.



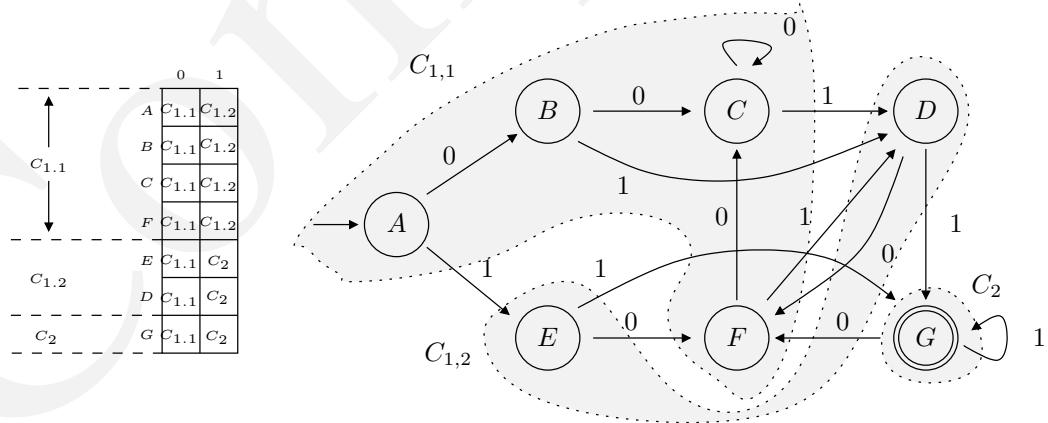
O conjunto C_1 , não sendo uma classe de equivalência, tem de ser desdobrado. Olhando para a tabela, verifica-se que as linhas referentes a C_1 caem em duas categorias, uma com os estados A , B , C e F e outra com os estados D e E . Divide-se então o conjunto C_1 em dois subconjuntos, representando estas duas categorias, passando Q' a ter 3 elementos:

$$C_{1.1} = \{A, B, C, F\}$$

$$C_{1.2} = \{D, E\}$$

$$C_2 = \{G\}$$

Reconstruindo a tabela e o grafo, obtém-se



Pode facilmente constatar-se que $C_{1.1}$ e $C_{1.2}$ são classes de equivalência, pelo que o processo de redução terminou. Se tal não acontecesse, os conjuntos de estados que não fossem classes de equivalência deveriam ser desdobrados, repetindo-se o processo enquanto necessário. Pode verificar que o autómato obtido é isomorfo com o apresentado no exemplo 5.5.

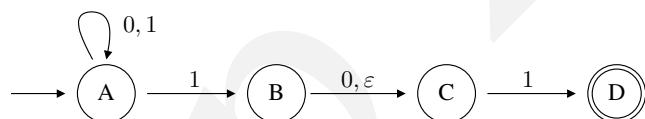
O algoritmo encontra sempre uma solução. Se se tentar reduzir um autómato finito determinista já reduzido, o processo de redução conduzirá ao próprio autómato (na realidade, a um que lhe é isomorfo).



Capítulo 6

Autómatos Finitos Não Deterministas (AFND)

Considere o seguinte autómato definido sobre o alfabeto $A = \{0, 1\}$.



Possui 3 características diferentes das assumidas na definição de autómato finito determinista:

- Não há nenhuma seta etiquetada com 1 a sair dos estados B e D , nem nenhuma seta etiquetada com 0 a sair dos estados C e D .
- Há duas setas etiquetadas com 1 a sair do estado A , um para A e outro para B .
- Há uma seta etiquetada com ε , a palavra vazia, a sair do estado B . Designar-se-ão este tipo de transições por **transições- ε** .

Estas características violam a definição de autómato finito determinista, mas são aceites pelos autómatos finitos não deterministas (AFND), que permitem que de cada estado possam sair zero ou mais setas etiquetadas com cada um dos símbolos do alfabeto de entrada ou com a palavra vazia. A multiplicidade permite que para fazer aceitar uma palavra se possa escolher entre caminhos alternativos. Para que uma palavra seja aceite basta que exista um caminho que conduza a uma estado de aceitação, mesmo que haja outros que conduzam a estados de não aceitação. Perante a entrada 1011 o autómato anterior pode realizar 4 caminhos alternativos:

$$\begin{aligned}
 A &\xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} A \xrightarrow{1} A \\
 A &\xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} A \xrightarrow{1} B \\
 A &\xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} A \xrightarrow{1} B \xrightarrow{\varepsilon} C \\
 A &\xrightarrow{1} A \xrightarrow{0} A \xrightarrow{1} B \xrightarrow{\varepsilon} C \xrightarrow{1} D
 \end{aligned}$$

um dos quais, o último, conduz ao estado de aceitação. Logo, porque há pelo menos um caminho que termina num estado de aceitação, a palavra 1011 é aceite. Este autómato reconhece todas as sequências binárias terminadas em 11 ou 101.

6.1 Definição de autómato finito não determinista

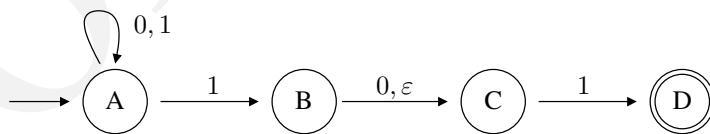
Um **autómato finito não determinista** (AFND) é um quíntuplo $M = (A, Q, q_0, \delta, F)$, em que:

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

A diferença relativamente à definição de autómato finito determinista verifica-se na relação δ . Por um lado, permite que as setas (transições) sejam etiquetadas com ε . Por outro lado, pelo facto de δ ser uma relação e não uma função, permite que o mesmo par $(q, a) \in Q \times A_\varepsilon$ possa ter 0 ou mais imagens.

Exemplo 6.1

À luz da definição anterior, apresente os vários elementos do AFND graficamente representado na figura seguinte.



Resposta:

Considerando que o AFND é representado pelo tuplo $M = (A, Q, q_0, \delta, F)$, tem-se

$$\begin{aligned}
 A &= \{0, 1\} \\
 Q &= \{A, B, C, D\} \\
 q_0 &= A \\
 F &= \{D\} \\
 \delta &= \{(A, 0, A), (A, 1, A), (A, 1, B), (B, \varepsilon, C), (B, 0, C), (C, 1, D)\}
 \end{aligned}$$

Note que a existência dos elementos $(A, 1, A)$ e $(A, 1, B)$ faz com que δ não seja uma função.

Alternativamente, é possível definir-se a relação de transição entre estados como uma função Δ que aplica o conjunto $Q \times A_\varepsilon$ em $\wp(Q)$, onde $\wp(Q)$ representa o conjunto dos subconjuntos de Q .

Exemplo 6.2

Usando esta última definição para a relação de transição entre estados, apresente os vários elementos do AFND graficamente representado no exemplo anterior (exemplo 6.1).

Resposta:

Considerando que o AFND é representado pelo tuplo $M = (A, Q, q_0, \delta, F)$, tem-se

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

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

$$q_0 = A$$

$$\begin{aligned} \Delta = \{ & (A, 0) \rightarrow \{A\}, \quad (A, 1) \rightarrow \{A, B\}, \quad (A, \varepsilon) \rightarrow \emptyset, \\ & (B, 0) \rightarrow \{C\}, \quad (B, 1) \rightarrow \emptyset, \quad (B, \varepsilon) \rightarrow \{C\}, \\ & (C, 0) \rightarrow \emptyset, \quad (C, 1) \rightarrow \{D\}, \quad (C, \varepsilon) \rightarrow \emptyset, \\ & (D, 0) \rightarrow \emptyset, \quad (D, 1) \rightarrow \emptyset, \quad (D, \varepsilon) \rightarrow \emptyset \} \end{aligned}$$

$$F = \{D\}$$

11

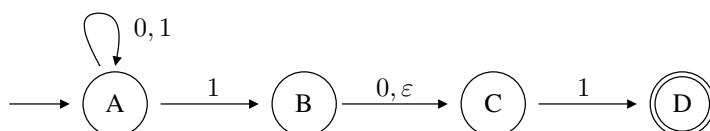
Neste caso, Δ é uma função que aplica $Q \times A_\varepsilon \rightarrow \wp(Q)$ e, como Q tem 4 elementos e $A_\varepsilon = \{0, 1, \varepsilon\}$ tem 3, Δ tem 12 elementos. Note que sendo Δ definida como uma função é preciso apresentar as imagens para todos os elementos de $Q \times A_\varepsilon$, mesmo que sejam o conjunto vazio. Em geral, torna-se mais clara a representação usando a relação de transição.

6.2 Árvore de caminhos

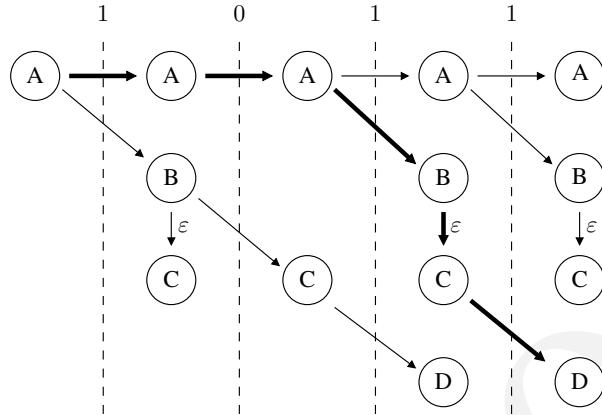
Uma forma simples de obter os diversos caminhos que um AFND pode realizar em resposta a uma dada palavra à entrada é traçando a *árvore de caminhos*. Corresponde a um grafo acíclico, em forma de árvore, tendo como raiz o estado inicial e onde se representam todos os estados alcançáveis por ocorrência da palavra de entrada.

Exemplo 6.3

Determine a árvore de caminhos que representa a resposta do autómato abaixo à palavra 1011.



Resposta:



A árvore de caminhos é uma estrutura em camadas (subconjunto de estados) alcançadas por força da ocorrência de cada símbolo da palavra de entrada. Uma palavra é aceite se o subconjunto de estados da última camada contiver pelo menos um estado de aceitação, ou seja, se a intersecção com o conjunto de aceitação não for o conjunto vazio. No exemplo anterior, o caminho que permite aceitar a palavra foi posto em realce. No exemplo, tenha também em atenção a forma como as transições- ε são tratadas. Elas ficam dentro da mesma camada (subconjunto) uma vez que não correspondem à ocorrência de um símbolo do alfabeto.

6.3 Linguagem reconhecida por um AFND

Diz-se que um AFND $M = (A, Q, q_0, \delta, F)$, **aceita** uma palavra $u \in A^*$ se e só 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, \dots, s_n , que satisfaça as seguintes condições:

1. $s_0 = q_0$;
2. qualquer que seja o $i = 1, \dots, n$, $(s_{i-1}, u_i, s_i) \in \delta$;
3. $s_n \in F$.

Caso contrário diz-se que M **rejeita** a entrada. Note que, nos autómatos finitos não deterministas, n pode ser maior que $|u|$, porque alguns dos u_i podem ser ε . Isto está patente no exemplo anterior (exemplo 6.3) no caminho de reconhecimento da palavra 1011.

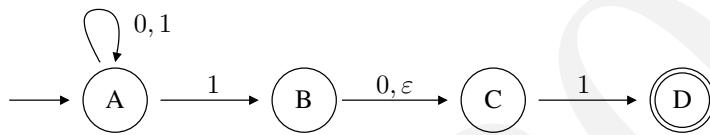
A linguagem reconhecida por M denota-se por $L(M) = \{u \mid M \text{ aceita } u\}$. Usar-se-á a notação $q_i \xrightarrow{u} q_j$ para representar a existência de uma palavra u que conduza do estado q_i ao estado q_j . Usando esta notação tem-se $L(M) = \{u \mid q_0 \xrightarrow{u} q_f \wedge q_f \in F\}$.

6.4 Fecho- ε

O conceito de fecho- ε (ε -closure, em inglês) é central à manipulação de AFND. Dado um AFND $M = (A, Q, q_0, \delta, F)$ qualquer, ε -closure : $Q \rightarrow \wp(Q)$ é uma função que associa a um dado estado $q \in Q$ o subconjunto de estados direta ou indiretamente alcançáveis a partir de q apenas por transições- ε , incluindo o próprio estado q .

Exemplo 6.4

Determine o ε -closure() de cada um dos estados do AFND seguinte.



Resposta:

$$\varepsilon\text{-closure}(A) = \{A\}$$

$$\varepsilon\text{-closure}(B) = \{B, C\}$$

$$\varepsilon\text{-closure}(C) = \{C\}$$

$$\varepsilon\text{-closure}(D) = \{D\}$$

É conveniente estender-se a definição de fecho- ε , considerando que o argumento da função é um subconjunto de estados em vez de apenas um. Neste caso, o resultado é a união dos fechos de cada um dos estados.

Capítulo 7

Equivalência entre AFD e AFND

A definição de AFND incorpora a definição de AFD, pelo que qualquer AFD é por definição um AFND. Na verdade, na definição de AFD, δ é uma função que aplica $Q \times A$ em Q , logo $\delta \subset (Q \times A \times Q)$. Mas, sendo $A \subset A_\varepsilon$, $(Q \times A \times Q) \subset (Q \times A_\varepsilon \times Q)$, pelo que a função δ dos AFD é um subconjunto da relação δ dos AFND.

7.1 Conversão de AFND em AFD

Prova-se também que dado um qualquer AFND é possível construir-se um AFD que reconhece exatamente a mesma linguagem. Olhando para uma árvore de caminhos, constata-se que por ação de um símbolo à entrada o autómato não determinista avança de um subconjunto de estados para outro, sendo que inicialmente se encontra no subconjunto constituído pelo estado inicial mais aqueles direta ou indiretamente alcançáveis por transições- ε , ou seja, o fecho- ε do estado inicial. É assim possível transformar um AFND num AFD cujos estados são subconjuntos de estados do AFND. Se o AFND tiver n estados, haverá 2^n subconjuntos de estados, pelo que o AFD equivalente terá no máximo 2^n estados. Ver-se-á adiante que, em geral, muitos destes estados não são alcançáveis a partir do estado inicial, pelo que podem ser descartados. Os estados de aceitação do AFD serão todos aqueles que contenham estados de aceitação do AFND, ou seja, aqueles cuja intersecção com o conjunto de aceitação do AFND não seja o conjunto vazio.

A estrutura de construção da árvore de caminhos permite obter a resposta de um AFND a cada símbolo do alfabeto de entrada, considerando que o AFND se encontra hipoteticamente num dado subconjunto de estados. É assim possível, dado um AFND $M = (A, Q, q_0, \delta, F)$, definir-se uma função $\delta' : \wp(Q) \times A \rightarrow \wp(Q)$, que representa as transições do autómato em termos de subconjuntos de estados.

Dado um AFND $M = (A, Q, q_0, \delta, F)$ qualquer, o autómato $M' = (A, Q', q'_0, \delta', F')$ onde:

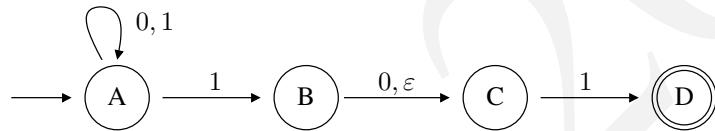
- $Q' = \wp(Q)$;

- $\delta' : \wp(Q) \times A \rightarrow \wp(Q)$, com $\delta'(p', a) = \bigcup_{p \in p'} (\varepsilon\text{-closure}(\Delta(p, a)))$
- $q'_0 = \varepsilon\text{-closure}(q_0)$; e
- $F' = \{q' : q' \in \wp(Q) \wedge q' \cap F \neq \emptyset\}$

é um autómato finito determinista equivalente a M . Nesta definição, optou-se por usar a função Δ e não a relação δ , porque simplifica a formulação da função δ' . Note que, na definição de δ' , p' representa um elemento de $\wp(Q)$, sendo portanto um subconjunto de estados de Q . Relembro que um subconjunto de estados é de aceitação no autómato determinista equivalente desde que contenha pelo menos um estado de aceitação do autómato inicial.

Exemplo 7.1

Usando o método anterior, construa-se um AFD equivalente ao AFND apresentado no exemplo 6.1, cuja representação gráfica se repete aqui por comodidade.



Resposta: Seja $M' = (A, Q', q'_0, \delta', F')$ o autómato pretendido. Tem-se

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

onde

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

Tem-se ainda $q'_0 = \varepsilon\text{-closure}(A) = X_1$ e

$$F' = \{X_8, X_9, X_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}\}$$

Finalmente, a função δ' é definida pela tabela seguinte

estado	0	1	estado	0	1	estado	0	1	estado	0	1
X_0	X_0	X_0	X_1	X_1	X_7	X_2	X_4	X_0	X_3	X_5	X_7
X_4	X_0	X_8	X_5	X_1	X_{15}	X_6	X_4	X_8	X_7	X_5	X_{15}
X_8	X_0	X_0	X_9	X_1	X_7	X_{10}	X_4	X_0	X_{11}	X_5	X_7
X_{12}	X_0	X_8	X_{13}	X_1	X_{15}	X_{14}	X_4	X_8	X_{15}	X_5	X_{15}

No exemplo anterior, observando a tabela de transição, é fácil constatar-se que a partir do estado inicial, X_1 , apenas os estados X_7 , X_5 e X_{15} são alcançáveis, pelo que os restantes podem ser retirados do

autómato. A obtenção de um AFD equivalente sem estados não alcançáveis (desnecessários) pode fazer-se por um método construtivo. A ideia é ir contruindo a função de transição δ' e o conjunto de estados Q' a partir do estado inicial. Sendo q_0 o estado inicial do AFND, o estado inicial do AFD equivalente é $q'_0 = \varepsilon\text{-closure}(q_0)$ e o seu conjunto de estados começa em $Q' = \{q'_0\}$. A partir deste podem calcular-se os estados alcançados por ocorrência dos vários símbolos do alfabeto, sendo acrescentados a Q' . O processo repete-se até que todos os elementos de Q' estejam processados e nenhum novo surja.

O algoritmo seguinte consagra este procedimento. Nele, considera-se a definição de AFND baseada na função Δ , função que, dados um estado e uma letra do alfabeto, devolve o conjunto de estados alcançados ($\delta : Q \times A \rightarrow \wp(Q)$), e a função $\varepsilon\text{-closure}(\cdot)$ que recebe como entrada um subconjunto de estados.

Algoritmo 7.1

NFA-to-DFA (input (A, Q, q_0, δ, F) , output $(A, Q', q'_0, \delta', F')$):

```

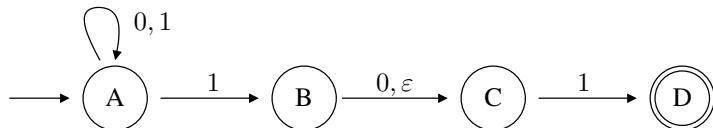
 $q'_0 \leftarrow \varepsilon\text{-closure}(\{q_0\})$ 
 $Q' \leftarrow \{q'_0\}$ 
 $O' \leftarrow Q'$ 
while  $O' \neq \emptyset$  do :
     $o' \leftarrow \text{take one element out of } O'$ 
    foreach  $a \in A$  do :
         $q' \leftarrow \bigcup_{o \in o'} \varepsilon\text{-closure}(\Delta(o, a))$ 
         $\delta'(q', a) \leftarrow q'$ 
        if  $q' \notin Q'$  then :
             $Q' \leftarrow Q' \cup q'$ 
             $O' \leftarrow O' \cup q'$ 
     $F' \leftarrow \emptyset$ 
    foreach  $q' \in Q'$  do :
        if  $(q' \cap F) \neq \emptyset$  then :
             $F' \leftarrow F' \cup q'$ 

```

O exemplo seguinte ilustra a aplicação deste algoritmo.

Exemplo 7.2

Usando o método construtivo, construa-se um AFD equivalente ao AFND apresentado no exemplo 6.1, cuja representação gráfica se repete aqui por comodidade.



Resposta: Seja $M' = (A, Q', q'_0, \delta', F')$ o autómato pretendido. Tem-se que

$$q'_0 = \varepsilon\text{-closure}(A) = \{A\} = X_1$$

De X_1 saem duas transições, uma etiquetada com 0 e outra com 1.

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

$$\delta'(X_1, 1) = \varepsilon\text{-closure}(\Delta(A, 1)) = \varepsilon\text{-closure}(\{A, B\}) = \{A, B, C\} = X_7$$

Partindo de X_7 tem-se

$$\begin{aligned}\delta'(X_7, 0) &= \varepsilon\text{-closure}(\Delta(A, 0)) \cup \varepsilon\text{-closure}(\Delta(B, 0)) \cup \varepsilon\text{-closure}(\Delta(C, 0)) \\ &= \varepsilon\text{-closure}(\{A\}) \cup \varepsilon\text{-closure}(\{C\}) \cup \varepsilon\text{-closure}(\{\}) = \{A\} \cup \{C\} \cup \{\} = X_5\end{aligned}$$

$$\begin{aligned}\delta'(X_7, 1) &= \varepsilon\text{-closure}(\Delta(A, 1)) \cup \varepsilon\text{-closure}(\Delta(B, 1)) \cup \varepsilon\text{-closure}(\Delta(C, 1)) \\ &= \varepsilon\text{-closure}(\{A, B\}) \cup \varepsilon\text{-closure}(\{\}) \cup \varepsilon\text{-closure}(\{D\}) \\ &= \{A, B, C\} \cup \{\} \cup \{D\} = X_{15}\end{aligned}$$

Procedendo de forma equivalente para X_5 e X_{15} , obtém-se

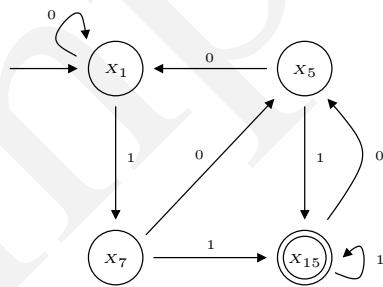
$$\delta'(X_5, 0) = \bigcup_{q \in X_5} (\varepsilon\text{-closure}(\Delta(q, 0))) = X_1$$

$$\delta'(X_5, 1) = \bigcup_{q \in X_5} (\varepsilon\text{-closure}(\Delta(q, 1))) = X_{15}$$

$$\delta'(X_{15}, 0) = \bigcup_{q \in X_{15}} (\varepsilon\text{-closure}(\Delta(q, 0))) = X_5$$

$$\delta'(X_{15}, 1) = \bigcup_{q \in X_{15}} (\varepsilon\text{-closure}(\Delta(q, 1))) = X_{15}$$

Sendo X_{15} o único estado que contém estados de aceitação do AFND, o AFD equivalente é o representado graficamente a seguir



Capítulo 8

Operações sobre AFD e AFND

A classe das linguagens regulares é fechada sob as operações de **reunião**, **concatenação**, **fecho de Kleene**, **complementação**, **diferença** e **intersecção**. Quer isto dizer que se M_1 e M_2 são dois autómatos finitos quaisquer, reconhecendo respetivamente as linguagens $L(M_1)$ e $L(M_2)$, existem autómatos finitos que reconhecem as linguagens $L(M_1) \cup L(M_2)$, $L(M_1)L(M_2)$, $L(M_1)^*$, $\overline{L(M_1)}$, $L(M_1) - L(M_2)$ e $L(M_1) \cap L(M_2)$.

8.1 Reunião de autómatos finitos

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, e sejam $L(M_1)$ e $L(M_2)$ as linguagens por eles reconhecidas, respetivamente. O AFND $M = (A, Q, q_0, \delta, F)$, onde

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

$$F = F_1 \cup F_2$$

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

reconhece a linguagem $L(M) = L(M_1) \cup L(M_2)$.

Na figura 8.1 ilustra-se graficamente a construção de M a partir de M_1 e M_2 . Por um lado, é intuitivo que qualquer sequência u aplicada a M , segue não deterministicamente os caminhos de M_1 e de M_2 . Logo, se u é aceite por um deles também o é por M . Por outro lado, também é intuitivo que M só aceita sequências que sejam aceites por M_1 ou por M_2 . Mas, prove-se formalmente esta equivalência.

Pretende-se provar que $L(M) = L(M_1) \cup L(M_2)$. Para isso basta provar que

$$L(M) \subseteq L(M_1) \cup L(M_2)$$

e que

$$L(M_1) \cup L(M_2) \subseteq L(M)$$

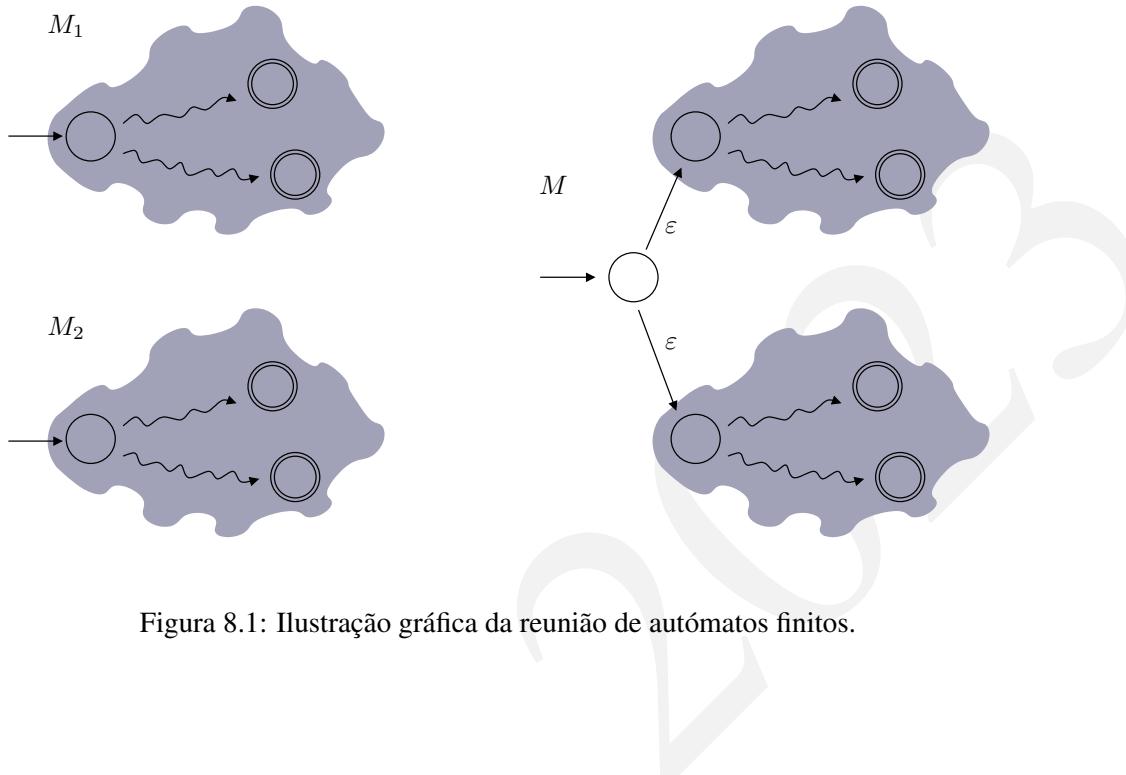


Figura 8.1: Ilustração gráfica da reunião de autómatos finitos.

A primeira condição corresponde a provar que

$$u \in L(M) \Rightarrow (u \in L(M_1) \vee u \in L(M_2))$$

o que se obtém do seguinte desenvolvimento

$$\begin{aligned} u \in L(M) &\Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F \\ &\Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_1 \cup F_2 \\ &\Rightarrow (q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_1) \vee (q_0 \xrightarrow{\varepsilon} q_2 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_2) \\ &\Rightarrow (q_1 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_1) \vee (q_2 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_2) \\ &\Rightarrow u \in L(M_1) \vee u \in L(M_2) \end{aligned}$$

A segunda condição corresponde a provar, para $i = 1, 2$, que

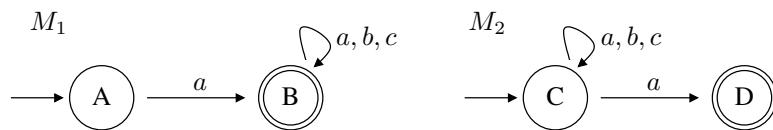
$$u \in L(M_i) \Rightarrow u \in L(M)$$

o que se obtém do seguinte desenvolvimento

$$\begin{aligned} u \in L(M_i) &\Rightarrow q_i \xrightarrow{u} q_f, \quad \text{com } q_f \in F_i \\ &\Rightarrow q_0 \xrightarrow{\varepsilon} q_i \xrightarrow{u} q_f, \quad \text{com } q_f \in F_i \\ &\Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_i \\ &\Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F \\ &\Rightarrow u \in L(M) \end{aligned}$$

Exemplo 8.1

A figura



representa graficamente os autómatos M_1 e M_2 , definidos sobre o alfabeto $A = \{a, b, c\}$, que reconhecem respetivamente as linguagens

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

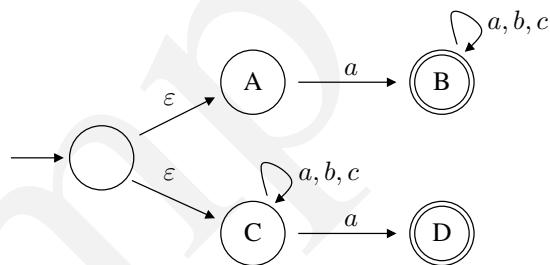
e

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

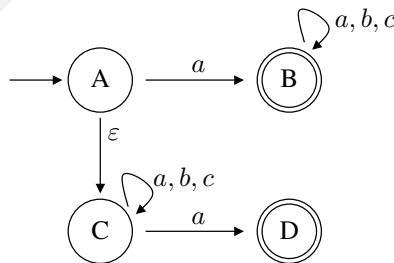
Usando a operação de reunião sobre autómatos determine o autómato M que reconhece a linguagem das palavras começadas ou terminadas por a.

