Instituto de Computação

Bacharelado em Ciência da Computação
Disciplina: Técnicas de Programação Avançada
Trabalho - 2013.2 – versão 1

A linguagem Logo é uma linguagem basicamente gráfica, que foi criada com base nas ideias de Seymour Papert, para que crianças pudessem aprender sobre computadores. O propósito da linguagem é fundamentalmente permitir o desenho de formas geométricas utilizando procedimentos computacionais. Algumas versões da linguagem Logo são bastante sofisticadas incluindo avaliadores de expressões, execução condicional de comandos, *loops* e comandos de entrada e saída, o que a torna uma linguagem completa.

Para interação com a interface gráfica e o desenho das formas, foi criado o conceito de uma "tartaruga" que se move através de comandos **forward** e **back** e que também é capaz de ser rotacionada de um certo ângulo, em torno do seu centro, através dos comandos **left** e **right**. Os comandos **forward**, **back**, **left** e **right** recebem como parâmetro um valor inteiro.

A tartaruga efetua os desenhos na medida em que se desloca deixando rastros retilíneos entre as posições corrente e a nova posição após a execução de um comando de movimentação. Ela também entende comandos extras como: **penup** – faz a tartaruga deslocar-se sem desenhar, **pendown** – faz a tartaruga deslocar-se desenhando, **hideturtle** – torna a tartaruga invisível, **showturtle** – torna a tartaruga visível, **pencolor** - define a cor da caneta associada a tartaruga e **clearscreen** - limpa a tela de desenho.

Além dos comandos básicos é possível efetuar a repetição de um conjunto de comandos agrupados entre chaves através do comando **repeat *n*** {**commands**}, onde **commands** é uma sequência de comandos e ***n*** o número de vezes em que os comandos são repetidos. Também é possível criar uma subrotina simples, associando um nome a um conjunto de comandos através do comando **to *name* {commands}**, onde ***name*** é o nome a ser dado para a subrotina, e **commands** uma lista de comandos no bloco delimitado pelas chaves. Uma vez criada, a subrotina pode ser utilizada como um comando convencional invocando seu nome no código.

A tabela Tabela 1 descreve os principais comandos da primeira versão da linguagem:

Tabela 1- Comandos da linguagem MiniLogo

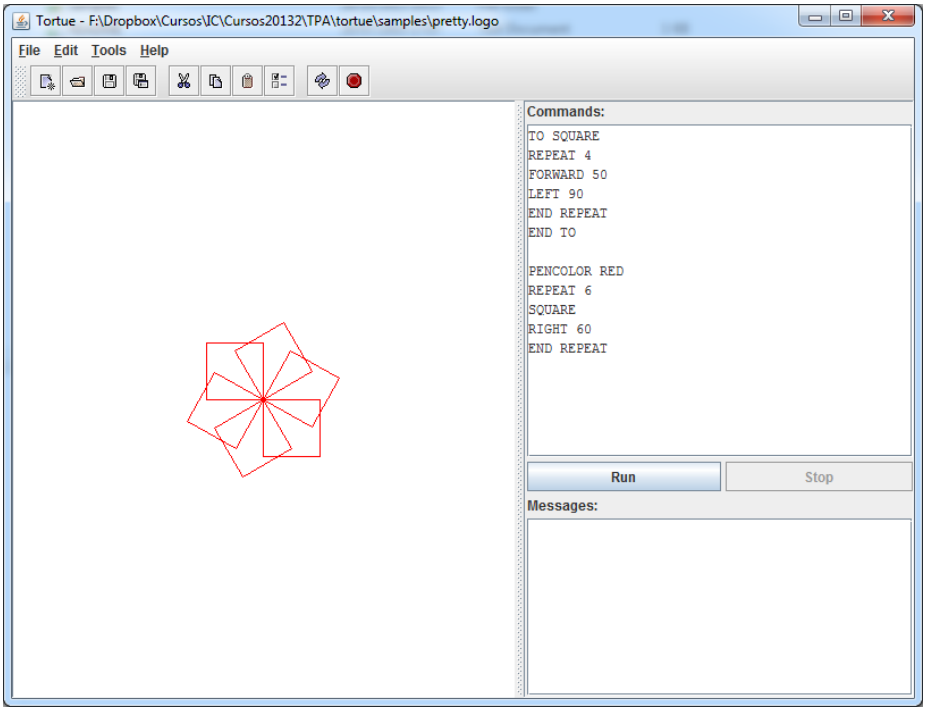
Comando	Ação	Exemplo
forward *n*	Avança a tartaruga de *n* posições	forwards 3;
back *n*	Retrocede a tartaruga de *n* posições	back 20;
left *n*	Rotaciona a tartaruga de *n* graus no sentido anti-horário	left 15;
right *n*	Rotaciona a tartaruga de *n* graus no sentido horário	right 10;
penup	Faz a tartaruga deslocar-se sem desenhar	penup;
pendown	Faz a tartaruga deslocar-se desenhando	pendown;
hideturtle	Esconde a tartaruga	hideturtle;
showturtle	Exibe a tartaruga	showturtle;
pencolor color	Muda a cor da tartaruga para color (color pode ser uma enumeração de cores disponíveis como, por exemplo, RED, GREEN e BLUE.	pencolor 0;
Clearscreen	Limpa a tela	clearscreen;

