PCS3216 – Sistemas de Programação

Prof. João José Neto

17 - Linguagens de alto nível

Parte 2 - **Linguagens de *script* e interpretadores**

Origens

- As linguagens de *script* surgiram pela primeira vez com a *Job Control* da IBM, usada para comandar a operação dos pioneiros *mainframes* dos anos 50;
- Nos anos 60 apareceu o *Shell Script*, usado para controlar os sistemas operacionais;
- Posteriormente, diversas outras linguagens de *script* foram sendo utilizadas, com diferentes propósitos;
- O nome se inspira no termo "*script*" dos cinemas.

O que caracteriza uma linguagem de *script*?

- É uma linguagem cujos comandos são processados um a um, com o auxílio de um programa denominado interpretador, capaz de reconhecê-los e de promover a execução de funções que cumprem as suas ordens.
- Em geral é utilizada para tornar automática a sequenciação da execução de operações que usualmente seriam comandadas por uma pessoa.
- Tais linguagens são **interpretadas**, ou seja, um programa **interpretador** (similar àqueles que implementam simuladores de máquinas virtuais) reconhece comandos e aciona rotinas que executam as operações a eles associadas.
- Um exemplo importante deste tipo de linguagem é a PHP, muito utilizada em programação para a Web.

Linguagem interpretada

- O processamento de programas escritos em uma linguagem de programação pode, usualmente, ser feito de duas formas: compilada e interpretada.
- Em geral, linguagens de *script* costumam ser interpretadas.
- A forma interpretada
  - Independe da plataforma (sistema hospedeiro).
  - A tipagem é dinâmica, ou seja, permite que uma variável mude de tipo ao longo da execução do programa, já que a determinação do tipo da variável fica postergado para a época da execução.
- A forma compilada
  - Exige um esforço maior do sistema para converter o programa para um formato no qual ele possa ser executado.
  - Na forma executável compilada, o desempenho do programa costuma resultar superior ao da forma equivalente interpretada.

Implementação

- As características das linguagens de *script* sugerem que sua implementação seja feita comando a comando, usando uma lógica similar à dos simuladores guiados por eventos.
- Uma parte do sistema deve se responsabilizar pela extração de comandos e sua decomposição em elementos mais básicos, enquanto outra parte recebe tais elementos e deve decidir quais ações o interpretador deve tomar em resposta ao comando, para finalmente acionar rotinas que executem as operações especificadas pelo comando.

Algumas considerações

- Detalha-se a seguir, estruturalmente, um mecanismo relativamente geral de interpretação para linguagens de *script*, sem particularizar a linguagem, adotando para sua implementação a técnica da simulação guiada por eventos.
- Cada módulo presente nos diagramas a seguir pode ser implementado a partir de um motor de eventos, particularizando-se para cada módulo apenas os respectivos conjuntos de eventos e os conjuntos das rotinas de tratamento a eles associadas.

Um primeiro detalhamento

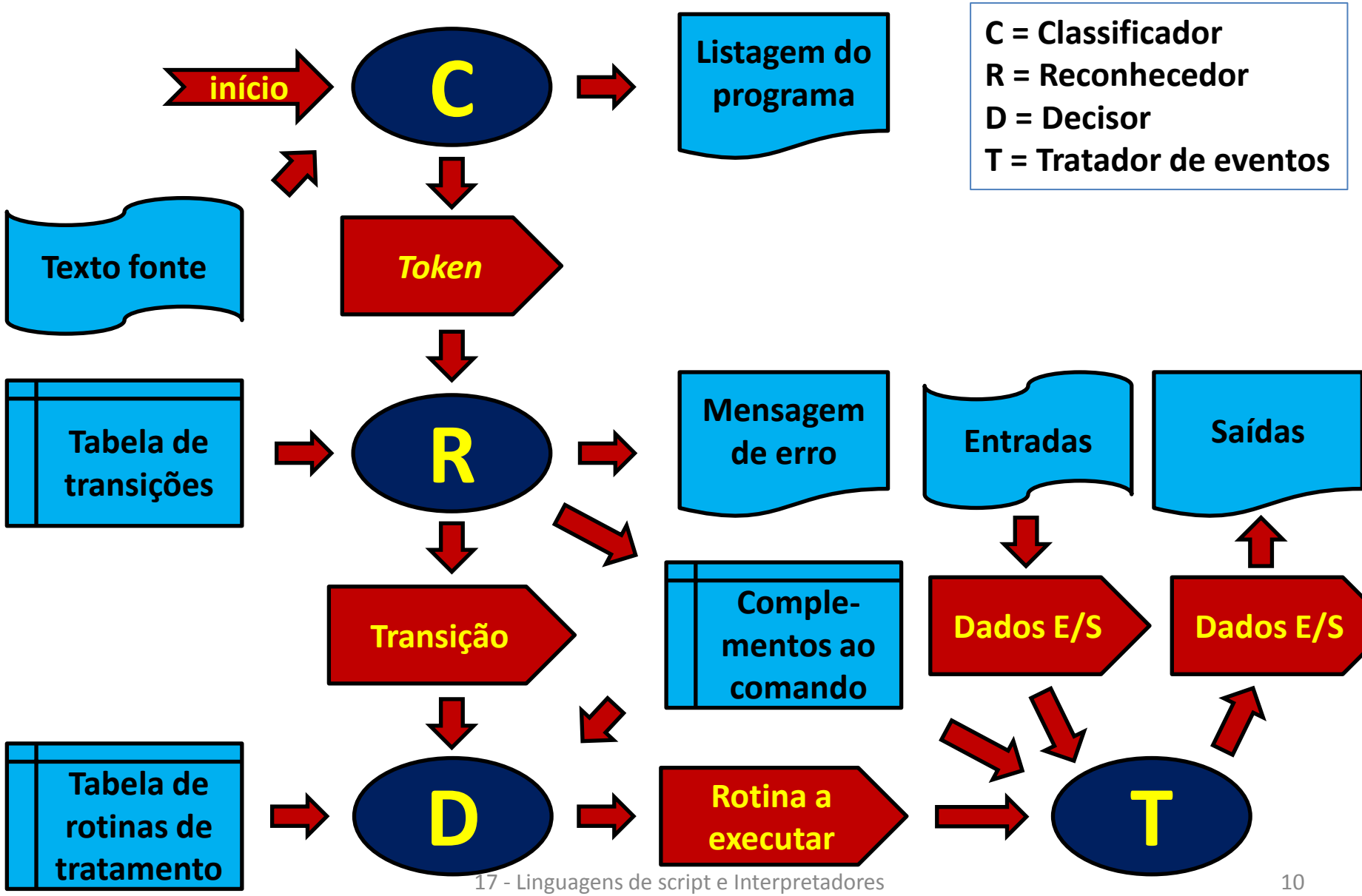
- Os módulos, simbolizados por elipses azuis nos diagramas de troca de mensagens mostrados adiante (**atenção**: ***não se trata de diagramas de blocos!***), comunicam-se entre si através de eventos, estes representados através de setas pentagonais.
- Esse esquema é facilmente implementado em linguagem orientada a objetos usando uma classe "motor de eventos", cujos objetos são os módulos.
- Tais objetos trocam mensagens uns com os outros, e essas mensagens fazem o papel de eventos.
- As setas do diagrama indicam o fluxo dos eventos entre os módulos, mas não implicam que exista uma sequencialidade temporal obrigatória.

ESBOÇO DO INTERPRETADOR

O diagrama geral do interpretador

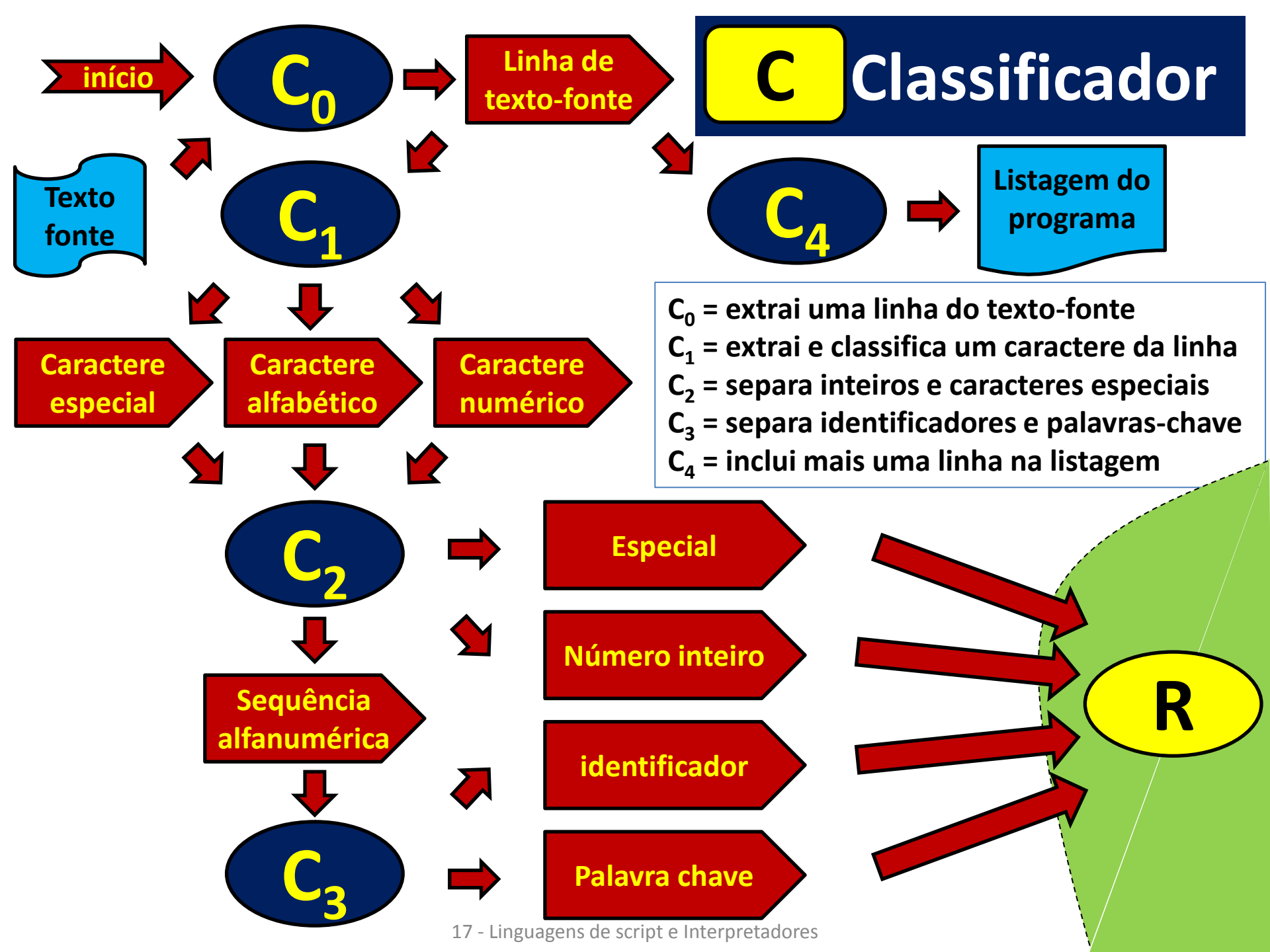
- Os **quatro módulos** envolvidos no interpretador são:
  - **Classificador** (**C**) – extração/classificação de caracteres/*tokens*
  - **Reconhecedor** (**R**) – aceitação/rejeição sintática
  - **Decisor** (**D**) – escolha do tratamento semântico a aplicar
  - **Tratador de eventos** (**T**) – aplicação do tratamento semântico
- Para ter uma visão global do funcionamento de um interpretador implementado dessa forma, o próximo slide mostra um **diagrama geral**, em que os módulos aparecem somente trocando informações entre si, sem detalhes internos.

