Construção de Compiladores

Prof. Dr. Daniel Lucrédio
DC - Departamento de Computação
UFSCar - Universidade Federal de São Carlos

Tópico 04 - Análise Sintática Descendente

Referências bibliográficas

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compiladores: Princípios, Técnicas e Ferramentas (2a. edição). Pearson, 2008.

Kenneth C. Louden. Compiladores: Princípios E Práticas (1a. edição). Cengage Learning, 2004.

Terence Parr. The Definitive Antlr 4 Reference (2a. edição). Pragmatic Bookshelf, 2013.

Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: the power of dynamic analysis. In Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14). Association for Computing Machinery, New York, NY, USA, 579–598. DOI:https://doi.org/10.1145/2660193.2660202

Análise sintática … recordando

- Vimos duas formas de reconhecer uma linguagem através de uma gramática
  - Inferência recursiva
  - Derivação

- Ex: Gramática para expressões aritméticas
  V = {E,I}
  T = {+,*,(,),a,b,0,1}
  P = conjunto de regras ao lado
  S = E

```
E → I
  | E + E
  | E * E
  | (E)
I → a
  | b
  | Ia
  | Ib
  | I0
  | I1
```

Análise sintática … recordando

- Inferência recursiva
  - Dada uma cadeia (conjunto de símbolos terminais)
  - Vamos do corpo para a cabeça

```
E → I
  | E + E
  | E * E
  | (E)
I → a
  | b
  | Ia
  | Ib
  | I0
  | I1
```

Ex: a*(a+b00)

a*(a+b00) ⇐ a*(a+I00) ⇐ a*(a+I0) ⇐
a*(a+I) ⇐ a*(a+E) ⇐ a*(I+E) ⇐ a*(E+E) ⇐
a*(E) ⇐ a*E ⇐ I*E ⇐ E*E ⇐ E

Análise sintática … recordando

- Derivação
  - Dada uma cadeia (conjunto de símbolos terminais)
  - Vamos da cabeça para o corpo

```
E →  I
   |  E + E
   |  E * E
   |  (E)
I →  a
   |  b
   |  Ia
   |  Ib
   |  I0
   |  I1
```

Ex: a*(a+b00)

E ⇒ E*E ⇒ I*E ⇒ a*E ⇒ a*(E) ⇒ a*(E+E) ⇒
a*(I+E) ⇒ a*(a+E) ⇒ a*(a+I) ⇒ a*(a+I0) ⇒
a*(a+I00) ⇒ a*(a+b00)

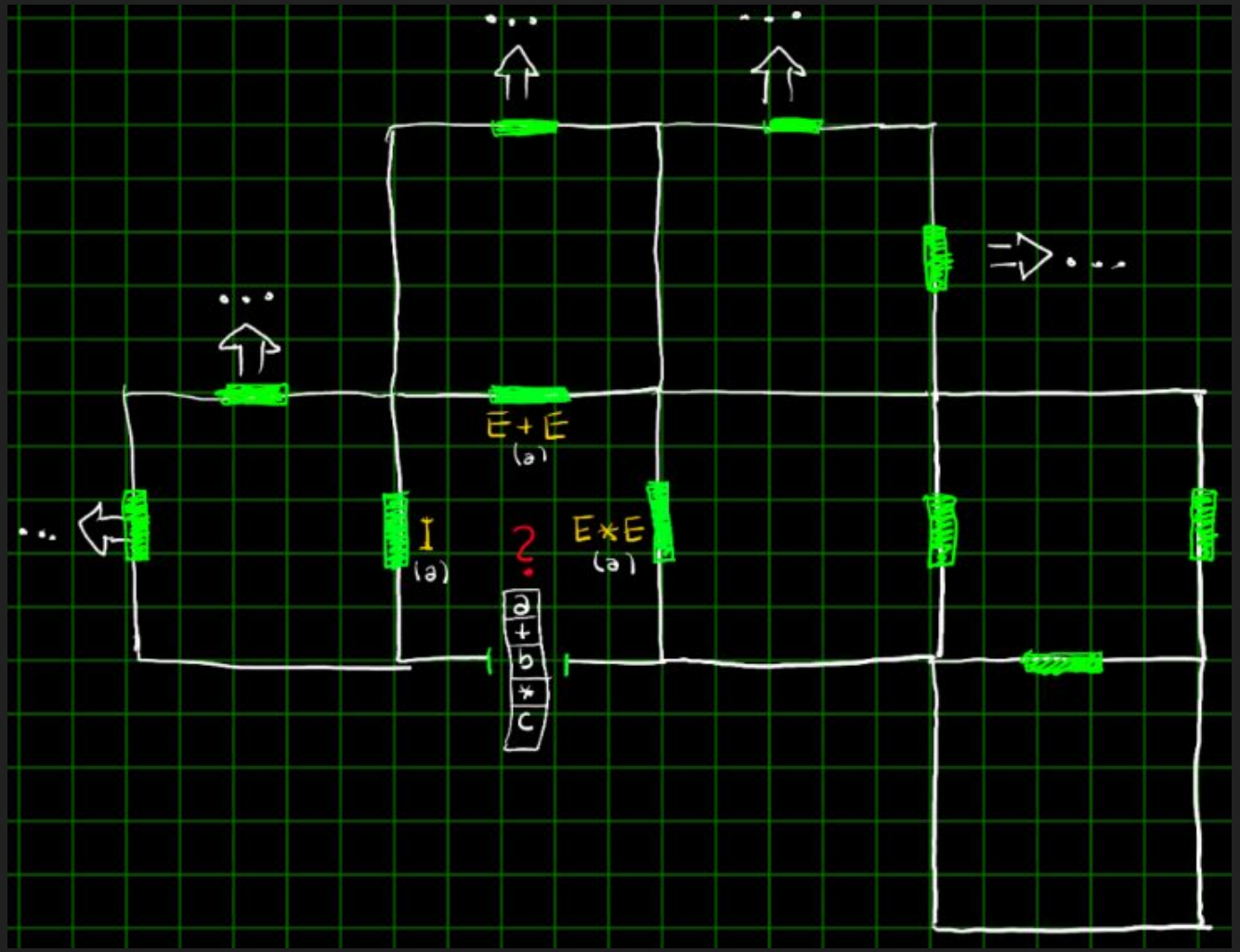
Análise sintática descendente

- Produz uma derivação mais à esquerda da cadeia
  - Os tokens são lidos da esquerda para a direita
  - Em cada passo,
    - o problema é determinar qual produção aplicar