repeat *n*{commands}	Repete n vezes a lista de comandos dentro do bloco delimitado por abre e fecha chaves	repeat 5{
		  forward 10;
		  right 5;
		  forward 10;
		  left 15;
		};

to *name*{commands}	Define uma subrotina declarada como *name* que executa os comandos especificados entre chaves quando invocada no programa. Rotinas somente podem ser declaradas no início doprograma.	to square{
		  repeat 4{
		  forward 50
		  right 90
		  };
		};

Escreva um sistema que implemente um interpretador para uma primeira versão simplificada da linguagem Logo, contemplando os comandos especificados acima. O sistema deve conter uma janela de visualização para os movimentos e desenhos da tartaruga. Além disso o sistema deve permitir a edição de um pequeno programa na forma textual, salvá-lo e carregá-lo conforme desejado pelo usuário.

A interação entre a interface gráfica, o modelo da tartaruga e o código fonte do programa em Logo deve ser feita segundo o padrão de projeto MVC. Na primeira versão do programa devem ser utilizados pelo menos dois padrões de projeto. Um exemplo de interface para linguagem Logo é a proposta no sistema Tortue (http://tortue.sourceforge.net/)



Procure projetar o sistema prevendo extensões na linguagem como declaração de variáveis de escopo global e possivelmente local, avaliação de expressões, comandos de seleção (if then else) e subrotinas com parâmetros.

Resumo sobre gramáticas, análise léxica e análise sintática

Uma gramática é um conjunto de regras que definem como um programa é bem formado. Uma gramática é composta de símbolos terminais, símbolos não-terminais e regras de derivação, isto é, regras que transforma um não-terminal em uma sequência válida de símbolos.

Eu um interpretador ou compilador associa-se aos lexemas de uma liguagem um valor numérico denominado token. Abaixo está descrita a gramática utilizada no trabalho e o conjunto de tokens.

Gramática da linguagem

program→ *list_ routine_decls list_commands* ´eof´
list_ routine_decls → *routine_delc list_routine_decls*|∈
routine_decl → ´to´ ID ´{´ *list_commands* ´}´ ´;´
list_commands → *command list_commandsA*
list_commandsA → *list_commands* |∈
command → ´forward´ NUM ´;´ |
→ ´backward´ NUM ´;´|
→ ´left´ NUM ´;´|
→ ´right´ NUM ´;´|
→ ´penup´ ´;´|
→ ´pendown´ ´;´|
→ ´showturtle´ ´;´|
→ ´hideturtle´ ´;´|
→ ´pencolor´ NUM ´;´|
→ ´clearscreen´ ´;´|
→ ´repeat´NUM ´{´ *list_commands* ´}´ ´;´ |

tokens = {FORWARD, BACKWARD, LEFT, RIGHT, PENUP, PENDOWN, SHOWTURTLE,
HIDETURTLE, PENCOLOR, CLEARSCREEN, REPEAT, TO, ID,
NUM, LEFT_BRACES,RIGHT_BRACES, SEMI_COMMA};

Análise léxica

É papel do analisador léxico ler a sequência de caracteres que compõem o código de entrada e verificar se eles formam lexemas da linguagem que devem ser retornados na forma de tokens. O analisador sintático recebe a sequência de tokens e verifica se ele está conforme segundo a especificação da gramática.