Diagrama geral do interpretador



O Classificador

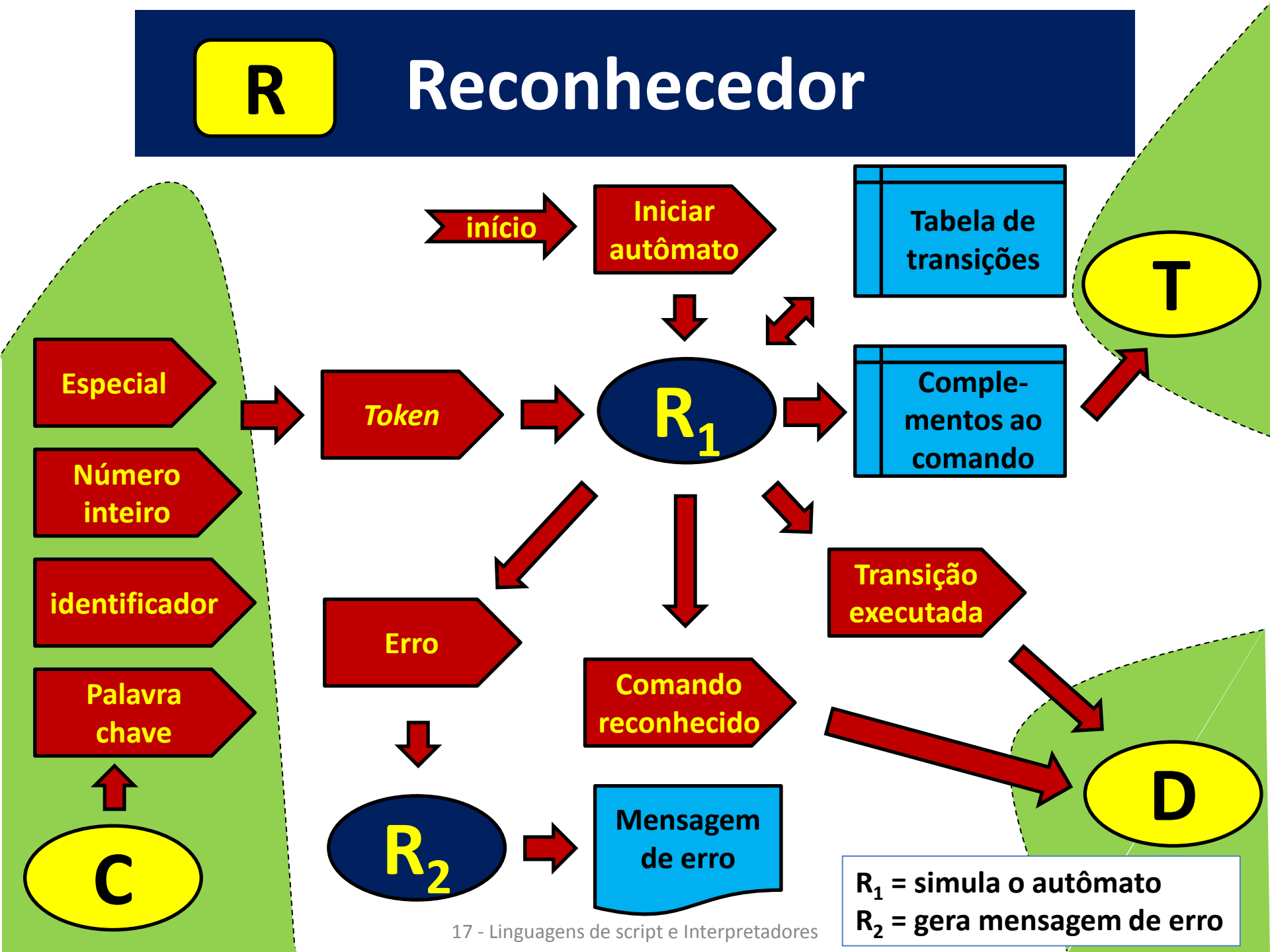
- O módulo **C – classificador** – efetua as seguintes tarefas:
  - lê um arquivo de entrada
  - Imprime a listagem do arquivo de entrada
  - decompõe o texto em caracteres isolados
  - classifica esses caracteres
  - Agrupa os caracteres com base na sua classe, obtendo ***tokens***:
    - **Caracteres especiais** isolados
    - Números **inteiros**
    - **Identificadores**
    - **Palavras chave**
  - Envia para o módulo **R** (**reconhecedor**) da linguagem uma mensagem para cada agrupamento assim extraído, representando eventos que simbolizam "***tokens***"


C Classificador
início
C_0
Linha de texto-fonte
Texto fonte
C_1
C_4
Listagem do programa
C_0 = extrai uma linha do texto-fonte
C_1 = extrai e classifica um caractere da linha
C_2 = separa inteiros e caracteres especiais
C_3 = separa identificadores e palavras-chave
C_4 = inclui mais uma linha na listagem
Caractere especial
Caractere alfabético
Caractere numérico
C_2
Especial
Número inteiro
Sequência alfanumérica
identificador
C_3
Palavra chave
R

O Reconhecedor

- Este módulo **R** – o **Reconhecedor** – destina-se a verificar e garantir que as sentenças da linguagem estejam sintaticamente corretas, de acordo com uma especificação formalizada através de um **autômato**.
- O autômato adequado para essas linguagens de script e outras linguagens regulares é o **autômato finito**, que aqui está formalizado por uma **tabela de transições** de estados.
- Ao iniciar o processamento, o reconhecedor posiciona o autômato em seu estado inicial, e dispara o reconhecimento, solicitando ao Classificador (**C**) um *token*.
- Recebido o *token*, o Classificador consulta a tabela de transições, **determina e aplica a transição adequada**, e envia essa informação ao módulo Decisor (**D**), e disponibiliza na tabela "Complementos ao comando" eventuais outros dados coletados ao longo da análise do comando em exame.
- Detectado algum **erro**, gera uma **mensagem** correspondente.

R Reconhecedor

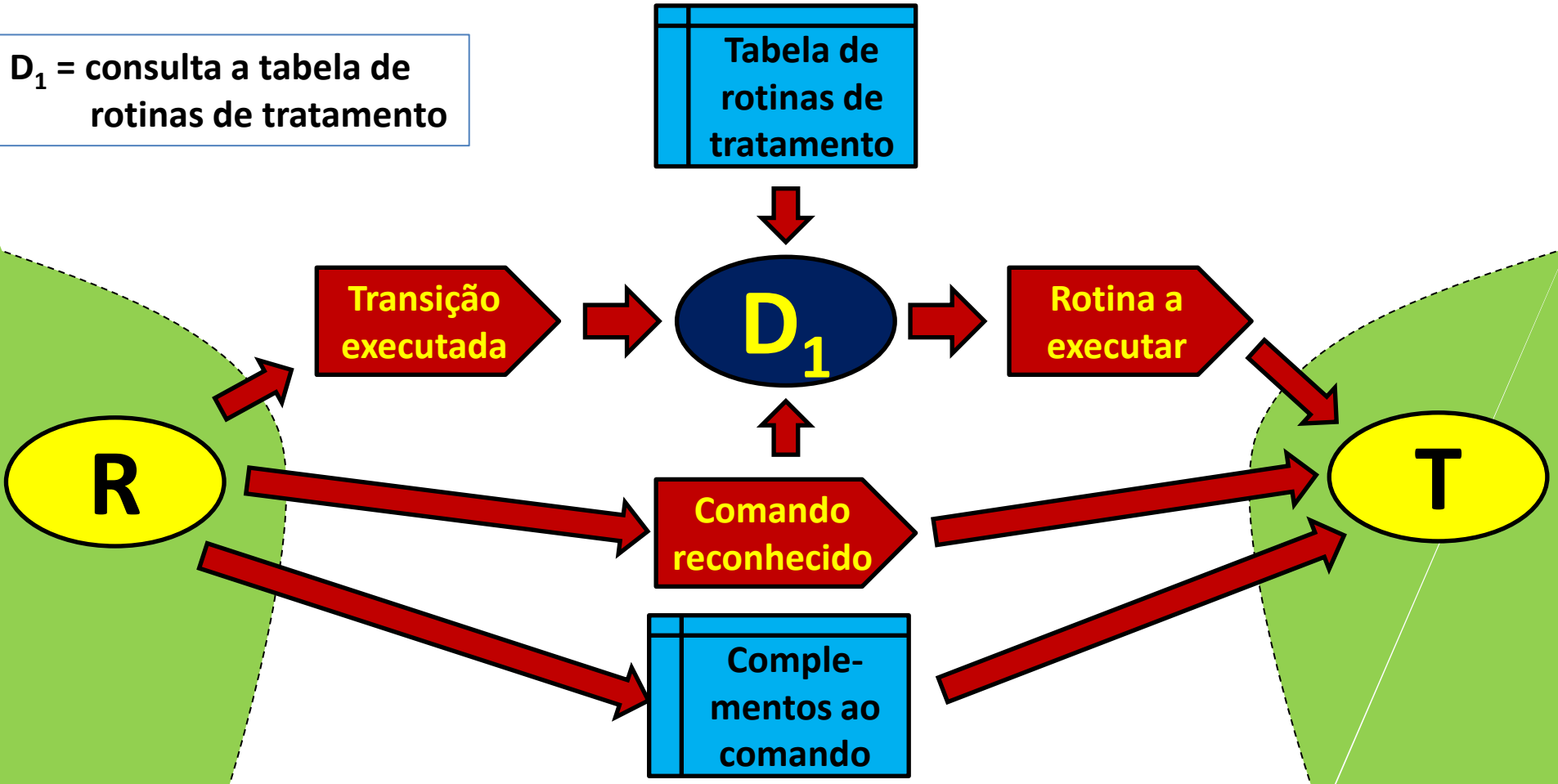


O Decisor

- Este módulo (**D**), como o nome sugere, é aquele que, a cada passo, toma a decisão de qual deve ser a **reação do interpretador** ao material recém-encontrado no programa fonte (ou seja, escolhe qual deve ser a **rotina de tratamento** a ser executada em **resposta** à recepção do último ***token***).
- O módulo **D** limita-se a receber, do reconhecedor (**R**), o evento que informa qual foi a última transição ocorrida, e, usando essa informação, consulta a **tabela das rotinas de tratamento** associadas às transições do autômato que define a linguagem, obtendo então a informação de **qual rotina deve ser executada** como reação a tal ocorrência.
- Essa informação, na forma de um evento de "**rotina a executar**", é repassada ao módulo **T** o qual finalmente acionará esta rotina, efetuando o tratamento desse evento.