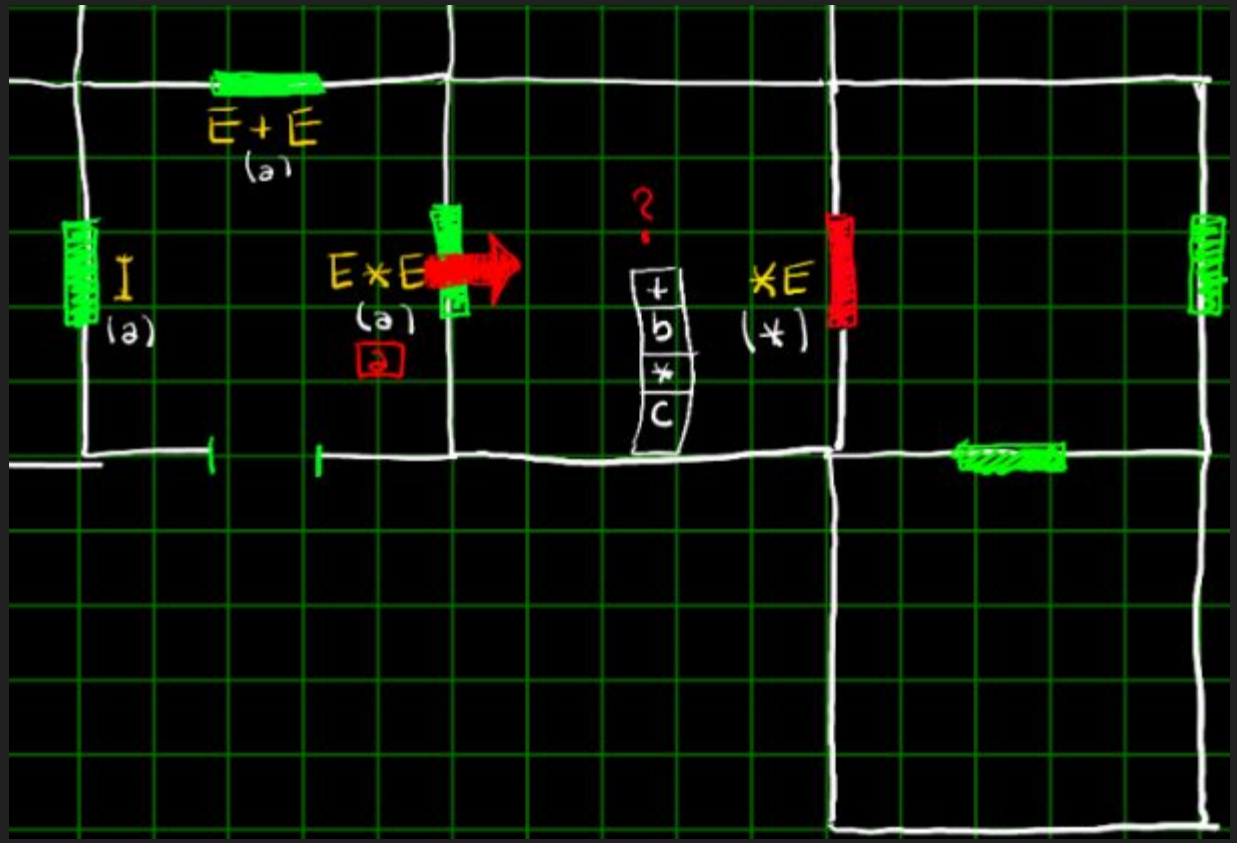
Ex: Entrada: a + b * c

- Token atual = a
- Símbolo inicial: E
- Possíveis produções de E:
  - I
  - E + E
  - E * E

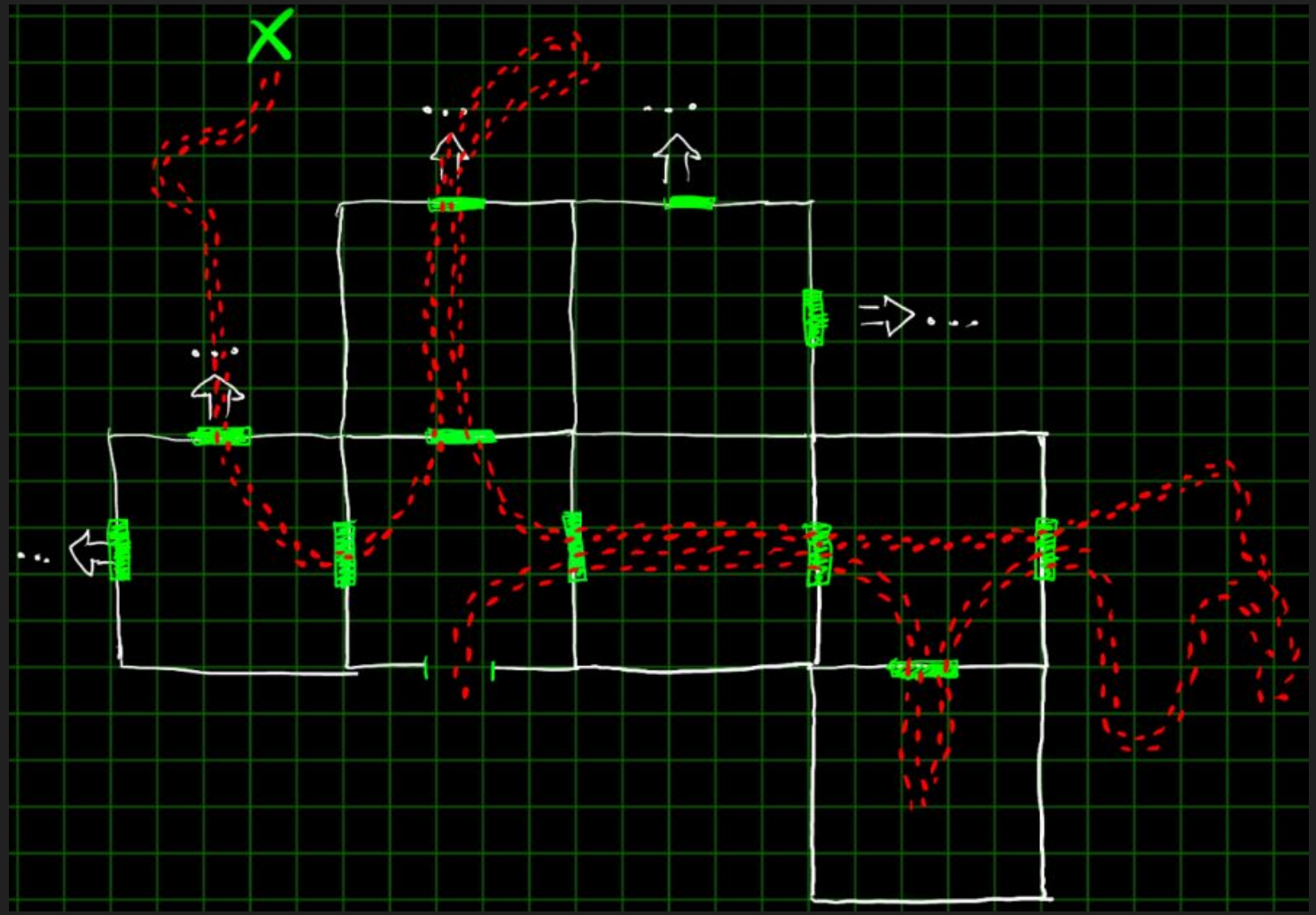

```
E →  I
   |  E + E
   |  E * E
   |  (E)
I →  a | b | c
```



- Minha próxima “chave” é o “a”
- Existem 3 portas que abrem com um “a”
- não-determinismo
- Vamos escolher uma!