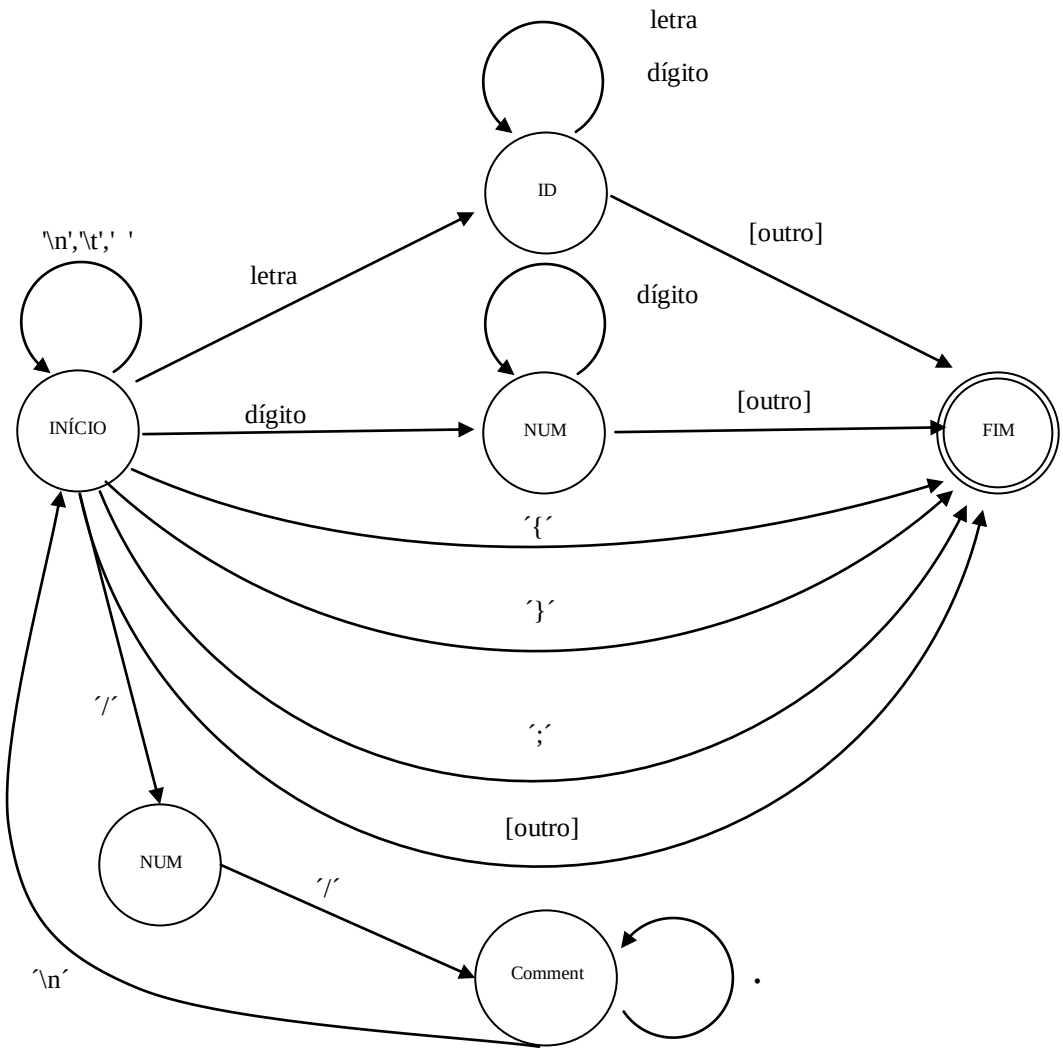
Os lexemas de uma linguagem são normalmente expressos através de expressões regulares que podem ser reconhecidas por autômatos finitos.

Autômato finito determinístico

O autômato finito determinístico é o modelo matemático de uma máquina que reconhece a linguagem regular formada por todas as sequencia de símbolos válidos (lexemas) da linguagem. Um autômato é definido por um conjunto de símbolos que determina seu alfabeto Σ, um conjunto de estados E, um estado inicial $\sigma_0 \in E$, um conjunto de estados finais Ω e regras de transição $T:E\times\Sigma\rightarrow E$, entre um par (estado, símbolo) e um novo estado.