D Decisor

D_1 = consulta a tabela de rotinas de tratamento

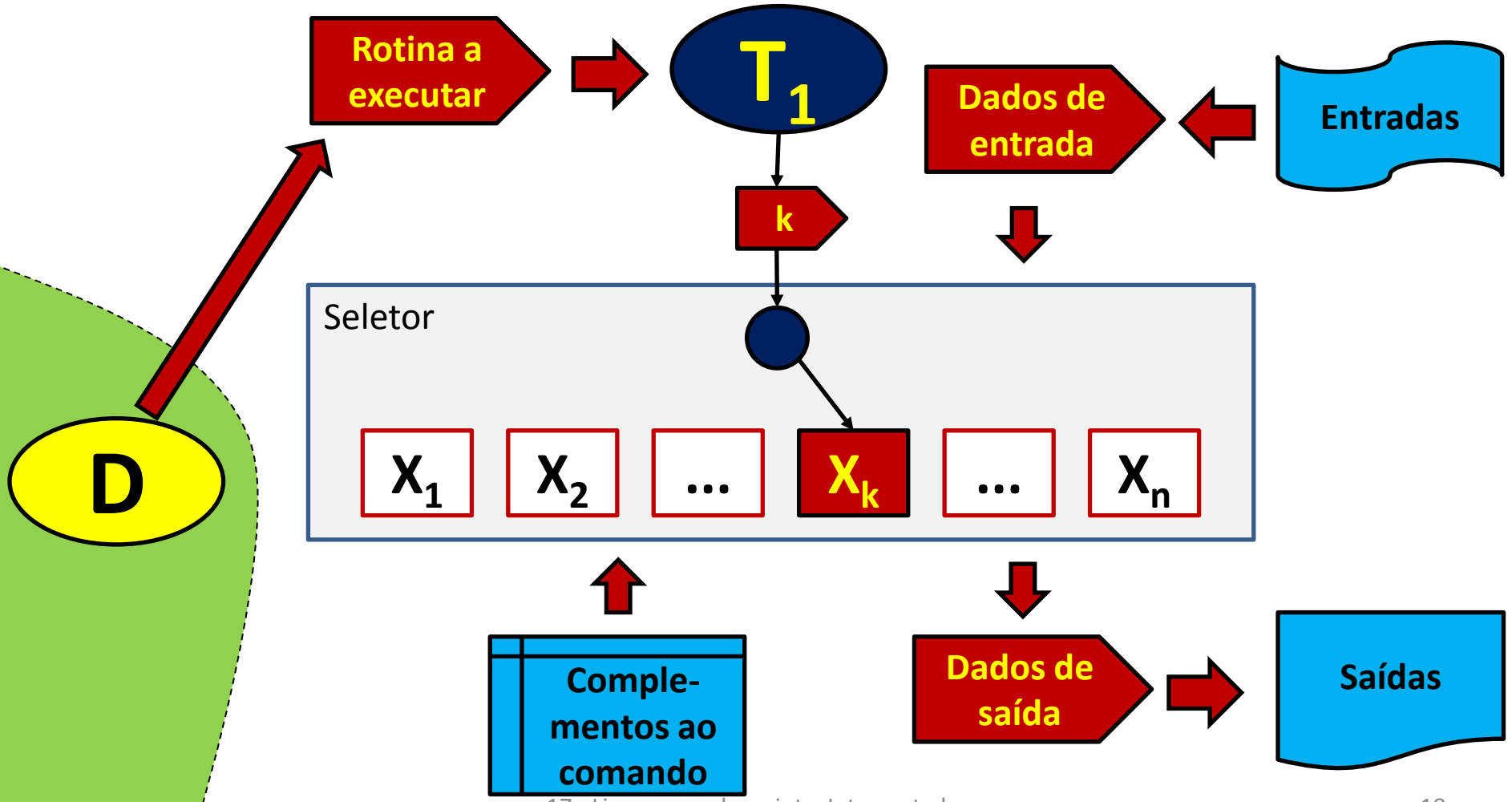


O Tratador de eventos

- Este último módulo é o **tratador de eventos** (**T**), no qual se concentra a parte do programa da qual mais depende o funcionamento do interpretador.
- Sua lógica de controle é simples: recebendo do decisor (**D**) o evento informativo da **rotina a ser executada** (**k**), o módulo tratador de eventos (**T**) simplesmente ativa a execução da rotina correta (X_k) selecionando-a do conjunto das rotinas de tratamento de eventos.
- Se necessário, informações complementares podem ser acessadas pelas rotinas de tratamento X_k em uma estrutura de dados "**complementos ao comando**", preenchida ao longo do processamento do comando.
- A **execução** das rotinas de tratamento efetuam as **operações** especificadas pelos **comandos do usuário**, inclusive as de entrada e saída, conforme o diagrama.

T Tratador de eventos

T_1 = aciona a rotina de tratamento solicitada



Primeiro exemplo – uma linguagem de controle

EXEMPLO 1 DE IMPLEMENTAÇÃO

Um exemplo ilustrativo

- O exemplo a seguir completa esta apresentação esboçando da implementação de uma linguagem de controle muito pequena e extremamente simples.
- Não se trata de uma linguagem com características adequadas para uso real, mas apenas um modelo primitivo para motivar a aplicação de uma técnica para o processamento de linguagens mais elaboradas.
- A finalidade deste exemplo é portanto apenas a de sugerir alguns dos detalhes da implementação de um interpretador para esse tipo de linguagem de script, complementando o estudo geral que fizemos até aqui.

O Interpretador

- Para completar o projeto deste interpretador, está faltando apenas programar as **rotinas de tratamento** para cada transição do autômato, e naturalmente, isso dependerá da linguagem que se estiver sendo implementada.
- Por isso, em sequência são projetadas **miniaturas de interpretadores**, para duas pequenas **linguagens de *script*,** aqui utilizadas como exemplos ilustrativos:
  - A primeira é uma **linguagem de controle** para o acionamento do software básico contido no sistema de programação.
  - A segunda é uma **linguagem de programação** incompleta, dotada de recursos sintáticos apenas suficientes para permitir a redação de programas pequenos e triviais.

Os comandos de controle

- Neste exemplo ilustrativo, a **sintaxe dos comandos** é extremamente simples:
  - O símbolo **$** inicia o comando
  - Uma palavra-chave identifica o comando: **LIN, DIR, DEL, RUN, END**
  - Um **identificador alfabético** completa os comandos **LIN**, **DEL** e **RUN**
- Nessa mini-linguagem há apenas **cinco comandos, triviais**:
  - $ **LIN** – login (identifica um usuário que acessa o sistema)
  - $ **DIR** – directory (lista a pasta de arquivos do sistema)
  - $ **DEL** – delete (remove um arquivo)
  - $ **RUN** – run (executa um programa contido em um arquivo)
  - $ **END** – end (encerra a execução de uma sequência de comandos)

Autômato para a linguagem de controle

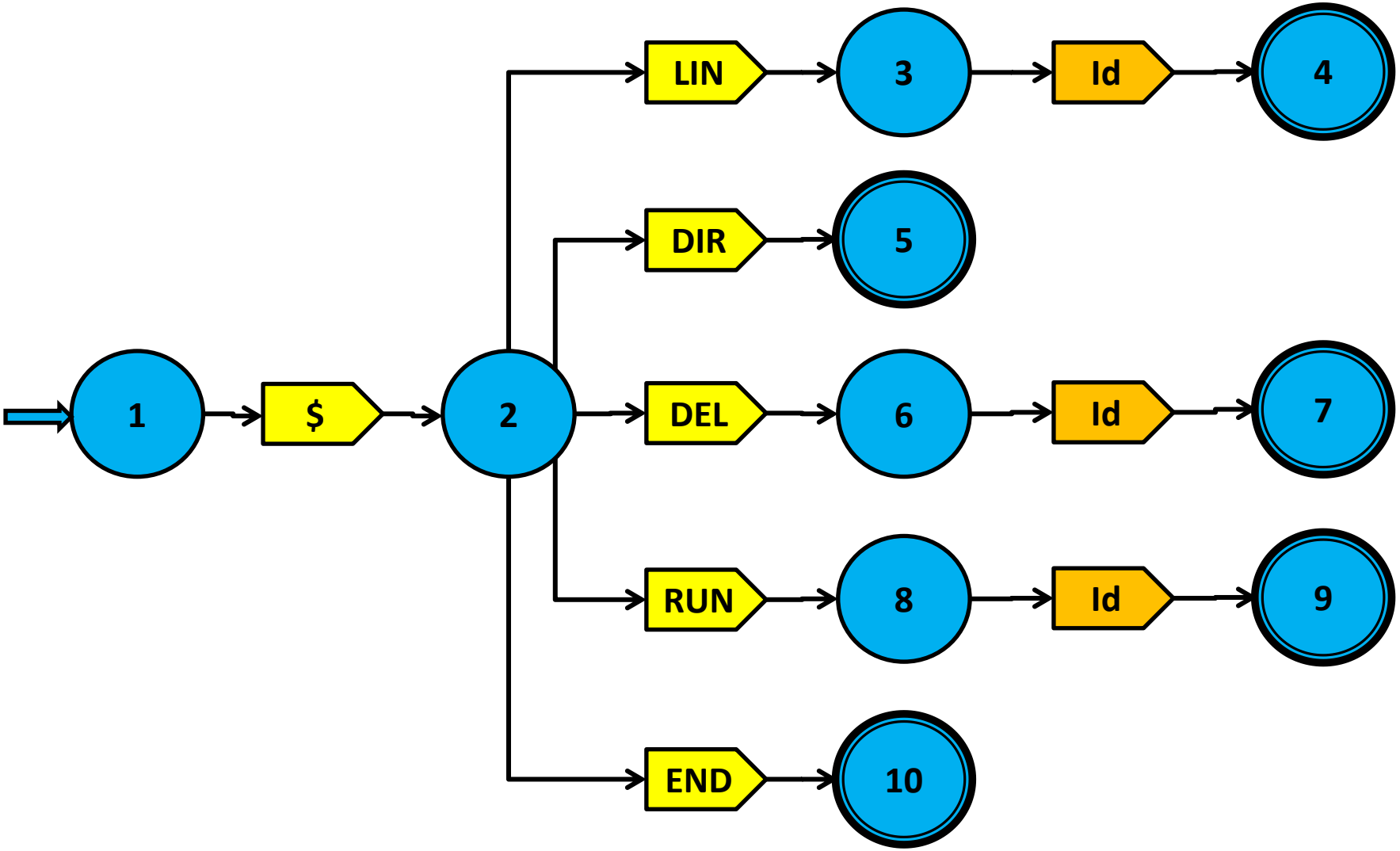


Tabela de transições

	$	LIN	DIR	DEL	RUN	END	id	outros
>1	2							
2		3	5	6	8	10		
3							4	
4>								
5>								
6							7	
7>								
8							9	
9>								
10>								

Lista de transições

```
>1      {$ 2}                  O estado 1 é o estado inicial
2       {LIN 3, DIR 5,
         DEL 6, RUN 8,
         END 10}
3       {id 4}
4>      {}                     O estado 4 termina o comando LIN
5>      {}                     O estado 5 termina o comando DIR
6       {id 7}
7>      {}                     O estado 7 termina o comando DEL
8       {id 9}
9>      {}                     O estado 9 termina o comando RUN
10>     {}                     O estado 10 termina o comando END
```

As rotinas de tratamento

- Para completar o perfil do interpretador, apresentam-se, em linguagem natural, esboços descritivos da lógica das rotinas de tratamento associadas às transições do reconhecedor dessa linguagem de controle.
- Para se obter um interpretador operante para essa linguagem, basta codificar essas rotinas de tratamento e incorporá-las ao reconhecedor sintático já apresentado. Isso não será detalhado aqui.
- Os slides a seguir mostram três representações equivalentes do mesmo interpretador de comandos de controle, usando notações diferentes: a de autômato, a de tabela de transições e a de lista de transições.

Interpretador da linguagem de controle

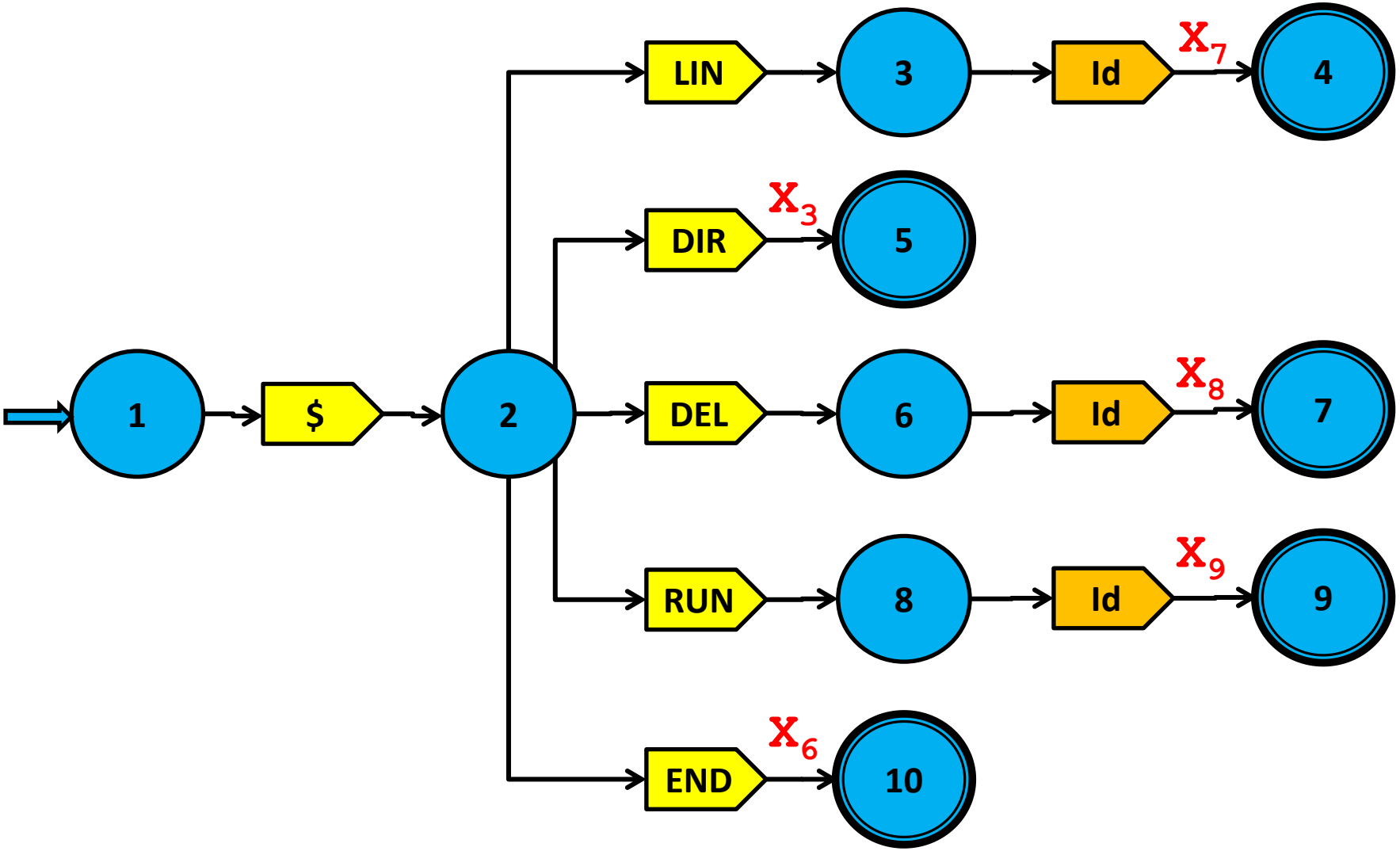


Tabela de transições com tratamentos

	$	LIN	DIR	DEL	RUN	END	id	outros
>1	2							
2		3	5 X_3	6	8	10 X_6		
3							4 X_7	
4>								
5>								
6							7 X_8	
7>								
8							9 X_9	
9>								
10>								

Os nomes das rotinas de tratamento estão escritas em vermelho, e sua descrição em alto nível se encontra nos slides que seguem. Ausência de nome indica nenhuma ação.