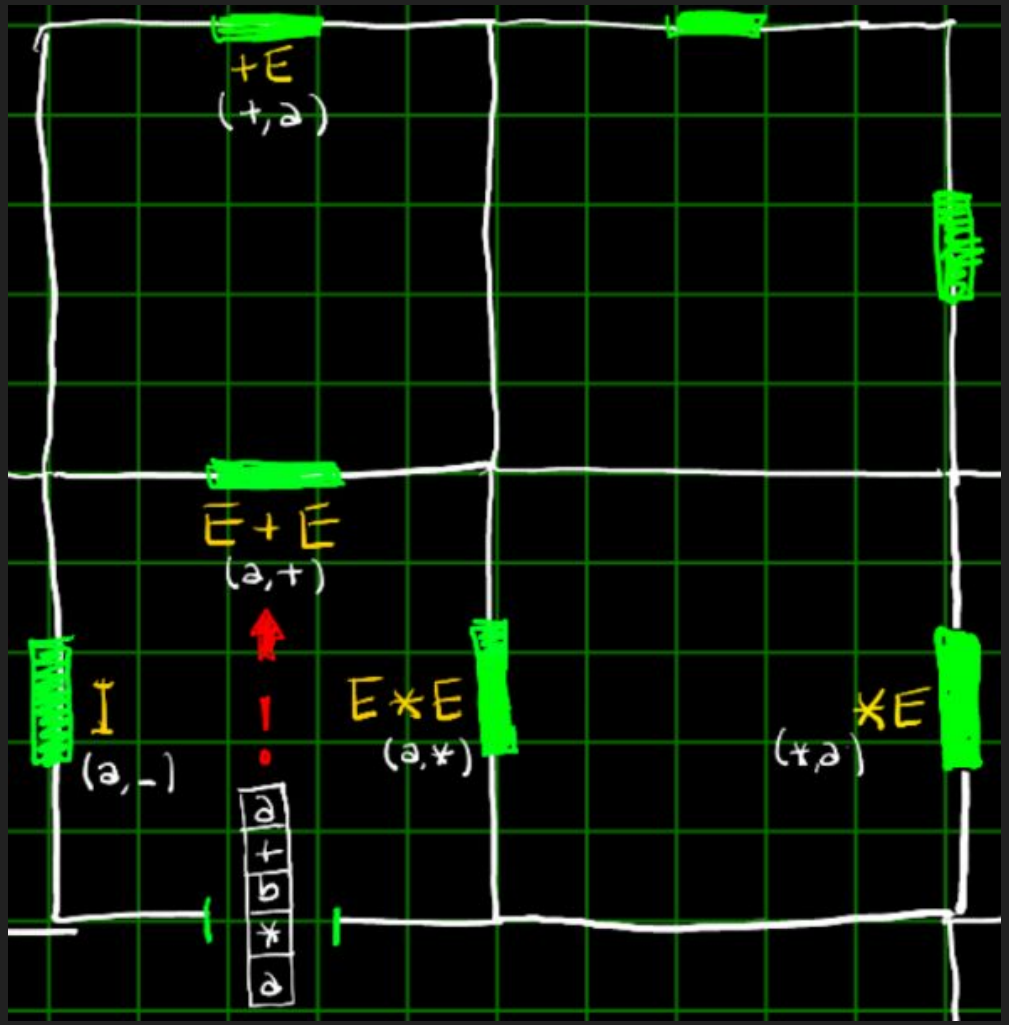
- Deixei o “a” na sala anterior
- Minha próxima “chave” é o “+”
- Só há uma porta, e ela abre com o “*”
- Beco sem saída!



- Abordagem com retrocesso (tentativa e erro)
- Tempo exponencial

Análise sintática descendente

- Outra opção é “adivinhar” a porta correta
  - Abordagem preditiva
- Como fazer isso?
  - Tentamos prever a porta correta com base na informação disponível “ao redor”
  - É uma “tentativa e erro” limitada
  - Tentamos uma das portas até um certo limite
- Na prática
  - É comum olhar apenas uma sala à frente



- Minha próxima “chave” é o “a”
  - A próxima depois desta é o “+”
- Existem 3 portas que abrem com um “a”
  - Mas apenas uma tem um “+” depois
- Determinismo!

Abordagem preditiva

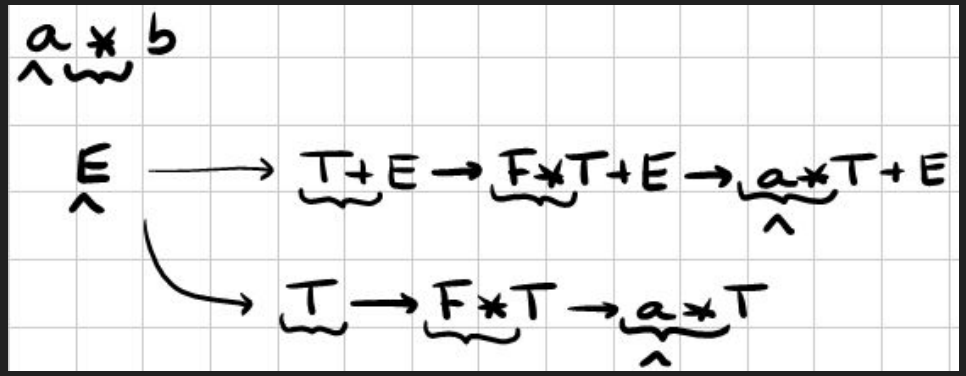
- Na construção do labirinto, tentamos “marcar” as portas com mais símbolos (K símbolos à frente)
- Dependendo das características do labirinto, sua execução pode ser 100% preditiva com um K determinado
  - Mas pode ser que não (mais sobre isso depois)