Resposta:

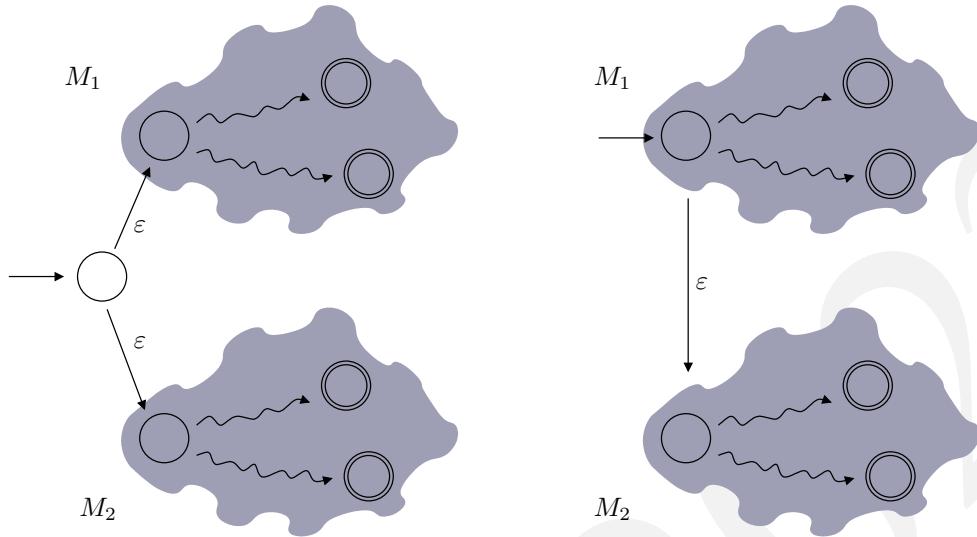
A aplicação direta do algoritmo de reunião resulta no autómato da figura abaixo



Alguma manipulação permite transformá-lo em

**Exercício 8.1**

A manipulação feita na resposta do exemplo anterior corresponde à ilustrada pelas duas composições da figura abaixo, definidas à volta dos autómatos M_1 e M_2 (a sombreado). Mostre que só são equivalentes se o estado inicial de M_1 não tiver setas a si dirigidas.



Considere que M_1 e M_2 são autómatos definidos sobre o alfabeto binário, sendo que M_1 reconhece as sequências com número par de uns e M_2 as sequências com número ímpar de zeros. Correspondendo à aplicação direta da reunião de autómatos finitos, a construção da esquerda reconhece as sequências com número par de uns ou ímpar de zeros. Por conseguinte, rejeita as sequências com número ímpar de uns e par de zeros. No entanto, estas sequências são aceites pelo autómato da direita. Mostre-o. Qual é a linguagem reconhecida pela construção da direita?

8.2 Concatenação de autómatos finitos

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, e sejam $L(M_1)$ e $L(M_2)$ as linguagens por eles reconhecidas, respectivamente. O AFND $M = (A, Q, q_0, \delta, F)$, onde

$$Q = Q_1 \cup Q_2$$

$$q_0 = q_1$$

$$F = F_2$$

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

reconhece a linguagem $L(M) = L(M_1).L(M_2)$, concatenação de L_1 com L_2 .

Na figura 8.2 ilustra-se graficamente a construção de M a partir de M_1 e M_2 . Para que M aceite uma palavra u é necessário percorrer um caminho que vá desde o seu estado inicial até um dos seus estados de aceitação. Ora, isto significa ir do estado inicial até um estado de aceitação de M_1 , passando-se, de seguida e por efeito de um ε , para o estado inicial de M_2 e finalmente indo deste até um dos estados de aceitação de M_2 . Ou seja, uma palavra u é aceite por M se se puder decompor u em duas partes v e w ($u = vw$), sendo v aceite por M_1 e w por M_2 . Prove-se formalmente esta equivalência.

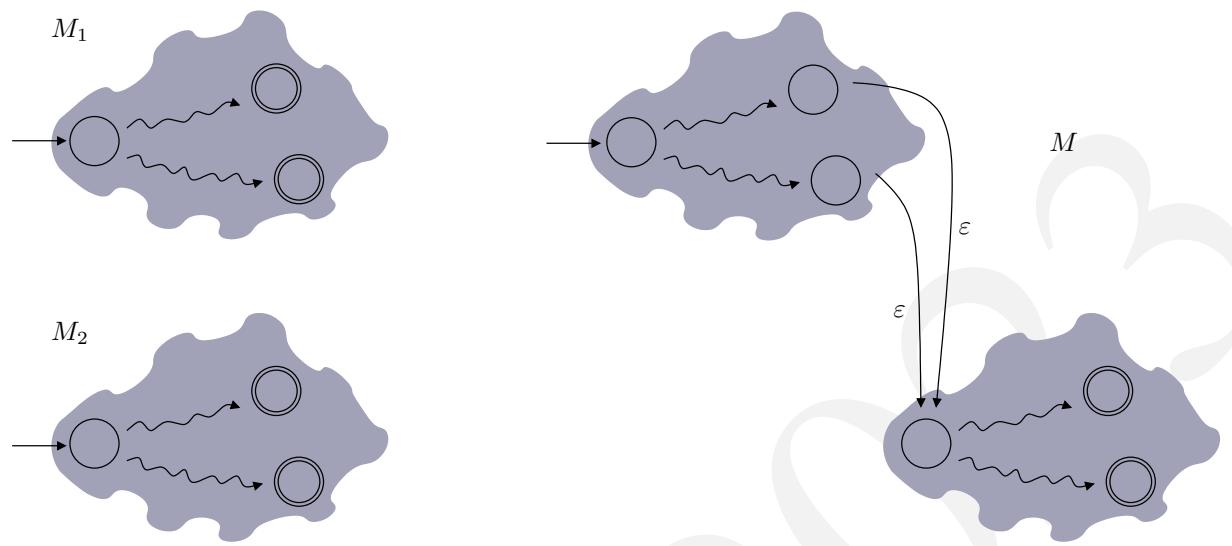


Figura 8.2: Ilustração gráfica da concatenação de autómatos finitos.

Pretende-se provar que $L(M) = L(M_1).L(M_2)$. Para isso basta provar que

$$L(M) \subseteq L(M_1).L(M_2)$$

e que

$$L(M_1).L(M_2) \subseteq L(M)$$

A primeira condição corresponde a provar que

$$u \in L(M) \Rightarrow (u \in L(M_1)L(M_2))$$

o que se obtém do seguinte desenvolvimento

$$\begin{aligned} u \in L(M) &\Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F \\ u \in L(M) &\Rightarrow q_0 \xrightarrow{u} q_f, \quad \text{com } q_f \in F_2 \\ &\Rightarrow q_0 \xrightarrow{v} q_v \xrightarrow{\varepsilon} q_2 \xrightarrow{w} q_f, \quad \text{com } u = vw \wedge q_v \in F_1 \wedge q_f \in F_2 \\ &\Rightarrow v \in L(M_1) \wedge w \in L(M_2), \quad \text{com } u = vw \\ &\Rightarrow u \in L(M_1)L(M_2) \end{aligned}$$

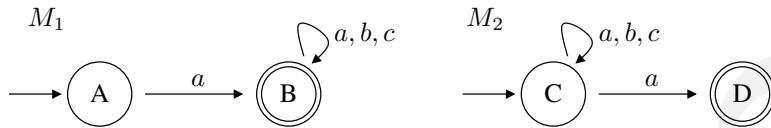
Exercício 8.2

Prove a segunda condição, isto é, prove que

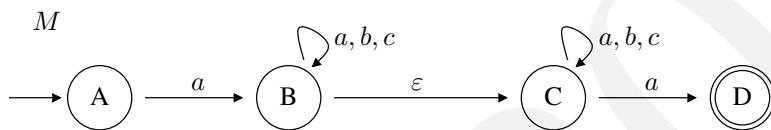
$$u \in L(M_1)L(M_2) \Rightarrow u \in L(M)$$

Exemplo 8.2

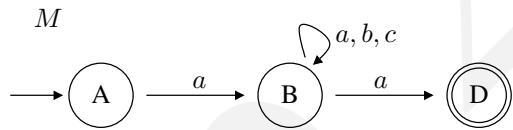
Usando o algoritmo apresentado, determine o autómato M que reconhece a linguagem $L(M_1).L(M_2)$, considerando os autómatos M_1 e M_2 do exemplo 8.1, que se repetem aqui por comodidade.



Resposta: A aplicação direta do algoritmo de concatenação resulta no autómato da figura abaixo



Note que B deixou de ser estado de aceitação. Alguma manipulação, permite que se chegue ao autómato

**Exercício 8.3**

Considerando os autómatos do exemplo 8.1 determine o autómato que reconhece a linguagem $L(M_2).L(M_1)$.

8.3 Fecho de Kleene de autómatos finitos

Seja $M_1 = (A, Q_1, q_1, \delta_1, F_1)$ um autómato (AFD ou AFND) qualquer, e seja $L(M_1)$ a linguagem por ele reconhecida. O AFND $M = (A, Q, q_0, \delta, F)$, onde

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

$$F = \{q_0\}$$

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

reconhece a linguagem $L(M) = (L(M_1))^*$, fecho de Kleene de M_1 .

Na figura 8.3 ilustra-se graficamente a construção de M a partir de M_1 . É claro da composição gráfica que M aceita ε ou qualquer palavra que percorra o caminho

$$q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{u_1} q_{f_1} \xrightarrow{\varepsilon} q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{u_2} \cdots \xrightarrow{\varepsilon} q_0 \xrightarrow{\varepsilon} q_1 \xrightarrow{u_n} q_{f_n} \xrightarrow{\varepsilon} q_0, \quad \text{com } q_{f_i} \in F_1$$

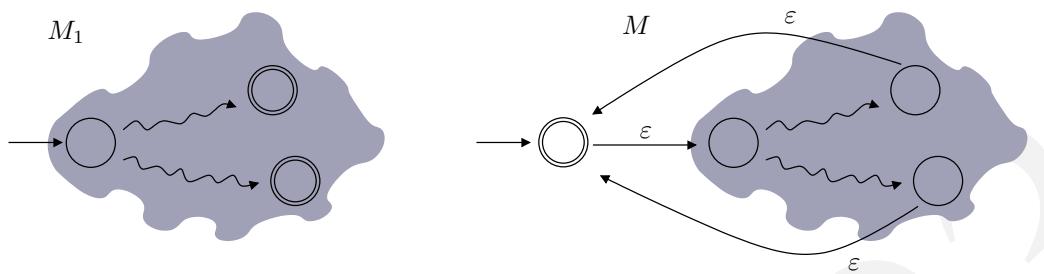


Figura 8.3: Ilustração gráfica do fecho de Kleene de autómatos finitos.

o que corresponde a zero ou mais concatenações de palavras de $L(M_1)$.

Exercício 8.4

Prove formalmente esta equivalência.

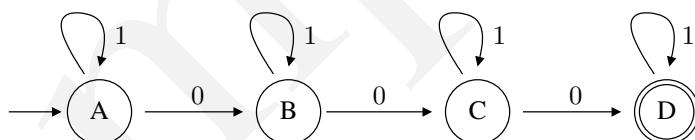
Exemplo 8.3

Sobre o alfabeto $A = \{0, 1\}$ construa um autómato que reconhece a linguagem

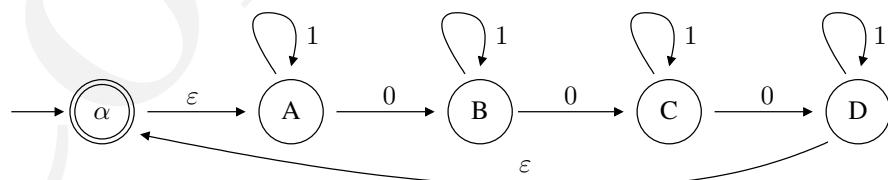
$$L = \{w \in A^* \mid \#(0, w) = 3\}$$

Com base nele construa o autómato que reconhece a linguagem L^* .

Resposta: A resposta à primeira questão é dada pelo autómato seguinte.

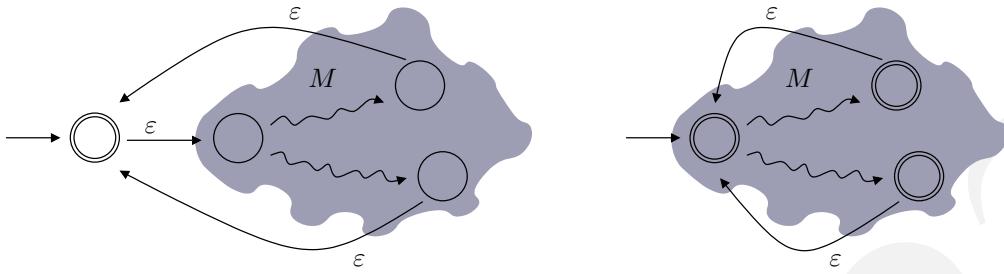


Aplicando-lhe o algoritmo de fecho obtém-se



Exercício 8.5

As duas construções da figura abaixo, definidas com base no autómato M (a parte a sombreado), parecem equivalentes, mas não o são. Apenas a da esquerda garante que o fecho de Kleene é bem construído sempre. Mostre que as duas construções só são equivalentes se o estado inicial de M for de aceitação ou se não houver transições a ele dirigidas.



- Considere que M é um autómato definido sobre o alfabeto $A = \{a, b, c\}$, reconhecendo as sequências começadas por a . Mostre que as linguagens reconhecidas pelas duas construções são a mesma.
- Considere que M é um autómato definido sobre o alfabeto $A = \{a, b, c\}$, reconhecendo as sequências terminadas por a . Mostre que as linguagens reconhecidas pelas duas construções são diferentes.

8.4 Complementação de autómatos finitos

Seja $M_1 = (A, Q_1, q_1, \delta_1, F_1)$ um AFD (note, determinista) qualquer, e seja $L(M_1)$ a linguagem por ele reconhecida. O AFD $M = (A, Q, q_0, \delta, F)$ onde

$$Q = Q_1$$

$$q_0 = q_1$$

$$F = Q_1 - F_1$$

$$\delta = \delta_1$$

reconhece a linguagem $L(M) = \overline{L(M_1)}$, ou seja, a linguagem complementar de $L(M_1)$ sobre o mesmo alfabeto. A única alteração efetuada é a complementação do conjunto de aceitação. Na realidade

$$u \in L(M) \Rightarrow \delta^*(q_0, u) \in F \Rightarrow \delta^*(q_0, u) \notin Q - F \Rightarrow u \notin L(M')$$

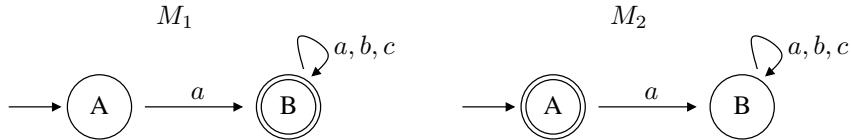
e

$$u \in L(M') \Rightarrow \delta^*(q_0, u) \in Q - F \Rightarrow \delta^*(q_0, u) \notin F \Rightarrow u \notin L(M)$$

A complementação dos estados de aceitação não funciona para AFND (autómatos não deterministas). **Se se quiser complementar um AFND deve-se primeiro convertê-lo para um AFD equivalente e depois complementar este.**

Exercício 8.6

Mostre que os dois autómatos da figura seguinte não são complementares um do outro. Para isso, basta apresentar uma palavra que ambos reconheçam ou ambos rejeitem.



Descreva a linguagem reconhecida pelo autómato da direita.

8.5 Intersecção de autómatos finitos

Dados dois autómatos \$M_1\$ e \$M_2\$ como construir um autómato \$M\$ tal que \$L(M) = L(M_1) \cap L(M_2)\$, i.e., que a linguagem reconhecida por \$M\$ seja a **intersecção** das linguagens reconhecidas por \$M_1\$ e \$M_2\$? Com base nas operações apresentadas anteriormente é possível chegar-se a tal autómato. Na realidade, por aplicação das leis de De Morgan,

$$L(M_1) \cap L(M_2) = \overline{L(M_1)} \cup \overline{L(M_2)}$$

Logo, sabendo-se complementar e reunir autómatos finitos, sabe-se fazer a sua intersecção. Note que esta abordagem envolve 3 complementações, o que pode obrigar à conversão de 3 AFND para os AFD equivalentes.

Considere-se agora uma abordagem direta para a obtenção da intersecção de autómatos finitos. Uma palavra é reconhecida pela intersecção de dois autómatos se e só se for reconhecida por cada um dos dois autómatos individualmente. Isto permite construir a intersecção à volta do produto cartesiano dos conjuntos de estados dos dois autómatos. Sejam \$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, e sejam \$L(M_1)\$ e \$L(M_2)\$ as linguagens por eles reconhecidas, respetivamente. O AFND \$M = (A, Q, q_0, \delta, F)\$, em que

$$Q = Q_1 \times Q_2$$

$$q_0 = (q_1, q_2)$$

$$F = F_1 \times F_2$$

$$\delta \subseteq (Q_1 \times Q_2) \times A_\varepsilon \times (Q_1 \times Q_2)$$

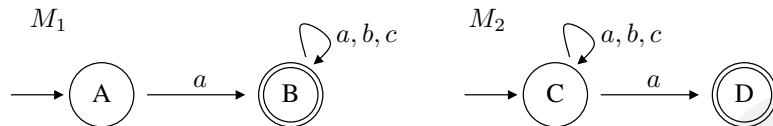
sendo \$\delta\$ definido de modo que:

- \$((q_i, q_j), a, (q'_i, q'_j)) \in \delta\$ sse \$(q_i, a, q'_i) \in \delta_1\$ e \$(q_j, a, q'_j) \in \delta_2\$, para todo o \$a \in A\$;
- \$((q_i, q_j), \varepsilon, (q'_i, q'_j)) \in \delta\$ sse \$(q_i, \varepsilon, q'_i) \in \delta_1\$, para todo o \$q_j \in Q_2\$;
- \$((q_i, q_j), \varepsilon, (q'_i, q'_j)) \in \delta\$ sse \$(q_j, \varepsilon, q'_j) \in \delta_2\$, para todo o \$q_i \in Q_1\$;
- \$((q_i, q_j), \varepsilon, (q'_i, q'_j)) \in \delta\$ sse \$(q_i, \varepsilon, q'_i) \in \delta_1\$ e \$(q_j, \varepsilon, q'_j) \in \delta_2\$;

reconhece a linguagem \$L(M) = L(M_1) \cap L(M_2)\$, ou seja, a linguagem intersecção de \$L(M_1)\$ e \$L(M_2)\$.

Exemplo 8.4

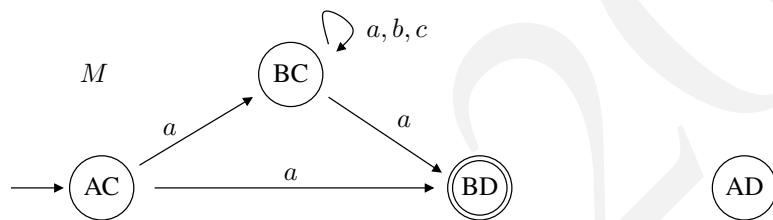
Sobre o alfabeto $A = \{a, b, c\}$ considere os autómatos M_1 e M_2 , graficamente apresentados a seguir, que reconhecem respetivamente as linguagens das palavras começadas e terminadas por a .



Construa um autómato que reconheça a linguagem $L = L(M_1) \cap L(M_2)$.

Resposta:

A figura seguinte mostra o autómato obtido pela aplicação do mecanismo de construção definido acima.

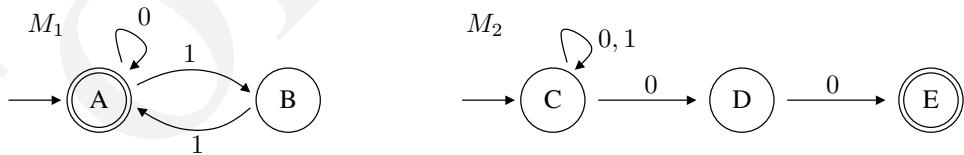


O estado AD não é alcançável a partir do estado inicial, pelo que pode ser eliminado da resposta, ficando-se com um autómato com 3 estados.

O exemplo anterior mostra que a aplicação do mecanismo de construção da intersecção de autómatos pode conduzir à introdução de estados não alcançáveis a partir do estado inicial. Isto propõe que a construção deve ser feita a partir do estado inicial, apenas se considerando os estados alcançáveis.

Exemplo 8.5

Sobre o alfabeto binário considere os autómatos M_1 e M_2 , graficamente apresentados a seguir

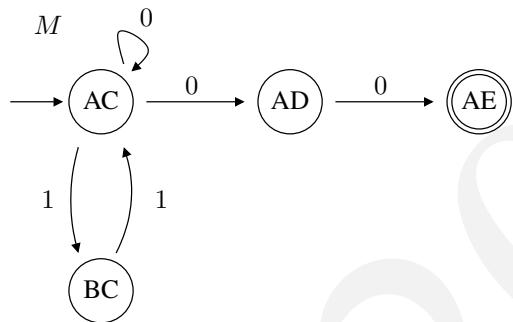


e determine a sua intersecção.

Resposta:

- O estado inicial do novo autómato é o estado $AC = (A, C)$. (Designar-se-á, no resto do exemplo, por XY o estado do autómato intersecção correspondente ao par (X, Y) .)
- Há uma transição a sair de A etiquetada com 0 — a transição $(A, 0, A)$ — e duas a sair de C — as transições $(C, 0, C)$ e $(C, 0, D)$. Logo, haverá duas transições a sair de AC etiquetadas com 0 — as transições $(AC, 0, AC)$ e $(AC, 0, AD)$.

- Etiquetadas com 1 há uma transição a sair de A , dirigida a B , e uma a sair de C , dirigida a C , pelo que o autómato intersecção terá uma única seta etiquetada com 1 a sair de AC — a seta $(AC, 1, BC)$. Com estas setas foram alcançados os estados AD e BC .
- Procedendo da mesma forma com estes estados e com outros que, eventualmente, se alcancem, obtém-se o autómato desenhado na figura seguinte, que possui apenas 4 estados e não os 6 do produto cartesiano.



É fácil constatar que M_1 reconhece as sequências binárias com número par de uns e M_2 as sequências binárias terminadas em 00. M , sendo a intersecção de M_1 e M_2 , reconhece as sequências binárias com um número par de uns terminadas em 00.

8.6 Diferença de autómatos

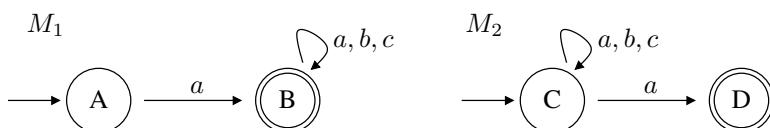
Dados dois autómatos M_1 e M_2 como construir um autómato M tal que $L(M) = L(M_1) - L(M_2)$, i.e., que a linguagem reconhecida por M seja a **diferença** entre linguagens reconhecidas por M_1 e M_2 ? Com base nas operações apresentadas anteriormente é possível chegar-se a tal autómato. Na realidade

$$L(M_1) - L(M_2) = L(M_1) \cap \overline{L(M_2)} = \overline{L(M_1)} \cup L(M_2)$$

Ora, já se mostrou como é que se constroem autómatos para a complementação e a intersecção ou para a reunião e a complementação.

Exercício 8.7

Sobre o alfabeto $A = \{a, b, c\}$ considere os autómatos M_1 e M_2 , graficamente apresentados a seguir, que reconhecem respetivamente as linguagens das palavras começadas e terminadas por a .



Construa um autómato que reconheça a linguagem $L = L(M_1) - L(M_2)$.

Capítulo 9

Equivalência entre Expressões Regulares e Autómatos Finitos

As expressões regulares e os autómatos finitos são equivalentes, no sentido em que descrevem a mesma classe de linguagens, as linguagens regulares. Assim sendo, é possível converter uma expressão regular dada num autómato finito que represente a mesma linguagem. Inversamente, é também possível converter um autómato finito dado numa expressão regular descrevendo a mesma linguagem.

9.1 Conversão de uma expressão regular num autómato finito

Como vimos na sua definição, uma expressão regular é construída à custa de elementos primitivos e dos operadores escolha ($|$), concatenação ($.$) e fecho ($*$). Os elementos primitivos correspondem à expressão \emptyset , representando o conjunto vazio, e às expressões do tipo a com $a \in A$, sendo A o alfabeto. É habitual considerar a expressão ε como um elemento primitivo, embora se saiba que $\varepsilon = \emptyset^*$.

É possível decompor uma expressão regular num conjunto de elementos primitivos interligados pelos operadores referidos acima. Dada uma expressão regular qualquer ela é:

1. um elemento primitivo;
2. ou uma expressão do tipo e^* , sendo e uma expressão regular qualquer;
3. ou uma expressão do tipo $e_1.e_2$, sendo e_1 e e_2 duas expressões regulares quaisquer; ou
4. ou uma expressão do tipo $e_1|e_2$, sendo e_1 e e_2 duas expressões regulares quaisquer;

Se se identificar os autómatos equivalentes das expressões primitivas, tem-se o problema da conversão de uma expressão regular para um autómato finito resolvido, visto que se sabe como fazer a reunião, concatenação e fecho de autómatos (ver capítulo 8).

9.1.1 Autómatos dos elementos primitivos

A tabela seguinte mostra os autómatos correspondentes às expressões regulares \emptyset , ε e a , com $a \in A$.

expressão regular	autómato finito
\emptyset	$\xrightarrow{\quad} \text{circle}$
ε	$\xrightarrow{\quad} \text{double circle}$
a	$\xrightarrow{\quad} \text{circle} \xrightarrow{a} \text{double circle}$

Na realidade, o autómato referente a ε pode ser obtido aplicando o fecho ao autómato de \emptyset .

9.1.2 Algoritmo de conversão

A transformação de uma expressão regular num autómato finito pode ser feita aplicando os seguintes passos:

1. Se a expressão regular é do tipo primitivo, o autómato correspondente pode ser obtido da tabela anterior.
2. Se é do tipo $e_1|e_2$, aplica-se este mesmo algoritmo na obtenção de autómatos para as expressões e_1 e e_2 e, de seguida, aplica-se a reunião de autómatos descrita na secção 8.1.
3. Se é do tipo e^* , aplica-se este mesmo algoritmo na obtenção de um autómato equivalente à expressão regular e e, de seguida, aplica-se o fecho de autómatos descrita na secção 8.3.
4. Finalmente, se é do tipo $e_1.e_2$, aplica-se este mesmo algoritmo na obtenção de autómatos para as expressões e_1 e e_2 e, de seguida, aplica-se a concatenação de autómatos descrita na secção 8.2.

Exemplo 9.1

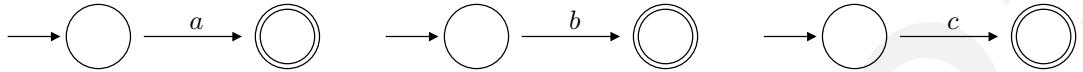
Construa um autómato equivalente à expressão regular $e = a|a(a|b|c)^*a$.

Resposta:

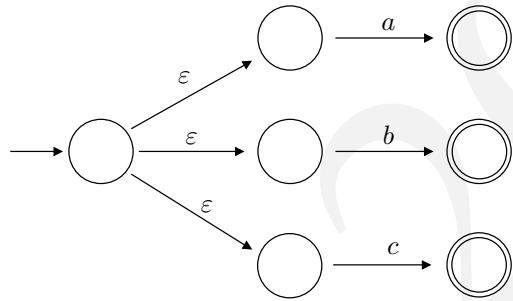
A expressão regular e é do tipo $e_1|e_2$, com $e_1 = a$ e $e_2 = a(a|b|c)^*a$, pelo que se deve aplicar a regra 4 do algoritmo. A expressão e_1 é do tipo primitivo, pelo que se pode aplicar a tabela para obter o autómato. A expressão e_2 resulta da concatenação de 3 expressões regulares, para as quais temos de obter autómatos. Apenas a segunda expressão da concatenação tem de ser considerada, visto já se ter os autómatos das outras. Tem-se então que converter a expressão regular $e_3 = (a|b|c)^*$,

que se obtém do fecho sobre o autómato que represente a expressão $e_4 = a|b|c$. Este, por sua vez, resulta da reunião dos autómatos das expressões primitivas a , b e c .

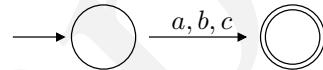
Uma vez definida a decomposição, a construção faz-se dos elementos primitivos para a expressão global. Os autómatos para as expressões a , b e c são



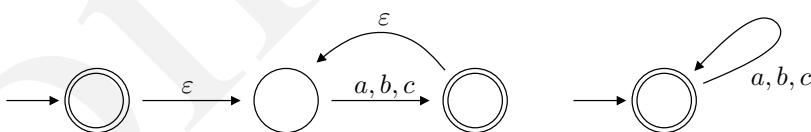
Fazendo a sua reunião obtém-se o autómato para e_4



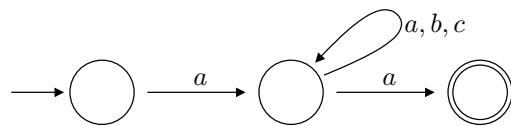
que pode ser simplificado para



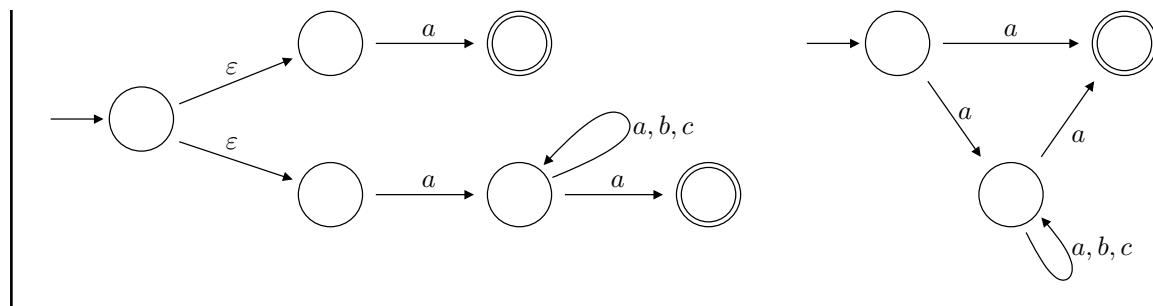
Aplicando o fecho a este autómato obtém-se o autómato para a expressão e_3 , que se apresenta a seguir tal como resulta do fecho (lado esquerdo) e após simplificação (lado direito).