Um autômato no qual, para cada par (estado, símbolo) existe uma única regra de transição é determinado autômato determinístico. Em uma linguagem de programação, identificadores, constantes (e todos os demais lexemas que podem ser manifestar de forma infinita) podem ser mapeados para o mesmo token. Para eliminar tal ambiguidade deve-se associar um token secundário a cada um deles.

Abaixo apresentamos um esboço de autômato finito determinístico para os lexemas da linguagem do trabalho:



Obs.: um ponto (na transição do comentário para si próprio) significa que qualquer caractere é admissível exceto '\n'.

Análise Sintática

A análise sintática é a parte do interpretador que é responsável por verificar a correta disposição dos lexemas na linguagem. Uma linguagem de programação é normalmente descrita através de uma gramática livre de contexto, a qual é reconhecida por um autômato de pilha. A análise sintática, basicamente verifica se existe uma árvore de derivação do programa de acordo com a gramática da linguagem.

Existem duas formas principais de se efetuar uma análise sintática: análise sintática *top-down* e *bottom-up*. Na análise *top-down*, o analisador inicia com o símbolo inicial da gramática, no nosso caso o não-terminal *program*, e procura inferir que regras da gramática podem ser

aplicadas. Na análise *bottom-up*, o analisador acumula símbolos lidos da entrada até que um lado direito de uma regra seja reconhecido. Quando um lado direito é reconhecido, ele cria um novo nó na árvore de derivação correspondente ao lado esquerdo da regra.

Neste trabalho utilizaremos uma análise *top-down*; em particular, utilizaremos um analisador descendente recursivo.

Na análise *top-down*, para decidir qual regra iremos aplicar, com base em um símbolo no lado direito, temos como informação disponível apenas o token corrente. Uma regra será aplicada se o token corrente for o início do lado direito da regra. Para isto definimos o conceito de iniciador *First*(α) de uma forma sentencial α, isto é uma cadeia arbitrária de símbolos, como

$$First(\alpha) = \{a \in \Sigma, S \Rightarrow^* a\beta\},$$

onde S é o símbolo inicial da gramática, Σ o conjunto símbolos (alfabeto), e $\Rightarrow$* uma derivação composta, isto é, uma ou mais aplicação de derivações.

Observe que, se um não-terminal específico é o lado esquerdo de mais de uma regra com símbolos iniciadores em comum, então não conseguiremos decidir qual regra aplicar. Impõe-se então a primeira restrição para os *First*(α), denominada 1ª restrição LL(1):

$$A \to \alpha \mid \beta = First(\alpha) \cap First(\beta) = \varnothing$$

Gramáticas que não satisfazem tal restrição não podem ser analisadas por analisadores sintáticos LL(1), isto é, analisadores que reproduzem em seu reconhecimento uma derivação mais a esquerda da entrada, lendo a entrada a partir da esquerda, consultando um símbolo por vez (left, left, one, isto é, LL(1)).

Os iniciadores de uma forma sentencial pode ser definidos de forma recursiva conforme abaixo:

$$Fist(a\alpha) = \{a\}$$
$$First(A\alpha) = \begin{cases} First(A)\}, \neg(A \Rightarrow \varepsilon) \\ Fist(A) \cup Fist(\alpha), A \Rightarrow \varepsilon \end{cases}$$
$$First(A) = \bigcup First(\alpha_i), A \to \alpha_i$$

Quando para um dado símbolo não-terminal existe um regra que leva em $\in$, isto é, uma regra vazia, uma situação nova surge, já que tal regra não possui iniciadores. Então tal regra será aplicada se o símbolo corrente for o iniciador do próximo símbolo, isto é, o símbolo que ocorre após o símbolo corrente. Chamamos os iniciadores do próximo símbolo de um símbolo corrente A de *Follow*(*A*):

$$A \Rightarrow *, Follow(A) = \{a \in \Sigma, S \Rightarrow^* \alpha A a \beta\}$$

A existência de regras que levam em $\in$, impõem uma nova restrição, a 2ª restrição LL(1):

$$Fist(A) \cap Follow(A) = \varnothing$$

Temos que tomar muito cuidado na definição da gramática, pois se a 2ª restrição LL(1) não for atendida, então o analisador LL(1) não terá como decidir se deve aplicar uma regra que leva em $\in$ou uma outra regra, para um símbolo que admite os dois tipos de regras.