Lista de transições com tratamentos

```
>1    {$ 2}
2     {LIN 3, DIR 5 X3, DEL 6, RUN 8, END 10 X6}
3     {id 4 X7}
4>    {}
5>    {}
6     {id 7 X8}
7>    {}
8     {id 9 X9}
9>    {}
10>   {}
```

LIN – login

- Formato: **$ LIN** usuário

ROTINA X_7 **transição 3-4 com identificador**

- Caso o usuário referenciado não conste na tabela de usuários do sistema, envia mensagem de erro.
- Caso contrário, estabelece um diálogo de login para o fornecimento de informações ao sistema:
  - Máximo tempo de utilização do sistema
  - Máximo de utilização de memória
  - Máximo de operações de entrada/saída

DIR – directory

- Formato: **$ DIR**

ROTINA X_3 **transição 2-5 com "DIR"**

- Lista na tela os nomes dos arquivos existentes na pasta do usuário corrente

DEL – delete

- Formato: **$ DEL** nome_do_arquivo

ROTINA X_8 **transição 6-7 com identificador**

- Se o arquivo cujo nome é referenciado neste comando existir na pasta do usuário corrente, ele será removido dessa pasta e movido para uma pasta de lixeira do sistema, para viabilizar uma eventual recuperação posterior.
- Esse arquivo pode ser de dados ou de programa.
- Caso tal arquivo não esteja presente na pasta do usuário, o comando não terá qualquer efeito.

RUN – run

- Formato: **$ RUN** nome_do_programa

ROTINA X_9 **transição 8-9 com identificador**

- Caso o programa referenciado seja um programa do usuário, executável na MVN, este comando:
  - Ativa o Loader para carregar o programa referenciado na memória do simulador MVN
  - Dá partida à simulação da execução do programa
- Caso seja algum programa da plataforma hospedeira, promove sua execução.
- Caso não se enquadre em nenhum desses dois casos, envia ao requisitante uma mensagem de erro.

END – end

- Formato: **$ END**

ROTINA X_6 **transição 2-10 com "END"**

- Relata ao usuário todas as informações referentes a essa sessão no sistema.
- Caso a sessão tenha excedido algum limite estabelecido no login, emite para o usuário uma mensagem de erro
- Recupera para o sistema todos os recursos que tenham sido alocados para a execução de programas ativados pelo usuário.
- Encerra a sessão do usuário corrente, efetuando seu logoff.

Para exercitar o interpretador

- Implementado o processador da linguagem de controle, exercite-o submetendo-lhe o seguinte script:

$ LIN NNNN	Teste com NNNN (conhecido e não)
$ DIR	Teste se imprime diretório correto
$ DEL AAAA	Teste se apaga AAAA e salva na lixeira
$ RUN BBBB	Teste se executa BBBB (existente e não)
$ END	Teste se encerra corretamente a sessão

Segundo exemplo – uma linguagem de programação

EXEMPLO 2 DE IMPLEMENTAÇÃO

Linguagens de alto nível

- Foram discutidos anteriormente nesta disciplina os principais atributos que caracterizam as linguagens de programação de alto nível.
- Para implementá-las usando a técnica da interpretação, convém que sua sintaxe seja simples, no sentido de que utilizem construtos sintáticos que tornem os comandos o mais independentes possível uns dos outros.

Interpretação de ling. de programação

- O uso de interpretadores no processamento de linguagens de programação só dá bons resultados quando a linguagem é formada de comandos suficientemente independentes e simples, pois o contrário exige técnicas artificiosas e disso resulta um processamento trabalhoso e lento.
- Os elementos das linguagens que mais impacto negativo causam no desempenho dos processos de interpretação são:
  - **Estruturas aninháveis**, envolvendo:
    - Declarações, escopos locais e globais (visibilidade)
    - Expressões aninháveis: aritméticas, booleanas
  - **Agrupamentos sintáticos** formados por múltiplos comandos
  - **Comandos** complexos, **com estrutura recursiva**
    - Condicionais
    - Iterativos
    - Blocos aninháveis

Interpretação de linguagens de script

- Linguagens de *script* procuram evitar tais elementos, favorecendo implementações que empregam com eficácia os interpretadores no lugar de compiladores.
- Assim, quando destinadas à codificação de programas em alto nível de abstração, as linguagens alvo de implementação interpretada costumam restringir os seus recursos oferecidos, assumindo a personalidade de linguagens de *script.*

Elementos típicos das linguagens

- **Sequências** – programas podem ser vistos como sequências de comandos independentes
- **Rótulos** – referências são permitidas apenas a comandos rotulados
- **Comandos imperativos** essenciais (variam entre linguagens):
  - **Atribuição** – associa o valor de uma expressão a uma variável.
  - **Entrada de dados** – associa a uma variável o valor de um dado lido a partir de um dispositivo de entrada.
  - **Saída de dados** – grava em um dispositivo de saída o valor calculado de uma expressão.
  - **Desvio** – desvia para um rótulo especificado o fluxo de execução do programa. No caso de desvios condicionais, o desvio se dá em função do resultado da avaliação de uma expressão booleana. Nas linguagens mais simples, essa expressão se limita a uma comparação entre dois valores.

Projeto de um interpretador

- A interpretação de linguagens de programação é efetuada de maneira similar à que foi exemplificada antes, para o script de controle apresentado.
- Apenas para orientar a implementação interpretada de linguagens de programação, descreve-se a seguir a interpretação de uma linguagem formada apenas de comandos de atribuição, a uma variável, de um valor resultante do cálculo de uma expressão aritmética.
- O diagrama de estados a seguir descreve essa pequena linguagem. É óbvio que, na prática, as linguagens de programação não podem ser tão simples, pois exigem muito mais recursos.
- O modelo simplificado aqui apresentado pode, no entanto, servir como base para as extensões que se fizerem necessárias em casos mais complexos.

Exemplo de implementação de expressões

INTERPRETAÇÃO DE EXPRESSÕES ARITMÉTICAS

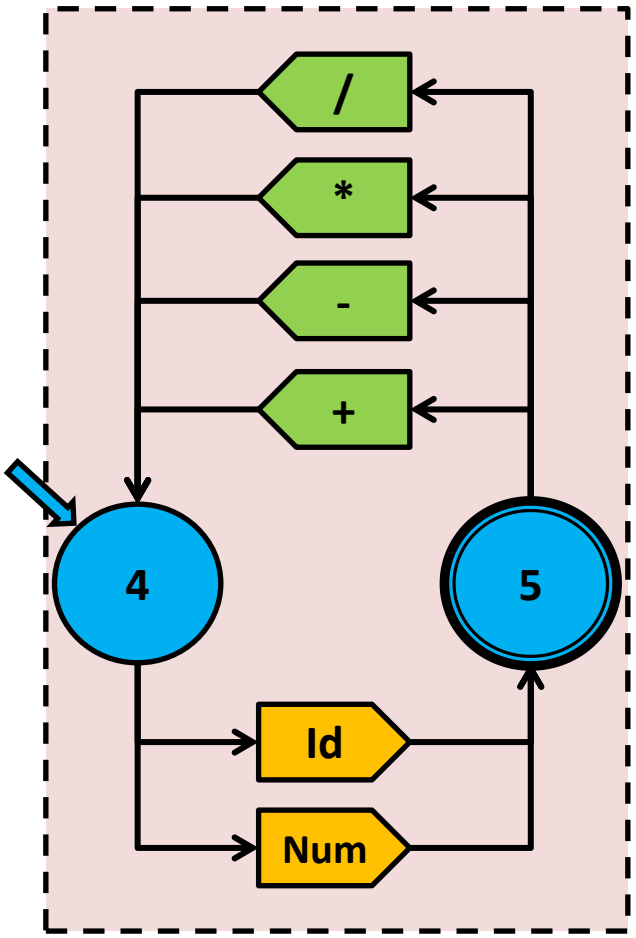
Expressões aritméticas

- Desde o surgimento das primeiras linguagens de alto nível, as expressões aritméticas têm exercido uma forte influência, não só na programação, mas até mesmo no projeto das características dessas linguagens, razão pela qual o processamento de expressões sempre foi estudado com muito cuidado.
- Em um estudo preliminar como este, o estudo de expressões não será profundo. Examinamos apenas o processamento de expressões que envolvem variáveis, números inteiros, e as quatro operações básicas. Deixamos de considerar os parênteses, as chamadas de funções, e operações aritméticas adicionais, todos obrigatórios nas linguagens de programação usuais.

Uma expressão aritmética simplificada

- Assim, a forma de expressão aritmética aqui utilizada, neste estudo preliminar sobre a implementação de linguagens de programação, inclui apenas os seguintes elementos:
  - Números inteiros.
  - Variáveis inteiras simples.
  - Operações aditivas: soma e subtração inteiros.
  - Operações multiplicativas: produto e divisão inteiros.
  - Precedência das operações multiplicativas sobre as aditivas.

Sintaxe da expressão simplificada



- O autômato ao lado reconhece a expressão aritmética simplificada aqui adotada, escrita em notação infixa.
- Inicia sua operação no estado 4, onde aceita um identificador ou um número inteiro, transitando para o estado (de aceitação) 5.
- No estado 5, se receber um sinal de operação, volta ao estado 4, ou então, em caso contrário, encerra com sucesso sua operação.