Exemplo preditivo

```
E → T + E | T
T → F * T | F
F → a | b | (E)

K = 2
```

- Entrada: a * b



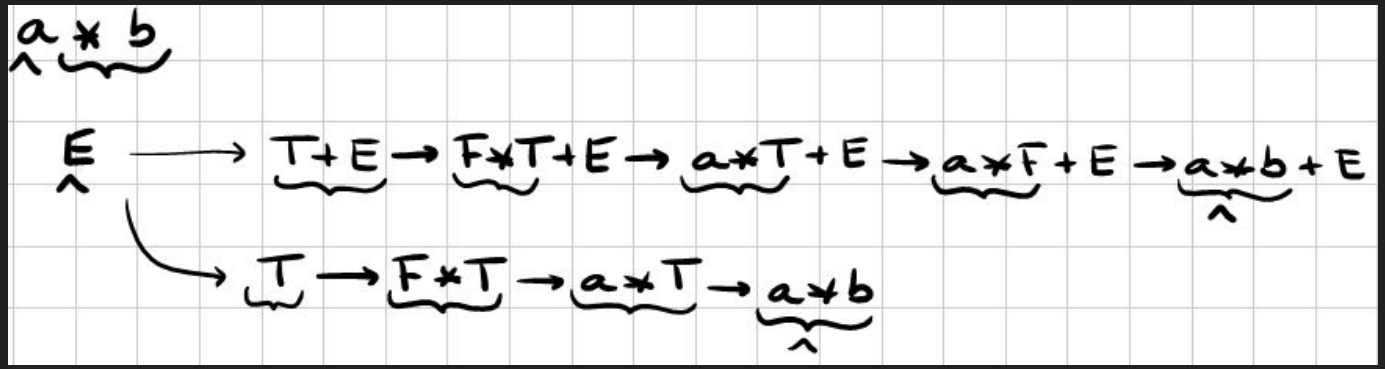
Olhando apenas dois símbolos à frente não é possível resolver o não-determinismo em E

Exemplo preditivo

```
E → T + E | T
T → F * T | F
F → a | b | (E)

K = 3
```

- Entrada: a * b



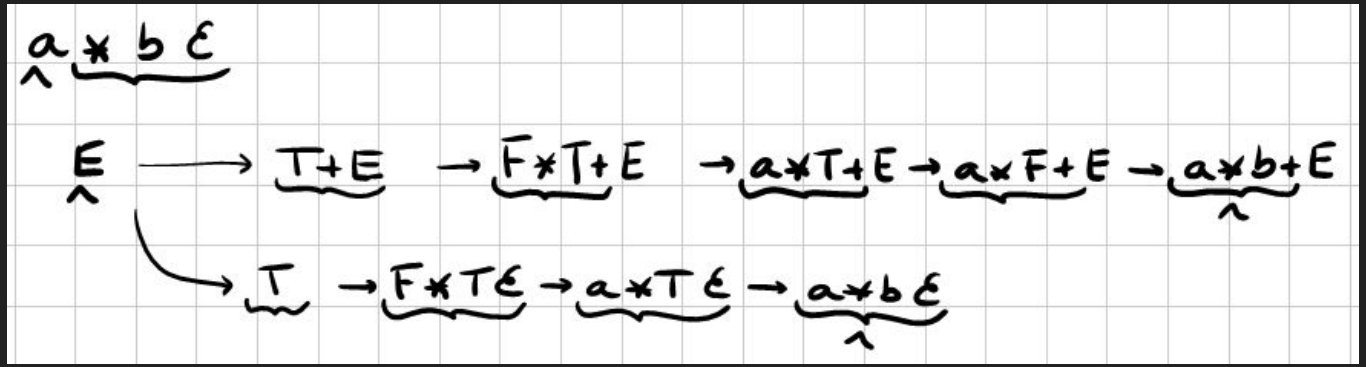
Olhando apenas três símbolos à frente não é possível resolver o não-determinismo em E

Exemplo preditivo

```
E → T + E | T
T → F * T | F
F → a | b | (E)

K = 4
```

- Entrada: a * b



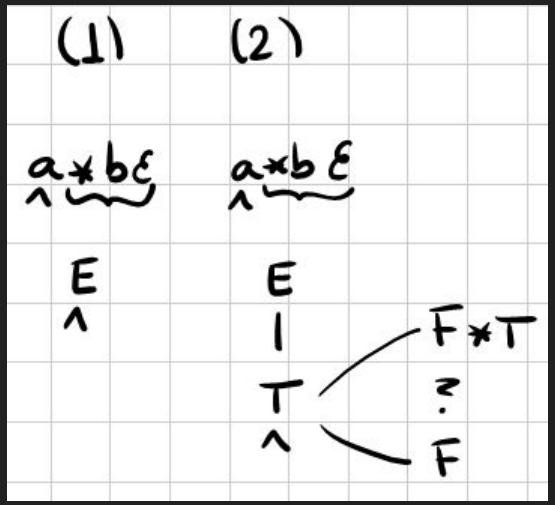
Olhando quatro símbolos à frente é possível resolver o não-determinismo em E (escolhendo E→T)

Exemplo preditivo

```
E → T + E | T
T → F * T | F
F → a | b | (E)

K = 4
```

- Entrada: a * b



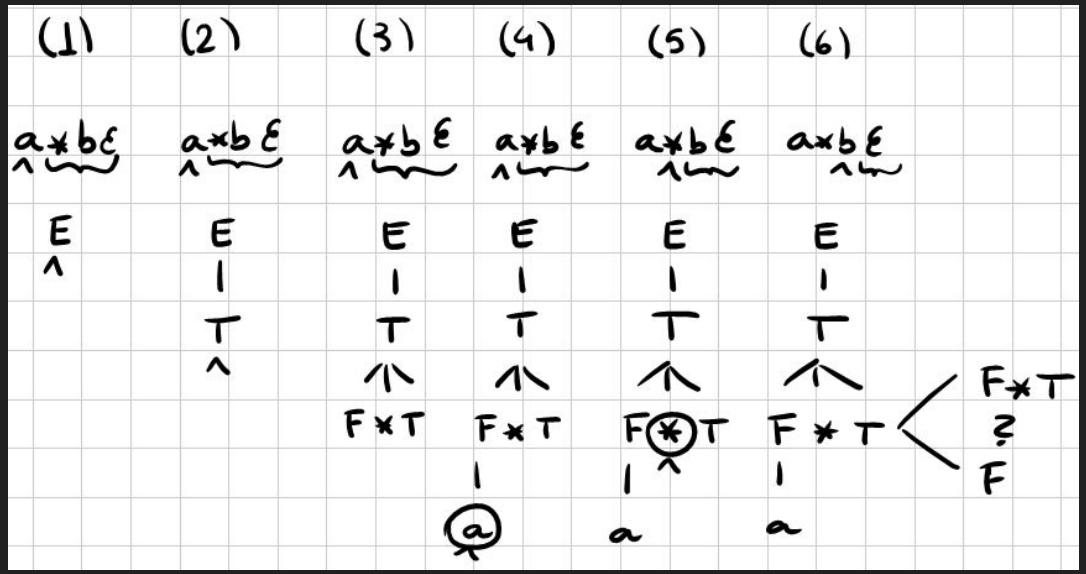
Olhando quatro símbolos à frente é possível resolver o não-determinismo em T (escolhendo T → F * T)