O autómato para e_2 resulta da concatenação dos autómatos para a , e_3 e a , obtendo-se, após simplificação, o autómato seguinte



Finalmente, o autómato para e obtém-se da reunião dos autómatos para e_2 e a , o que resulta no autómato seguinte, lado esquerdo (o lado direito representa o mesmo autómato após simplificação).



9.2 Conversão de um autómato finito numa expressão regular

9.2.1 Autómato finito generalizado

A conversão de autómatos finitos em expressões regulares baseia-se num novo tipo de autómatos, designados **autómatos finitos generalizados**. Este autómatos são semelhantes aos autómatos finitos não-deterministas mas em que as etiquetas dos arcos são expressões regulares.

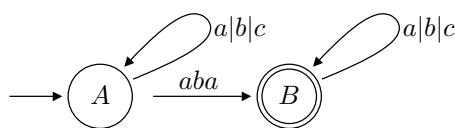
Um **autómato finito generalizado** (AFG) é um quíntuplo $M = (A, Q, q_0, \delta, F)$, em que:

- 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 a transição entre estados, sendo E o conjunto das expressões regulares definidas sobre A ; e
- $F \subseteq Q$ é o conjunto dos estados de aceitação.

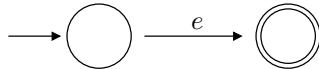
Note que com base nesta definição os automáticos finitos deterministas e não deterministas são autómatos finitos generalizados.

Exemplo 9.2

O autómato finito generalizado representado graficamente abaixo representa o conjunto das palavras, definidas sobre o alfabeto $A = \{a, b, c\}$, que contêm a sub-palavra aba .



Imagine que consegue transformar um autómato finito generalizado dado — pense, por exemplo, no autómato da figura anterior — num outro com a configuração dada pela figura seguinte



sendo e uma expressão regular qualquer. Então, e é uma expressão regular equivalente ao autómato dado. Um autómato como o da figura designa-se por **autómato finito generalizado reduzido**.

9.2.2 Algoritmo de conversão

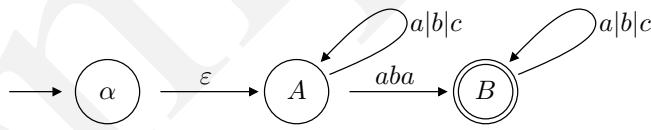
Assim sendo, a obtenção de uma expressão regular equivalente a um autómato qualquer dado, resume-se à transformação desse autómato num autómato finito generalizado reduzido. A redução de um AFG pode ser feita aplicando o seguinte procedimento:

1. transformação de um AFG noutro cujo estado inicial não tenha transições a ele dirigidas;
2. transformação de um AFG noutro com um único estado de aceitação que não tenha transições que dele partam;
3. eliminação, um a um, por transformações de equivalência, dos restantes estados.

Se o estado inicial de um AFG possui transições a ele dirigidas, a transformação definida pelo ponto 1 do procedimento anterior faz-se acrescentando um novo estado, que passa a ser o estado inicial do AFG transformado, e uma nova transição, etiquetada com ε , ligando este estado ao antigo estado inicial.

Exemplo 9.3

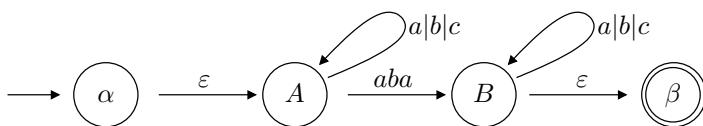
O AFG do exemplo 9.2 tem transições dirigidas ao seu estado inicial. O AFG representado abaixo, é-lhe equivalente e tal já não acontece.



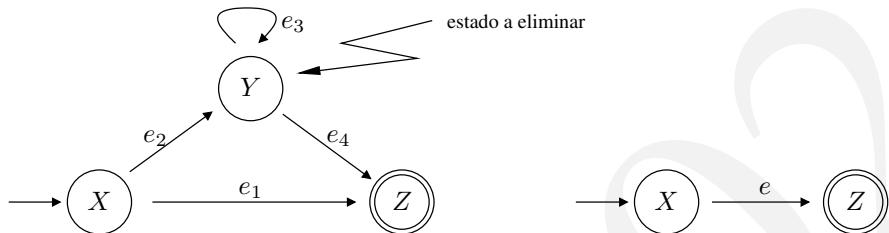
Se um AFG possui mais que um estado de aceitação ou se possui apenas um com transições que dele partem, a transformação definida pelo ponto 2 do procedimento de redução faz-se acrescentando um novo estado, que passa a ser o único estado de aceitação do AFG transformado, e novas transições etiquetadas com ε ligando os antigos estados de aceitação ao novo.

Exemplo 9.4

O AFG do exemplo 9.3 tem um único estado de aceitação mas tem transições que dele partem. O AFG representado abaixo, é-lhe equivalente e tal já não acontece.



Está-se agora em condições de aplicar sucessivamente o ponto 3 do procedimento de redução, no sentido de eliminar os estados intermédios (A e B , no caso do exemplo). A ordem pela qual se deve proceder à sua eliminação é arbitrária, em termos da obtenção de uma expressão regular equivalente, mas pode afetar a complexidade das operações a realizar. A eliminação de estados preconizada pelo ponto 3 do procedimento de transformação está esquematizada na figura seguinte

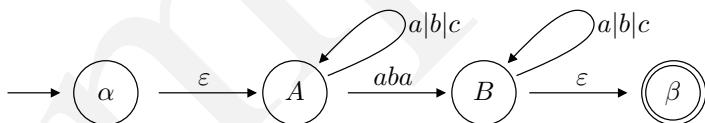


A situação inicial é representada pelo autómato da esquerda, sendo Y o estado que se pretende eliminar. É possível ir de X para Z passando por Y . Isso corresponde a sequências que encaixem na expressão regular $e_2e_3^*e_4$. Se se eliminar Y , estas sequências têm de ser colocadas diretamente entre X e Z . Se já existir uma transição entre X e Y , esta nova terá que ficar em paralelo. Chega-se assim à situação ilustrada à direita, sendo e dado por

$$e = e_1 | e_2e_3^*e_4$$

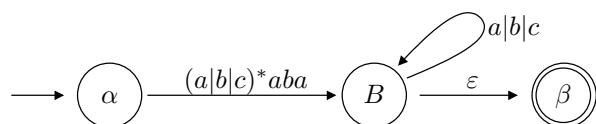
Exemplo 9.5

Remova os estados A e B do AFG do exemplo 9.4, redesenhado aqui por comodidade

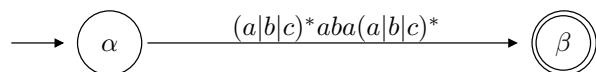


Resposta:

Comece-se por remover o estado A . Os estados α , A e B desempenham respetivamente os papéis dos estados X , Y e Z no esquema relativo ao ponto 3 do procedimento de redução. Fazendo a correspondência, tem-se $e_1 = \emptyset$, $e_2 = \epsilon$, $e_3 = a|b|c$ e $e_4 = aba$, pelo que o AFG após supressão do estado A se transforma no AFG seguinte, onde $e = \emptyset | \epsilon(a|b|c)^*aba = (a|b|c)^*aba$.



De seguida elimina-se o estado B . Neste caso os estados α , B e β desempenham respetivamente os papéis dos estados X , Y e Z , resultando em



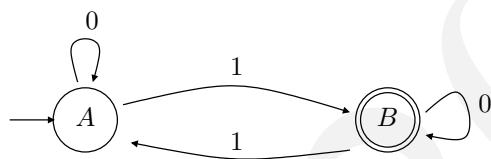
pelo que a expressão pretendida é

$$e = (a|b|c)^* aba(a|b|c)^*$$

O exemplo anterior não deixa transparecer alguma complexidade que pode aparecer no processo de remoção de estados. O estado a remover pode participar em vários triângulos $X - Y - Z$, o que obriga a calcular várias expressões regulares por cada estado a remover.

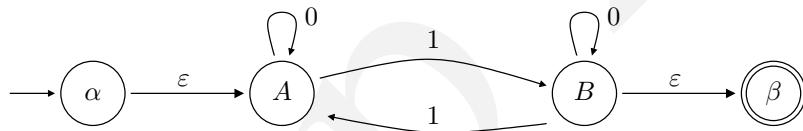
Exemplo 9.6

Considere o autómato da figura seguinte e obtenha uma expressão regular equivalente



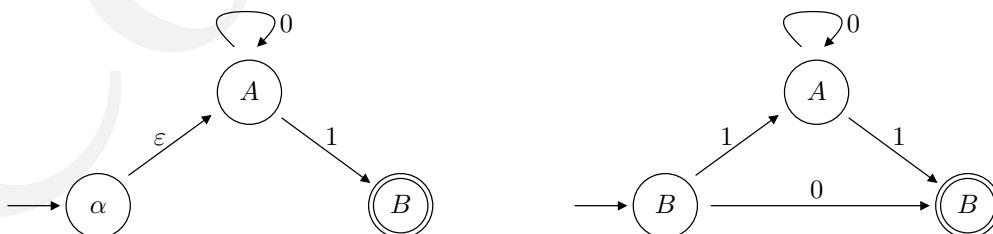
Resposta:

Comece-se por acrescentar novos estados inicial e de aceitação em consequência da aplicação dos pontos 1 e 2 do procedimento de redução. Obtém-se

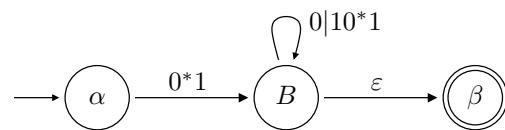


O estado A tem duas transições a si dirigidas, vindas de α e B , e uma que dele parte, dirigida a B . A sua supressão vai obrigar ao acrescento de duas transições, uma de α para B e outra de B para B . O estado B tem duas transições que dele saem, uma dirigida a A e outra a β , e uma a si dirigida, vinda de A . A sua supressão vai obrigar ao acrescento de duas transições, uma de A para β e outra de A para A . Sendo o custo de supressão de cada um dos dois estados igual, a ordem parece irrelevante. Opte-se então pela supressão de A em primeiro lugar.

Há dois triângulos $X - Y - Z$ a considerar, um formado pelos estados α , A e B e outro pelos estados B , A e B . Estes triângulos estão ilustrados na figura seguinte.



Da supressão de A resultam um arco entre os estados α e B e outro de B para ele próprio. O primeiro terá a etiqueta $e_1 = 0^*1$ e o segundo a etiqueta $e_2 = 0|10^*1$. Note que o arco original de B para B desaparece em consequência da supressão de A , sendo substituído pelo novo com etiqueta e_2 . Após a supressão obtém-se o AFG seguinte



Suprimindo B obtém-se facilmente a expressão pretendida

$$e = 0^*1(0|10^*1)^*$$

Exercício 9.1

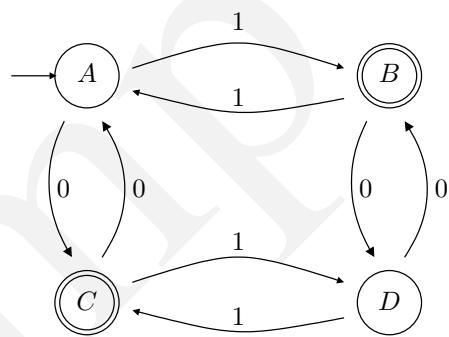
Repita o exercício do exemplo 9.6 mas eliminando em primeiro lugar o estado B .

Resposta:

Irá obter uma expressão regular diferente $((0|10^*1)^*10^*)$, mas que é equivalente à obtida no exemplo anterior.

Exercício 9.2

Determine uma expressão regular equivalente do autómato finito seguinte

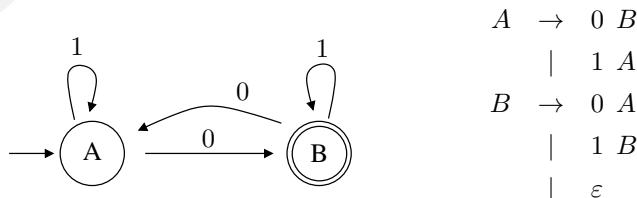


Capítulo 10

Equivalência entre Gramáticas Regulares e Autómatos Finitos

As gramáticas regulares e os autómatos finitos são equivalentes, no sentido em que descrevem a mesma classe de linguagens, as linguagens regulares. Assim sendo, é possível converter-se uma gramática regular dada num autómato finito que represente a mesma linguagem. Inversamente, é também possível converter-se um autómato finito dado numa gramática regular que descreva a mesma linguagem.

Considere a figura seguinte, onde se representa um autómato finito (neste caso determinista) e uma gramática regular, ambos representando as sequências binárias com um número ímpar de uns.



A analogia entre ambos é óbvia, o que permite definir os algoritmos de conversão.

10.1 Conversão de AF em GR

Algoritmo 10.1 (Conversão de AF em GR)

AFND-GR (input (A, Q, q_0, δ, F) , output (T, N, P, S)):

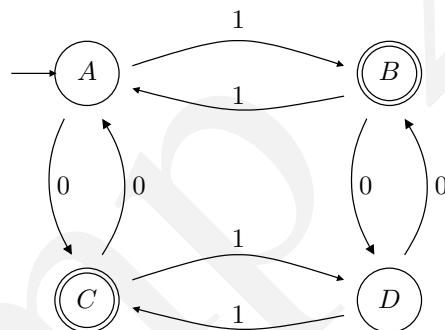
```

 $T \leftarrow A$ 
 $N \leftarrow Q$ 
 $S \leftarrow q_0$ 
 $P \leftarrow \{\}$ 
foreach  $(p, a, q) \in \delta$ 
     $P \leftarrow P \cup (p \rightarrow a q)$ 
foreach  $p \in F$ 
     $P \leftarrow P \cup (p \rightarrow \epsilon)$ 

```

Exemplo 10.1

Determine uma gramática regular equivalente do autómato finito seguinte



Resposta: Aplicando diretamente o algoritmo 10.1 e considerando que a gramática pretendida corresponde ao tuplo (T, N, P, S) , tem-se

$$\begin{aligned}
T &= \{0, 1\} \\
N &= \{A, B, C, D\} \\
S &= A \\
P &= \{A \rightarrow 0C, A \rightarrow 1B, B \rightarrow 0D, B \rightarrow 1A \\
&\quad C \rightarrow A, C \rightarrow 1D, D \rightarrow 0B, D \rightarrow 1C \\
&\quad B \rightarrow \epsilon, C \rightarrow \epsilon\}
\end{aligned}$$

Rescrevendo a gramática no formato mais habitual, tem-se

$$\begin{aligned}
A &\rightarrow 0C \mid 1B \\
B &\rightarrow 0D \mid 1A \mid \epsilon \\
C &\rightarrow 0A \mid 1D \mid \epsilon \\
D &\rightarrow 0B \mid 1C
\end{aligned}$$

10.2 Conversão de GR em AF

Para a conversão de uma gramática regular num autómato finito, podem considerar-se duas situações:

- conversão para um autómato finito generalizado;
- conversão para um autómato finito não generalizado;

O primeiro caso é bastante simples e é dado pelo seguinte algoritmo.

Algoritmo 10.2 (Conversão de GR em AFG)

GR-AFG (input (T, N, P, S) , output (A, Q, q_0, δ, F)):

```

 $A \leftarrow T$ 
 $Q \leftarrow N \cup \{q_f\}$ , com  $q_f \notin N$ 
 $q_0 \leftarrow S$ 
 $\delta \leftarrow \{\}$ 
foreach  $(X \rightarrow \omega Y) \in P$ , com  $\omega \in T^*$  and  $Y \in N$  // produções com não terminal no fim
     $\delta \leftarrow \delta \cup (X, \omega, Y)$ 
foreach  $(X \rightarrow \omega) \in P$ , com  $\omega \in T^*$  // produções só com terminais (0 ou mais)
     $\delta \leftarrow \delta \cup (X, \omega, q_f)$ 
 $F \leftarrow \{q_f\}$ 
```

O exemplo seguinte ilustra a aplicação deste algoritmo.

Exemplo 10.2

Determine um autómato finito generalizado equivalente à gramática regular

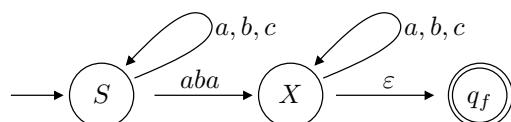
$$\begin{aligned} S &\rightarrow a \ S \mid b \ S \mid c \ S \mid aba \ X \\ X &\rightarrow a \ X \mid b \ X \mid c \ X \mid \epsilon \end{aligned}$$

Resposta:

A aplicação direta do algoritmo 10.2 resulta no autómato $M = (A, Q, q_0, \delta, F)$, com

$$\begin{aligned} A &= \{a, b, c\} \\ Q &= \{S, X, q_f\} \\ q_0 &= S \\ \delta &= \{(S, a, S), (S, b, S), (S, c, S), (S, aba, X), \\ &\quad (X, a, X), (X, b, X), (X, c, X), (X, \epsilon, q_f)\} \\ F &= \{q_f\} \end{aligned}$$

Representado-o graficamente, tem-se



No caso de se pretender um autómato não generalizado como resultado, é preciso tratar adequadamente as produções dos tipos $X \rightarrow \omega Y$ e $X \rightarrow \omega$ quando ω possui 2 ou mais símbolos terminais. É fácil perceber que se $\omega = a_1 a_2$, ou seja, se contiver 2 símbolos terminais, a produção anterior pode ser transformada em

$$X \rightarrow a_1 X_1$$

$$X_1 \rightarrow a_2 Y$$

em que X_1 corresponde a um símbolo não terminal novo. Se ω tiver mais que 2 símbolos terminais, o procedimento é semelhante. Se se transformar previamente a gramática de modo a que, em todas as produções, $|\omega| = 1$, a algoritmo anterior (algoritmo 10.2) conduz a um autómato finito não generalizado.

Exemplo 10.3

Determine um autómato finito não generalizado equivalente à gramática regular

$$S \rightarrow a S | b S | c S | a b a X$$

$$X \rightarrow a X | b X | c X | \varepsilon$$

Resposta:

Comece-se por transformar a gramática de modo a quebrar a sequência aba na produção $S \rightarrow a b a X$, o que resulta em

$$S \rightarrow a S | b S | c S | a S_1$$

$$S_1 \rightarrow b S_2$$

$$S_2 \rightarrow a X$$

$$X \rightarrow a X | b X | c X | \varepsilon$$

A aplicação direta do algoritmo 10.2 a esta última gramática resulta no autómato $M = (A, Q, q_0, \delta, F)$, com

$$A = \{a, b, c\}$$

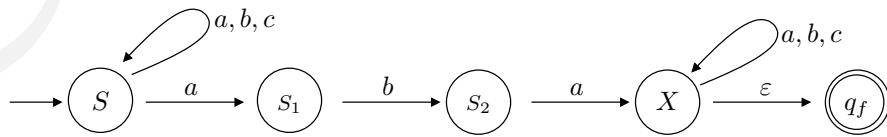
$$Q = \{S, S_1, S_2, X, q_f\}$$

$$q_0 = S$$

$$\begin{aligned} \delta = & \{(S, a, S), (S, b, S), (S, c, S), (S, a, S_1), (S_1, b, S_2), (S_2, a, X) \\ & (X, a, X), (X, b, X), (X, c, X), (X, \varepsilon, q_f)\} \end{aligned}$$

$$F = \{q_f\}$$

Representado-o graficamente, tem-se





Compiladores

Autómatos finitos

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

DETI, Universidade de Aveiro

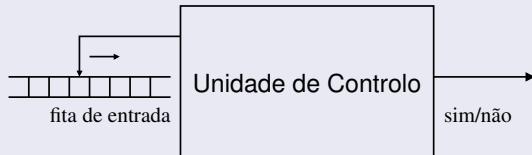
Ano letivo de 2022-2023

Sumário

- ① Autómato finito determinista (AFD)
- ② Redução de autómato finito determinista
- ③ Autómato finito não determinista (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

Autómato finito

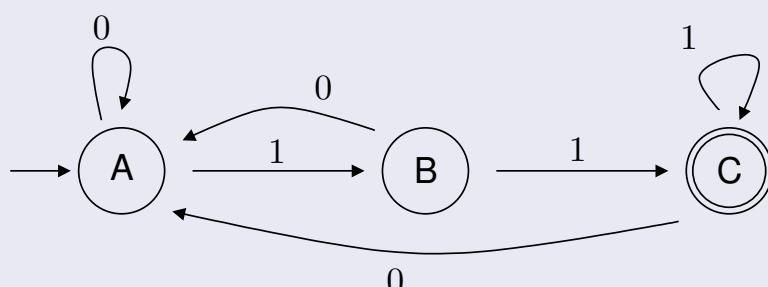
Um **autómato finito** é um mecanismo reconhecedor das palavras de uma linguagem regular



- 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 autómatos finitos podem ser **deterministas, não deterministas ou generalizados**

Autómato finito determinista

Um **autómato finito determinista** é um autómato finito

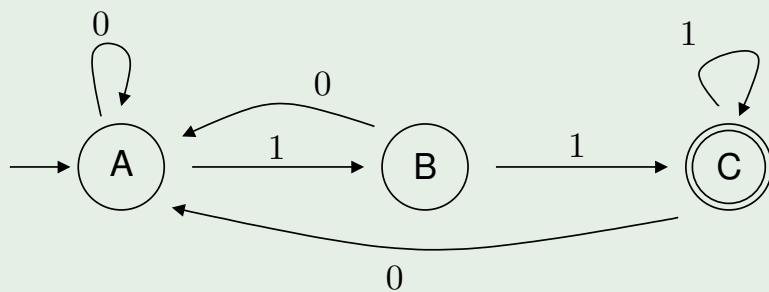


onde

- 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 autómato.

Autómato finito determinista: exemplo (1)

Q Que palavras binárias são reconhecidas pelo autómato seguinte?

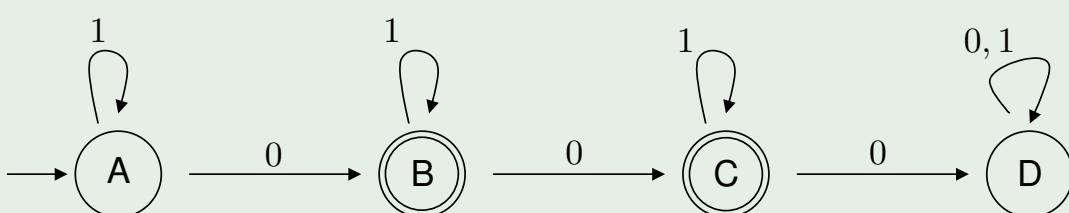


R Todas as palavras terminadas em 11.

E Obtenha uma expressão regular que represente a mesma linguagem.

Autómato finito determinista: exemplo (2)

Q Que palavras binárias são reconhecidas pelo autómato seguinte?

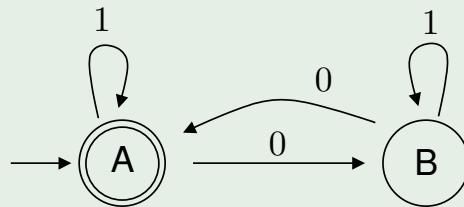


R Todas as palavras com apenas 1 ou 2 zeros.

E Obtenha uma expressão regular que represente a mesma linguagem.

Autómato finito determinista: exemplo (3)

Q Que palavras binárias são reconhecidas pelo autómato seguinte?



R as sequências binárias com um número par de zeros.

E Obtenha uma expressão regular que represente a mesma linguagem.

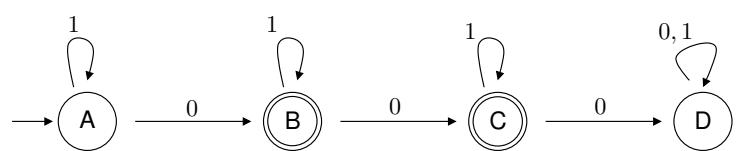
Definição de autómato finito determinista

D Um **autómato finito determinista** (AFD) é um quíntuplo

$M = (A, Q, q_0, \delta, F)$, 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 autómato finito determinista

D Um **autómato finito determinista** (AFD) é um quíntuplo

$$M = (A, Q, q_0, \delta, F), \text{ em que:}$$

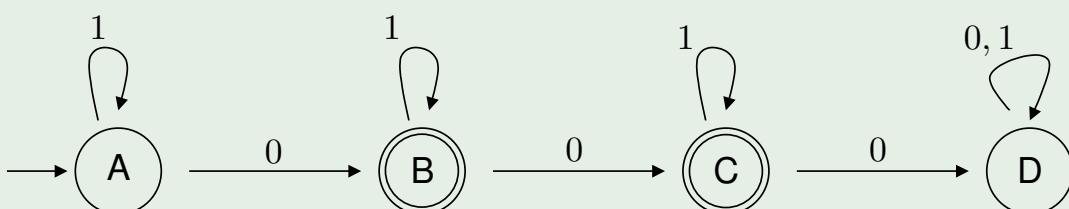
- A é o alfabeto de entrada;
- Q é um conjunto finito não vazio de estados;
- $q_0 \in Q$ é o estado inicial;
- $\delta : Q \times A \rightarrow Q$ é uma função que determina a transição entre estados; e
- $F \subseteq Q$ é o conjunto dos estados de aceitação.

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 $(q, a, q) \in Q \times A \times Q$

Autómato finito determinista: exemplo (4)

Q Represente textualmente o AFD seguinte.



R

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

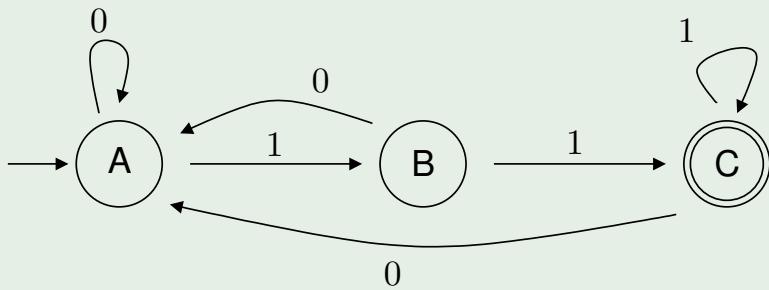
- $A = \{0, 1\}$
 - $Q = \{A, B, C, D\}$
 - $q_0 = A$
 - $F = \{B, C\}$
- $\delta = \{(A, 0, B), (A, 1, A), (B, 0, C), (B, 1, B), (C, 0, D), (C, 1, C), (D, 0, D), (D, 1, D)\}$

$$\bullet \delta =$$

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

Autómato finito determinista: exemplo (5)

Q Represente textualmente o AFD seguinte.



R

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

- $A = \{0, 1\}$
- $Q = \{A, B, C\}$
- $q_0 = A$
- $F = \{C\}$

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

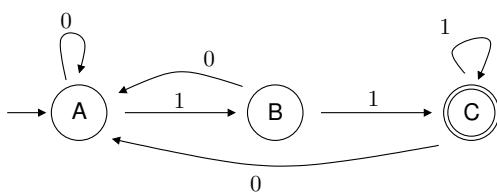
- $\delta = \begin{array}{|c|c|c|} \hline & 0 & 1 \\ \hline A & A & B \\ \hline B & A & C \\ \hline C & A & C \\ \hline \end{array}$

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.

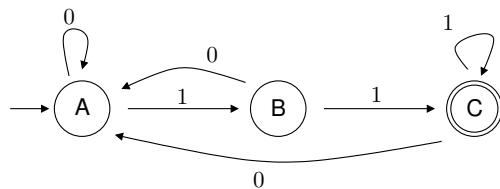
- A palavra $\omega_1 = 0101$ faz o caminho $A \xrightarrow{0} A \xrightarrow{1} B \xrightarrow{0} A \xrightarrow{1} B$
 - como B não é de aceitação, ω_1 não pertence à linguagem
- A palavra $\omega_2 = 0011$ faz o caminho $A \xrightarrow{0} A \xrightarrow{0} A \xrightarrow{1} B \xrightarrow{1} C$
 - 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
 - ① $\delta^*(q, \varepsilon) = q$
 - ② $\delta^*(q, av) = \delta^*(\delta(q, a), v)$, com $a \in A \wedge v \in A^*$
- M aceita u se $\delta^*(q_0, u) \in F$.
- $L(M) = \{u \in A^* : M \text{ aceita } u\} = \{u \in A^* : \delta^*(q_0, u) \in F\}$

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



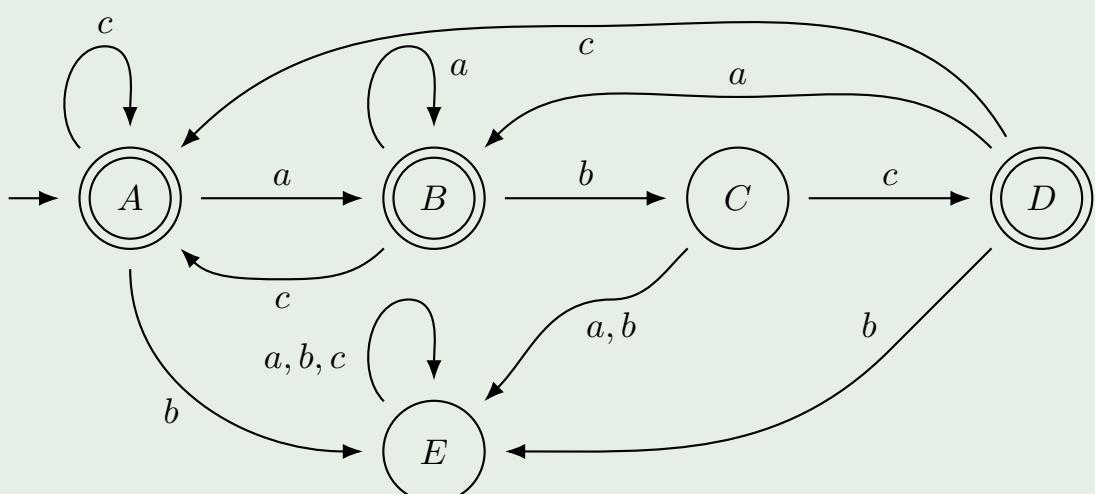
Autómato finito determinista: exemplo (6)

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 autómato que reconheça L .