A sequenciação das operações infixas

- Há diversas técnicas para interpretar expressões aritméticas, dentre as quais uma está sendo aqui adotada, sem consideração de mérito.
- Na interpretação de expressões infixas, a sequência das operações que incidem sobre seus operandos deve ser reordenada, para atender as regras de precedência convencionadas para sua avaliação.
- No nosso caso, a ausência de parênteses nas expressões reduz essa reordenação à priorização da aplicação dos operadores multiplicativos sobre a dos aditivos:

  multiplicativo > aditivo > final

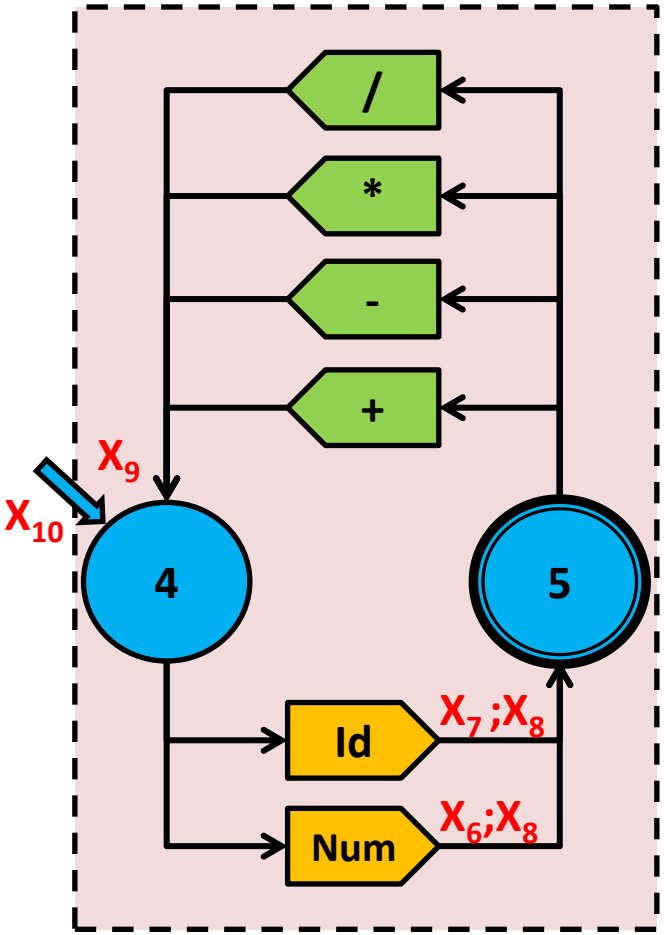
- Para realizar essa operação, duas pilhas são empregadas:
  - uma delas para memorizar operandos ainda não utilizados, e
  - a outra, para memorizar operadores já analisados, mas ainda não aplicados.

O procedimento de sequenciação

ROTINA X_{10} **à entrada do estado inicial (4)**

- Para dar partida ao algoritmo, ao início do processamento da expressão, empilham-se:
  - um valor 0 (zero) na pilha de operandos, iniciando o valor acumulado da expressão até o momento
  - um operador aditivo de soma (+) na pilha dos operadores pendentes.
- A seguir, vão sendo extraídos, alternadamente, operandos e operadores da expressão, executando-se algumas operações de acordo com os elementos extraídos e com os conteúdos dos topos das duas pilhas.
- Ao ser finalizado o processamento de uma expressão correta, não podem restar pendências, portanto ambas as pilhas deverão estar vazias.
- Descrevem-se a seguir as ações a aplicar em cada caso.

Autômato com rotinas semânticas



Rotinas semânticas

- $X_6\ X_8$ transição 4-5 com Num
- $X_7\ X_8$ transição 4-5 com Id
- X_9 transição 5-4 com operador
- X_{10} 1ª entrada no estado inicial (4)

Associação de valores aos operandos

- Estando no estado 4 e encontrado um operando (identificador ou número), determina-se o seu valor, que é empilhado na pilha de operandos, e transita-se para o estado 5.

ROTINA X_6 **transição 4-5 com Num**

- Se o operando for um **Num** (operando numérico), converte-se a sequência de caracteres numéricos para um valor binário aplicando-se uma rotina de conversão decimal ASCII para binário.

ROTINA X_7 **transição 4-5 com Id**

- Se o operando for um **Id** (identificador, ou seja, nome de uma variável) já conhecido, o seu valor se obtém consultando o último valor a ele associado na tabela das variáveis do interpretador.
- Se o **id** encontrado não for um identificador conhecido, ele deverá ser incluído na tabela de identificadores do interpretador, e um valor zero será arbitrariamente atribuído a ele.

ROTINA X_8 **entrando no estado 5, logo após executar X_6 ou X_7**

- Guarda-se o valor do operando, assim obtido, na variável **VALOR** do interpretador, e empilha-se este valor na pilha de operandos da expressão em processamento.

Dinâmica da interpretação da expressão

- Continuando a interpretação da expressão, no estado 5 recebe-se um operador, multiplicativo (* ou /), aditivo (+ ou -) ou ou então um final de expressão (⊥).
- Da análise conjunta desse item com o estado da pilha de operadores, aplica-se uma ação adequada (empilhamento de operador, ou aplicação de operador pendente sobre operandos ainda não processados).
- Toda vez que um operador pendente for desempilhado para ser aplicado, os dois valores empilhados mais recentemente na pilha de operandos serão desempilhados, e após aplicar-se sobre eles o operador pendente, empilha-se finalmente na pilha de operandos o resultado assim calculado.
- A tabela a seguir sintetiza todas essas ações aplicáveis.

Ações a executar em cada situação

- Assim, sempre que um operador (X) for extraído para análise, deverá ser comparado com o operador (Y) presente no topo da pilha de operadores, e a ação a ser tomada deverá seguir a tabela abaixo:

ROTINA X_9 **transição de 5 para 4**

Operador recém-extraído da expressão (X↓) \ Operador presente no topo da pilha de operadores (Y→)	+	–	*	/
+	Opera + Empilha +	Opera – Empilha +	Opera * Empilha +	Opera / Empilha +
–	Opera + Empiha –	Opera – Empilha –	Opera * Empilha –	Opera / Empilha –
*	Empilha *	Empilha *	Opera * Empilha *	Opera / Empilha *
/	Empilha /	Empilha /	Opera * Empilha /	Opera / Empilha /
outros	Opera tudo	Opera tudo	Opera tudo	Opera tudo

- **Opera** – **Desempilha** e aplica operador contido no topo da pilha sobre os dois operandos, **desempilha** esses dois operandos e **empilha** em seu lugar o resultado da operação.
- **Opera tudo** – Para cada operador na pilha, aplica a rotina **Opera**.
- **Empilha** – Enquanto o operador recém-extraído tiver precedência menor que o operador do topo da pilha, aplica a rotina **Opera**. Por último, empilha-o na pilha de operadores.

Exemplo

Vamos simular aqui o processamento, passo a passo, da expressão

5 + 4 – 3 * 6 / 2 + 1 ⊥

Tokens	Estado/ Rotina	Pilha de operadores	Pilha de operandos	Ações executadas pelas rotinas
Início	4/X_{10}	+	0	Empilha + e Empilha 0 p/ iniciar
5	5/X_6;X_8	+	0 5	Empilha operando (5)
+	4/X_7;X_8	+	(0+5)	Opera(+) e Empilha operador (+)
4	5/X_6;X_8	+	5 4	Empilha operando (4)
–	4/X_7;X_8	–	(5+4)	Opera(+) e Empilha operador(-)
3	5/X_6;X_8	–	9 3	Empilha operando (3)
*	4/X_7;X_8	– *	9 3	Empilha operador(*)
6	5/X_6;X_8	– *	9 3 6	Empilha operando (6)
/	4/X_7;X_8	– /	9 (3*6)	Opera(*) e Empilha operador(/)
2	5/X_6;X_8	– /	9 18 2	Empilha operando (2)
+	4/X_7;X_8	–	9 (18/2)	Opera(/) [– e + mesma precedência]
		+	(9-9)	Opera(-) e Empilha operador (+)
1	5/X_6;X_8	+	0 1	Empilha operando (1)
⊥ Final	X_9		(0+1)	Opera tudo e termina.
			1	1 é o valor final da expressão

Exemplo de implementação de comandos imperativos

INTERPRETAÇÃO DE LINGUAGEM IMPERATIVA

Interpretação de comandos imperativos

- A forma mais simples de promover a interpretação de comandos imperativos consiste em executar uma rotina previamente construída, a eles equivalente.
- Durante a análise do programa, dados referentes à particular utilização do comando são coletados, preenchendo-se com eles as lacunas dessa rotina, que assim fica preparada para produzir os resultados específicos dela esperados.
- Uma vez executada a rotina assim construída, suas informações específicas são descartadas, liberando o código original para ser reutilizado.
- A seguir, o reconhecedor da linguagem é mostrado em três formas (sem rotinas semânticas): diagrama de estados, tabela de transições e lista de transições.

Autômato reconhecedor sintático da linguagem de alto nível

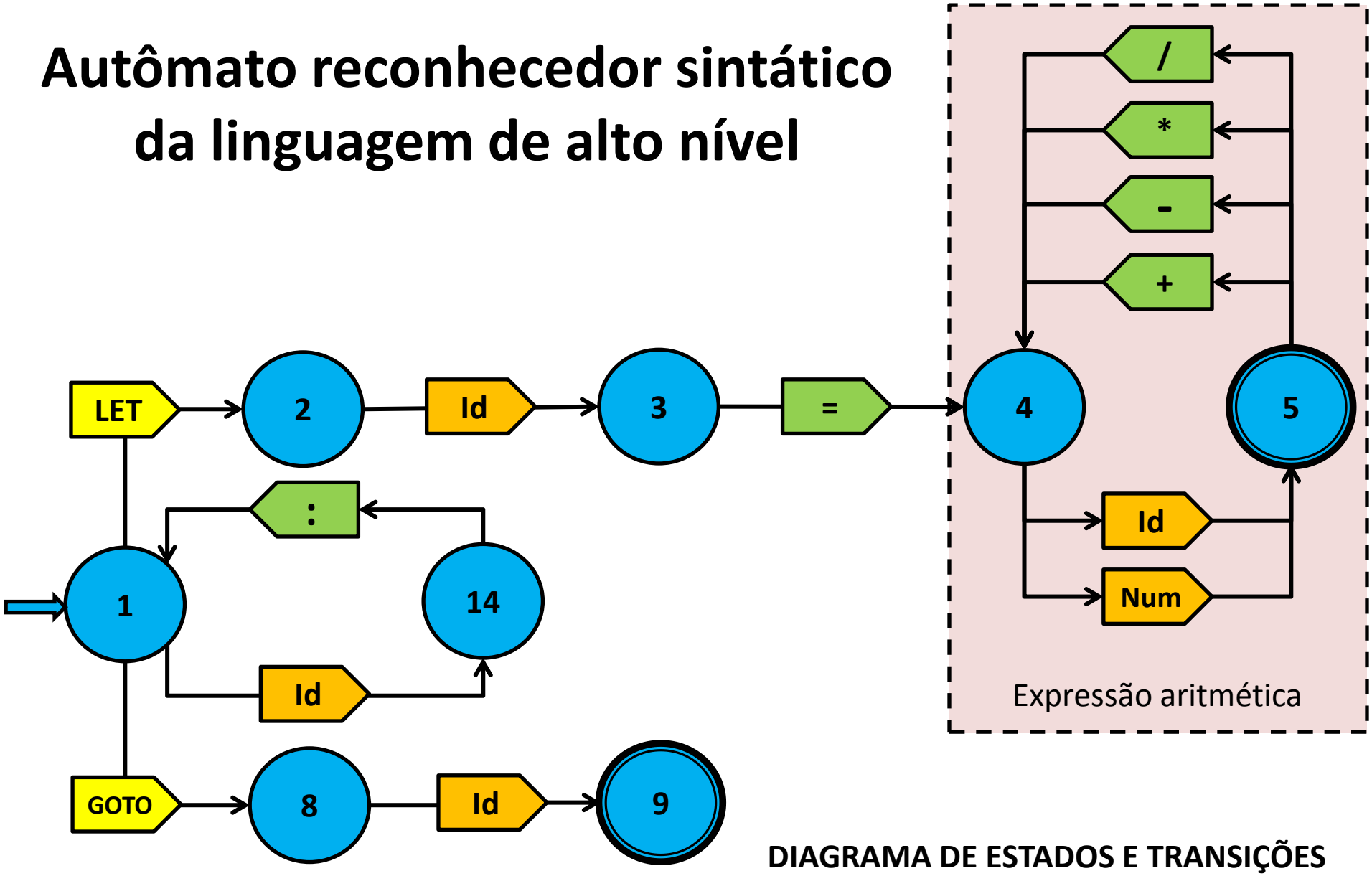


DIAGRAMA DE ESTADOS E TRANSIÇÕES

Tabela de transições do autômato

	LET	GOTO	Id	Num	:	=	+	-	*	/	outros
>1	2	8	14								
2			3								
3						4					
4			5	5							
5>							4	4	4	4	
8			9								
9>											
14					1						

TABELA DE TRANSIÇÕES

Lista de transições do autômato

```
>1   {LET 2, GOTO 8, id 14}
2    {id 3}
3    {= 4}
4    {id 5, num 5}
5>   {+ 4, - 4, * 4, / 4}
8    {id 9}
9>   {}
14   {: 1}
```