Exemplo preditivo

```
E → T + E | T
T → F * T | F
F → a | b | (E)

K = 4
```

- Entrada: a * b



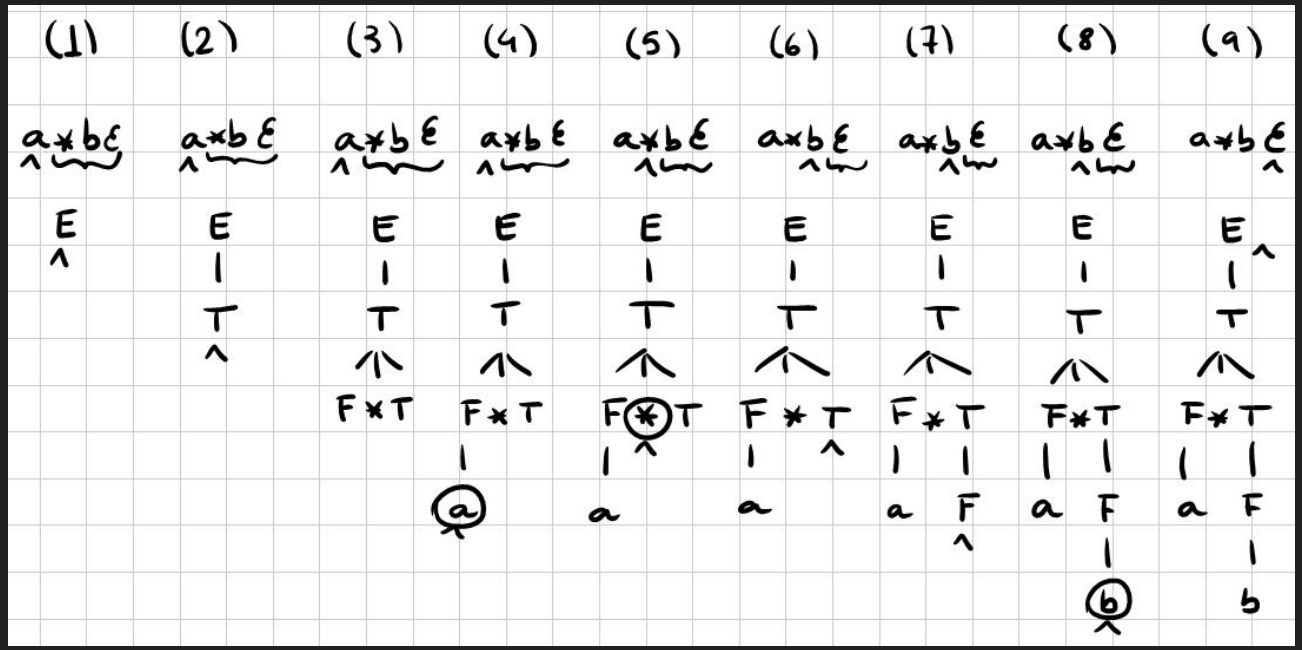
Olhando quatro símbolos à frente é possível resolver o novo não-determinismo em T (escolhendo T → F)

Exemplo preditivo

```
E → T + E | T
T → F * T | F
F → a | b | (E)

K = 4
```

- Entrada: a * b



Cadeia reconhecida

Análise sintática descendente

- A abordagem preditiva é mais eficiente
  - Ela não precisa tentar todos os caminhos
  - Portanto, o tempo de execução não é exponencial
    - Como a força bruta da abordagem com retrocesso
  - Porém, ela é mais cautelosa/preguiçosa ...

Análise sintática descendente

- Não é de todo labirinto que ela encontra a saída
  - Apenas alguns labirintos “especiais”
  - Com propriedades interessantes
    - Como pontos de não-determinismo limitados a 1, 2 ou k salas adjacentes
- Deve existir uma garantia de que naquele labirinto:
  - Para TODA sala com mais de uma porta com a mesma palavra …
  - ... é SEMPRE possível determinar que a escolha da porta foi correta olhando no máximo k salas à frente

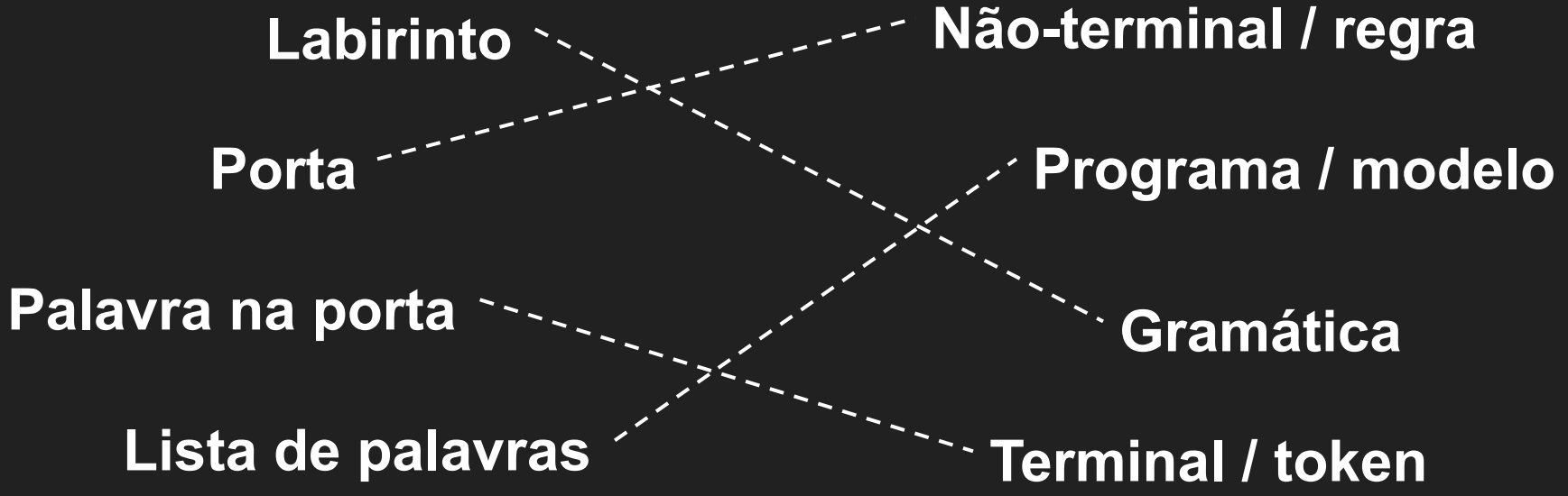
Análise sintática descendente

- Mas chega de labirintos, portas, palavras e listas
- Vamos fazer as associações:

Labirinto	**Não-terminal / regra**
Porta	**Programa / modelo**
Palavra na porta	**Gramática**
Lista de palavras	**Terminal / token**

Análise sintática descendente

- Mas chega de labirintos, portas, palavras e listas
- Vamos fazer as associações:



Análise sintática descendente

Veremos algumas técnicas de implementação deste tipo de analisador sintático

Preditivo de descida recursiva – LL(1)

Preditivo de descida não-recursiva (usando pilha) – LL(1)

Técnica do macaco treinado – LL(*)