R



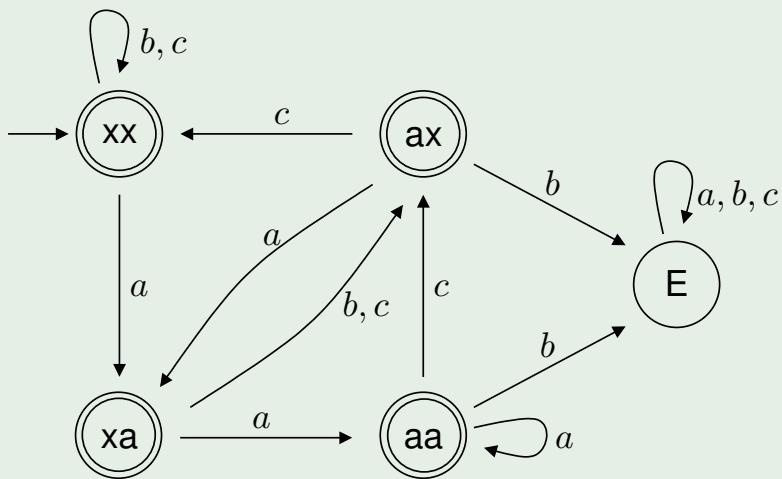
Autómato finito determinista: exemplo (7)

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

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

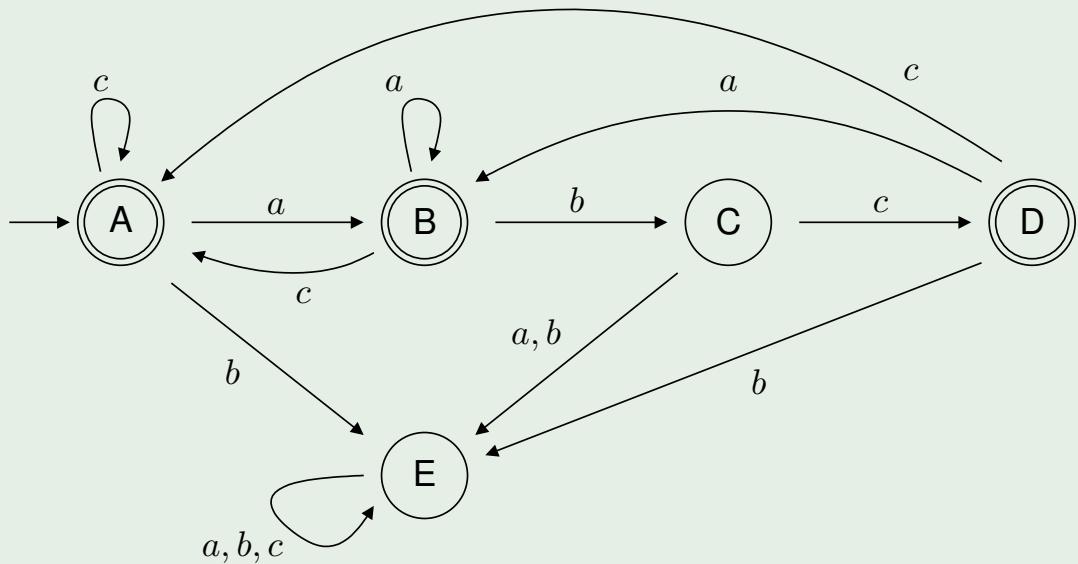
Projecte um autómato que reconheça L .

R

???

Redução de autómato finito determinista (1)

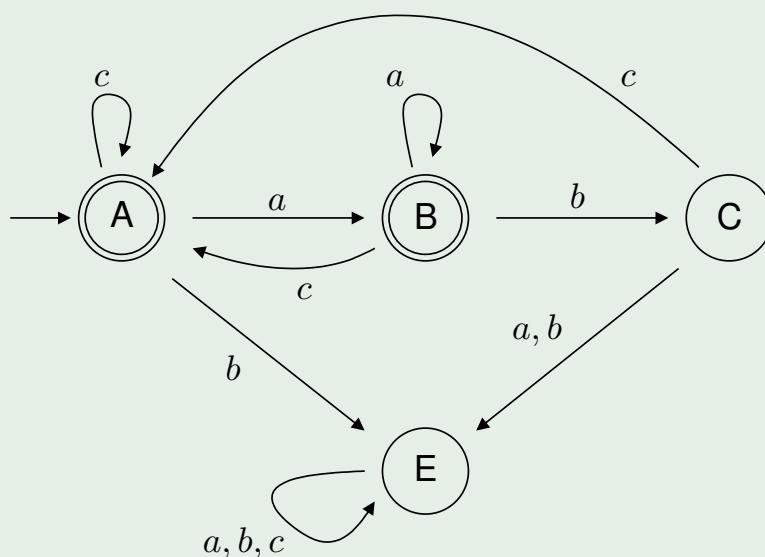
Q Considere o autómato seguinte (o do exemplo 6) e compare os estados A e D. Que pode concluir?



- São equivalentes. Por conseguinte, podem ser fundidos

Redução de autómato finito determinista (2)

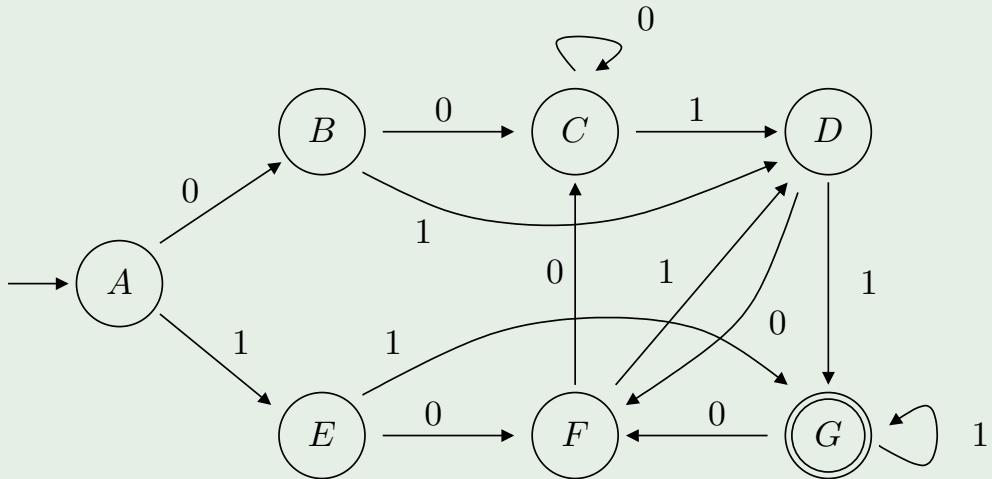
- O que resulta em



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

Algoritmo de Redução de AFD (1)

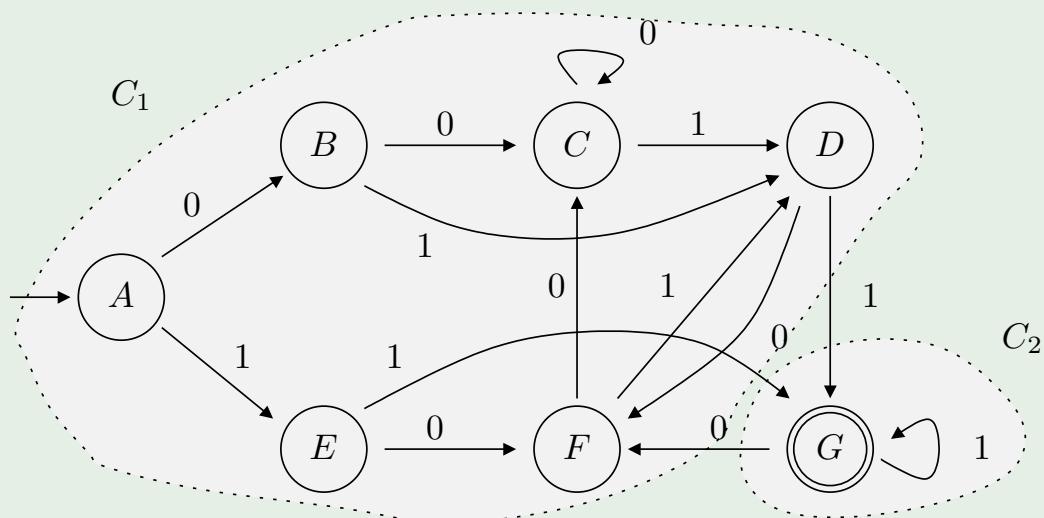
- Como proceder para reduzir um AFD?



- 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)

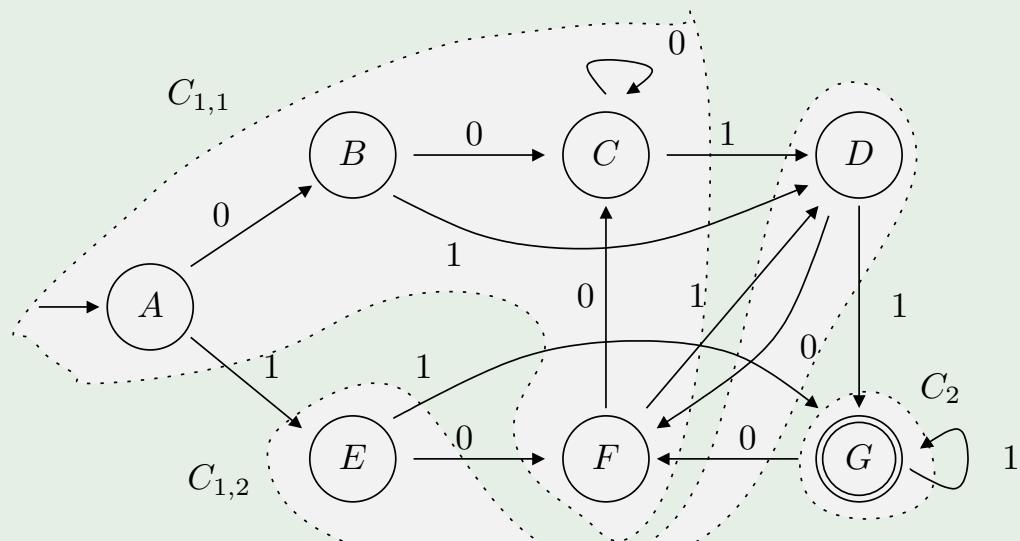
- Obtêm-se $C_1 = \{A, B, C, D, E, F\}$ e $C_2 = \{G\}$.



- 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)

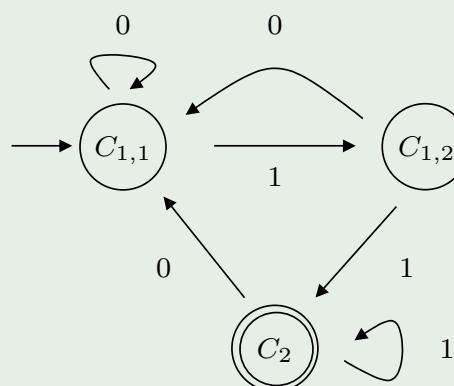
- Dividindo C_1 em $C_{1,1} = \{A, B, C, F\}$ e $C_{1,2} = \{D, E\}$ obtém-se



- 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 autómato.

Algoritmo de Redução de AFD (4)

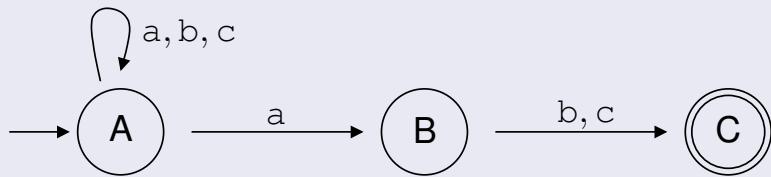
- Autómato reduzido



- Nos apontamentos encontra uma versão não gráfica do algoritmo.

Autómato finito não determinista

Um **autómato finito não determinista** é um autómato finito

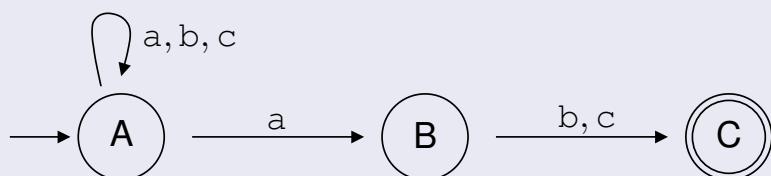


onde

- 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 autómato.
-
- 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 ?



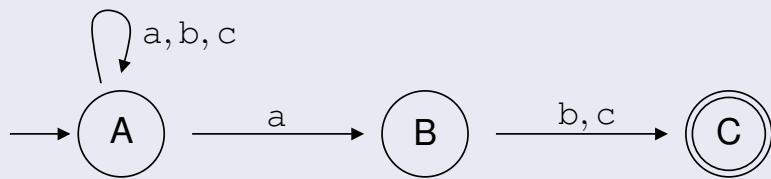
- Há 3 caminhos alternativos

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

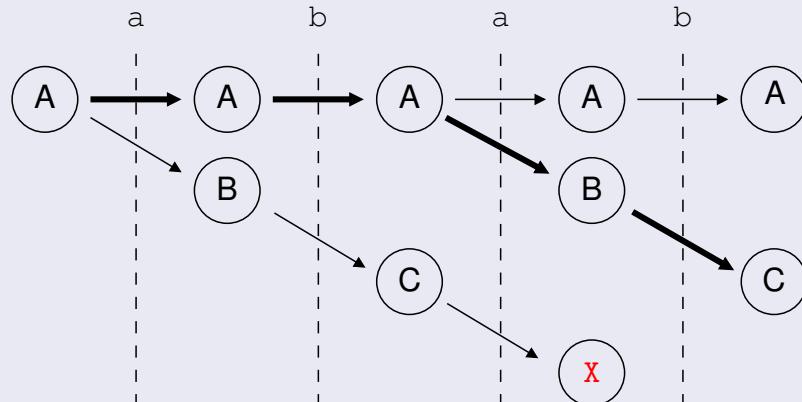
- Como há um caminho que conduz a um estado de aceitação a palavra é reconhecida pelo autómato

AFND: caminhos alternativos

- Analise o processo de reconhecimento da palavra abab ?

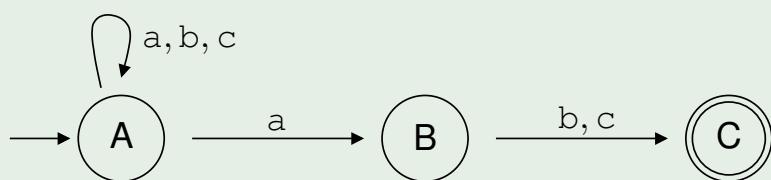


- Que se podem representar de forma arbórea



AFND: exemplo

Q Que palavras são reconhecidas pelo autómato seguinte?



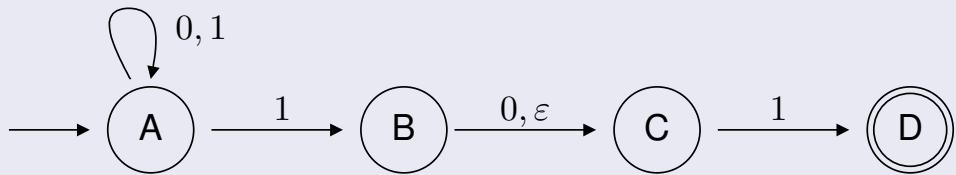
R Todas as palavras que terminarem em ab ou ac

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

- 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 autómato através do caminho

$$A \xrightarrow{1} B \xrightarrow{0} C \xrightarrow{1} D$$

- A palavra 11 é reconhecida pelo autómato através do caminho

$$A \xrightarrow{1} B \xrightarrow{\varepsilon} C \xrightarrow{1} D$$

porque $11 = 1\varepsilon 1$

- Este autómato 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

D Um **autómato finito não determinista** (AFND) é um quíntuplo

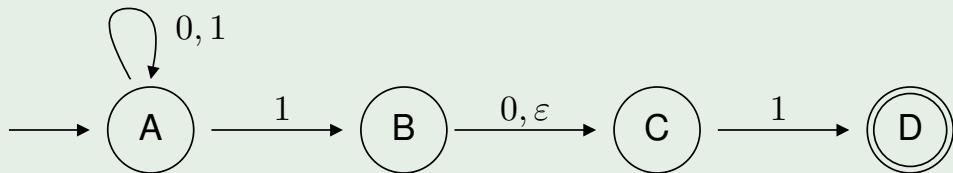
$$M = (A, Q, q_0, \delta, F),$$
 em que:

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

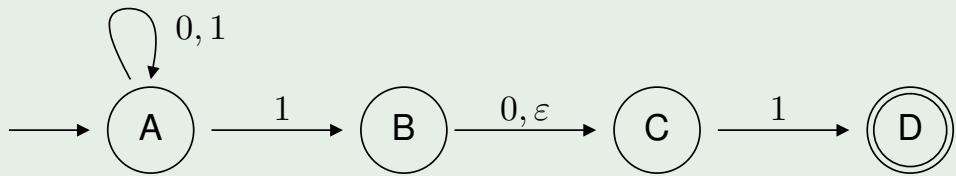
AFND: linguagem reconhecida

- 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, \dots, s_n , que satisfaça as seguintes condições:
 - 1 $s_0 = q_0$;
 - 2 qualquer que seja o $i = 1, \dots, n$, $(s_{i-1}, u_i, s_i) \in \delta$;
 - 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

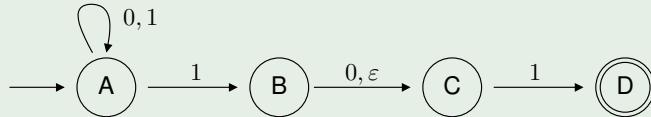
- A classe das linguagens cobertas por um AFD é a mesma que a classe das linguagens cobertas por um AFND
- Isto significa que:
 - Se M é um AFD, então $\exists_{M' \in \text{AFND}} : L(M') = L(M)$.
 - Se M é um AFND, então $\exists_{M' \in \text{AFD}} : L(M') = L(M)$.

- 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

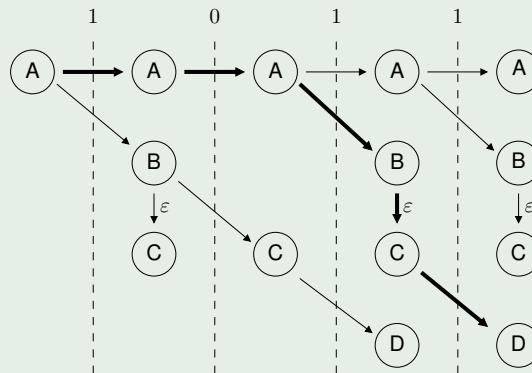
Equivalente AFD de um AFND (1)

- 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



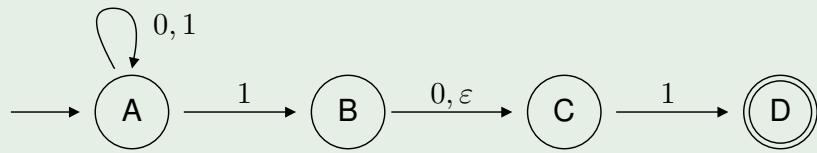
Equivalente AFD de um AFND (2)

- 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

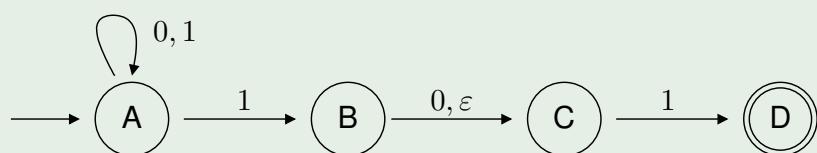
- $Q' = \{X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, x_{10}, X_{11}, X_{12}, X_{13}, X_{14}, X_{15}\}$, com

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

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

Equivalente AFD de um AFND: exemplo

Q Determinar um AFD equivalente ao AFND seguinte ?



R

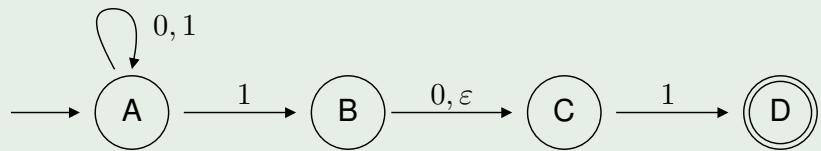
- $\delta' =$

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 ?

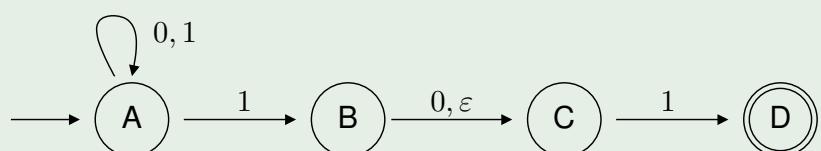


R

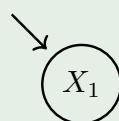
- Consegue-se o mesmo resultado através de um processo construtivo.

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



R

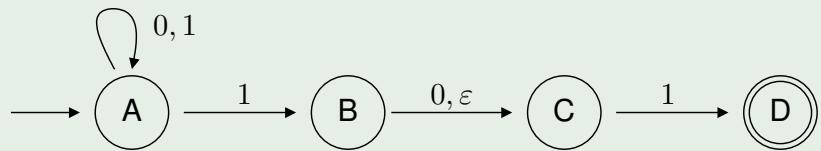


$$X_1 = \{A\}$$

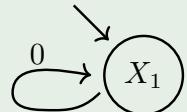
- Comece-se com o estado inicial ($X_1 = \{A\}$)

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



R

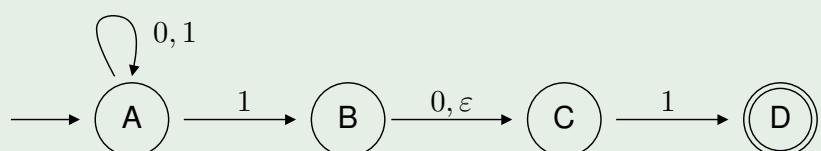


$$X_1 = \{A\}$$

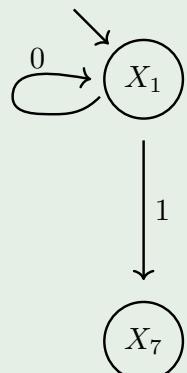
- $\delta'(X_1, 0) = \varepsilon\text{-closure}(A) = \{A\}$

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



R

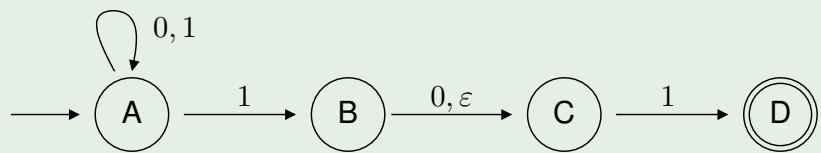


$$X_1 = \{A\}$$
$$X_7 = \{A, B, C\}$$

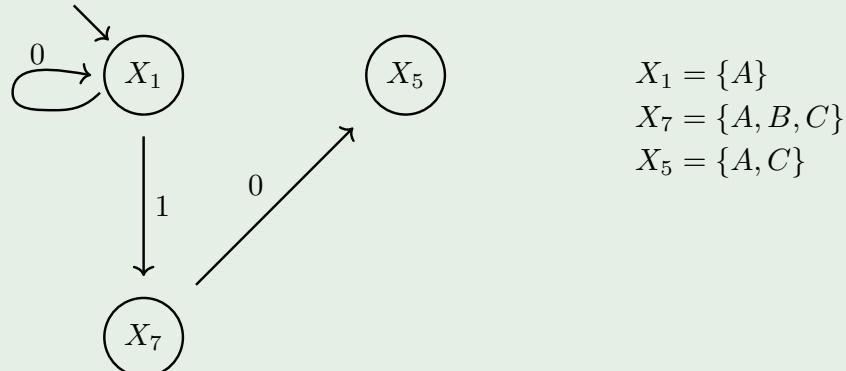
- $\delta'(X_1, 1) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(B) = \{A\} \cup \{B, C\} = \{A, B, C\}$

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



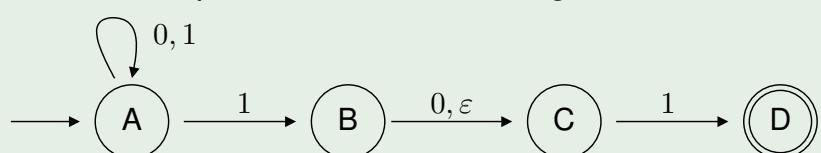
R



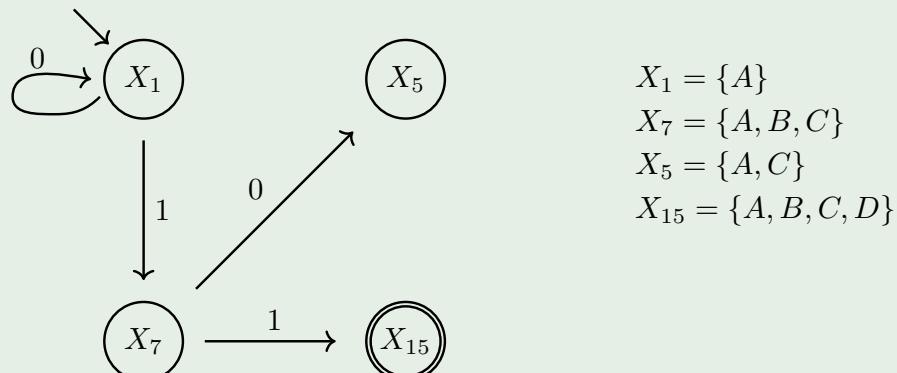
- $\delta'(X_7, 0) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(C) = \{A\} \cup \{C\} = \{A, C\}$

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



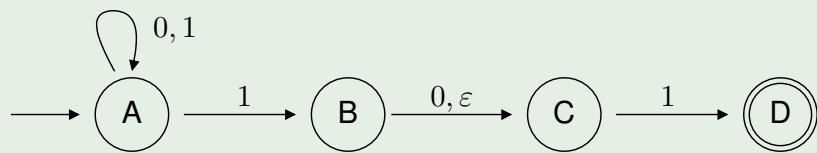
R



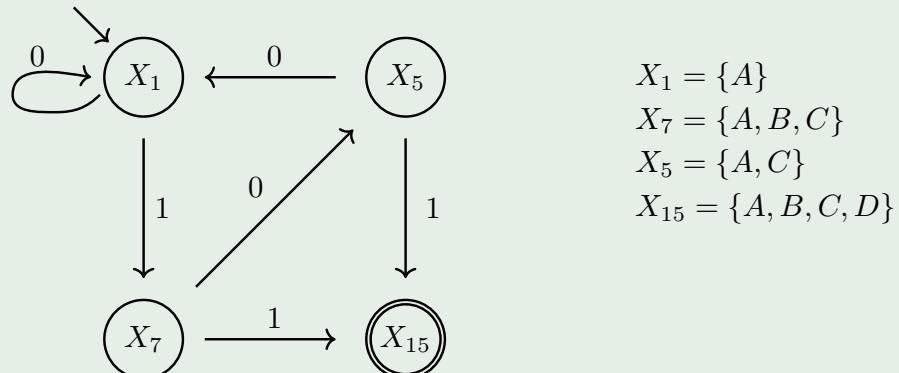
- $\delta'(X_7, 1) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(B) \cup \varepsilon\text{-closure}(D) = \{A\} \cup \{B, C\} \cup \{D\} = \{A, B, C, D\}$
- É de aceitação porque $\{A, B, C, D\} \cap \{D\} \neq \emptyset$

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



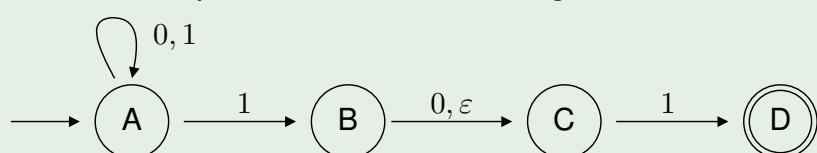
R



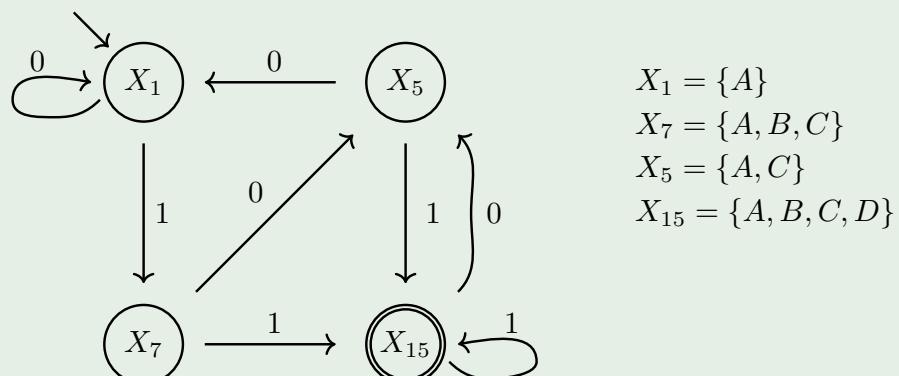
- $\delta'(X_5, 0) = \varepsilon\text{-closure}(A) = \{A\}$
- $\delta'(X_5, 1) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(B) \cup \varepsilon\text{-closure}(D) = \{A\} \cup \{B, C\} \cup \{D\} = \{A, B, C, D\}$

Equivalente AFD de um AFND: exemplo (2)

Q Determinar um AFD equivalente ao AFND seguinte ?