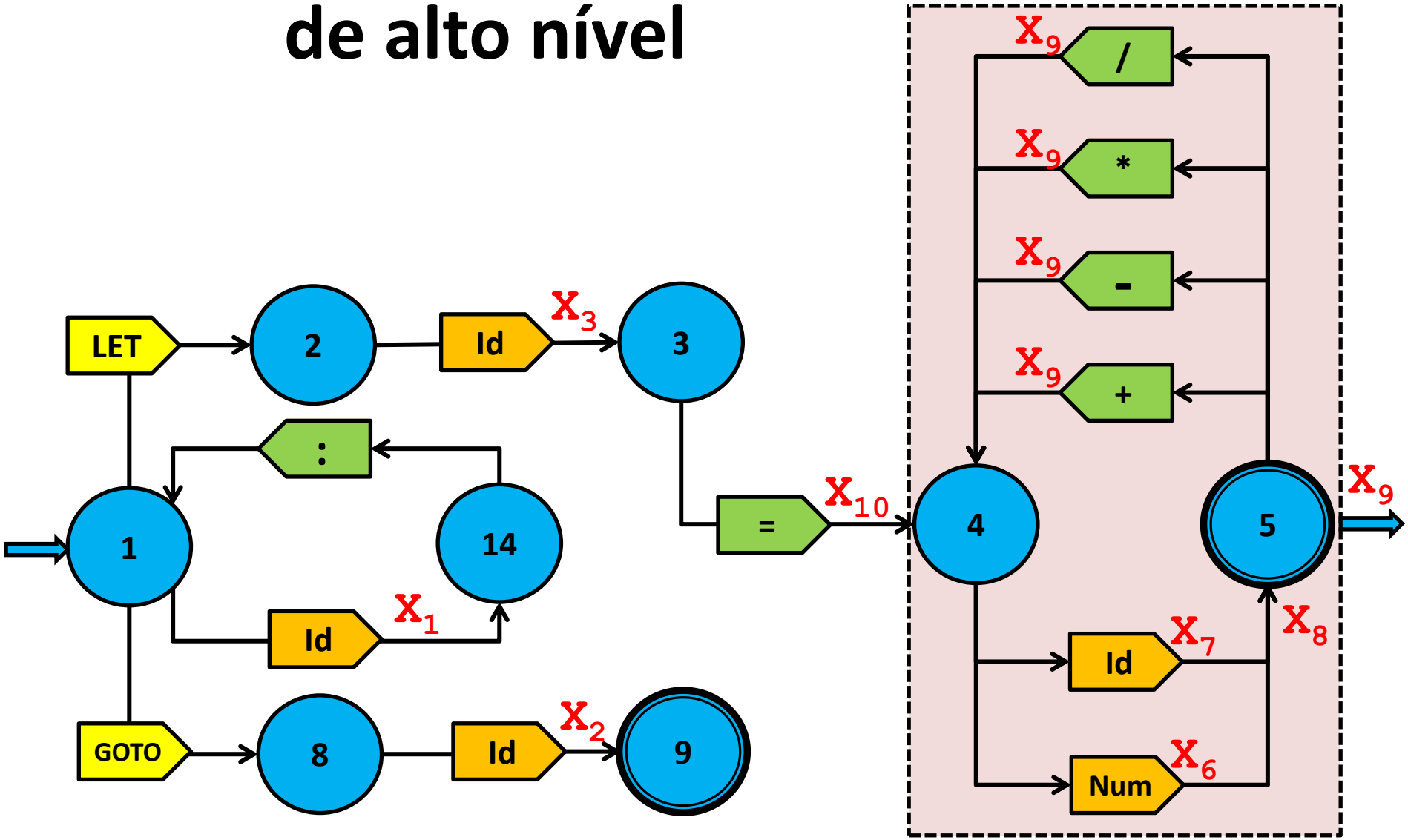
LISTA DE TRANSIÇÕES

INTERPRETAÇÃO DOS COMANDOS

Interpretação dos comandos

- Na sequência, esboça-se o conjunto de rotinas (semânticas) de tratamento para os diversos elementos da linguagem anteriormente formalizada.

Interpretador da linguagem de alto nível



Definição de rótulos

- Os rótulos são associados aos respectivos comandos na transição **1-14**, quando um identificador é encontrado no programa fonte em um contexto de associação a um comando.

ROTINA X_1 **transição 1-14 com id**

- O tratamento mais simples dessa definição de um rótulo é memorizá-lo numa tabela de nomes, associando-o à posição (número da linha) do programa fonte em que ele foi encontrado.
- Caso o rótulo em questão já tenha sido definido anteriormente, cabe emitir uma mensagem de erro.

Comando de desvio

- Nesta linguagem todos os comandos de desvio são incondicionais: o comando **GO TO**.

ROTINA X_2 **transição 8-9 com id**

- Ao ser detectada a referência a um rótulo, na transição **8-9**, o identificador especificado deve ser procurado na tabela de nomes, e aí deve ser obtido o número da linha associada à definição desse rótulo, da qual deverá ser extraído o próximo comando a ser interpretado (isto já implementa o desvio do comando **GO TO**).
- Caso o identificador se refira a um objeto de outro tipo no programa, uma mensagem de erro deve ser emitida.
- Caso tal identificador não conste na tabela, registra-se a pendência de que, até o final do programa, deverá surgir, na parte do programa-fonte ainda não analisada, a definição do rótulo não encontrado.

Comando de atribuição

- O comando de atribuição tem duas partes: o identificador que indica o destino da atribuição, e uma expressão, que fornece o valor a ser atribuído.
- Na transição **2-3** memoriza-se o identificador destino da atribuição. **ROTINA X_3** **transição 2-3 com id**
- Na transição **3-4** promove-se o preenchimento inicial das pilhas de operador e de operando, para dar partida ao cálculo da expressão. **ROTINA X_{10}** **transição 3-4 com id**
- Nas transições **4-5** e **5-4** é feito o cálculo do valor da expressão (detalhado anteriormente: expressão aritmética).
- Na saída pelo estado **5**, o valor da expressão e o identificador destino já são conhecidos, podendo-se depositar o valor calculado nesse destino, completando a interpretação do comando de atribuição. **ROTINA X_9** **saída pelo estado 5**

INSERÇÃO DAS ROTINAS DE TRATAMENTO DO INTERPRETADOR

Representação formal do interpretador

- Os três slides a seguir mostram **três formas** equivalentes de **representação formal** do interpretador completo, compreendendo tanto a **sintaxe** da linguagem como o seu tratamento, que tem por meta promover a execução das ações necessárias à execução das ações simbolizadas pelas instruções do programa. Naturalmente, na prática somente uma dessas representações é suficiente.

Interpretador, com tratamento semântico: diagrama de estados

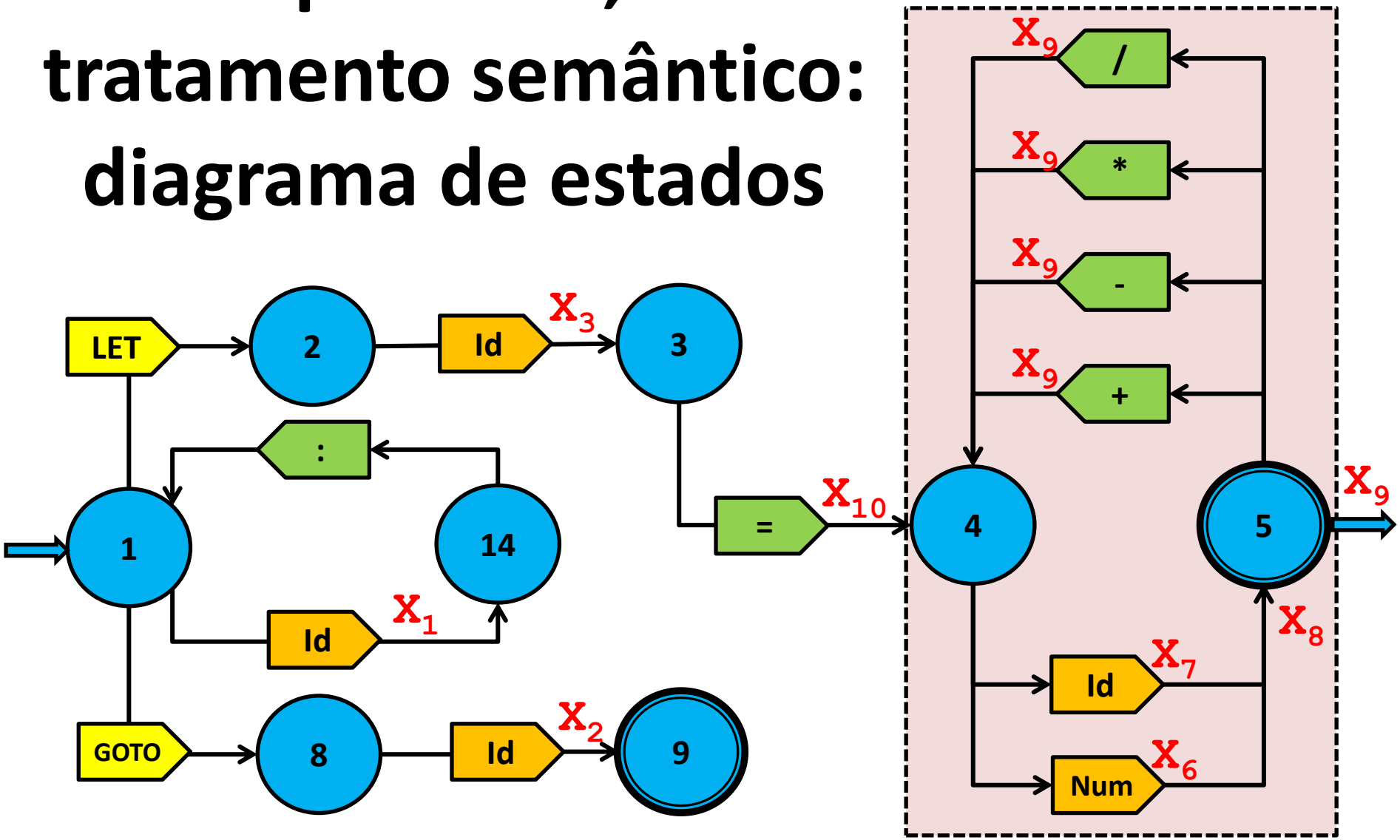


Tabela de transições do interpretador, com tratamento semântico

	LET	GOTO	Id	Num	:	=	+	-	*	/	outros
>1	2	8	14 X_1								
2			3 X_3								
3						4 X_{10}					
4			5 X_7	5 X_6							
5>							4 X_9	4 X_9	4 X_9	4 X_9	X_8 X_9
8			9 X_2								
9>											
14					1						

Lista de transições com os tratamentos semânticos do interpretador

>1	{LET 2, GOTO 8, id 14 X_1}
2	{id 3 X_3}
3	{= 4 X_{10}}
4	{id 5 X_7, num 5 X_6}
5>X_8X_9	{+ 4 X_9, - 4 X_9, * 4 X_9, / 4 X_9}
8	{id 9 X_2}
9>	{}
14	{: 1}

EXEMPLO DE INTERPRETAÇÃO DE UM PROGRAMA-FONTE

Exemplo

- Aqui é apresentado um pequeno exemplo completo de programa com apenas três comandos (no caso, um loop infinito) sendo avaliado usando o interpretador aqui construído.

```
1          LET A = 10;
2  LOOP : LET A = A+7;
3          GO TO LOOP;
```

1

LET A = 10;