Adaptive LL(*) - All-Star

Gramáticas LL

Conjuntos primeiros e seguidores

Conjuntos gerados com base em funções associadas a uma gramática

Ajudam na construção de analisadores descendentes e ascendentes

Ajudam a escolher qual produção aplicar

Conjuntos primeiros

α é uma cadeia de símbolos gramaticais (terminais e não-terminais)

primeiros(**α**) =

conjunto de terminais que começam as cadeias derivadas de **α**

Conjuntos primeiros

Gramática
S → A B
A → a A | a | d
B → b B | c | A | Cd
C → x | y | ε
D → ε

- primeiros(abcd) = { a }
- primeiros(ABC) = { a,d }
- primeiros(AxCd) = { a,d }
- primeiros(BzSB) = { b,c,a,d,x,y }
- primeiros(CzSB) = { x,y,z }
- primeiros(SABC) = { a,d }
- primeiros(C) = { x,y,ε }
- primeiros(DAB) = { a,d }
- primeiros(DCe) = { x,y,e }
- primeiros(DC) = { x,y,ε }

Conjuntos seguidores

A é um não-terminal

`seguidores(A)`

- É o conjunto de terminais que podem aparecer imediatamente após A em alguma forma sentencial
- Em outras palavras, é o conjunto de terminais a tal que existe uma derivação na forma $S \overset{*}{\Rightarrow} \alpha A a \beta$
  - $\{a \mid S \overset{*}{\Rightarrow} \alpha A a \beta\}$

Conjuntos seguidores

- Símbolo especial $ (fim de cadeia)
  - Se A pode ser o símbolo mais à direita em alguma forma sentencial, $ está em seguidores(A)

Conjuntos seguidores

Gramática
S → A B
A → a A | a | d
B → b B | c | A | Cd
C → x | y | ε
D → ε

- seguidores(S) = { $ }
- seguidores(A) = { b,c,a,d,x,y,$ }
- seguidores(B) = { $ }
- seguidores(C) = { d }
- seguidores(D) = {} (não existem! Na verdade D é inalcançável a partir de S)

Gramáticas LL(k)

- LL(k) = classe de gramáticas / analisador sintático
  - Left-to-right = da esquerda para a direita
  - Leftmost derivation = derivação mais à esquerda
  - k = # símbolos à frente para previsão correta
- k > 1 geralmente acarreta em baixa eficiência
  - Por isso, estudaremos o caso LL(1)
    - LL(1) é bastante rica para a maioria das construções de linguagens de programação

Gramáticas LL(1)

- Uma gramática G é LL(1) sse para duas produções distintas $A \rightarrow \alpha \mid \beta$:
  - α e β não derivam cadeias começando com o mesmo terminal
  - No máximo um dentre α e β deriva a cadeia vazia
  - Se $\beta \overset{*}{\Rightarrow} \varepsilon$, então α não deriva nenhuma cadeia começando com um terminal em seguidores(A)
    - O correspondente vale para α

Gramáticas LL(1)

primeiros(α) e primeiros(β) são disjuntos

- Uma gramática G é LL(1) sse para duas produções distintas $A \rightarrow \alpha \mid \beta$:
  - α e β não derivam cadeias começando com o mesmo terminal
  - No máximo um dentre α e β deriva a cadeia vazia
  - Se $\beta \Rightarrow^* \varepsilon$, então α não deriva nenhuma cadeia começando com um terminal em seguidores(A)
    - O correspondente vale para α

Gramáticas LL(1)

- Ex: a gramática a seguir não é LL(1)

```
declaracao → if-decl | 'outra'
if-decl → 'if' '(' exp ')' declaracao else-parte
else-parte → 'else' declaracao | ε
exp → '0' | '1'
comando → declaracao | if-decl
```

primeiros(declaracao)={'outra','if'}

primeiros(if-decl)={'if'}

Gramáticas LL(1)

- Ex: a gramática a seguir não é LL(1)

```
ExprRel → TermoRel ExprRel2
ExprRel2 → 'OU' FatorRel ExprRel2 | ε
FatorRel → '(' ExprRel ')' | Expr '<' Expr
Expr → Termo Expr2
Expr2 → '+' Termo Expr2 | ε
Termo → Fator Termo2
Termo2 → '*' Fator Termo2 | ε
Fator → '(' Expr ')' | id
```

```
primeiros('(' ExprRel ')')={'('}
primeiros(Expr '<' Expr)={id, '('}
```

Gramáticas LL(1)

- Uma gramática G é LL(1) sse para todas as produções distintas $A \rightarrow \alpha \mid \beta$:
  - α e β não derivam cadeias começando com o mesmo terminal
  - No máximo um dentre α e β deriva a cadeia vazia
  - Se $\beta \Rightarrow^* \varepsilon$, então α não deriva nenhuma cadeia começando com um terminal em seguidores(A)
    - O correspondente vale para α

Quer dizer que se ε está em `primeiros(β)`, então `primeiros(α)` e `seguidores(A)` são disjuntos

Gramáticas LL(1)

- Ex: a gramática a seguir não é LL(1)

```
listaComandos → comando ';' listaComandos | '{' listaComandos '}'
comando → if-decl | 'outra' | ε
if-decl → 'if' '(' exp ')' then-parte
then-parte → comandoThen | listaComandos
comandoThen → 'soma' | ε
exp → '0' | '1'
```

primeiros(comandoThen)={'soma',ε}

primeiros(listaComandos)={'if','outra',';','{'}

seguidores(then-parte)={';'}

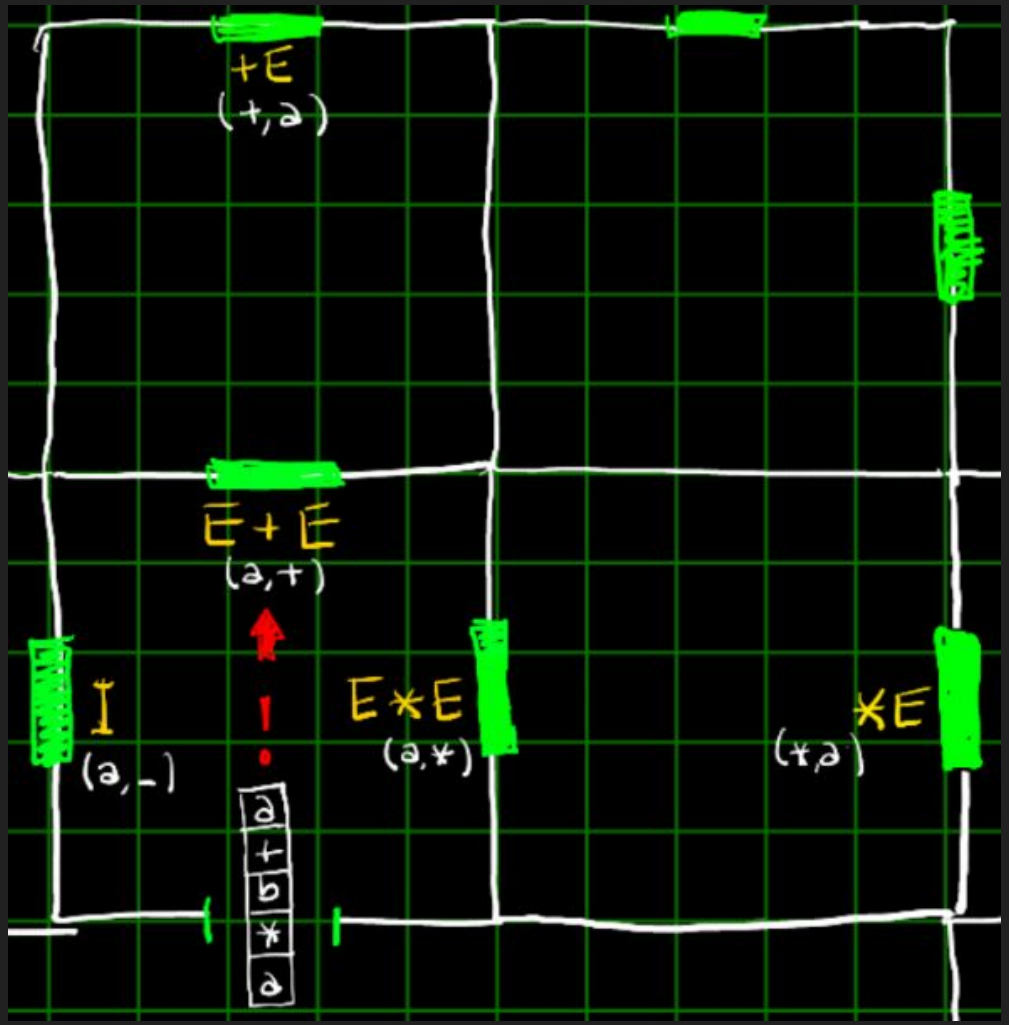
Gramáticas LL(1)