R

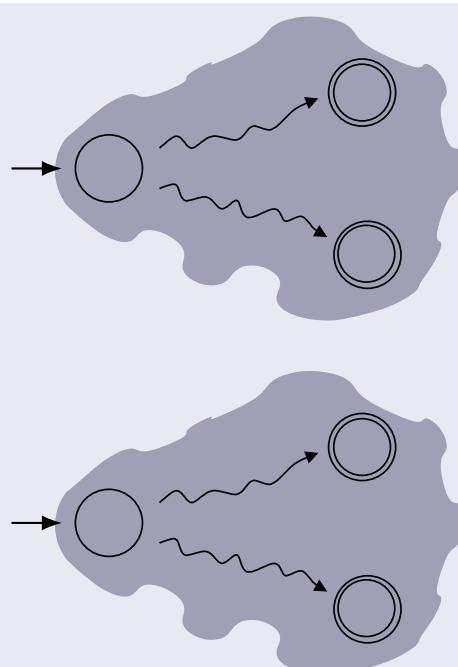


- $\delta'(X_{15}, 0) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(C) = \{A\} \cup \{C\} = \{A, C\}$
- $\delta'(X_{15}, 1) = \varepsilon\text{-closure}(A) \cup \varepsilon\text{-closure}(B) \cup \varepsilon\text{-closure}(D) = \{A\} \cup \{B, C\} \cup \{D\} = \{A, B, C, D\}$

Operações sobre AFD e AFND

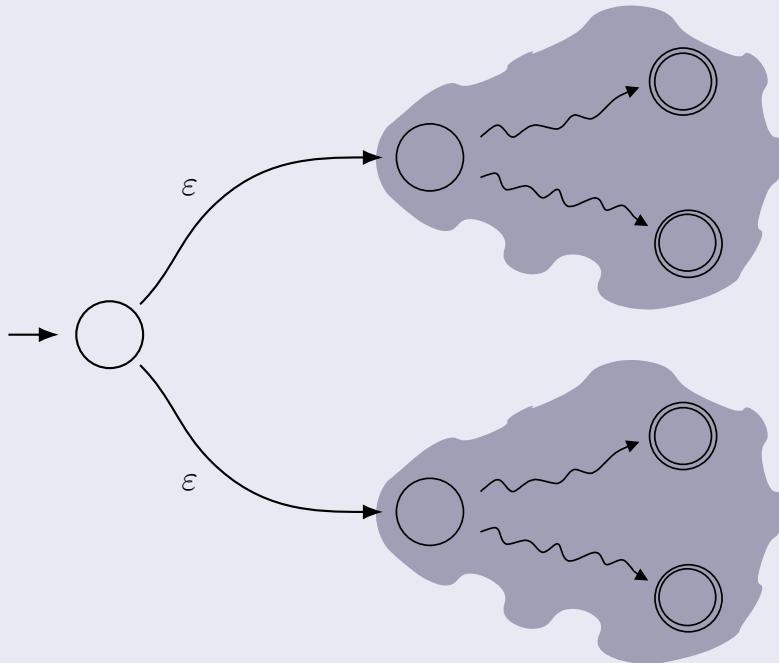
- Os automátos finitos (AF) são fechados sobre as operações de:
 - Reunião
 - Concatenação
 - Fecho
 - Intersecção
 - Complementação

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:

$$L_1 = \{\omega a \mid \omega \in A^*\} \quad L_2 = \{a\omega \mid \omega \in A^*\}$$

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 ?

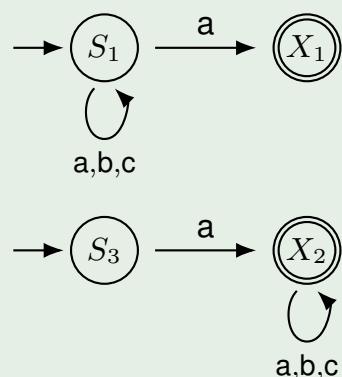
Reunião de AF: exemplo (1)

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

$$L_1 = \{\omega a \mid \omega \in A^*\} \quad L_2 = \{a\omega \mid \omega \in A^*\}$$

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)

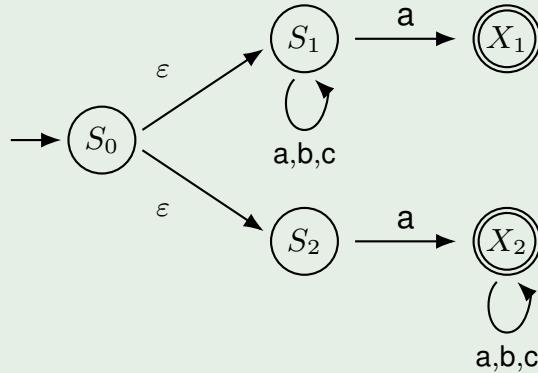
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

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

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

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

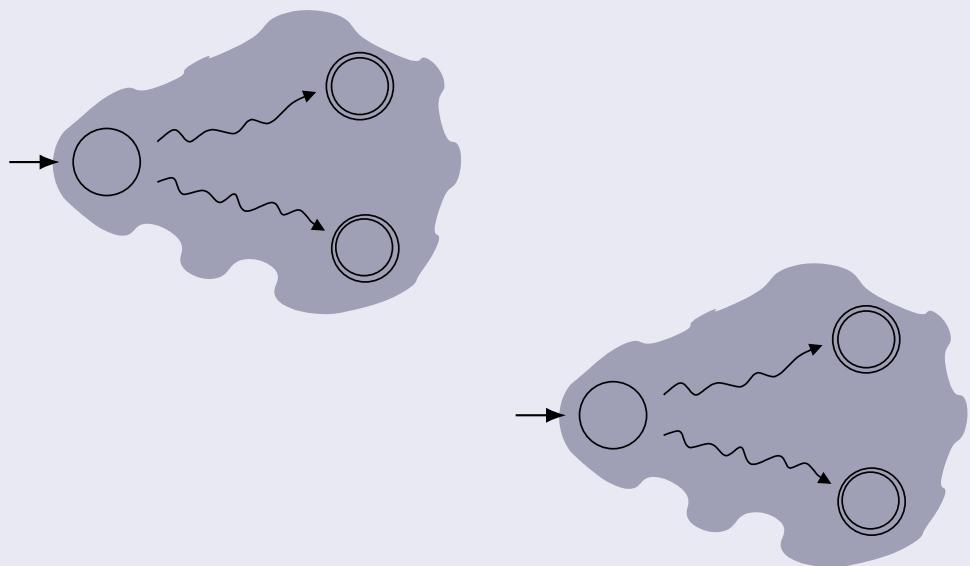
$$Q = \{S_0, S_1, X_1, S_2, X_2\}, \quad q_0 = S_0, \quad F = \{X_1, X_2\},$$

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

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

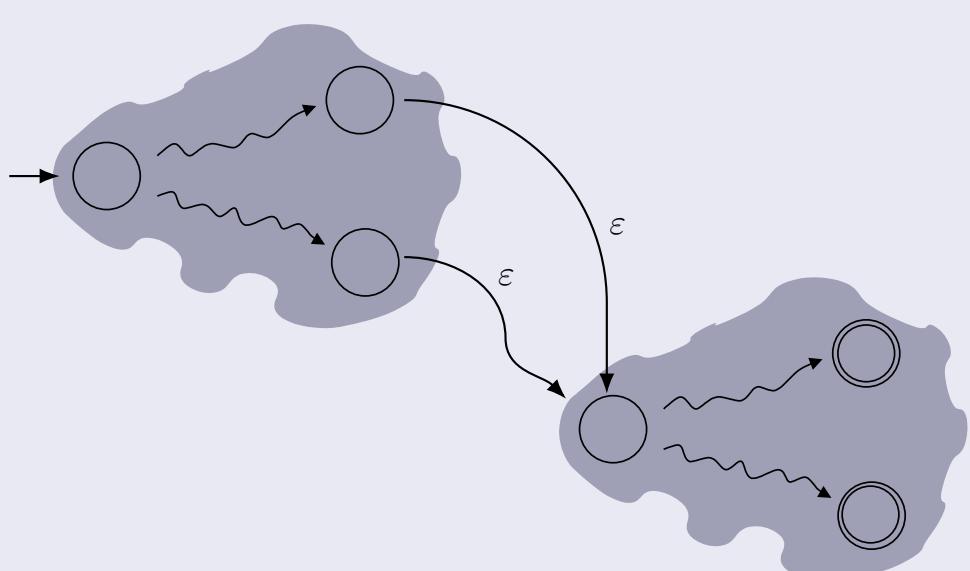
- Alternativamente, pode ser escrito de forma textual

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 $L = L_1 \cdot L_2$.

\mathcal{R}

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

Concatenação de AF: exemplo

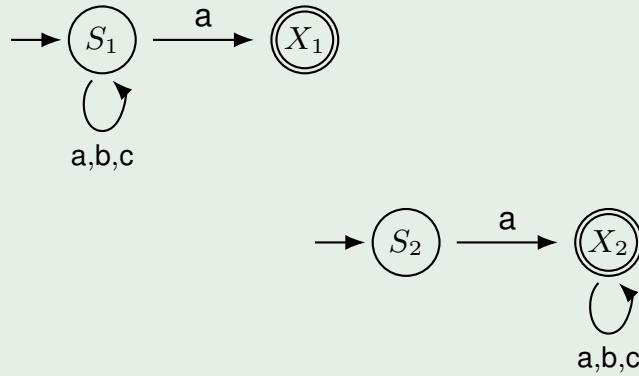
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

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

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

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

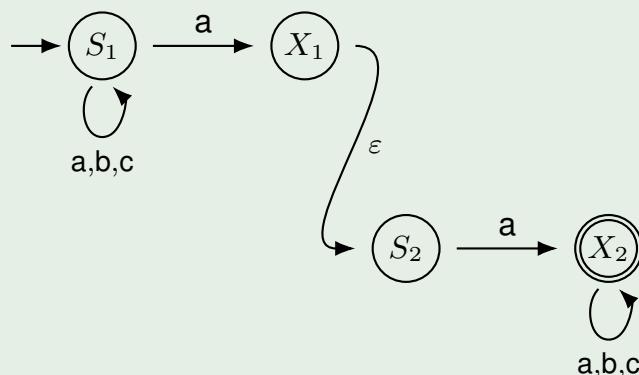
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

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

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

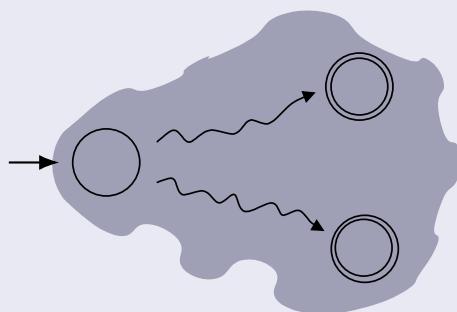
Determine um AF que reconheça $L = L_1 \cdot L_2$.

R



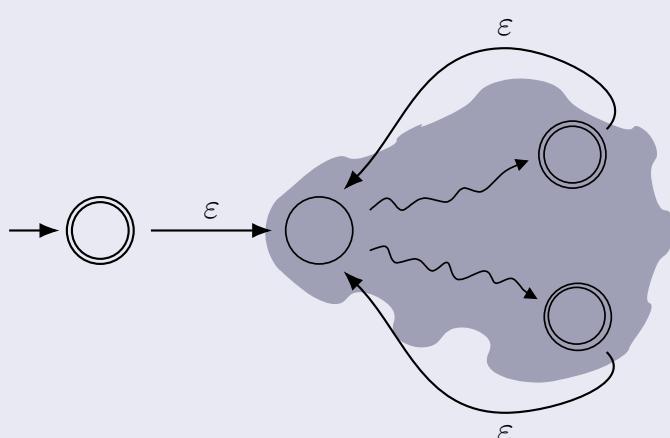
- X_1 deixa de ser de aceitação; S_2 deixa de ser de entrada
- acrescenta-se uma transição- ϵ de X_1 para S_2

Fecho de AF



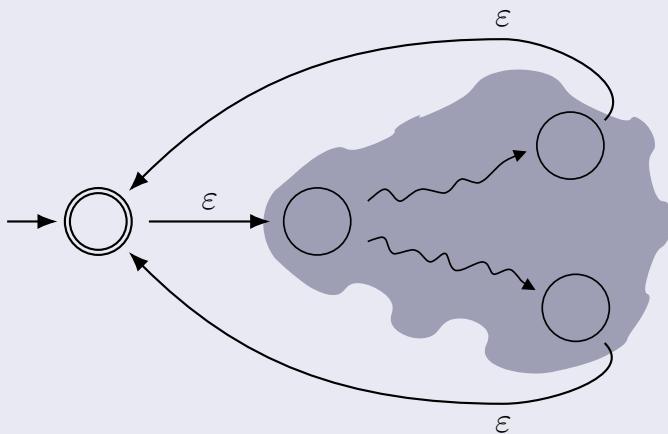
- Como criar um AF que represente o fecho deste AF?

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

\mathcal{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

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

$$F = \{q_0\}$$

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

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 ?

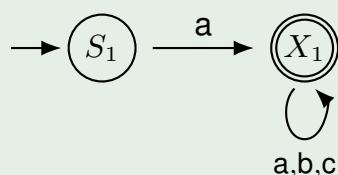
Fecho de AF: exemplo

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

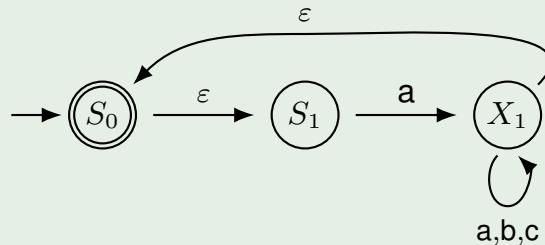
Fecho de AF: exemplo

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:

$$L_1 = \{\omega a \mid \omega \in A^*\} \qquad \qquad L_2 = \{a\omega \mid \omega \in A^*\}$$

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

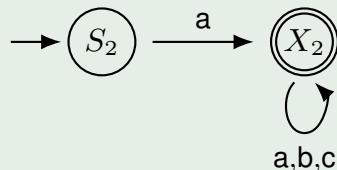
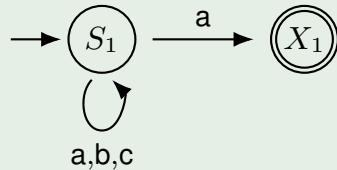
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

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

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

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

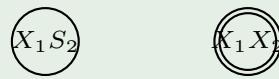
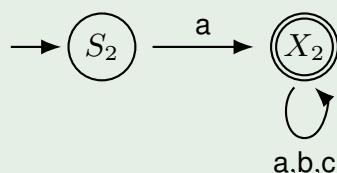
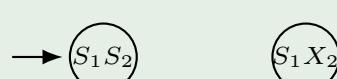
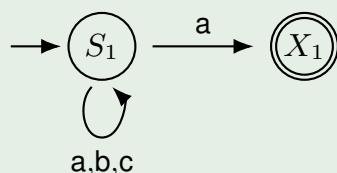
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

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

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

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

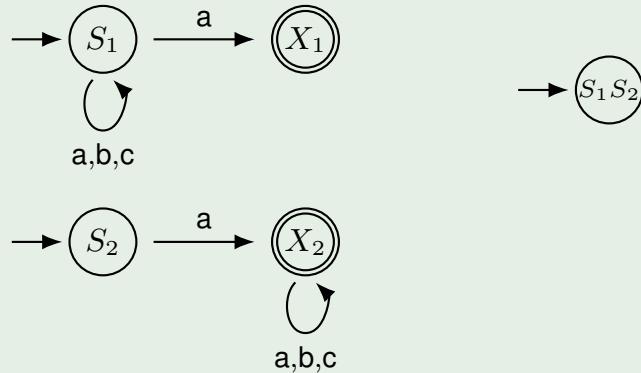
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

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

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

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

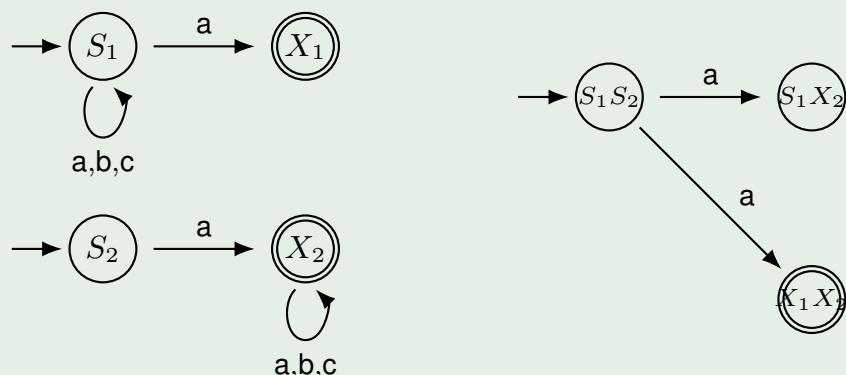
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

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

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

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

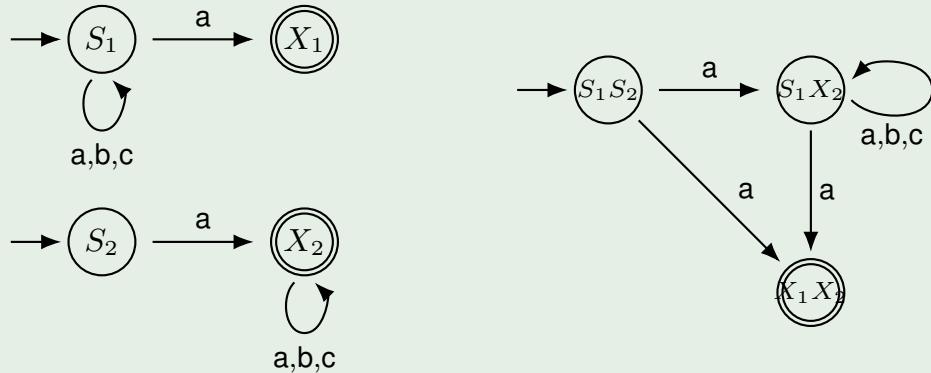
Q Sobre o alfabeto $A = \{a, b, c\}$, sejam L_1 e L_2 as duas linguagens seguintes:

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

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

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 $S_1 X_2 \xrightarrow{x} S_1 X_2$, para $x \in \{a, b, c\}$
- de $S_1 \xrightarrow{a} X_1$ e $X_2 \xrightarrow{a} X_2$ aparece $S_1 X_2 \xrightarrow{a} X_1 X_2$

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

$$Q = Q_1 \times Q_2$$

$$q_0 = (q_1, q_2)$$

$$F = F_1 \times F_2$$

$$\delta \subseteq (Q_1 \times Q_2) \times A_\varepsilon \times (Q_1 \times Q_2)$$

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 autómato finito determinista (em sentido estrito, ie. com todos os estados representados) basta complementar o conjunto de aceitação
- Para o caso de um autómato 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 ?

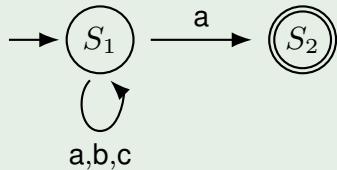
Complementação de AF: exemplo

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

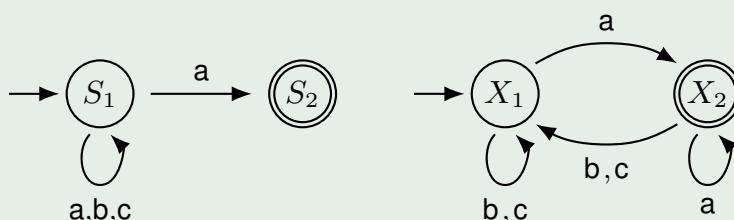
Complementação de AF: exemplo

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

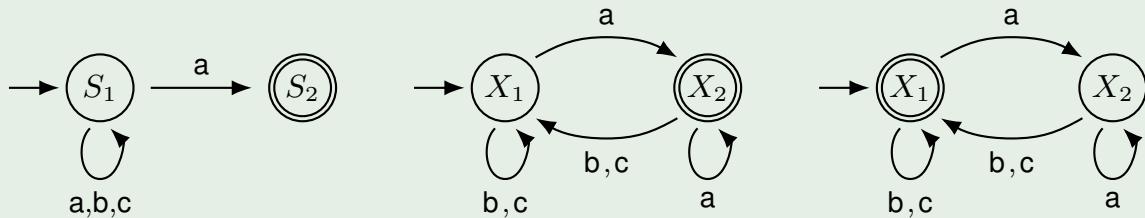
Complementação de AF: exemplo

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
- $L_3 = L_1 \cup L_2$
- $L_4 = L_1 \cdot L_2$
- $L_6 = \underline{L_1} \cap L_2$
- $L_7 = \underline{L_2}$
- $L_8 = (L_4 \cup L_3)^*$

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

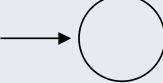
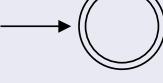
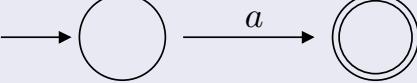
Conversão de uma ER num AF

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

Autómatos dos elementos primitivos

expressão regular	autómato finito
\emptyset	
ϵ	
a	

- Na realidade, o autómato referente a ϵ pode ser obtido aplicando o fecho ao autómato de \emptyset

Conversão de uma ER num AF

Algoritmo de conversão

- Se a expressão regular é do tipo primitivo, o autómato correspondente pode ser obtido da tabela anterior
- Se é do tipo e^* , aplica-se este mesmo algoritmo na obtenção de um autómato equivalente à expressão regular e e, de seguida, aplica-se o fecho de autómatos
- Se é do tipo e_1e_2 , aplica-se este mesmo algoritmo na obtenção de autómatos para as expressões e_1 e e_2 e, de seguida, aplica-se a concatenação de autómatos
- Finalmente, se é do tipo $e_1|e_2$, aplica-se este mesmo algoritmo na obtenção de autómatos para as expressões e_1 e e_2 e, de seguida, aplica-se a reunião de autómatos

- 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

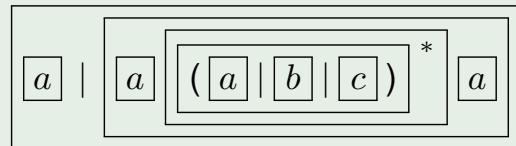
Conversão de uma ER num AF

Exemplo

Q Construa um autómato equivalente à expressão regular $e = a|a(a|b|c)^*a$

R

1 Decomposição



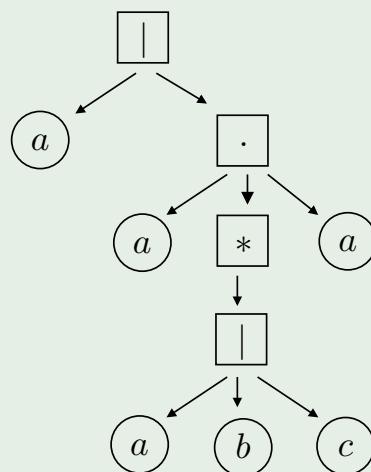
Conversão de uma ER num AF

Exemplo

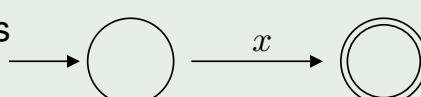
Q Construa um autómato equivalente à expressão regular $e = a|a(a|b|c)^*a$

R

1 Decomposição



2 Autómatos primitivos

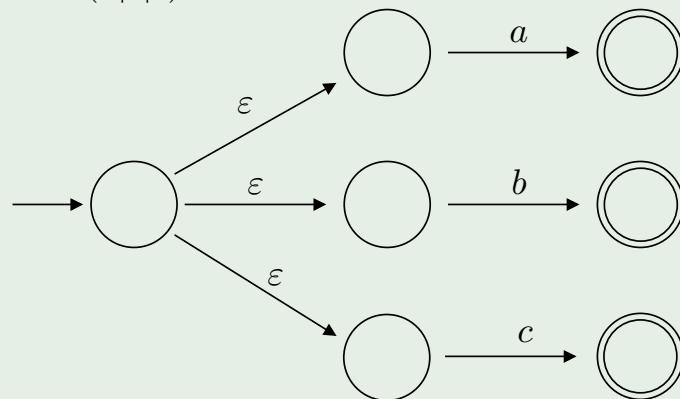


com $x = \{a, b, c\}$

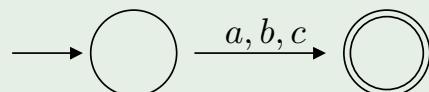
Conversão de uma ER num AF

Exemplo

- 3 Reunião para obter $(a|b|c)$



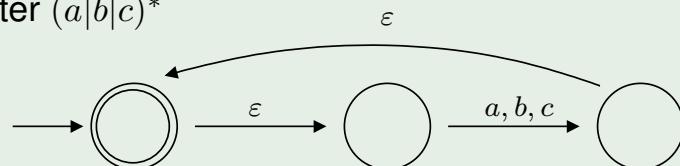
- 4 Simplificando



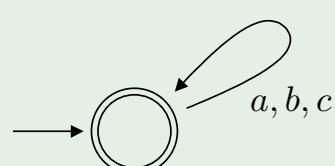
Conversão de uma ER num AF

Exemplo

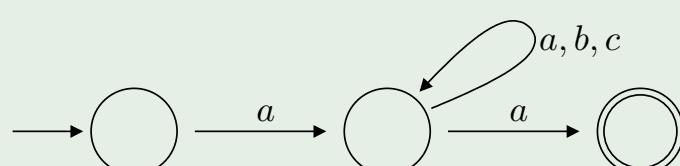
- 5 Fecho para obter $(a|b|c)^*$



- 6 Simplificando



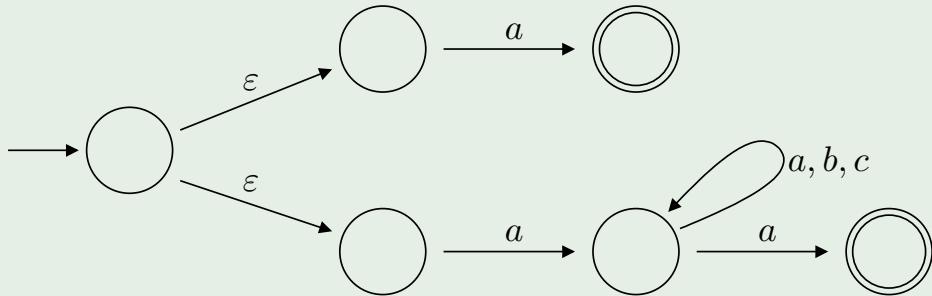
- 7 Concatenação (já com simplificação) para obter $a(a|b|c)^*a$



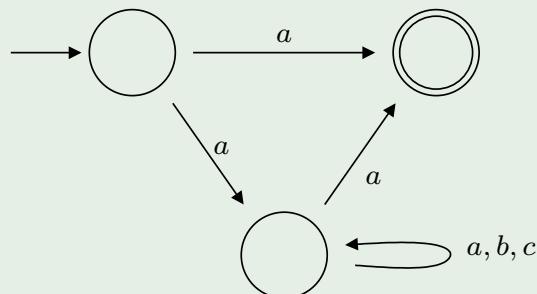
Conversão de uma ER num AF

Exemplo

- 8 Finalmente obtenção de $a|a(a|b|c)^*a$



- ## 9 Simplificando



Autómato finito generalizado (AFG)

Definição

D Um autómato finito generalizado (AFG) é um quíntuplo

$M = (A, Q, q_0, \delta, F)$, em que:

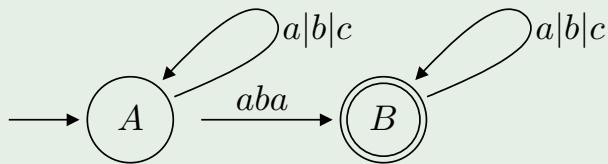
- A é o alfabeto de entrada
 - Q é um conjunto finito não vazio de estados
 - $q_0 \in Q$ é o estado inicial
 - $\delta \subseteq (Q \times E \times Q)$ é a relação de transição entre estados, sendo E o conjunto das expressões regulares definidas sobre A
 - $F \subseteq Q$ é o conjunto dos estados de 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 autómatos finitos generalizados