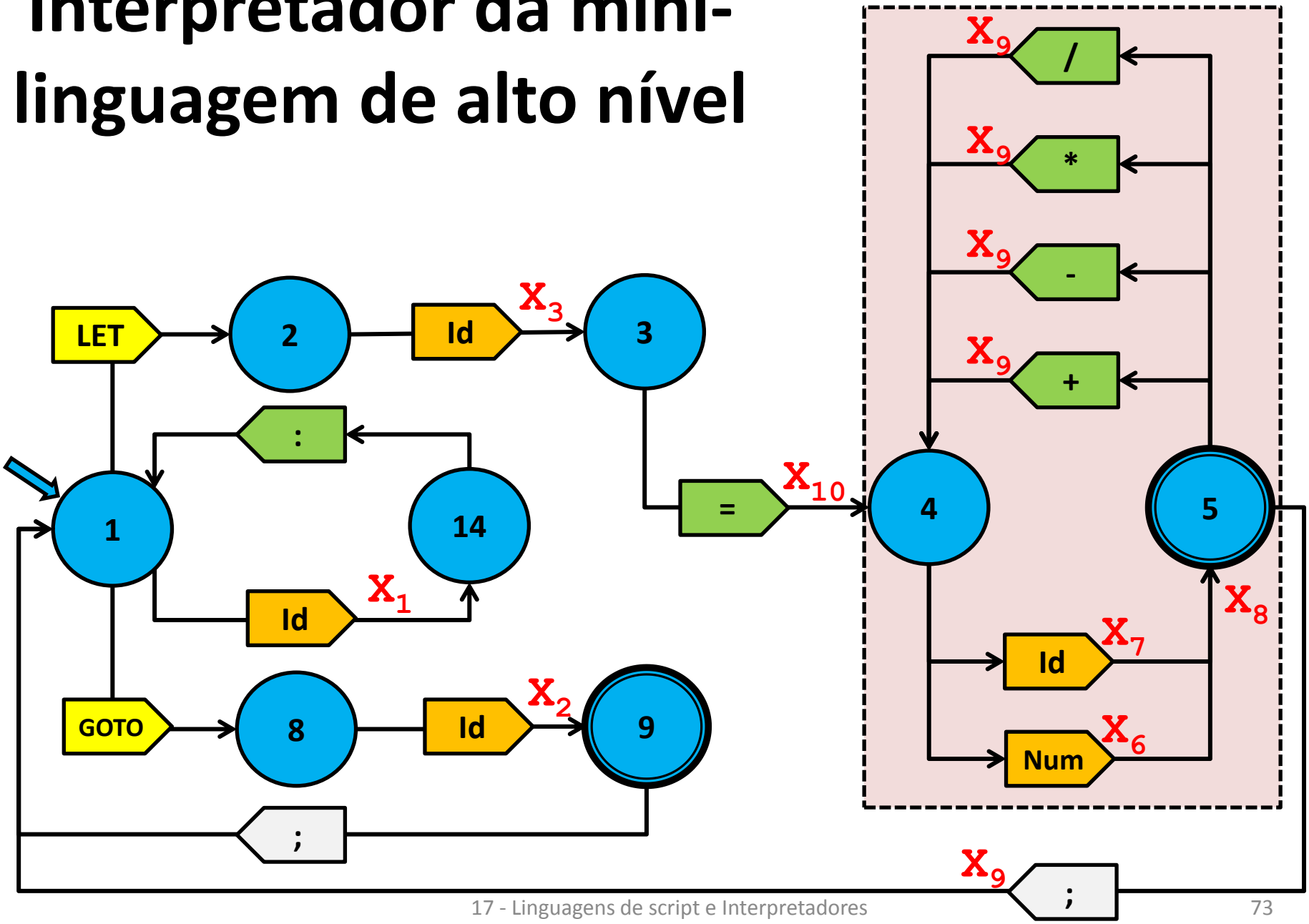
est.	átomo	trat.	efeito
1	LET		
2	A	X_3	destino=A
3	=	X_{10}	zera a expressão
4	10	X_6	empilha operando
		X_8	aplica (0+10)=10
5	;	X_9	atribui 10 a A.
1			

2

LOOP : LET A = A+7;

est.	átomo	trat.	Efeito
1	LOOP	X_1	LOOP←linha 2
14	:	-	-
1	LET	-	-
2	A	X_3	destino=A
3	=	X_{10}	zera a expressão
4	A	X_7	empilha valor(A)=10
		X_8	aplica (0 + 10)=10
5	+	X_9	empilha +
4	7	X_6	empilha 7
		X_8	aplica (10 + 7)=17
5	;	X_9	atribui 17 a A
1			

Interpretador da mini-linguagem de alto nível



3

GO TO LOOP;

estado	átomo	trat.	Efeito
1	GO TO	-	-
8	LOOP	X_2	reposiciona: linha 2
9	;	-	-
1			

2

LOOP : LET A = A+7;

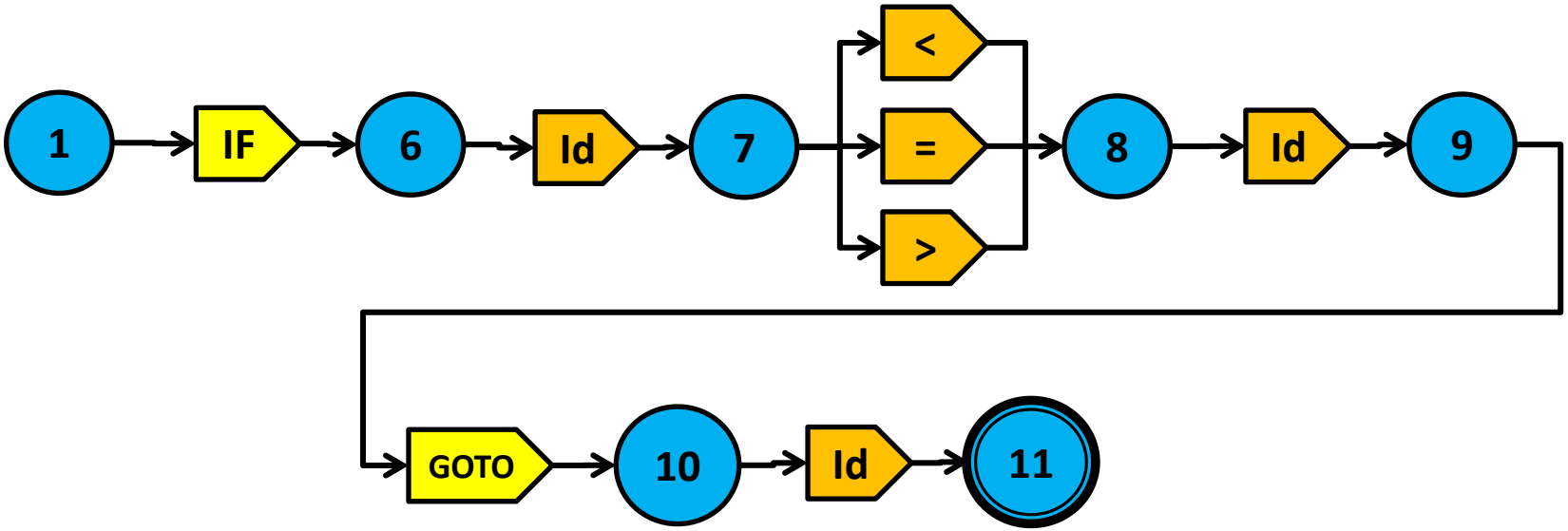
est.	átomo	trat.	Efeito
1	LOOP	X_1	LOOP←linha 2
14	:	-	-
1	LET	-	-
2	A	X_3	destino=A
3	=	X_{10}	zera a expressão
4	A	X_7	empilha valor(A)=17
		X_8	aplica (0 + 17)=17
5	+	X_9	empilha +
4	7	X_6	empilha 7
		X_8	aplica (17 + 7)=24
5	;	X_9	atribui 24 a A
1			

E assim por diante...

EXEMPLO DE UMA AMPLIAÇÃO: ADIÇÃO DE UM COMANDO CONDICIONAL

Inclusão de um comando condicional

- Exemplifica-se aqui uma ampliação do projeto, através do acréscimo de um novo comando à linguagem.
- Inclui-se um **desvio condicional** (**IF..GOTO**) simplificado, cuja sintaxe é mostrada no diagrama de estados abaixo, que será integrado ao autômato já construído (para maior conforto no uso do autômato, os estados foram renomeados):



Incorporação do novo comando

- O reconhecedor do comando condicional assim construído é incorporado ao autômato reconhecedor da linguagem, já implementado, e o autômato consolidado resultante assume o aspecto mostrado no próximo slide.
- Os estados do autômato original foram rebatizados apenas por conveniência para o manuseio e utilização do reconhecedor.

Diagrama de estados do Interpretador ampliado

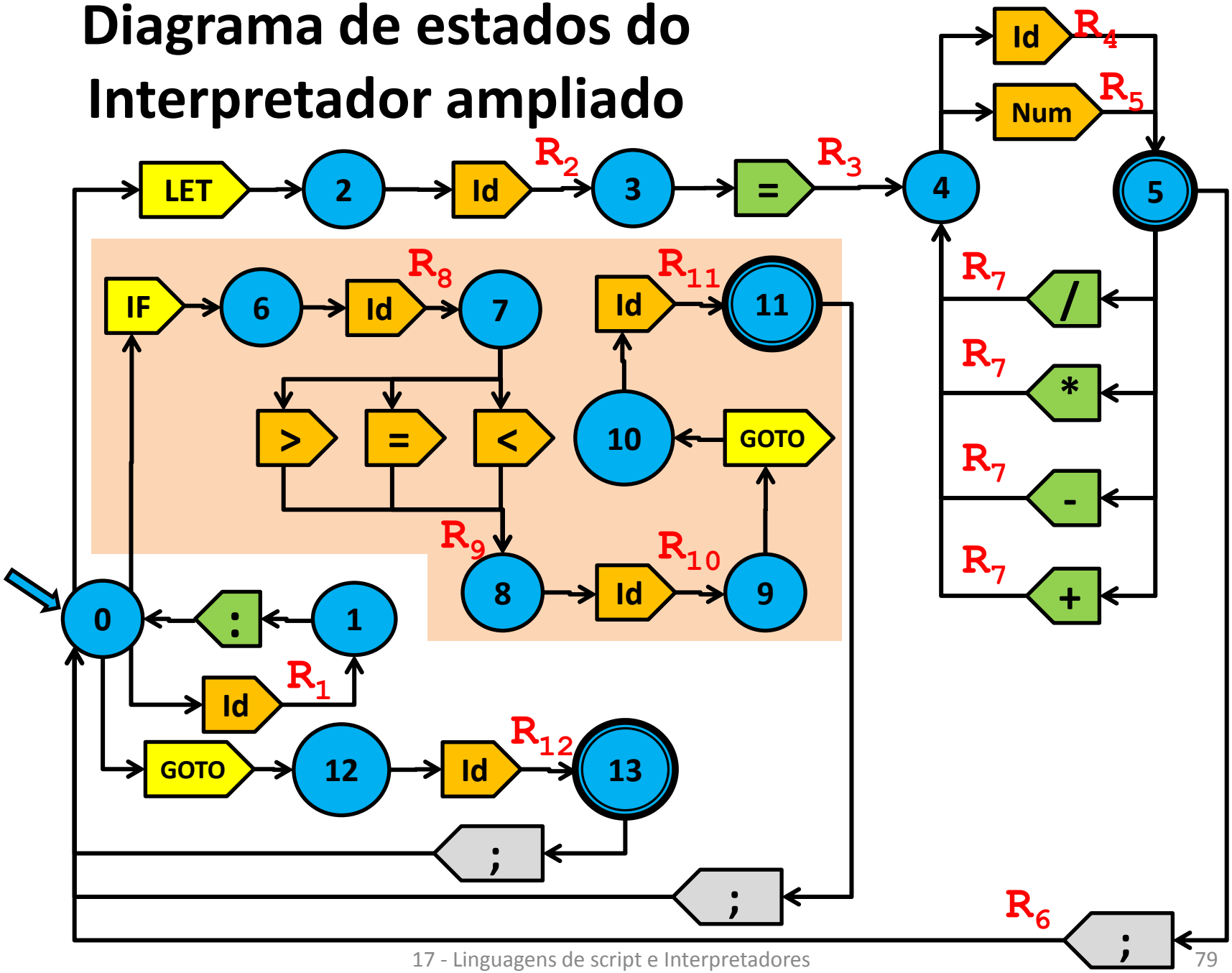


Tabela de transição, ampliada

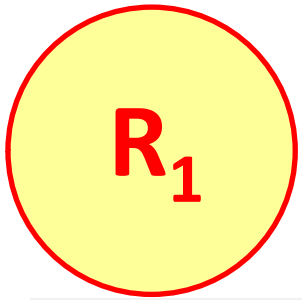
	Id	:	;	LET	Num	+	-	*	/	IF	>	=	<	GOTO
0	1 R_1			2						6				12
1		0												
2	3 R_2													
3												4 R_3		
4	5 R_4				5 R_5									
5			0 R_6			4 R_7	4 R_7	4 R_7	4 R_7					
6	7 R_8													
7											8 R_9	8 R_9	8 R_9	
8	9 R_{10}													
9														10
10	11 R_{11}													
11			0											
12	13 R_{12}													
13			0											

Lista de transições, ampliada

```
>0     {Id 1 R1, LET 2,   IF 6,  GOTO 12}
1      {: 0}
2      {Id 3 R2}
3      {= 4 R3}
4      {Id 5 R4,  Num 5 R5}
5>     {; 0 R6,  + 4 R7,  - 4 R7,  * 4 R7,  / 4 R7}
6      {Id 7 R8}
7      {> 8 R9,  = 8 R9,  < 8 R9}
8      {Id 9 R10}
9      {GOTO 10}
10     {Id 11 R11}
11>    {; 0}
12     {Id 13 R12}
13>    {; 0}
```

Semântica do interpretador final

- Para completar a modificação, é preciso acrescentar rotinas semânticas ao autômato.
- A codificação não está incluída nesta apresentação.
- Encerrando, então, os slides a seguir descrevem sumariamente todas as rotinas semânticas citadas para o autômato apresentado, que agora está praticamente pronto para ser codificado.