- Estas condições garantem ser possível escolher a produção apropriada olhando apenas o símbolo atual

- Ex:

```
cmd → ‘if’ ‘(‘ expr ‘)’ cmd else cmd
        | ‘while’ ‘(‘ expr ‘)’ cmd
        | ‘{‘ listaCmd ‘}’
```

- Se o símbolo atual for ‘if’, ‘while’ ou ‘{‘ é possível decidir exatamente qual alternativa usar



- Minha próxima “chave” é o “a”
  - A próxima depois desta é o “+”
- Existem 3 portas que abrem com um “a”
  - Mas apenas uma tem um “+” depois
- Determinismo
  - Mas somente com k=2
- Esse labirinto não é LL(1)

Tabela LL(1)

- Tabela de predição
  - Array bidimensional M[A,a]
  - Linha corresponde à produção atual
  - Coluna corresponde ao próximo terminal

	Terminais e $
Não-terminais	

Produção a ser escolhida

Tabela LL(1)

- Ideia geral

  - A produção A → α é escolhida se o próximo símbolo de entrada “a” estiver em `primeiros(α)`

  - Se α = ε ou α *⇒ ε, escolhemos A → α se
    - o próximo símbolo de entrada “a” estiver em `seguidores(A)`
    - se $ foi alcançado e $ está em `seguidores(A)`

Tabela LL(1)

- Objetivo
  - Colocar na tabela as possibilidades de escolha de produção, dados:
    - Um símbolo da entrada (terminal)
      - que representa o próximo símbolo a ser lido
    - A produção (não-terminal) sendo expandida
      - que representa qual símbolo precisa ser substituído no processo de derivação

Tabela LL(1)

- Algoritmo
  1. Para cada terminal “a” em `primeiros(α)`
     - Adicione A → α em M[A,a]
  2. Se ε está em `primeiros(α)` então, para cada terminal “b” em `seguidores(A)`
     - Adicione A → α em M[A,b]
  3. Se ε está em `primeiros(α)` e $ está em `seguidores(A)`
     - Adicione A → α em M[A,$]

- Células vazias correspondem a erro sintático

Exemplo Tabela LL(1)

- E → TE’
- E’ → +TE’ | ε
- T → FT’
- T’ → *FT’ | ε
- F → (E) | id

```
primeiros(E)={(,id}
primeiros(T)={(,id}
primeiros(F)={(,id}
primeiros(“(“)={(}
primeiros(id)={id}
primeiros(E’)={+,ε}
primeiros(+)={+}
primeiros(T’)={*,ε}
primeiros(*)={*}
primeiros(TE’)={(,id}
primeiros(+TE’)={+}
primeiros(FT’)={(,id}
primeiros(*FT’)={*}
primeiros(“(“E”)”)={(}
primeiros(E’T)={+,(,id}
primeiros(T’E’)={*,+,ε}
```

```
seguidores(E)={$,)}
seguidores(E’)={$,)}
seguidores(T)={+,$,)}
seguidores(T’)={+,$,)}
seguidores(F)={*,+,$,)}
```

Exemplo Tabela LL(1)

	+	*	(	)	id	$
E			E→TE’		E→TE’	
E’	E’→+TE’			E’→ε		E’→ε
T			T→FT’		T→FT’	
T’	T’→ε	T’→*FT’		T’→ε		T’→ε
F			F→(E)		F→id	

Exercício Tabela LL(1)

- Construa a tabela LL(1) da gramática

```
S  → iEtSS' | a
S' → eS | ε
E  → b
```

```
primeiros(iEtSS')={i}
primeiros(a)={a}
primeiros(eS)={e}
primeiros(b)={b}

seguidores(S)={$,e}
seguidores(S')={$,e}
seguidores(E)={t}
```

Tabela LL(1)

- Resposta

	i	t	e	a	b	$
S	S→iEtSS’			S→a		
S’			S’→eS S’→ε			S’→ε
E					E→b	

Não-determinismo causado pela ambiguidade da gramática

Tabela LL(1)

- O algoritmo anterior serve, portanto, para detectar se uma gramática é LL(1)
  - Células com múltiplos valores são indícios de:
    - Ambiguidade
    - Necessidade de fatoração
    - Recursão à esquerda

- Pode ajudar a transformar uma gramática em LL(1)
  - Mas existem gramáticas que nunca podem ser transformadas em LL(1)
  - Gramática do exercício anterior é um exemplo