Autómato 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

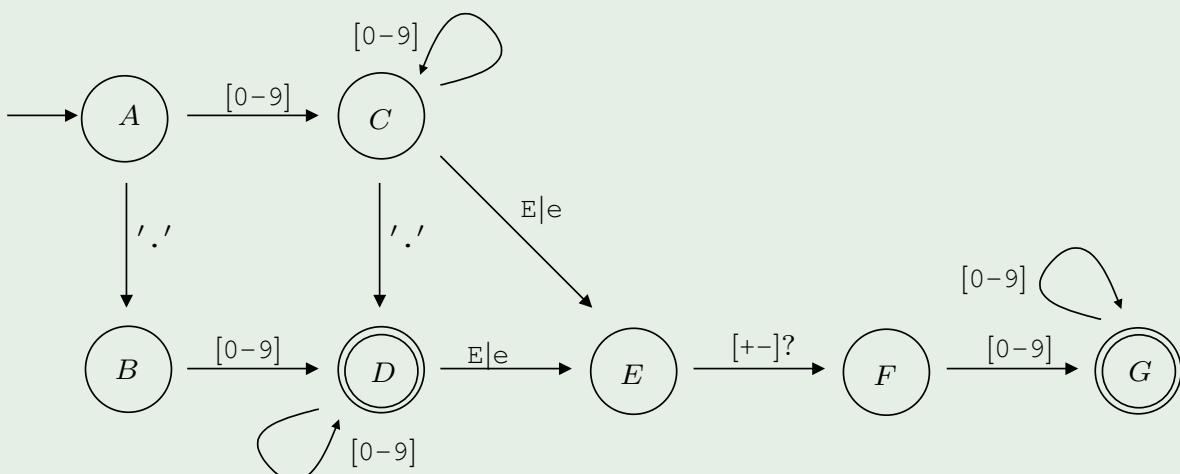


- Note que a etiqueta das transições $A \rightarrow A$ e $B \rightarrow B$ é $a|b|c$ (uma expressão regular) e não a, b, c (que representa 3 transições, uma em a , uma em b e uma em c)

Autómato finito generalizado (AFG)

Exemplo

- O AFG seguinte representa as constantes reais em C

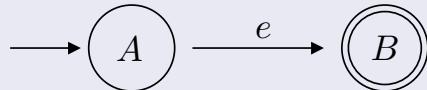


- Note que se usou ' . ' e não . , porque o último é uma expressão regular que representa qualquer letra do alfabeto

Conversão de um AFG numa ER

Abordagem

D UM AFG com a forma



designa-se por **autómato 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 autómato
- 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

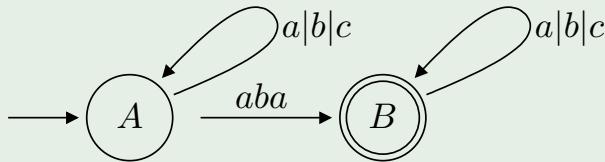
Conversão de um AFG numa ER

Ilustração com um exemplo

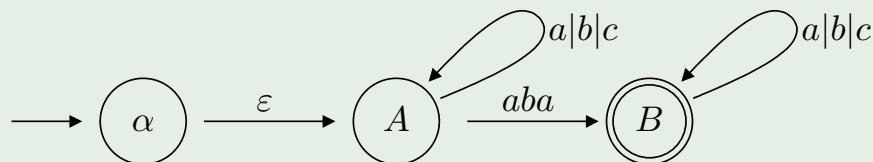
- 1 transformação de um AFG noutro cujo estado inicial **não tenha transições a chegar**

- Se necessário, acrescenta-se um novo estado inicial com uma transição em ε para o antigo

antes



depois



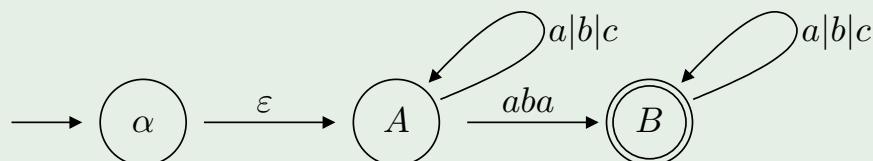
Conversão de um AFG numa ER

Ilustração com um exemplo

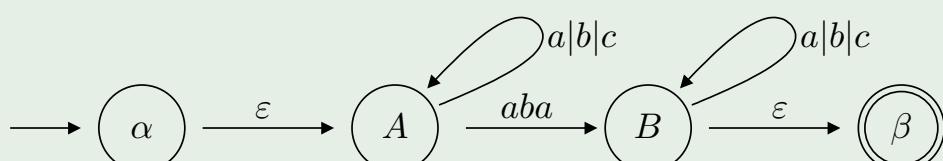
- 2 transformação de um AFG noutro com **um único estado de aceitação e sem transições de saída**

- 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



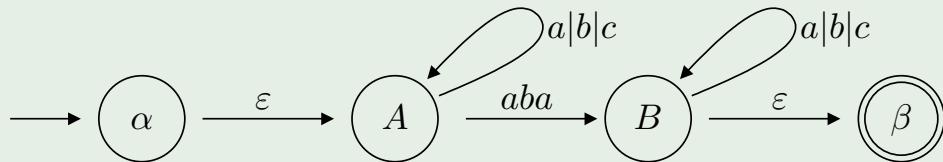
Conversão de um AFG numa ER

Ilustração com um exemplo

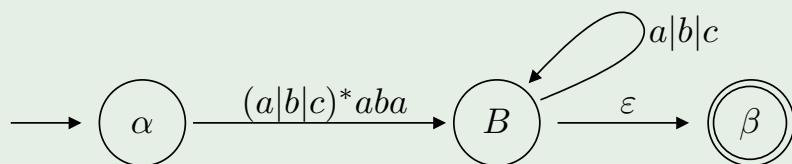
3 Eliminação dos restantes estados

- 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



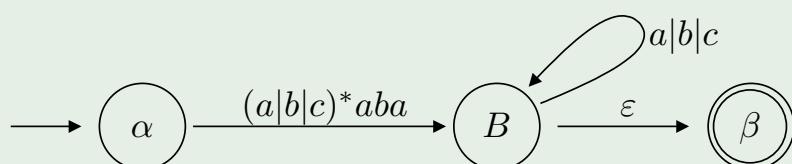
Conversão de um AFG numa ER

Ilustração com um exemplo

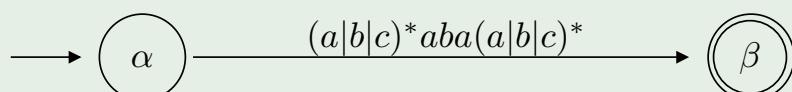
3 Eliminação dos restantes estados

- 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

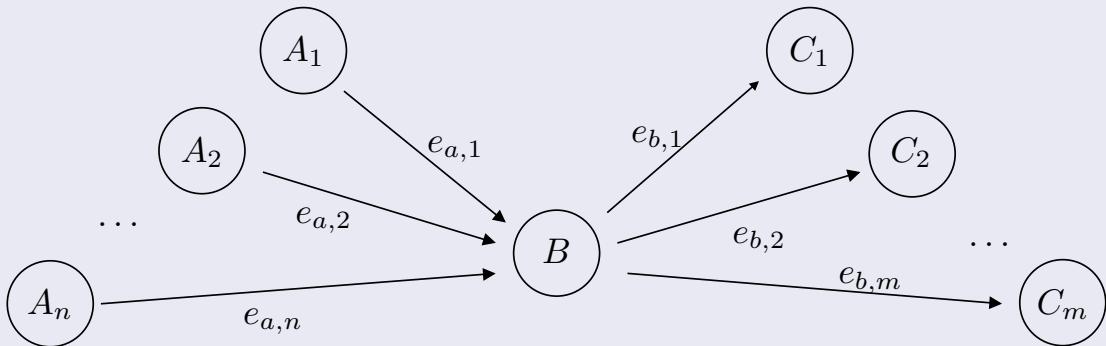


- Sendo $(a|b|c)^*aba(a|b|c)^*$ a expressão regular pretendida

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

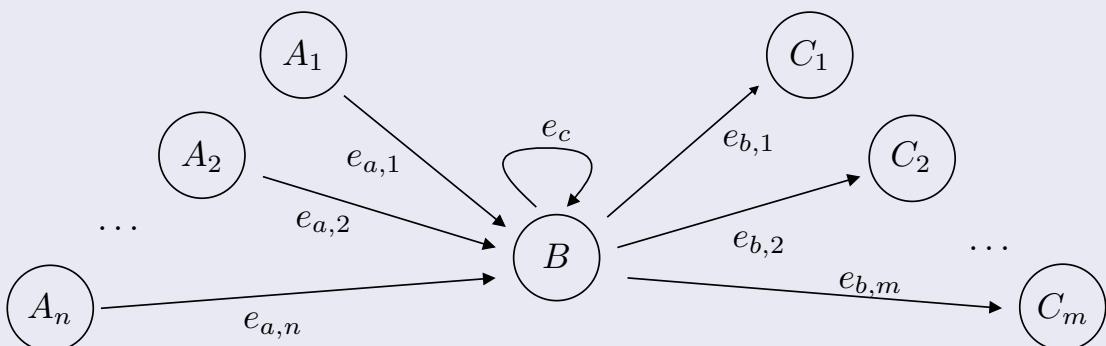


- Pode acontecer que haja $A_i = C_j$
- Para ir de A_i para C_j através de B , para $i = 1, 2, \dots, n$ e $j = 1, 2, \dots, m$, é preciso uma palavra que encaixe na expressão regular $(e_{a,i})(e_{b,j})$
- Então, se se retirar B , é preciso acrescentar uma transição de A_i para C_j que contemple essas 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

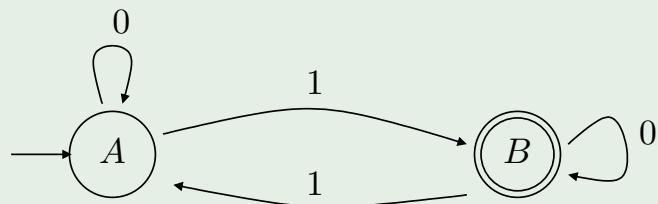


- Pode acontecer que haja $A_i = C_j$
- Para ir de A_i para C_j através de B , para $i = 1, 2, \dots, n$ e $j = 1, 2, \dots, m$, é preciso uma palavra que encaixe na expressão regular $(e_{a,i})(e_c)^*(e_{b,j})$
- Então, se se retirar B , é preciso acrescentar uma transição de A_i para C_j que contemple 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

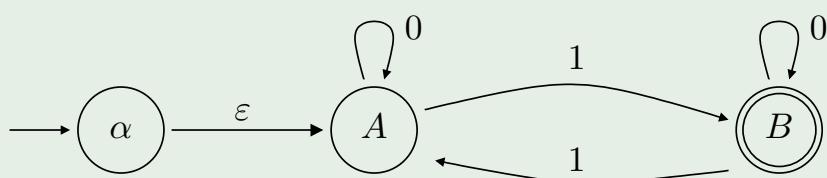
Exercício

Q Obtenha uma ER equivalente ao AF seguinte



R Aplique-se passo a passo o algoritmo de conversão

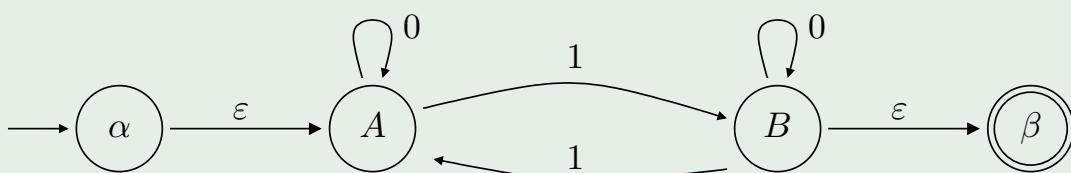
- Porque o estado inicial possui uma transição a entrar, deve substituir-se o estado inicial, de acordo com o passo 1 do algoritmo



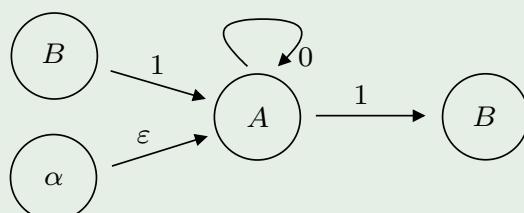
Exemplo de conversão de um AFG numa ER

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.

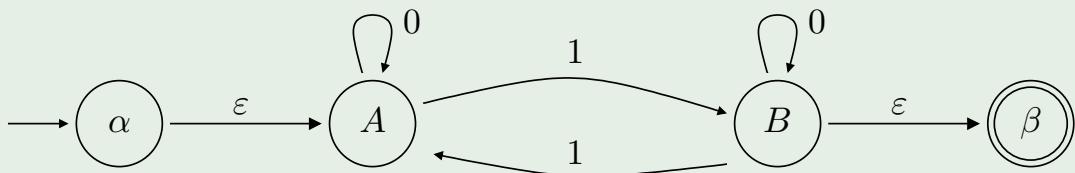


- Note que B aparece à esquerda e à direita

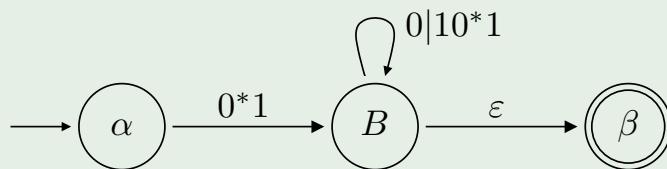
Exemplo de conversão de um AFG numa ER

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 $0^*1(0|10^*)^*$

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

A Seja $M = (A, Q, q_0, \delta, F)$ um autómato finito qualquer.

A GR $G = (T, N, P, S)$, onde

$$T = A$$

$$N = Q$$

$$S = q_0$$

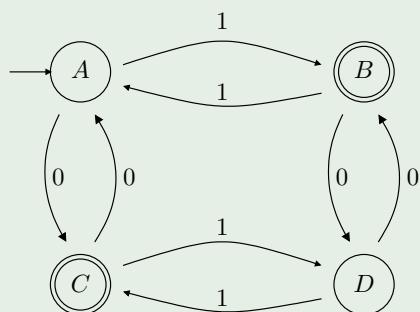
$$\begin{aligned}P = \{p \rightarrow aq : p, q \in Q \wedge a \in T \wedge (p, a, q) \in \delta\} \\ \cup \{p \rightarrow \varepsilon : p \in F\}\end{aligned}$$

representa a mesma linguagem que M , isto é, $L(G) = L(M)$.

Conversão de um AF numa GR

Exemplo

Q Determine uma GR equivalente ao AF



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

- A Seja $G = (T, N, P, S)$ uma gramática regular qualquer.
O AF $M = (A, Q, q_0, \delta, F)$, onde

$$A = T$$

$$Q = N \cup \{q_f\}, \quad \text{com } q_f \notin N$$

$$q_0 = S$$

$$F = \{q_f\}$$

$$\begin{aligned} \delta = & \{(q_i, e, q_j) : q_i, q_j \in N \wedge e \in T^* \wedge q_i \xrightarrow{e} q_j \in P\} \\ & \cup \{(q, e, q_f) : q \in N \wedge e \in T^* \wedge q \xrightarrow{e} q_f \in F\} \end{aligned}$$

representa a mesma linguagem que G , isto é, $L(M) = L(G)$.

Conversão de uma GR num AFG

Exemplo

- Q Determine um AFG equivalente à GR

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

R

Sendo $M = (A, Q, q_0, \delta, F)$ o AFG equivalente, tem-se

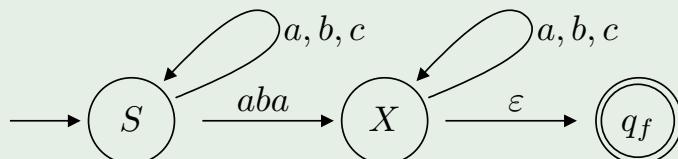
$$A = \{a, b, c\}$$

$$Q = \{S, X, q_f\}$$

$$q_0 = S$$

$$\begin{aligned} \delta = & \{(S, a, S), (S, b, S), (S, c, S), (S, aba, X), \\ & (X, a, X), (X, b, X), (X, c, X), (X, \varepsilon, q_f)\} \end{aligned}$$

$$F = \{q_f\}$$





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

$$D \rightarrow T L ;$$

$$T \rightarrow i \mid r$$

$$L \rightarrow v \mid L , v$$

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
$$D \Rightarrow TL ; \Rightarrow TL , v ; \Rightarrow T v , v ; \Rightarrow i v , v ;$$
- 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

$$D \rightarrow T L ;$$

$$T \rightarrow i \mid r$$

$$L \rightarrow v \mid L , v$$

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

$$D \Rightarrow TL ; \Rightarrow TL , v ; \Rightarrow T v , v ; \Rightarrow i v , v ;$$

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

$$\begin{array}{l} S \rightarrow i \ c \ S \\ | \\ i \ c \ S \ e \ S \\ | \\ a \end{array}$$

e aplique-se o procedimento anterior à palavra `i c i c a e a`

pilha	entrada	próxima ação
	<code>i c i c a e a \$</code>	deslocamento
<code>i</code>	<code>c i c a e a \$</code>	deslocamento
<code>i c</code>	<code>i c a e a \$</code>	deslocamento
<code>i c i</code>	<code>c a e a \$</code>	deslocamento
<code>i c i c</code>	<code>a e a \$</code>	deslocamento
<code>i c i c a</code>	<code>e a \$</code>	redução por $S \rightarrow a$
<code>i c i c S</code>	<code>e a \$</code>	conflito: – redução por $S \rightarrow i \ c \ S$ – deslocamento para tentar $S \rightarrow i \ c \ S \ e \ S$

- Esta gramática representa uma estrutura típica em linguagens de programação.
Qual?

Análise sintática ascendente

Ilustração de conflito entre reduções

- Considere a gramática

$$\begin{array}{l} S \rightarrow A \\ | \\ B \\ A \rightarrow c \\ | \\ A \ a \\ B \rightarrow c \\ | \\ B \ b \end{array}$$

e aplique-se o procedimento anterior à palavra `c`

pilha	entrada	próxima ação
	<code>c \$</code>	deslocamento
<code>c</code>	<code>\$</code>	conflito: – redução usando $A \rightarrow c$ – redução usando $B \rightarrow c$

Análise sintática ascendente

Ilustração de falso conflito

- Considere a gramática

$$\begin{array}{l} S \rightarrow a \\ | \\ < S > \\ | \\ a P \\ | \\ < S > S \\ P \rightarrow < S > \\ | \\ < S > S \end{array}$$

e aplique-se o procedimento de reconhecimento à palavra “a < a > a”

pilha	entrada	próxima ação
a	a < a > a \$	deslocamento

- 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.

$$\begin{array}{l} S \rightarrow a \\ | \quad i \text{ c } S \\ | \quad i \text{ c } S' \text{ e } S \\ S' \rightarrow a \\ | \quad i \text{ c } S' \text{ e } S' \end{array}$$

$$\begin{array}{l} S \rightarrow a \\ | \quad i \text{ c } S \\ | \quad i \text{ c } S \text{ e } S \end{array}$$

Análise sintática ascendente

if..then..else sem conflitos

- Considere a gramática seguinte e processe-se a palavra “icicicaea”

$$\begin{array}{l} S \rightarrow a \mid i \text{ c } S \mid i \text{ c } S' \text{ e } S \\ S' \rightarrow a \mid i \text{ c } S' \text{ e } S' \end{array}$$

pilha	entrada	próxima ação
	icicicaea \$	deslocamento
i	cicicaea \$	deslocamento
ic	icaea \$	deslocamento
ici	aea \$	deslocamento
icic	a ea \$	deslocamento
icica	ea \$	redução por $S' \rightarrow a$ // $e \in \text{follow}(S')$, $e \notin \text{follow}(S)$
icicS'	e a \$	deslocamento
icicS'e	a \$	deslocamento
icicS'ea	\$	redução por $S \rightarrow a$ // $\$ \in \text{follow}(S)$, $\$ \notin \text{follow}(S')$
icicS'eS	\$	redução por $S \rightarrow i \text{ c } S' \text{ e } S$
icS	\$	redução por $S \rightarrow i \text{ c } S$
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:
$$A \rightarrow \cdot \ B_1 \ B_2 \ B_3$$
$$A \rightarrow B_1 \ \cdot \ B_2 \ B_3$$
$$A \rightarrow B_1 \ B_2 \ \cdot \ B_3$$
$$A \rightarrow B_1 \ B_2 \ B_3 \cdot$$
- 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
$$S \rightarrow E$$
$$E \rightarrow a \mid (\ E \)$$
- Reconhecer a palavra $u = u_1 u_2 \cdots u_n$, significa reduzir $u \$$ a $S \$$, então, o estado inicial no processo de reconhecimento pode ser definido por
$$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

$$Z_4 = \{ E \rightarrow (\cdot E) \}$$

- Apenas evolui por ocorrência de)

$$\delta(Z_4,) = \{ E \rightarrow (\cdot E) \} = Z_5 \quad \text{um estado novo}$$

- Z_5 apenas possui um item terminal, que representa uma situação passível de redução pela regra $E \rightarrow (\cdot E)$

- 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

$$Z_0 = \{ S \rightarrow \cdot E \$ \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (\cdot E) \}$$

$$Z_1 = \delta(Z_0, E) = \{ S \rightarrow E \cdot \$ \}$$

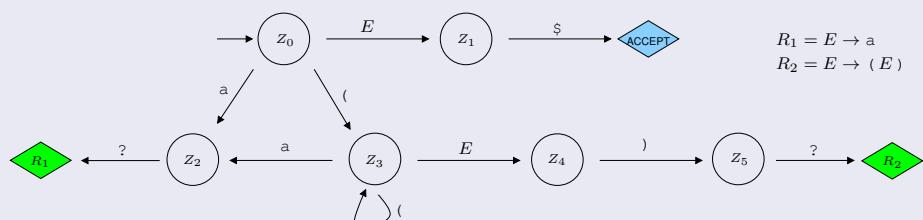
$$Z_2 = \delta(Z_0, a) = \{ E \rightarrow a \cdot \}$$

$$Z_3 = \delta(Z_0, ()) = \{ E \rightarrow (\cdot E) \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (\cdot E) \}$$

$$Z_4 = \delta(Z_3, E) = \{ E \rightarrow (\cdot E) \}$$

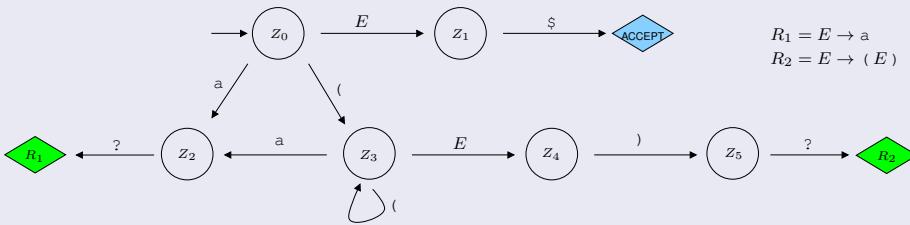
$$Z_5 = \delta(Z_4, ()) = \{ E \rightarrow (\cdot E) \}$$

- Representando na forma de um autómato, tem-se



Conjunto dos conjuntos de itens

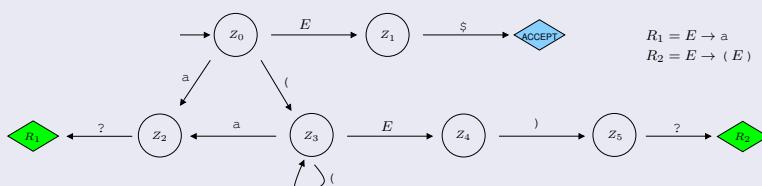
Ilustração com um exemplo (cont.)



- Neste autómato, 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 autómato de pilha que permite fazer o reconhecimento da linguagem

Tabela de decisão de um reconhecedor ascendente

Introdução



- O autómato 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

$$Z_0 = \{ S \rightarrow \cdot E \$ \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

$$Z_1 = \delta(Z_0, E) = \{ S \rightarrow E \cdot \$ \}$$

$$Z_2 = \delta(Z_0, a) = \{ E \rightarrow a \cdot \}$$

$$Z_3 = \delta(Z_0, ()) = \{ E \rightarrow (\cdot E) \} \cup \{ E \rightarrow \cdot a, E \rightarrow \cdot (E) \}$$

$$Z_4 = \delta(Z_3, E) = \{ E \rightarrow (E \cdot) \}$$

$$Z_5 = \delta(Z_4, ()) = \{ E \rightarrow (E) \cdot \}$$

- É o correspondente autómato de pilha

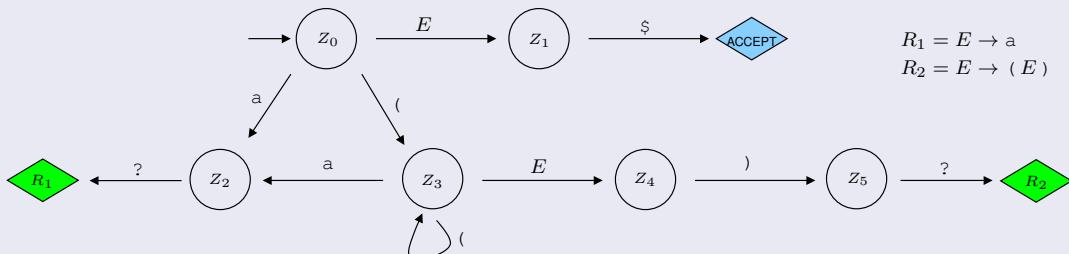
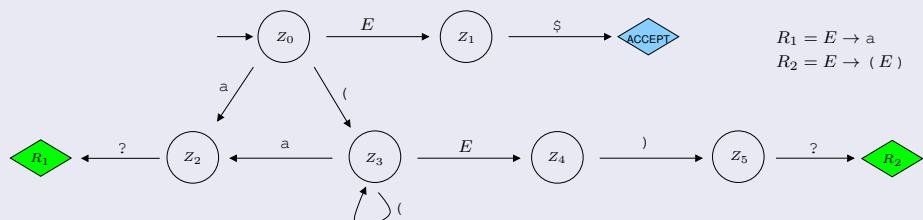


Tabela de decisão de um reconhecedor ascendente

Exemplo

- A este autómato 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 → a		reduce, E → a	
Z ₃	shift, Z ₂	shift, Z ₃				Z ₄
Z ₄			shift, Z ₅			
Z ₅			reduce, E → (E)	reduce, E → (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	$((a))\$$	ACTION(Z_0 , $()$) = (shift, Z_3)
$Z_0 Z_3$	$(a))\$$	ACTION(Z_3 , $()$) = (shift, Z_3)
$Z_0 Z_3 Z_3$	$a))\$$	ACTION(Z_3 , a) = (shift, Z_2)
$Z_0 Z_3 Z_3 Z_2$	$)\$$	ACTION(Z_2 , $)$) = (reduce $E \rightarrow a$) (1 pop)
$Z_0 Z_3 Z_3 Z_2$	$[E]$	GOTO(Z_3 , E) = Z_4 (push Z_4)
$Z_0 Z_3 Z_3 Z_4$	$)\$$	ACTION(Z_4 , $)$) = (shift, Z_5)
$Z_0 Z_3 Z_3 Z_4 Z_5$	$\$$	ACTION(Z_5 , $)$) = (reduce $E \rightarrow (E)$) (3 pops)
$Z_0 Z_3 Z_3 Z_4 Z_5$	$[E]$	GOTO(Z_3 , E) = Z_4 (push Z_4)
$Z_0 Z_3 Z_4$	$\$$	ACTION(Z_4 , $)$) = (shift, Z_5)
$Z_0 Z_3 Z_4 Z_5$	$\$$	ACTION(Z_5 , $\$$) = (reduce $E \rightarrow (E)$) (3 pops)
$Z_0 Z_3 Z_4 Z_5$	$[E]$	GOTO(Z_0 , E) = Z_1 (push Z_1)
$Z_0 Z_1$	$\$$	ACTION(Z_1 , $\$$) = 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

$$Z_0 = \{S_0 \rightarrow \cdot S \$\}$$

$$\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot aP, S \rightarrow \cdot (S)S\} \quad \text{fecho}$$

$$\delta(Z_0, S) = \{S_0 \rightarrow S \cdot \$\} = Z_1 \quad \text{um estado novo}$$

$$\delta(Z_0, a) = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} \quad \text{um estado novo}$$

$$\cup \{P \rightarrow \cdot (S), P \rightarrow \cdot (S)S\} = Z_2 \quad \text{fecho}$$

$$\delta(Z_0, ()) = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S)S\} \quad \text{um estado novo}$$

$$\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot aP, S \rightarrow \cdot (S)S\} = Z_3 \quad \text{fecho}$$

$$\delta(Z_2, P) = \{S \rightarrow aP \cdot\} = Z_4 \quad \text{um estado novo}$$

$$\delta(Z_2, ()) = \{P \rightarrow (\cdot S), P \rightarrow (\cdot S)S\} \quad \text{um estado novo}$$

$$\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot aP, S \rightarrow \cdot (S)S\} = Z_5 \quad \text{fecho}$$

$$\delta(Z_3, S) = \{S \rightarrow (S \cdot), S \rightarrow (S \cdot)S\} = Z_6 \quad \text{um estado novo}$$

$$\delta(Z_3, a) = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} = Z_2 \quad \text{um estado repetido}$$

$$\delta(Z_3, ()) = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S)S\} = Z_3 \quad \text{um estado repetido}$$

$$S_0 \rightarrow S$$

$$S \rightarrow a \mid (S) \mid aP \mid (S)S$$

$$P \rightarrow (S) \mid (S)S$$

Tabela de decisão de um reconhecedor ascendente

Exemplo #3 (cont.)