Da mesma forma que para os símbolos iniciadores, podemos definir os terminadores de modo recursivo, conforme abaixo:

$$\forall B \to \alpha A \beta \in P \begin{cases} Follow(A) \supset First(\beta), \neg\beta \Rightarrow \varepsilon \\ Follow(A) \supset First(\beta) \cup Follow(B), \beta \Rightarrow \varepsilon \end{cases}$$

Seguem abaixo os iniciadores e terminadores de todos os símbolos não-terminais da gramática considerada no trabalho:

Firsts:

first(*program*) = first(*list_routine_decls*) $\cup$ first(*list_commands*) = {´to´,´forward´, ´backward´, ´left´, ´right´, ´penup, ´pendown´, ´showturtle´, ´hideturtle´, ´pencolor´,´clearscreen´,´repeat´}

first(*list_routine_decls*) = first(*routine_decls*) = {to}

first(*routine_decls*) = {to}

first(*list_commands*) = first(*command*) = {´forward´, ´backward´, ´left´, ´right´, ´penup´, ´pendown´, ´showturtle´, ´hideturtle´, ´pencolor´, ´clearscreen´,´repeat´}

first(*list_commandsA*) = first(*list_commands*) = {´forward´, ´backward´, ´left´, ´right´, ´penup´, ´pendown´, ´showturtle´, ´hideturtle´, ´pencolor´, ´clearscreen´,´repeat´}

first(*command*) = {´forward´, ´backward´, ´left´, ´right´, ´penup´, ´pendown´, ´showturtle´, ´hideturtle´, ´pencolor´, ´clearscreen´,´repeat´}

Follows:

follow(*program*) = $\varnothing$
follow(*list_routine_decls*) = first(*list_commands*) = first(*command*) = {´forward´, ´backward´, ´left´, ´right´, ´penup´, ´pendown´,´showturtle´, ´hideturtle´, ´pencolor´, ´clearscreen´,´repeat´};

follow(*list_routine_declsA*) = follow(*list_routine_decls*) = {´forward´, ´backward´, ´left´,
´right´, ´penup´,
´pendown´,´showturtle´,
´hideturtle´, ´pencolor´,
´clearscreen´,´repeat´};

follow(*routine_decls*) = {to}

follow(*list_commands*) = first(´eof´)∪first{´}´} = {´eof´,´}´}
follow(*list_commandsA*) = follow(*list_commands*) = {´eof´, ´}´}
follow(*command*) = first(*list_commandsA*) ∪ follow(*list_commands*) =
{ ´forward´, ´backward´, ´left´,
´right´, penup´,´pendown´,´showturtle´,
´hideturtle´, ´pencolor´, ´clearscreen´
´repeat´, ´to´,´eof´,´, ´}´}

Com base nos iniciadores e terminadores podemos definir como aplicar as regras de transição na forma de uma tabela que será utilizada pelo autômato de pilha.

Tokens	*program*	*list_commands*	*list_commandsA*	*command*
´forward´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´forward´ NUM ´;´
´backward´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´backward´ NUM ´;´
´left´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´left´ NUM ´;´
´right´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´right´ NUM ´;´
´penup´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´penup´ ´;´
´pendown´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´pendown´ ´;´
´showturtle´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´showturtle´ ´;´
´hideturtle´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´hideturtle´ ´;´
´pencolor´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´pencolor´ NUM ´;´
´clearscreen´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´clearscreen´
´repeat´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´repeat´ NUM ´{´ *list_commands* ´}´ ´;´
´to´	*list_commands* ´eof´	*command list_commandsA*	*list_commands*	´to´ ID ´{´ *list_commands* ´}´ ´;´
´{´				
´}´			∈	
´;´				
´eof´			∈	
ID				
NUM				

Tokens	*list_routine_decls*	*routine_decls*		
´forward´	∈			
´backward´	∈			
´left´	∈			
´right´	∈			
´penup´	∈			
´pendown´	∈			
´showturtle´	∈			
´hideturtle´	∈			
´pencolor´	∈			
´clearscreen´	∈			
´repeat´	∈			
´to´	*routine_decls*	to´ ID ´{´ *list_commands* ´}´ ´;´		
´{´				
´}´				
´;´				
´eof´				
ID				
NUM				