(0-1 Id)

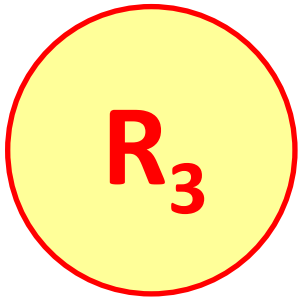
Tratamento da **definição de um rótulo**:

- Se **Id** já estiver na tabela de símbolos definida,
  - Enviar mensagem de erro e terminar
- Caso contrário:
  - Memorizar o rótulo encontrado na tabela de símbolos
  - Marcar como rótulo
  - Associar-lhe o número da linha do script
  - Não alterar o comando corrente a interpretar

R_2

(2-3 id)

- Memorizar, em uma variável **Id_1**, o valor associado ao Identificador encontrado, para que possa receber posteriormente o resultado do cálculo da expressão aritmética.



(3-4 =)

Inicialização do tratamento da expressão aritmética:

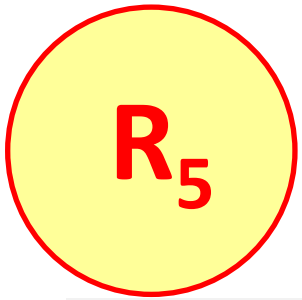
- Iniciar em vazio a pilha de operandos
- Empilhar zero na pilha de operandos
- Iniciar em vazio a pilha de operadores
- Empilhar “+” na pilha de operadores

R_4

(4-5 Id)

Utilizar como operando o **Id** encontrado:

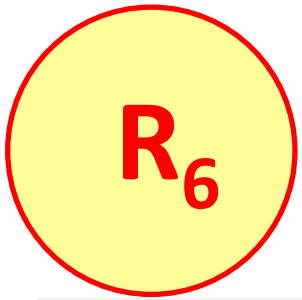
- Empilhar o valor associado ao Identificador na pilha de operandos
- Se o topo da pilha de operadores for um “*” ou um “/”, aplicar o operador memorizado anteriormente sobre os dois elementos do topo da pilha de operandos, desempilhá-los e empilhar na pilha de operandos o resultado da operação



(4-5 Num)

Utilizar como operando o **Num** encontrado:

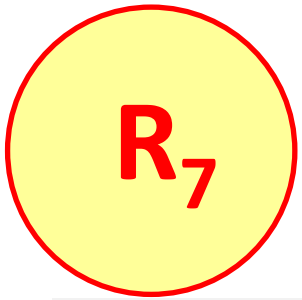
- Empilhar na pilha de operandos o valor associado ao Número inteiro encontrado
- Se o topo da pilha de operadores contiver um "*" ou um "/", aplicar sobre os dois elementos do topo da pilha de operandos, desempilhá-los e empilhar na pilha de operandos o resultado da operação



(5-0 ;)

Resolução das pendências, ao final da expressão:

- Aplicar cada operador empilhado sobre os dois operandos mais recentemente empilhados, desempilhá-los e empilhar o resultado da operação em seu lugar, e repetir o processo até que as pilhas estejam vazias
- Guardar o último resultado na variável-destino da atribuição, memorizada em Id_1



(5-4 operador)

Tratamento dos quatro **operadores aritméticos**:

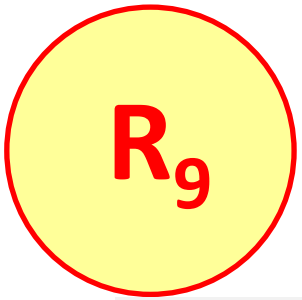
- Se a operação encontrada for multiplicativa, aplicá-la sobre os dois elementos do topo da pilha de operandos, e substituí-los empilhando o resultado da operação.
- Se for aditiva, e a piha de operadores já tiver uma operação empilhada, aplicá-la e empilhar a nova operação encontrada.

R_8

(6-7 Id)

Tratamento da **primeira variável da comparação**:

- Memorizar em $\mathbf{id_1}$, para uso posterior, o identificador encontrado, que representa o primeiro operando da expressão de comparação



(7-8 comparador)

Tratamento do **comparador** encontrado:

- Memorizar em **Opcomp** o operador de comparação encontrado, para aplicação posterior sobre seus operandos

R_{10}

(8-9 Id)

Tratamento da **segunda variável da comparação**:

- Seja $\mathbf{id_2}$ o valor da variável encontrada
- Avaliar a diferença ($\mathbf{id_1 - id_2}$)
- Analisar (tabela abaixo) o valor dessa diferença de acordo com a operação de comparação (**Opcomp**) memorizada anteriormente, para determinar o resultado da comparação, e guardar esse resultado (**TRUE** ou **FALSE**) em **Comp**

	$Id_1 > id_2$	$Id_1 = id_2$	$Id_1 < id_2$
$Id_1 - id_2 > 0$	TRUE	FALSE	FALSE
$Id_1 - id_2 = 0$	FALSE	TRUE	FALSE
$Id_1 - id_2 < 0$	FALSE	FALSE	TRUE

comparação

diferença

(10-11 id)

Tratamento do **rótulo-alvo do desvio** condicional:

- Seja **id_3** o rótulo encontrado
- Se ainda não consta na tabela de símbolos,
  - Inserir e marcar como rótulo indefinido
  - Memorizar o ponto de retorno à execução
  - Buscar a definição do rótulo no texto fonte e defini-lo na tabela
  - Voltar a executar a partir do ponto memorizado.
- Se **Comp=TRUE**, desviar para **id_3**, caso contrário nada faz.

R_{12}

(12-13 id)

Tratamento de **referência a um rótulo** em **GOTO**

- Seja $\mathbf{id_3}$ o rótulo encontrado
- Se $\mathbf{id_3}$ ainda não consta na tabela de símbolos,
  - Inserir e marcar como rótulo indefinido
  - Memorizar o ponto de retorno à execução
  - Buscar a definição do rótulo no texto fonte e defini-lo na tabela
  - Voltar a executar a partir do ponto memorizado.

Exemplo

- Aqui é apresentado um exemplo ampliado parecido com o anteriormente estudado para a linguagem inicial, porém utilizando desvio condicional em lugar do desvio incondicional anteriormente existente.

```
0                  .........
1                  LET A = 10;
2                  LET B = 30;
3          LOOP:   LET A = A+7;
4                  IF A < B GO TO LOOP;
5                  .........
```

1

LET A = 10;

est.	átomo	trat.	efeito
0	LET		
2	Id	R_2	Id = "A" Destino = A
3	=	R_3	inicializa cálculo da expressão
4	Num	R_5	Num = 10 empilha operando
			exec. oper. pendente: (0+10) = 10
5	;	R_6	atribui resultado (10) a Destino (A)
			[Valor (A) = 10]
0			

2

LET B = 30;

est.	átomo	trat.	efeito
0	LET		
2	Id	R_2	Id = “B” Destino=B
3	=	R_3	inicializa cálculo da expressão
4	30	R_5	empilha operando 30
			aplica op. pendente (0+30) = 30
5	;	R_6	atribui 30 a B [Valor(B) = 30]
0			

3

LOOP : LET A = A+7;

est.	átomo	trat.	efeito	
0	Id	R_1	Id = “LOOP”	Define rótulo (linha 3)
1	:	-	-	
0	LET	-	-	
2	Id	R_2	Id=“A”	Destino = A
3	=	R_3	inicializa cálculo da expressão	
4	Id	R_4	Id=“A”	empilha valor (A) = 10
			aplica op. pendente	(0 + 10) = 10
5	+	R_7	empilha operador	“+”
4	Num	R_5	Num=7	empilha 7
			aplica op. pend.	(10 + 7) = 17
5	;	R_6	atribui 17 a A	[Valor(A) = 17]
0				

4

IF A < B GO TO LOOP;

estado	átomo	trat.	efeito
0	IF	-	-
6	Id	R_8	Id_1 = "A" [Valor (A) = 17]
7	<	R_9	Opcomp = "<"
8	Id	R_{10}	Id_2 = "B" Comp = A<B = 17<30 [logo, Comp = TRUE]
9	GOTO	-	-
10	Id	R_{11}	Id3 = "LOOP"; ir para LOOP, pois Comp=TRUE [LOOP está na linha 3]
11	;	-	-
0			

3

LOOP : LET A = A+7;

est.	átomo	trat.	efeito
0	Id	R_1	Id = “LOOP” Define rótulo (linha 3)
1	:	-	-
0	LET	-	-
2	Id	R_2	Id=“A” Destino = A
3	=	R_3	inicializa cálculo da expressão
4	Id	R_4	Id=“A” empilha valor (A) = 17
			aplica op. pendente (0 + 17) = 17
5	+	R_7	empilha operador “+”
4	Num	R_5	Num=7 empilha 7
			aplica op. pend. (17 + 7) = 24
5	;	R_6	atribui 24 a A [Valor (A) = 24]
0			

4

IF A < B GO TO LOOP;

estado	átomo	trat.	efeito
0	IF	-	-
6	Id	R_8	Id_1 = "A" [Valor (A) = 24]
7	<	R_9	Opcomp = "<"
8	Id	R_{10}	Id_2 = "B" Comp = A<B = 24<30 [Comp = TRUE]
9	GOTO	-	-
10	Id	R_{11}	Id3 = "LOOP"; ir p/ LOOP pois Comp=TRUE [LOOP está na linha 3]
11	;	-	-
0			

LOOP : LET A = A+7;

est.	átomo	trat.	efeito	
0	Id	R_1	Id = “LOOP”	Define rótulo (linha 3)
1	:	-	-	
0	LET	-	-	
2	Id	R_2	Id=“A”	Destino = A
3	=	R_3	inicializa cálculo da expressão	
4	Id	R_4	Id=“A”	empilha valor (A) = 24
			aplica op. pendente	(0 + 24) = 24
5	+	R_7	empilha operador	“+”
4	Num	R_5	Num=7	empilha 7
			aplica op. pend.	(24 + 7) = 31
5	;	R_6	atribui 31 a A	[Valor (A) = 31]
0				

4

IF A < B GO TO LOOP;

estado	átomo	trat.	efeito
0	IF	-	-
6	Id	R_8	Id_1 = "A" [Valor (A) = 31]
7	<	R_9	Opcomp = "<"
8	Id	R_{10}	Id_2 = "B" Comp = A<B = 31<30 [Comp = FALSE]
9	GOTO	-	-
10	Id	R_{11}	Id3 = "LOOP"; não ir p/ LOOP pois Comp=FALSE
11	;	-	-
0			

[e portanto o programa deve prosseguir na linha seguinte (linha 5)]

Conclusão

- A aplicação repetida desses procedimentos de interpretação propiciam assim a execução da sequência de comandos que representa o programa que está sendo interpretado.
- Isso é suficiente para dar uma ideia de como é feito o processamento interpretado de uma linguagem de script, ainda que muito desprovida de recursos.
- Em linguagens mais elaboradas, podem surgir problemas mais complexos a serem resolvidos, o que pode requerer procedimentos de interpretação mais sofisticados.
- Uma linguagem histórica de uso prático, muito representativa dessa classe de linguagens com implementação interpretada é a linguagem BASIC, que pode ser citada como um excelente modelo para projetos dessa natureza.

FIM