- continuando, apenas mostrando os elementos envolvidos em processamento

$Z_2 = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} \cup \dots$	
$Z_3 = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} \cup \dots$	
$Z_5 = \{P \rightarrow (\cdot S), P \rightarrow (\cdot S) S\}$	
$\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot a P, S \rightarrow \cdot (S) S\}$	
$Z_6 = \{S \rightarrow (S \cdot), S \rightarrow (S \cdot) S\}$	
$\delta(Z_5, S) = \{P \rightarrow (S \cdot), P \rightarrow (S \cdot) S\} = Z_7$	<i>um estado novo</i>
$\delta(Z_5, a) = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} = Z_2$	<i>um estado repetido</i>
$\delta(Z_5, ()) = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} = Z_3$	<i>um estado repetido</i>
$\delta(Z_6, ()) = \{S \rightarrow (S \cdot), S \rightarrow (S \cdot) S\}$	<i>um estado novo</i>
$\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot a P, S \rightarrow \cdot (S) S\} = Z_8$	
$\delta(Z_7, ()) = \{P \rightarrow (S \cdot), P \rightarrow (S \cdot) S\}$	<i>um estado novo</i>
$\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot a P, S \rightarrow \cdot (S) S\} = Z_9$	

$S_0 \rightarrow S$
 $S \rightarrow a \mid (S) \mid a P \mid (S) S$
 $P \rightarrow (S) \mid (S) S$

Tabela de decisão de um reconhecedor ascendente

Exemplo #3 (cont.)

- continuando...

$Z_2 = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} \cup \dots$	
$Z_3 = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} \cup \dots$	
$Z_8 = \{S \rightarrow (S \cdot), S \rightarrow (S \cdot) S\}$	
$\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot a P, S \rightarrow \cdot (S) S\}$	
$Z_9 = \{P \rightarrow (S \cdot), P \rightarrow (S \cdot) S\}$	
$\cup \{S \rightarrow \cdot a, S \rightarrow \cdot (S), S \rightarrow \cdot a P, S \rightarrow \cdot (S) S\}$	
$\delta(Z_8, S) = S \rightarrow (S \cdot) S \cdot \} = Z_{10}$	<i>um estado novo</i>
$\delta(Z_8, a) = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} = Z_2$	<i>um estado repetido</i>
$\delta(Z_8, ()) = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} = Z_3$	<i>um estado repetido</i>
$\delta(Z_9, S) = \{P \rightarrow (S \cdot) S \cdot\} = Z_{11}$	<i>um estado novo</i>
$\delta(Z_9, a) = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} = Z_2$	<i>um estado repetido</i>
$\delta(Z_9, ()) = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} = Z_3$	<i>um estado repetido</i>

$S_0 \rightarrow S$
 $S \rightarrow a \mid (S) \mid a P \mid (S) S$
 $P \rightarrow (S) \mid (S) S$

Tabela de decisão de um reconhecedor ascendente

Exemplo #3 (cont.)

- O que resulta em

$Z_0 = \{S_0 \rightarrow \cdot S \$\} \cup \dots$	$\delta(Z_0, S) = Z_1$	$\delta(Z_0, a) = Z_2$	$\delta(Z_0, ()) = Z_3$
$Z_1 = \{S_0 \rightarrow S \cdot \$\}$			
$Z_2 = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} \cup \dots$		$\delta(Z_2, P) = Z_4$	$\delta(Z_2, ()) = Z_5$
$Z_3 = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} \cup \dots$	$\delta(Z_3, S) = Z_6$	$\delta(Z_3, a) = Z_2$	$\delta(Z_3, ()) = Z_3$
$Z_4 = \{S \rightarrow a P \cdot\}$			
$Z_5 = \{P \rightarrow (\cdot S), P \rightarrow (\cdot S) S\} \cup \dots$	$\delta(Z_5, S) = Z_7$	$\delta(Z_5, a) = Z_2$	$\delta(Z_5, ()) = Z_3$
$Z_6 = \{S \rightarrow (S \cdot), S \rightarrow (S \cdot) S\}$			$\delta(Z_6, ()) = Z_8$
$Z_7 = \{P \rightarrow (S \cdot), P \rightarrow (S \cdot) S\}$			$\delta(Z_7, ()) = Z_9$
$Z_8 = \{S \rightarrow (S) \cdot, S \rightarrow (S) \cdot S\} \cup \dots$	$\delta(Z_8, S) = Z_{10}$	$\delta(Z_8, a) = Z_2$	$\delta(Z_8, ()) = Z_3$
$Z_9 = \{P \rightarrow (S) \cdot, P \rightarrow (S) \cdot S\} \cup \dots$	$\delta(Z_9, S) = Z_{11}$	$\delta(Z_9, a) = Z_2$	$\delta(Z_9, ()) = Z_3$
$Z_{10} = \{S \rightarrow (S) S \cdot\}$			
$Z_{11} = \{P \rightarrow (S) S \cdot\}$			

$S_0 \rightarrow S$
 $S \rightarrow a \mid (S) \mid a P \mid (S) S$
 $P \rightarrow (S) \mid (S) S$

Tabela de decisão de um reconhecedor ascendente

Exemplo #3 (cont.)

- O que resulta em

$Z_0 = \{S_0 \rightarrow \cdot S \$\} \cup \dots$	$\delta(Z_0, S) = Z_1$	$\delta(Z_0, a) = Z_2$	$\delta(Z_0, ()) = Z_3$
$Z_1 = \{S_0 \rightarrow S \cdot \$\}$			
$Z_2 = \{S \rightarrow a \cdot, S \rightarrow a \cdot P\} \cup \dots$		$\delta(Z_2, P) = Z_4$	$\delta(Z_2, ()) = Z_5$
$Z_3 = \{S \rightarrow (\cdot S), S \rightarrow (\cdot S) S\} \cup \dots$	$\delta(Z_3, S) = Z_6$	$\delta(Z_3, a) = Z_2$	$\delta(Z_3, ()) = Z_3$
$Z_4 = \{S \rightarrow a P \cdot\}$			
$Z_5 = \{P \rightarrow (\cdot S), P \rightarrow (\cdot S) S\} \cup \dots$	$\delta(Z_5, S) = Z_7$	$\delta(Z_5, a) = Z_2$	$\delta(Z_5, ()) = Z_3$
$Z_6 = \{S \rightarrow (S \cdot), S \rightarrow (S \cdot) S\}$			$\delta(Z_6, ()) = Z_8$
$Z_7 = \{P \rightarrow (S \cdot), P \rightarrow (S \cdot) S\}$			$\delta(Z_7, ()) = Z_9$
$Z_8 = \{S \rightarrow (S) \cdot, S \rightarrow (S) \cdot S\} \cup \dots$	$\delta(Z_8, S) = Z_{10}$	$\delta(Z_8, a) = Z_2$	$\delta(Z_8, ()) = Z_3$
$Z_9 = \{P \rightarrow (S) \cdot, P \rightarrow (S) \cdot S\} \cup \dots$	$\delta(Z_9, S) = Z_{11}$	$\delta(Z_9, a) = Z_2$	$\delta(Z_9, ()) = Z_3$
$Z_{10} = \{S \rightarrow (S) S \cdot\}$			
$Z_{11} = \{P \rightarrow (S) S \cdot\}$			

$R_1 = S \rightarrow a$
 $R_2 = S \rightarrow (S)$
 $R_3 = S \rightarrow a P$
 $R_4 = S \rightarrow (S) S$
 $R_5 = P \rightarrow (S)$
 $R_6 = P \rightarrow (S) S$

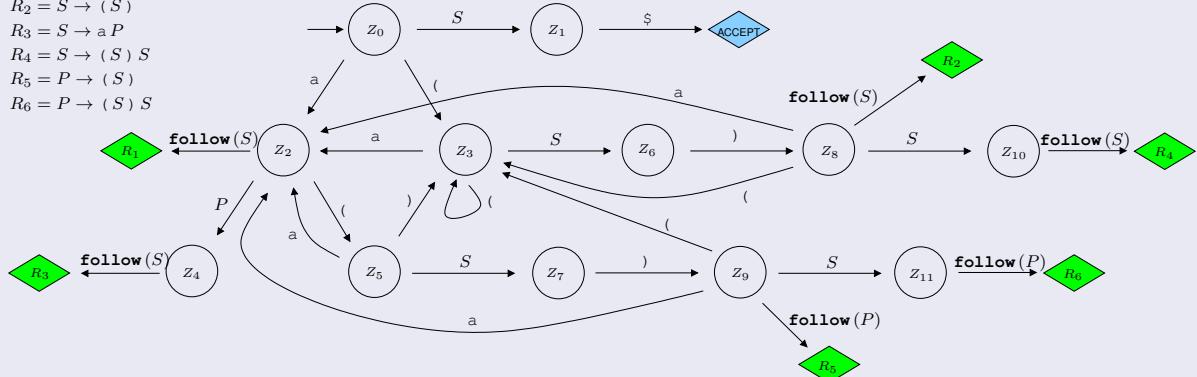


Tabela de decisão de um reconhecedor ascendente

Exemplo #3 (cont.)

- E finalmente a tabela de decisão

	a	()	\$	S	P
Z_0	<i>shift, Z₂</i>	<i>shift, Z₃</i>			Z_1	
Z_1				<i>ACCEPT</i>		
Z_2		<i>shift, Z₅</i>	<i>reduce S → a</i>	<i>reduce S → a</i>		Z_4
Z_3	<i>shift, Z₂</i>	<i>shift, Z₃</i>				Z_6
Z_4			<i>reduce S → a P</i>	<i>reduce S → a P</i>		
Z_5	<i>shift, Z₂</i>	<i>shift, Z₃</i>				Z_7
Z_6			<i>shift, Z₈</i>			
Z_7			<i>shift, Z₉</i>			
Z_8	<i>shift, Z₂</i>	<i>shift, Z₃</i>	<i>reduce S → (S)</i>	<i>reduce S → (S)</i>	Z_{10}	
Z_9	<i>shift, Z₂</i>	<i>shift, Z₃</i>	<i>reduce P → (S)</i>	<i>reduce P → (S)</i>	Z_{11}	
Z_{10}			<i>reduce S → (S) S</i>	<i>reduce S → (S) S</i>		
Z_{11}			<i>reduce P → (S) S</i>	<i>reduce P → (S) S</i>		

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

$$S \rightarrow \epsilon \mid S B a \mid S A b$$

$$A \rightarrow a \mid A A b$$

$$B \rightarrow B B a \mid b$$



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**

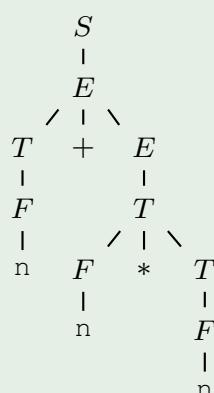
Análise sintática descendente

Exemplo

- Considere a gramática

$$\begin{aligned}S &\rightarrow E \\E &\rightarrow T + E \mid T \\T &\rightarrow F * T \mid F \\F &\rightarrow n \mid (E)\end{aligned}$$

- 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**

Analisador (*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 .

```
void S(void)
{
    switch(lookahead)
    {
        case 'a':
            eat('a'); S(); eat('b');
            break;
        default:
            epsilon();
            break;
    }
}

int lookahead;
int main()
{
    while (1)
    {
        printf(">> ");
        adv();
        S();
        eat('\n');
        printf("\n");
    }
    return 0;
}

void eat(int c)
{
    if (lookahead != c) error();
    adv();
}

void epsilon()
{
}

void error()
{
    printf("Unexpected symbol\n");
    exit(1);
}

void adv()
{
    lookahead = getchar();
}
```

Analisador (*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 $S \rightarrow \epsilon$ 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.

Analisador (*parser*) recursivo-descendente preditivo

Exemplo #2

- Sobre o alfabeto {a, b}, considere linguagem

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

descrita pela gramática

$$S \rightarrow \epsilon \mid a B S \mid b A S$$

$$A \rightarrow a \mid b A A$$

$$B \rightarrow a B B \mid b$$

- 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 $S \rightarrow \epsilon$ se for \$ (este último, mais tarde saber-se-á porquê)

Analisador (*parser*) recursivo-descendente preditivo

Exemplo #2a

- Sobre o alfabeto $\{a, b\}$, considere linguagem

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

descrita pela gramática

$$S \rightarrow \epsilon \mid a B \mid b A$$

$$A \rightarrow a S \mid b A A$$

$$B \rightarrow a B B \mid b S$$

- 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 $S \rightarrow \epsilon$

- Escolher $S \rightarrow \epsilon$ 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 $S \rightarrow \epsilon$

- 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

Analisador (*parser*) recursivo-descendente preditivo

Exemplo #2b

- Sobre o alfabeto $\{a, b\}$, considere linguagem

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

descrita pela gramática

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

- Construa um programa em que os símbolos não terminais sejam funções recursivas que reconheça a linguagem L

- Tal como no caso anterior, escolher $S \rightarrow \epsilon$ quando lookahead for $\$$ pode não resolver

Analisador (parser) recursivo-descendente preditivo

Exemplo #3

- Sobre o alfabeto $\{a, b\}$, considere linguagem

$$L = \{a^n b^n : n \geq 1\}$$

descrita pela gramática

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

- 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 esqueda, 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 $S \rightarrow a \ b$

Analisador (parser) recursivo-descendente preditivo

Exemplo #4

- Sobre o alfabeto $\{a, b\}$, considere linguagem

$$L = \{(ab)^n : n \geq 1\}$$

descrita pela gramática

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

- 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

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

- 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

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

- Esta gramática permite a construção de um programa preditivo com *lookahead* de 1

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

$$A \rightarrow A \alpha \\ | \\ \beta$$

- Aplicando a primeira produção n vezes e a seguir a segunda, obtem-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

$$A \rightarrow \beta X \\ X \rightarrow \varepsilon \\ | \quad \alpha X$$

-
- Em ANTLR seria possível fazer-se $A \rightarrow \beta (\alpha)^*$

Eliminação de recursividade à esquerda

Exemplo com recursividade direta simples

- Para a gramática

$$S \rightarrow S \text{ a b} \\ | \quad \text{c b a}$$

obtenha-se uma gramática equivalente sem recursividade à esquerda

- Aplicando a estratégia anterior, tem-se

$$S \Rightarrow S \underbrace{\text{a b}}_{\alpha} \Rightarrow S \underbrace{\text{a b}}_{\alpha} \cdots \underbrace{\text{a b}}_{\alpha} \Rightarrow \underbrace{\text{c b a}}_{\beta} \underbrace{\text{a b}}_{\alpha} \cdots \underbrace{\text{a b}}_{\alpha}$$

- Ou seja

$$S = (\underbrace{\text{c b a}}_{\beta}) (\underbrace{\text{a b}}_{\alpha})^n, \quad n \geq 0$$

- Que corresponde à gramática

$$S \rightarrow \text{c b a} X \\ X \rightarrow \varepsilon \\ | \quad \text{a b} X$$

Eliminação de recursividade à esquerda

Recursividade direta múltipla

- A gramática seguinte, onde α_i e β_j representam sequências de símbolos terminais e/ou não terminais, com os β_j não começando por A , representa genericamente a recursividade direta múltipla à esquerda

$$A \rightarrow A \alpha_1 \mid A \alpha_2 \mid \dots \mid A \alpha_n$$

$$\mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m$$

- Aplicando a estratégia anterior, tem-se

$$A = (\beta_1 \mid \beta_2 \mid \dots \mid \beta_m)(\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n)^k \quad k \geq 0$$

- Que corresponde à gramática

$$A \rightarrow \beta_1 X \mid \beta_2 X \mid \dots \mid \beta_m X$$

$$X \rightarrow \varepsilon$$

$$\mid \alpha_1 X \mid \alpha_2 X \mid \dots \mid \alpha_n X$$

-
- Em ANTLR seria possível fazer-se $(\beta_1 \mid \beta_2 \mid \dots \mid \beta_m)(\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n)^*$

Eliminação de recursividade à esquerda

Exemplo com recursividade direta múltipla

- Obtenha-se uma gramática equivalente à seguinte sem recursividade à esquerda

$$S \rightarrow S \text{ a b} \mid S \text{ c}$$

$$\mid \text{b b} \mid \text{c c}$$

- As palavras da linguagem são da forma

$$S = (\text{b b} \mid \text{c c}) (\text{a b} \mid \text{c})^k, \quad k \geq 0$$

- Obtendo-se a gramática

$$S \rightarrow \text{b b} X \mid \text{c c} X$$

$$X \rightarrow \varepsilon$$

$$\mid \text{a b} X \mid \text{c} X$$

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

$$\begin{aligned} S &\rightarrow A \ a \mid b \\ A &\rightarrow A \ c \mid S \ d \mid \varepsilon \end{aligned}$$

- O resultado seria

$$\begin{aligned} S &\rightarrow A \ a \mid b \\ A &\rightarrow S \ d \ X \mid X \\ X &\rightarrow \varepsilon \mid c \ X \end{aligned}$$

- 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 α_i , β_i , \dots , Ω_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

$$\begin{aligned} S &\rightarrow A \ a \mid b \\ A &\rightarrow A \ c \mid S \ d \mid \epsilon \end{aligned}$$

- 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

$$\begin{aligned} S &\rightarrow A \ a \mid b \\ A &\rightarrow A \ c \mid A \ a \ d \mid b \ d \mid \epsilon \end{aligned}$$

- Elimina-se a recursividade à esquerda direta das produções começadas por A , obtendo-se

$$\begin{aligned} S &\rightarrow A \ a \mid b \\ A &\rightarrow b \ d \ X \mid X \\ X &\rightarrow \epsilon \mid c \ X \mid a \ d \ X \end{aligned}$$

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:

predict ($A \rightarrow \alpha$) =

$$\left\{ \begin{array}{ll} \textbf{first}(\alpha) & \epsilon \notin \textbf{first}(\alpha) \\ (\textbf{first}(\alpha) - \{\epsilon\}) \cup \textbf{follow}(A) & \epsilon \in \textbf{first}(\alpha) \end{array} \right.$$

- 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:

$$\textbf{first}(\alpha) = \{t \in T : \alpha \Rightarrow^* t\omega \wedge \omega \in T^*\} \cup \{\epsilon : \alpha \Rightarrow^* \epsilon\}$$

- O conjunto **follow** (A) representa as letras (símbolos terminais) que podem aparecer imediatamente à frente de A numa derivação. Define-se por:

$$\textbf{follow}(A) = \{t \in T_{\$} : S \$ \Rightarrow^* \gamma A t \omega\} \quad \text{com} \quad T_{\$} = \{T \cup \$\}$$

Conjunto **first**

Algoritmo de cálculo

- Trata-se de um algoritmo recursivo

```
first( $\alpha$ ) {  
    if ( $\alpha = \varepsilon$ ) then  
        return  $\{\varepsilon\}$   
     $h = \text{head } (\alpha)$       # com  $|h| = 1$   
     $\omega = \text{tail } (\alpha)$      # tal que  $\alpha = h \omega$   
    if ( $h \in T$ ) then  
        return  $\{h\}$   
    else  
        return  $\bigcup_{(h \rightarrow \beta_i) \in P} \text{first } (\beta_i \omega)$       # 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

$$\text{first}(a \ S) = \{a\}.$$

- Determine o conjunto **first**($B \ C$)

- Porque $B \ C$ começa pelo símbolo não terminal B

$$\text{first}(B \ C) = \text{first}(C) \cup \text{first}(b \ S \ C)$$

- Porque C começa pelo símbolo não terminal C

$$\text{first}(C) = \text{first}(c) \cup \text{first}(c \ S)$$

$$\therefore \text{first}(B \ C) = \text{first}(c) \cup \text{first}(c \ S) \cup \text{first}(b \ S \ C) = \{b, c\}$$

- Note que, embora B se possa transformar em ε , $\varepsilon \notin \text{first}(B \ C)$

- Por essa razão, $\text{first}(B \ C) \neq \text{first}(B) \cup \text{first}(C)$

Conjunto **first**

Exemplo #2

- Considere a gramática

$$S \rightarrow a \ S \mid B \ C$$

$$B \rightarrow \epsilon \mid b \ S$$

$$C \rightarrow \epsilon \mid c \ S$$

- Determine o conjunto **first** ($B C$)

- Porque $B C$ começa pelo símbolo não terminal B

$$\text{first}(B C) = \text{first}(C) \cup \text{first}(b S C)$$

- Porque C começa pelo símbolo não terminal C

$$\text{first}(C) = \text{first}(\epsilon) \cup \text{first}(c S)$$

$$\text{first}(B C) = \text{first}(\epsilon) \cup \text{first}(c S) \cup \text{first}(b S C)$$

$$= \{\epsilon, b, c\}$$

-
- Note que a gramática não é a mesma
 - Note que $\epsilon \in \text{first}(B C)$ apenas porque todo o $B C$ 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 $\$ \in \text{follow}(S)$

② **if** ($A \rightarrow \alpha B \in P$) **then**
 $\text{follow}(B) \supseteq \text{follow}(A)$

③ **if** ($A \rightarrow \alpha B \beta \in P$) $\wedge (\epsilon \notin \text{first}(\beta))$ **then**
 $\text{follow}(B) \supseteq \text{first}(\beta)$

④ **if** ($A \rightarrow \alpha B \beta \in P$) $\wedge (\epsilon \in \text{first}(\beta))$ **then**
 $\text{follow}(B) \supseteq ((\text{first}(\beta) - \{\epsilon\}) \cup \text{follow}(A))$

- 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

$$S \rightarrow a S \mid B C$$

$$B \rightarrow \epsilon \mid b S$$

$$C \rightarrow c \mid c S$$

- Determine o conjunto **follow**(*B*)

- Procuram-se ocorrências de *B* no lado direito das produções. Há uma: *BC*

- A produção $S \rightarrow BC$ 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

$$S \rightarrow a S \mid B C$$

$$B \rightarrow \epsilon \mid b S$$

$$C \rightarrow \epsilon \mid c S$$

- Determine o conjunto **follow**(*B*)

- A produção $S \rightarrow BC$ encaixa nas regras 3 ou 4, dependendo de o ϵ pertencer ou não ao **first**(*C*)

- **first**(*C*) = { ϵ , c}

- $\therefore \text{follow}(B) \supseteq ((\text{first}(C) - \{\epsilon\}) \cup \text{follow}(S))$ [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 $S \rightarrow BC$ 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

$$S \rightarrow a S \mid B C$$

$$B \rightarrow b \mid b S$$

$$C \rightarrow \epsilon \mid S c$$

- Determine o conjunto **follow**(B)

- A produção $S \rightarrow BC$ encaixa nas regras 3 ou 4, dependendo de o ϵ pertencer ou não ao **first**(C)
- $\text{first}(C) = \{\epsilon, a, b\}$
- $\therefore \text{follow}(B) \supseteq (\text{first}(C) - \{\epsilon\}) \cup \text{follow}(S)$
- 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) ∈ (N × T$)
    τ(n, t) = ∅          # começa-se com as células vazias
    foreach (A → α) ∈ P
        foreach t ∈ predict(A → α)
            add (A → α) to τ(A, t)
```

Tabela de decisão

Exemplo #1

- Considere a gramática

$$S \rightarrow a S b \mid \epsilon$$

- Preencha a tabela de decisão de um reconhecedor descendente desta linguagem com *lookahead* de 1

$$\text{first}(a S b) = \{a\}$$

$$\therefore \text{predict}(S \rightarrow a S b) = \{a\}$$

Tabela de decisão

$$\text{first}(\epsilon) = \{\epsilon\}$$

$$\text{follow}(S) = \{\$, b\}$$

$$\therefore \text{predict}(S \rightarrow \epsilon) = \{\$\}, \{b\}$$

	a	b	\$
S	a S b	\epsilon	\epsilon

- 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

$$S \rightarrow \epsilon \mid a B S \mid b A S$$

$$A \rightarrow a \mid b A A$$

$$B \rightarrow a B B \mid b$$

- Preencha a tabela de decisão de um reconhecedor descendente desta linguagem com *lookahead* de 1

$$\text{predict}(S \rightarrow a B S) = \{a\}$$

$$\text{predict}(S \rightarrow b A S) = \{b\}$$

$$\text{predict}(S \rightarrow \epsilon) = \{\$\}$$

$$\text{predict}(A \rightarrow a) = \{a\}$$

$$\text{predict}(A \rightarrow b A A) = \{b\}$$

$$\text{predict}(B \rightarrow b) = \{b\}$$

$$\text{predict}(B \rightarrow a B B) = \{a\}$$

Tabela de decisão

	a	b	\$
S	a B S	b A S	\epsilon
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

$$S \rightarrow \epsilon \mid a B \mid b A$$

$$A \rightarrow a S \mid b A A$$

$$B \rightarrow a B B \mid b S$$

- Preencha a tabela de decisão de um reconhecedor descendente desta linguagem com *lookahead* de 1

$$\text{predict}(S \rightarrow a B) = \{a\}$$

$$\text{predict}(S \rightarrow b A) = \{b\}$$

$$\text{predict}(S \rightarrow \epsilon) = \{a, b, \$\}$$

$$\text{predict}(A \rightarrow a S) = \{a\}$$

$$\text{predict}(A \rightarrow b A A) = \{b\}$$

$$\text{predict}(B \rightarrow b S) = \{b\}$$

$$\text{predict}(B \rightarrow a B B) = \{a\}$$

Tabela de decisão

	a	b	\$
S	$a B, \epsilon$	$b A, \epsilon$	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

$$S \rightarrow \epsilon$$

$$\mid a S b S$$

$$\mid b S a S$$

- Preencha a tabela de decisão de um reconhecedor descendente desta linguagem com *lookahead* de 1

$$\text{predict}(S \rightarrow a S b S) = \{a\}$$

$$\text{predict}(S \rightarrow b S a S) = \{b\}$$

$$\text{predict}(S \rightarrow \epsilon) = \{a, b, \$\}$$

Tabela de decisão

	a	b	\$
S	$a A b S, \epsilon$	$b S a S, \epsilon$	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.)

predict($D \rightarrow TL ;$) = ?
first($TL ;$) = ?
first(T) = **first**(i) \cup **first**(f) = { i } \cup { f }
 \therefore **first**($TL ;$) = { i, f }
 \therefore **predict**($D \rightarrow TL ;$) == { i, f }

predict($T \rightarrow i$) = ?
first(i) = { i }
 \therefore **predict**($T \rightarrow i$) = { i }

predict($T \rightarrow f$) = { f }
predict($L \rightarrow vX$) = ?
first(vX) = { v }
 \therefore **predict**($L \rightarrow vX$) = { v }

predict($X \rightarrow \epsilon$) = ?
first(ϵ) = { ϵ }
 \therefore **predict**($X \rightarrow \epsilon$) = **follow**(X)
follow(X) = **follow**(L) = { $;$ }
 \therefore **predict**($X \rightarrow \epsilon$) = { $;$ }
predict($X \rightarrow , L$) = { $,$ }

Tabela de decisão

Exemplo #3 (cont.)