Algoritmo correspondente ao automato de pilha que reconhece programas da gramática dada:

Algoritmo: AnalisadorSintáticoTopDown(subst,S)

---

subst: tabela de substituições
S: símbolo inicial

```
push( S );
a = nextToken();
do {
        x = pop();
        if ( IS_TERMINAL(x) ) {
                if ( x == a ) {
                        a = nextToken();
                } else {
                        syntaxError();
                        break;
                }
        } else {
                a = subst[x][a];
                if ( IS_NULL(a) ) {
                        syntaxError();
                        break;
                } else {
                        push( a );
                }
        }
} while( x != ‘eof’ );
```

É possível transformar o analisador sintático baseado em uma pilha em um código recursivo, onde para cada não terminal existe um procedimento que o interpreta (ver código fornecido em anexo).

Exemplo do algoritmo aplicado a análise sintática do programa abaixo:

```
 to square{
   repeat 4{
      forward 50;
      right 90;
   };
};
square;
```

TO	TO	TO	TO	TO	ID = ´square´	LEFT_BRACES	REPEAT
				´to´			
				ID	ID		
				´{´	´{´	´{´	
				list_commands	*list_commands*	*list_commands*	*list_commands*
				´}´	´}´	´}´	´}´
	list_routine_decls		*command*	´;´	´;´	´;´	´;´
	list_commands	*list_commands*	*list_commandsA*	*list_commands*	*list_commands*	*list_commands*	*list_commands*
program	´eof´	´eof´	´eof´	´eof´	´eof´	´eof´	´eof´

REPEAT	REPEAT	NUM = ´4´	RIGHT_BRACES	FORWARD	FORWARD	FORWARD	NUM =´50´
	´repeat´					´forward´	
	NUM	NUM				NUM	NUM
	´{´	´{´	´{´		´command´	´;´	´;´
	list_commands	*list_commands*	*list_commands*	*list_commands*	*list_commandsA*	*list_commandsA*	*list_commandsA*
	´}´	´}´	´}´	´}´	´}´	´}´	´}´
´command´	´;´	´;´	´;´	´;´	´;´	´;´	´;´
list_commandsA	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*
´}´	´}´	´}´	´}´	´}´	´}´	´}´	´}´
´;´	´;´	´;´	´;´	´;´	´;´	´;´	´;´
list_commands	*list_commands*	*list_commands*	*list_commands*	*list_commands*	*list_commands*	*list_commands*	*list_commands*
´eof´	´eof´	´eof´	´eof´	´eof´	´eof´	´eof´	´eof´

SEMI_COMMA	RIGHT	RIGHT	RIGHT	RIGHT	NUM = ´90´	SEMI_COMMA	RIGHT_BRACES
				´right´			
				NUM	NUM		
´;´			´command´	´;´	´;´	´;´	
list_commandsA	*list_commandsA*	*list_commands*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*
´}´	´}´	´}´	´}´	´}´	´}´	´}´	´}´
´;´	´;´	´;´	´;´	´;´	´;´	´;´	´;´
list_commandsA	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*
´}´	´}´	´}´	´}´	´}´	´}´	´}´	´}´
´;´	´;´	´;´	´;´	´;´	´;´	´;´	´;´
list_commandsA	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*
´eof´	´eof´	´eof´	´eof´	´eof´	´eof´	´eof´	´eof´

RIGHT_BRACES	SEMI_COMMA	RIGHT_BRACES	RIGHT_BRACES	SEMI_COMMA	ID = 'square'	ID = 'square'	ID = 'square'

´}´							
´;´	´;´						
list_commandsA	*list_commandsA*	*list_commandsA*					
´}´	´}´	´}´	´}´				
´;´	´;´	´;´	´;´	´;´			´command´
list_commandsA	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commandsA*	*list_commands*	*list_commandA*
´eof´	´eof´	´eof´	´eof´	´eof´	´eof´	´eof´	´eof´

ID = 'square'	SEMI_COMMA	´eof´	´eof´				
'ÍD'							
';'	´;´						
list_commandsA	*list_commandsA*	*list_commandsA*					
´eof´	´eof´	´eof´	´eof´				