$D \rightarrow T \ L \ ;$

$T \rightarrow i$

| f

$L \rightarrow v \ X$

$X \rightarrow$

| , 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

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

$$E \rightarrow E + T \mid T$$

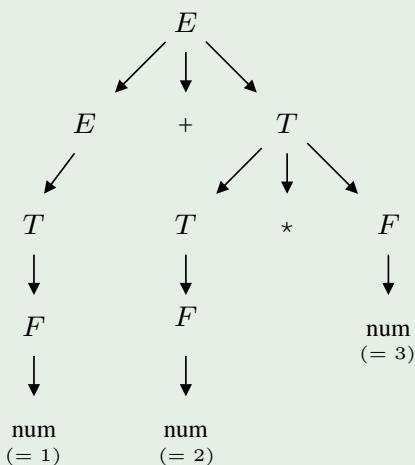
um número $T \rightarrow T * F \mid F$

$$F \rightarrow \text{num} \mid (E)$$

- 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

$$E \rightarrow E + T \mid T$$

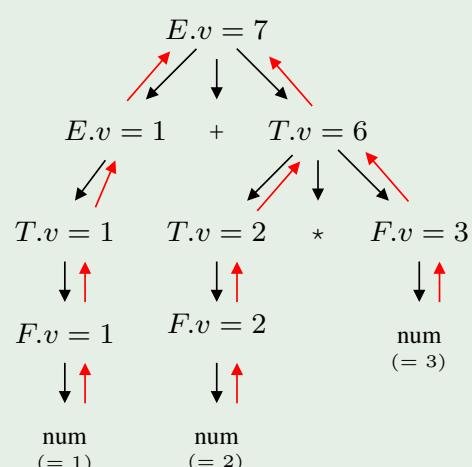
um número $T \rightarrow T * F \mid F$

$$F \rightarrow \text{num} \mid (E)$$

- 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



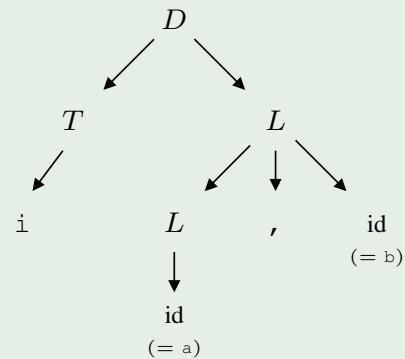
- As setas vermelhas representam dependência entre atributos
 - o sentido indica qual influencia qual

Conteúdo semântico

Ilustração com uma declaração de variáveis

- Considere a gramática seguinte, onde id é um *token* que representa o nome de uma variável

$$\begin{aligned} D &\rightarrow T \ L \\ T &\rightarrow i \mid f \\ L &\rightarrow \text{id} \mid L \ , \ \text{id} \end{aligned}$$



- desenhe-se a árvore de derivação da palavra $i \ a, b$
- Associe-se
 - a T e L um atributo t que armazene o tipo

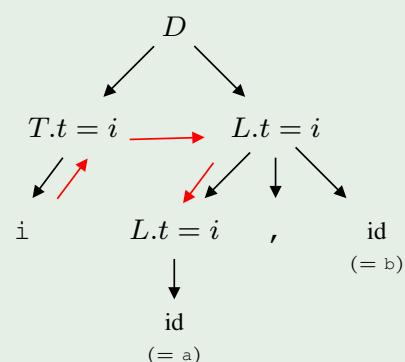
Conteúdo semântico

Ilustração com uma declaração de variáveis

- Considere a gramática seguinte, onde id é um *token* que representa o nome de uma variável

$$\begin{aligned} D &\rightarrow T \ L \\ T &\rightarrow i \mid f \\ L &\rightarrow \text{id} \mid L \ , \ \text{id} \end{aligned}$$

- 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

$$b = f(c_1, c_2, \dots, c_n)$$

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, \dots, c_n são atributos dos símbolos que ocorrem na produção

- Podem ainda associar-se regras semânticas com efeitos colaterais

$$g(c_1, c_2, \dots, c_n)$$

- Embora este caso possa considerar-se o anterior atuando sobre um atributo fictício

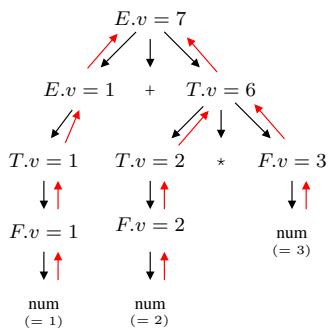
Gramática de atributos

Tipos de atributos

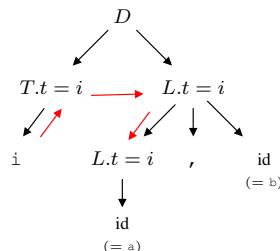
- Os atributos podem ser classificados como **sintetizados** ou **herdados**
- Considere-se uma produção $A \rightarrow B_1 B_2 \cdots B_n \in P$, com $B_i \in (T \cup N)^*$, e uma função de cálculo de um atributo associada a essa produção

$$b = f(c_1, c_2, \dots, c_n)$$

- 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



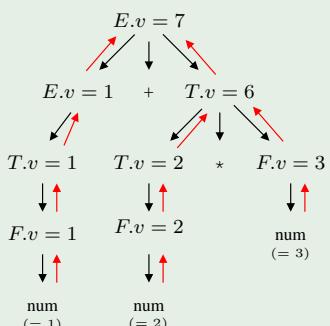
- $T.t$ é sintetizado
- $L.t$ é herdado

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
$F \rightarrow \text{num}$	$F.v = \text{num}.v$
$F \rightarrow (E)$	$F.v = E.v$
$T \rightarrow F$	$T.v = F.v$
$T_1 \rightarrow T_2 * F$	$T_1.v = T_2.v * F.v$
$E \rightarrow T$	$E.v = T.v$
$E_1 \rightarrow E_2 + T$	$E_1.v = E_2.v + T.v$

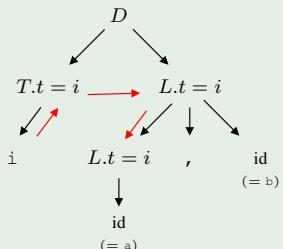
- Note que se assume que o símbolo terminal `num` tem um atributo chamado v com o valor correspondente.
- O ANTLR não suporta atributos nos terminais (*tokens*)

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
$T \rightarrow i$	$T.t = \text{int}$
$T \rightarrow f$	$T.t = \text{float}$
$D \rightarrow T \ L$	$L.t = T.t$
$L_1 \rightarrow L_2 , \ \text{id}$	$L_2.t = L_1.t$ addsym(id.n, L ₁ .t)
$L \rightarrow \text{id}$	addsym(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`)

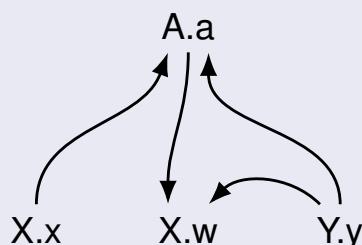
Avaliação dirigida pela sintaxe

- 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**

$$A \rightarrow X \ Y$$

$$A.a = f(X.x, Y.y)$$

$$X.w = q(A.a, Y.y)$$



- Aqui as setas apontam no sentido das dependências



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 escrita), 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**, respectivamente.
 - No caso geral $\alpha \in (N \cup T)^*$ $\times N \times (N \cup T)^*$ e $\beta = (N \cup T)^*$.
 - 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

- 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 $aabcabc$ por aplicação sucessiva de produções da gramática

R

$$\begin{aligned} S &\Rightarrow aB \Rightarrow aaBB \Rightarrow aabSB \Rightarrow aabcSB \Rightarrow aabcB \Rightarrow aabcbS \\ &\Rightarrow aabcbcS \Rightarrow aabcbc \end{aligned}$$

- Acabou de se obter uma **derivação à esquerda** da palavra $aabcbc$
- 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

D Chama-se **derivação** a uma sucessão de zero ou mais derivações diretas, denotando-se

$$\alpha \Rightarrow^* \beta \quad \equiv \quad \alpha = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_n = \beta$$

onde n é o comprimento da derivação.

D Chama-se **derivação à esquerda** a uma sucessão de zero ou mais derivações diretas à esquerda, denotando-se

$$\alpha \stackrel{E}{\Rightarrow}^* \beta \quad \equiv \quad \alpha = \alpha_0 \stackrel{E}{\Rightarrow} \alpha_1 \stackrel{E}{\Rightarrow} \dots \stackrel{E}{\Rightarrow} \alpha_n = \beta$$

onde n é o comprimento da derivação.

D Chama-se **derivação à direita** a uma sucessão de zero ou mais derivações diretas à direita, denotando-se

$$\alpha \stackrel{D}{\Rightarrow}^* \beta \quad \equiv \quad \alpha = \gamma_0 \stackrel{D}{\Rightarrow} \gamma_1 \stackrel{D}{\Rightarrow} \dots \stackrel{D}{\Rightarrow} \gamma_n = \beta$$

onde n é o comprimento da derivação.

Derivação

Exemplo

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 $aabcabc$

R

à esquerda

$$\begin{aligned} S &\Rightarrow aB \Rightarrow aaBB \Rightarrow aabSB \Rightarrow aabcSB \\ &\Rightarrow aabcB \Rightarrow aabcbS \Rightarrow aabcabcS \Rightarrow aabcabc \end{aligned}$$

à direita

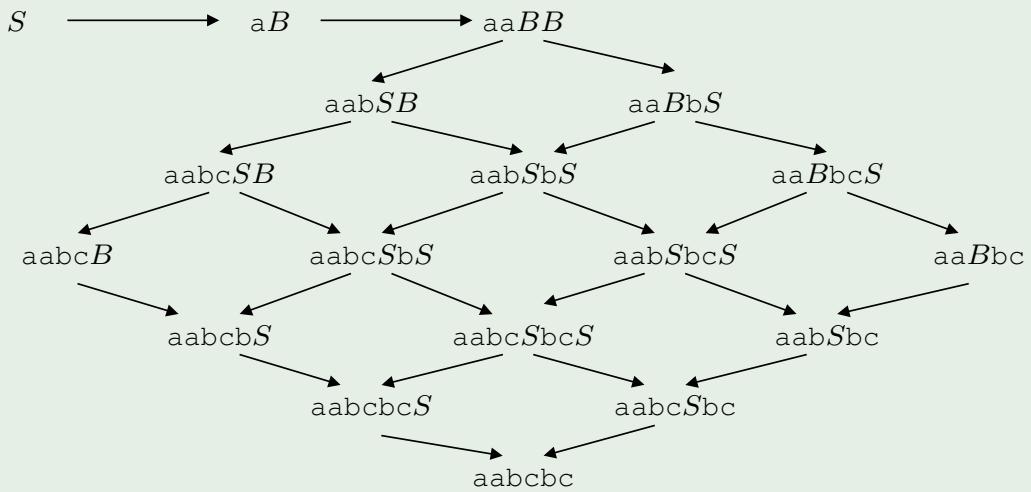
$$\begin{aligned} S &\Rightarrow aB \Rightarrow aaBB \Rightarrow aaBbS \Rightarrow aaBbcS \\ &\Rightarrow aaBbc \Rightarrow aabSbc \Rightarrow aabcSbc \Rightarrow aabcabc \end{aligned}$$

- Note que se usou \Rightarrow em vez de $\stackrel{D}{\Rightarrow}$ e $\stackrel{E}{\Rightarrow}$

Derivação

Alternativas de derivação

- O grafo seguinte capta as alternativas de derivação. Considera-se novamente a palavra $aabcabc$ e a gramática anterior



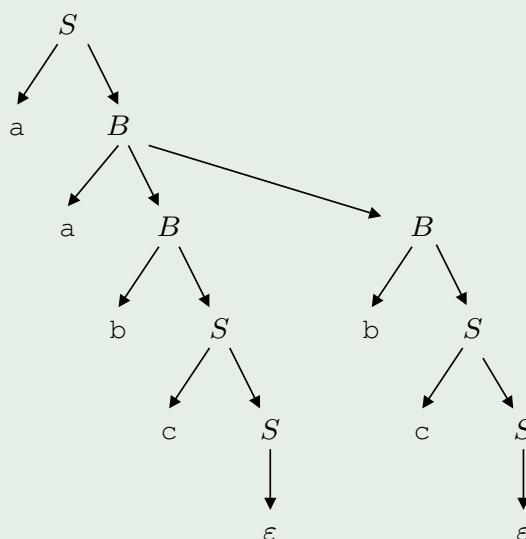
- Identifique os caminhos que correspondem às derivações à direita e à esquerda

Derivação

Árvore de derivação

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 $aabcabc$ na gramática anterior é

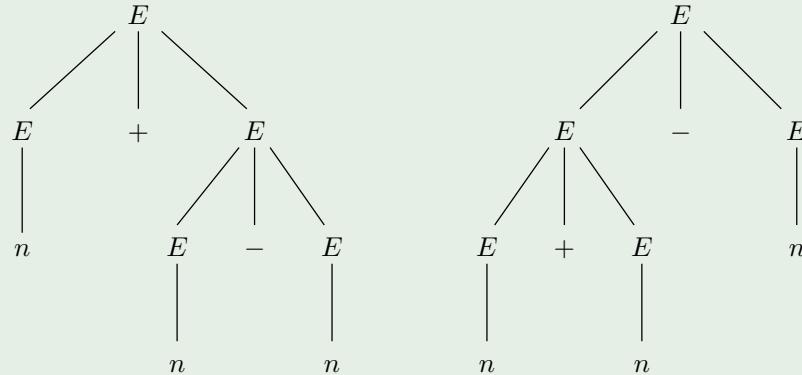


Ambiguidade

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 $n+n-n$

Podem obter-se duas árvores de derivação diferentes



-
- Pode haver duas interpretações diferentes para a palavra; há **ambiguidade**

Ambiguidade

Definição

- Diz-se que uma palavra é derivada **ambiguamente** se possuir duas ou mais árvores de derivação distintas
- 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 **inherentemente 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

R Considere-se novamente a gramática

$$S \rightarrow S + S \mid S - S \mid (S) \mid n$$

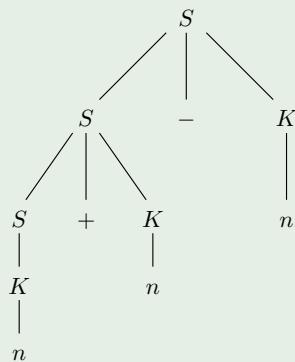
e obtenha-se uma gramática não ambígua equivalente

R

$$S \rightarrow K \mid S + K \mid S - K$$

$$K \rightarrow n \mid (S)$$

Q Desenhe a árvore de derivação da palavra $n+n-n$ na nova gramática



Projeto de gramáticas

Exemplo #1, solução #1

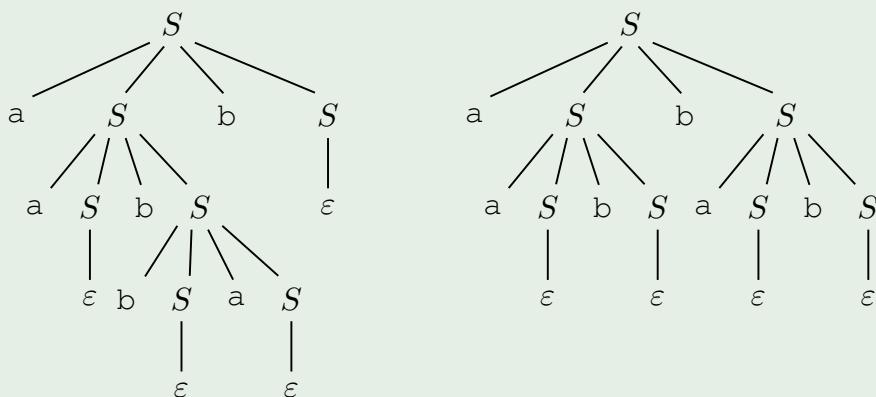
Q Sobre o conjunto de terminais $T = \{a, b\}$, determine uma gramática livre de contexto que represente a linguagem

$$L_1 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$$

R₁

$$S \rightarrow \epsilon \mid a \ S \ b \ S \mid b \ S \ a \ S$$

Q A gramática é ambígua? Analise a palavra aabbab



Projeto de gramáticas

Exemplo #1, solução #2

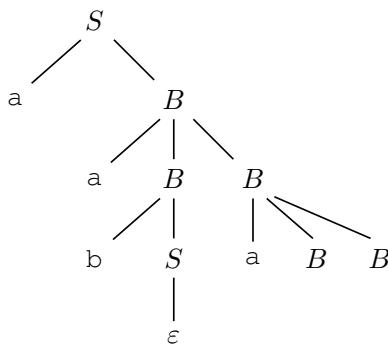
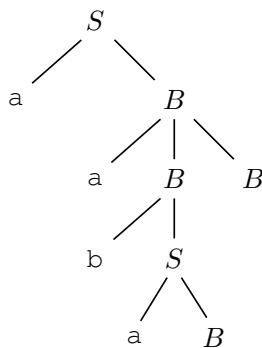
Q Sobre o conjunto de terminais $T = \{a, b\}$, determine uma gramática livre de contexto que represente a linguagem

$$L_1 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$$

R₂

$$\begin{array}{l} S \rightarrow \varepsilon \mid a \ B \mid b \ A \\ A \rightarrow a \ S \mid b \ A \ A \\ B \rightarrow a \ B \ B \mid b \ S \end{array}$$

Q A gramática é ambígua?
Analise a palavra `aababb`.



- 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

$$L_1 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$$

3

$$\begin{array}{l} S \rightarrow \varepsilon \mid a \ B \ S \mid b \ A \ S \\ A \rightarrow a \mid b \ A \ A \\ B \rightarrow a \ B \ B \mid b \end{array}$$

Q A gramática é ambígua? Analise a palavra `aababb`

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

$$L_2 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$$

\mathcal{R}

$$S \rightarrow \epsilon \mid a \ B \ S \mid b \ A \ S \mid c \ S$$

$$A \rightarrow a \mid b \ A \ A \mid c \ A$$

$$B \rightarrow a \ B \ B \mid b \mid c \ B$$

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

$$L_3 = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega) \wedge \forall_{i \leq |\omega|} \#(a, \text{prefix}(i, \omega)) \geq \#(b, \text{prefix}(i, \omega))\}$$

\mathcal{R}_1

$$S \rightarrow \epsilon \mid a \ S \ b \ S \mid c \ S$$

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

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

- 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

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

- 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

$$\begin{aligned} T &= T_1 \cup T_2 \\ N &= N_1 \cup N_2 \cup \{S\} \text{ com } S \notin (N_1 \cup N_2) \\ P &= \{S \rightarrow S_1, S \rightarrow S_2\} \cup P_1 \cup P_2 \end{aligned}$$

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

$X = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega)\}$	$\begin{array}{l} X \rightarrow \varepsilon \mid a \ B \mid b \ A \mid c \ X \\ A \rightarrow a \ X \mid b \ A \ A \mid c \ A \\ B \rightarrow a \ B \ B \mid b \ X \mid c \ B \end{array}$
$A = \{\omega \in T^* : \#(a, \omega) = \#(b, \omega) + 1\}$	Basta usar o A anterior como símbolo inicial
$L = X \cup A^*$	$\begin{array}{l} S \rightarrow \varepsilon \mid A \ S \mid X \\ X \rightarrow \varepsilon \mid a \ B \mid b \ A \mid c \ X \\ A \rightarrow a \ X \mid b \ A \ A \mid c \ A \\ B \rightarrow a \ B \ B \mid b \ X \mid c \ B \end{array}$

- 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

$$\begin{aligned} T &= T_1 \\ N &= N_1 \cup \{S\} \text{ com } S \notin N_1 \\ P &= \{S \rightarrow \varepsilon, S \rightarrow S_1 S\} \cup P_1 \end{aligned}$$

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

$$\begin{aligned}S &\rightarrow a \ A \ b \mid b \ B \\A &\rightarrow c \ C \mid b \ B \mid d \\B &\rightarrow d \ D \mid b \\C &\rightarrow A \ C \mid B \ D \mid S \ D \\D &\rightarrow A \ D \mid B \ C \mid C \ S \\E &\rightarrow a \ A \mid b \ B \mid \epsilon\end{aligned}$$

- 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
            if  $A \notin N_p$  then
                 $N_p \leftarrow N_p \cup \{A\}$ 
                nothingAdded  $\leftarrow$  false
            else
                 $P_p \leftarrow P_p - \{A \rightarrow \alpha\}$ 
        end if
    end foreach
until nothingAdded or  $N_p = N$ 
# se todos são terminais ou produtivos, A é produtivo
# se ainda não pertence aos produtivos
# é lá colocado
# e é preciso repetir o processo
# a produção já não precisa de ser processada mais
```

Símbolos acessíveis e inacessíveis

Exemplo de ilustração

Q Sobre o conjunto de terminais $T = \{a, b, c, d\}$, considere a gramática

$$\begin{aligned}S &\rightarrow a \ A \ b \mid b \ B \\A &\rightarrow c \ C \mid b \ B \mid d \\B &\rightarrow d \ D \mid b \\C &\rightarrow A \ C \mid B \ D \mid S \ D \\D &\rightarrow A \ D \mid B \ C \mid C \ S \\E &\rightarrow a \ A \mid b \ B \mid \epsilon\end{aligned}$$

- 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

$$S \in V_A$$

if $A \rightarrow \alpha B \beta \in P$ **and** $A \in V_A$ **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  $\alpha$  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 $S \rightarrow a \ A \ b \wedge a, b \in (T \cup N_p)^* \implies N_p \leftarrow N_p \cup \{S\}$
- 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

$$S \rightarrow a A b \mid b B$$

$$A \rightarrow b B \mid d$$

$$B \rightarrow b$$

$$E \rightarrow a A \mid b B \mid \epsilon$$

- 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

Linguagens Regulares, Expressões Regulares e Gramáticas Regulares

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

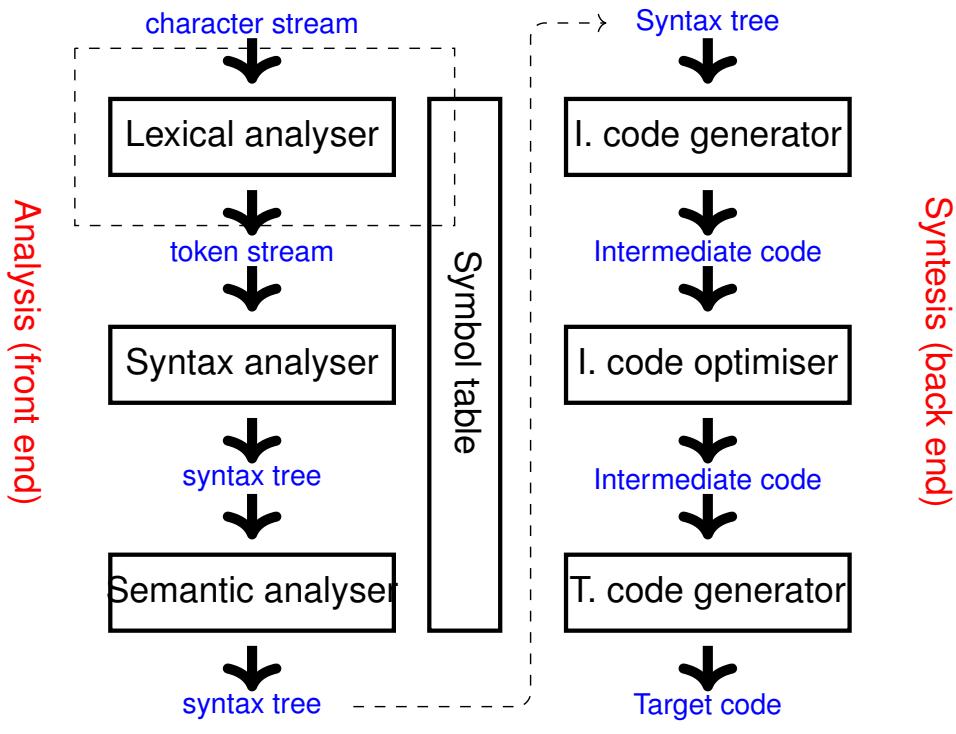
DETI, Universidade de Aveiro

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 $\langle \text{token-name}, \text{attribute-value} \rangle$
 - *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: $e_1\epsilon = \epsilon e_1 = e_1$
 - existência de elemento absorvente: $e_1\emptyset = \emptyset e_1 = \emptyset$
 - **não goza da propriedade comutativa**

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

Expressões regulares

Propriedades distributivas

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

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

Expressões regulares

Propriedades da operação de fecho de Kleene

- A operação de fecho goza das propriedades:
 - $(e^*)^* = e^*$
 - $(e_1^* | e_2^*)^* = (e_1 | e_2)^*$
 - $(e_1 | e_2^*)^* = (e_1 | e_2)^*$
 - $(e_1^* | e_2)^* = (e_1 | e_2)^*$
- Mas atenção:
 - $(e_1 | e_2)^* \neq e_1^* | e_2^*$
 - $(e_1 e_2)^* \neq e_1^* e_2^*$

- Exemplo:

- $b(a^*)^* = b a^*$
- $(a^* | b^*)^* = (a | b)^*$
- $(a | b^*)^* = (a | b)^*$
- $(a^* | b)^* = (a | b)^*$
- $(a | b)^* \neq a^* | b^*$
- $(a b)^* \neq a^* b^*$

Expressões regulares

Exemplos

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

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

R $1^* 0 1^* 0 1^*$

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

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

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 a e z . 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:

$$[^a_1a_2a_3 \cdots a_n], \quad [^a_1-a_n]$$

Em ANTLR:

- $x..y$ é equivalente a $[x-y]$
- $\sim [abc]$ é equivalente a $[^abc]$

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

- $^$ 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

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

R $1^*01^*01^* = (1^*0)(1^*0)1^* = (1^*0)\{2\}1^*$

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

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

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

Gramáticas regulares

Introdução

- Exemplo de gramática regular

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

- Exemplo de gramática **não** regular

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

-
- 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

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

\mathcal{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

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

\mathcal{R}

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

-
- 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

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

$$T = T_1 \cup T_2$$

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

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

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

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

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

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

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

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

-
- 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

$$\begin{aligned} T &= T_1 \cup T_2 \\ N &= N_1 \cup N_2 \\ P &= \{A \rightarrow \omega S_2 : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^*\} \\ &\quad \cup \{A \rightarrow \omega : (A \rightarrow \omega) \in P_1 \wedge \omega \in T_1^* N_1\} \\ &\quad \cup P_2 \\ S &= S_1 \end{aligned}$$

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

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

- 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

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

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

- 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

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

$$T = T_1$$

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

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

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

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

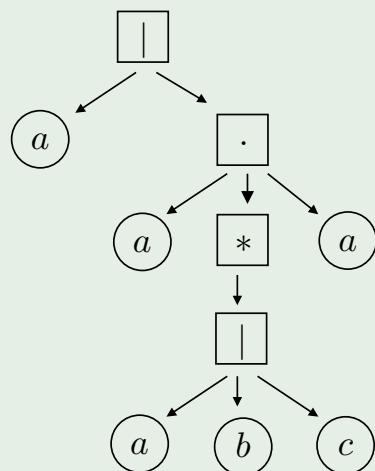
é regular e gera a linguagem $L = (L(G_1))^*$.

- 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

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

R

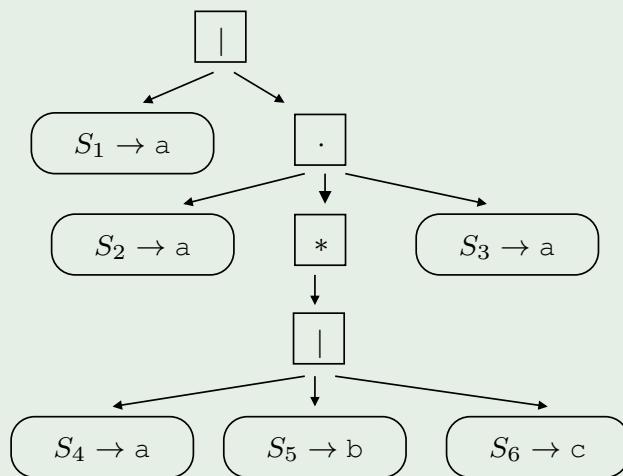


- Coloque-se de forma arbórea

Conversão de uma ER em uma GR exemplo

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

R

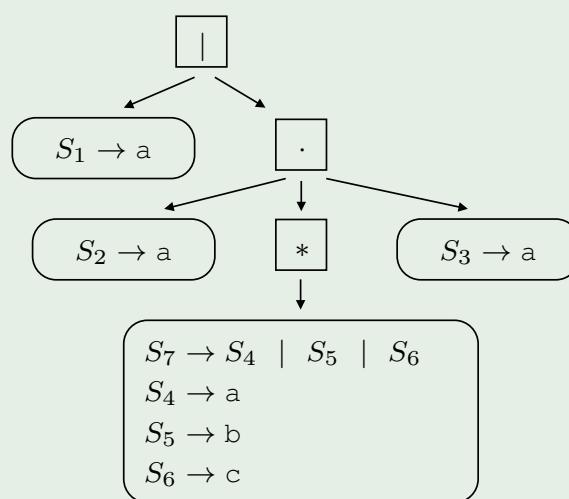


- Após converter as folhas (elementos primitivos) em GR

Conversão de uma ER em uma GR exemplo

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

R

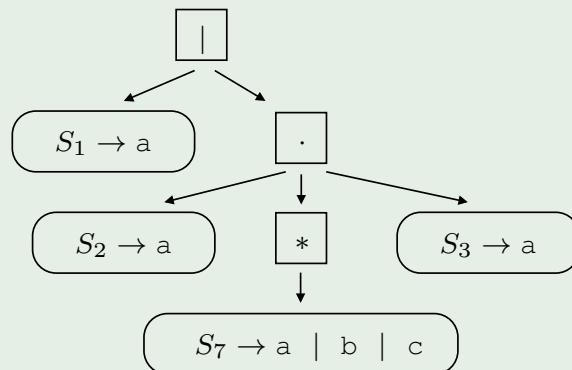


- Após aplicar a escolha (reunião) de baixo

Conversão de uma ER em uma GR exemplo

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

R

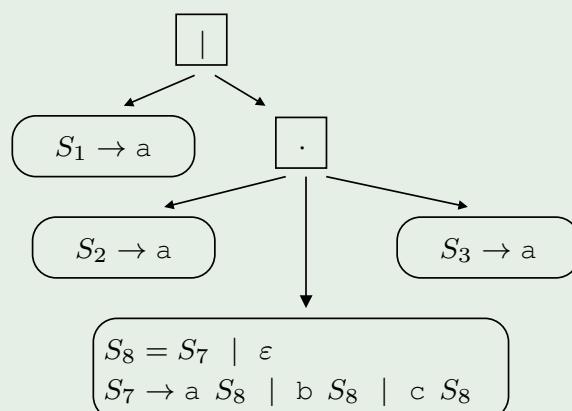


- Simplificando

Conversão de uma ER em uma GR exemplo

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

R

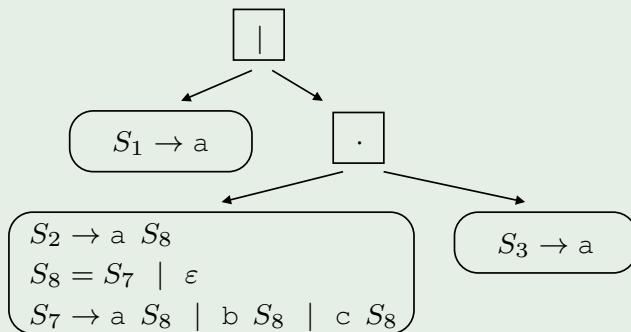


- Após aplicar o fecho

Conversão de uma ER em uma GR exemplo

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

R

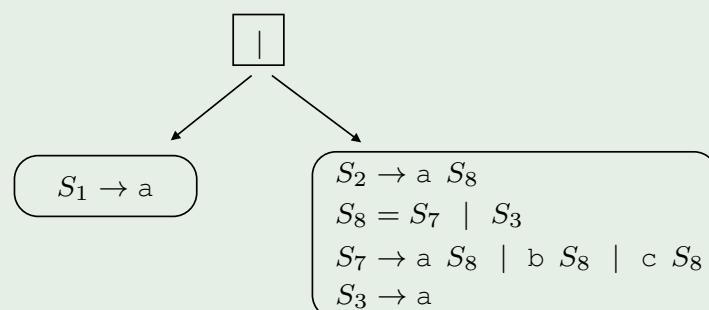


- Após aplicar a concatenação da esquerda

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 $e_1 \cdot e_2$, 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	$S \rightarrow \epsilon$
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 $e_1 \cdot e_2$, 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

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

R Abordagem admitindo expressões regulares nas produções das gramáticas

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

- *acrescentou-se um novo símbolo inicial de forma a garantir que não aparece do lado direito*

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

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

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

- *transformaram-se as produções $E \rightarrow \epsilon \ S$, $S \rightarrow (a|c) \ S$ e $S \rightarrow aba \ X$ em $E \rightarrow (a|c)^* aba \ X$*
- *Note que o $(a|c)$ passou a $(a|c)^*$*

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

- *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

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

R Abordagem transformando a gramática num conjunto e triplos

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

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

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

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

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

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

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

- 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

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:

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

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

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