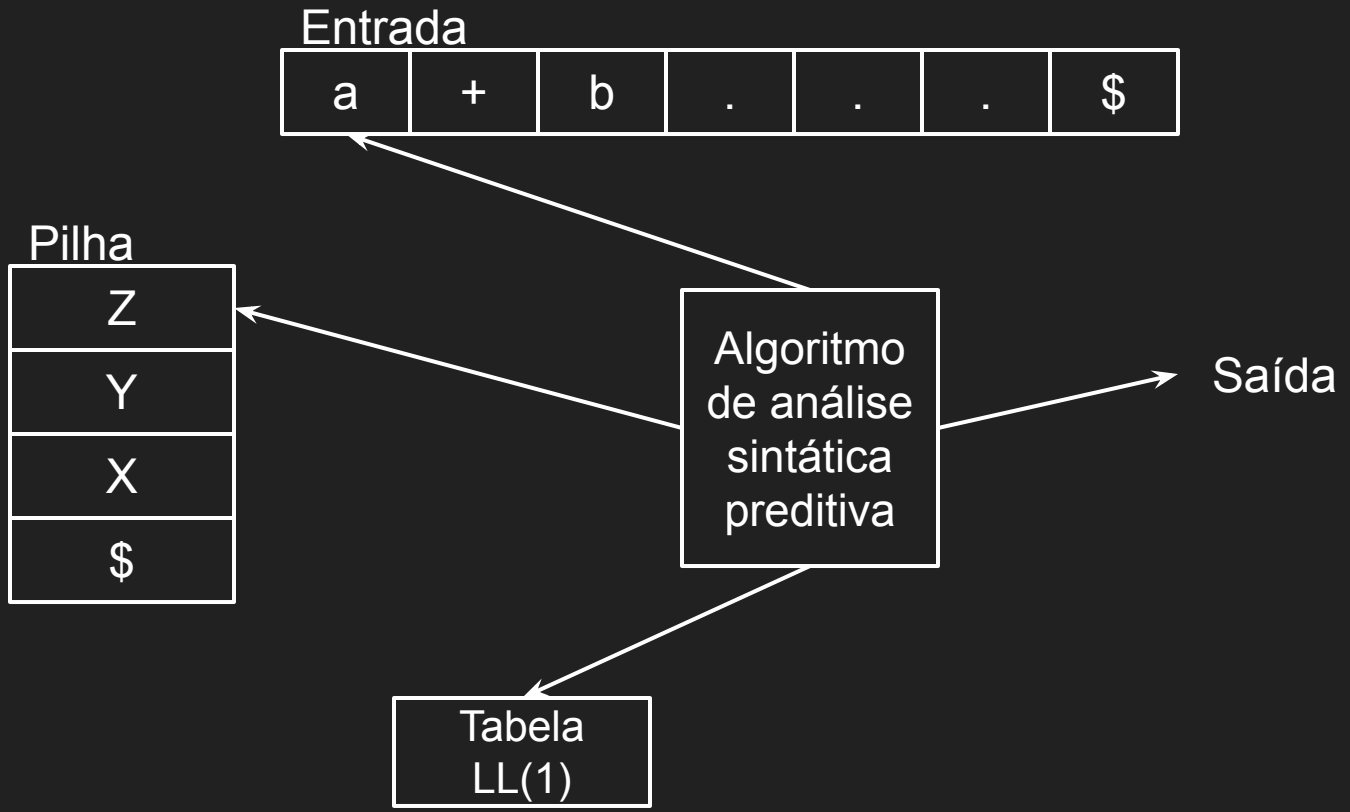
Lembrando um pouco de LFA

- Estamos lidando com gramáticas livres de contexto
  - Gramáticas LL(1), para ser mais específico

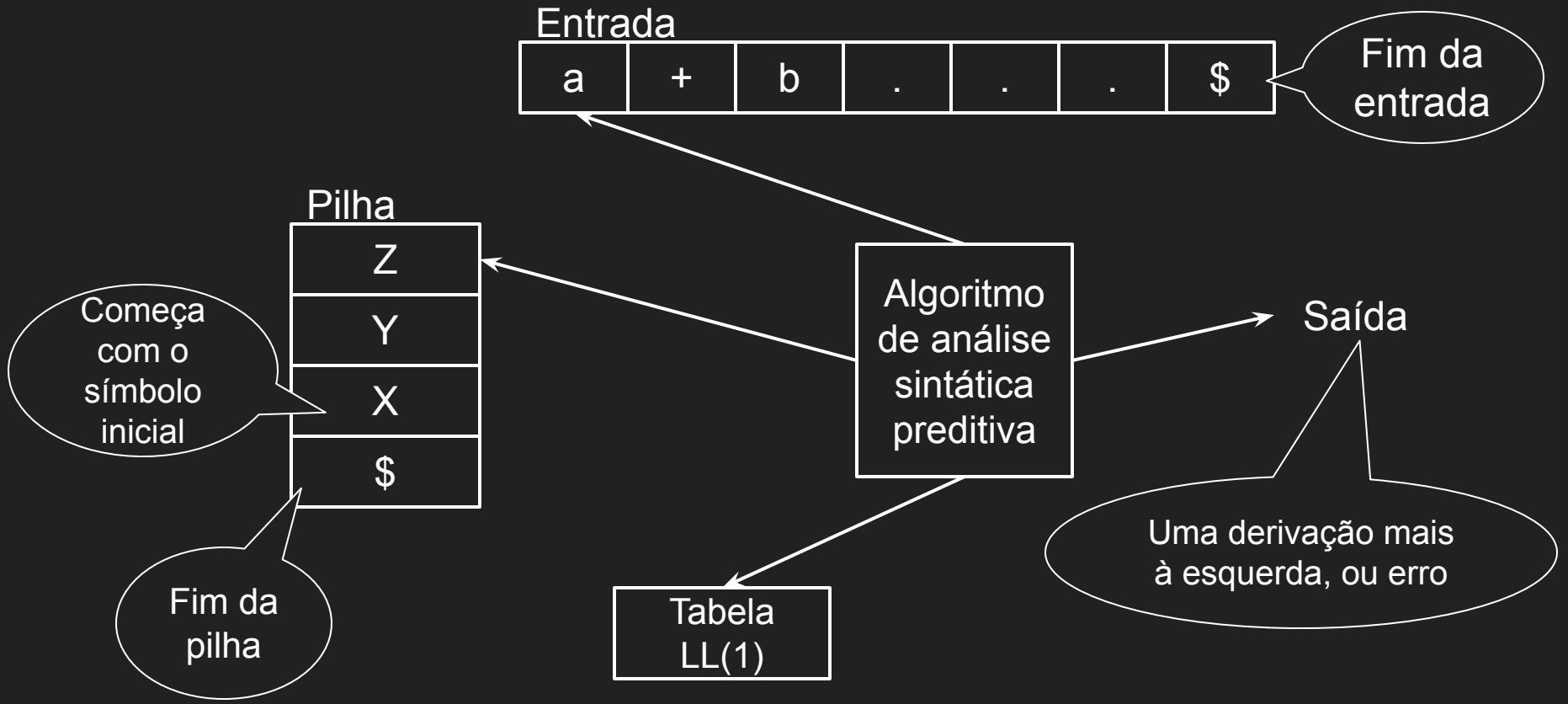
- A máquina capaz de processar este tipo de linguagem é um PDA (ou autômato com pilha)

- Portanto, vamos usar um PDA para implementar um analisador sintático LL

Análise preditiva sem recursão



Análise preditiva sem recursão



Algoritmo de análise sintática preditiva

Condições iniciais:

entrada = w$

símbolo S no topo da pilha, sobre $

Algoritmo:

```
1.    ip = primeiro símbolo de w
2.    X = topo da pilha // inicialmente, X = S
3.    enquanto(X != $) { // pilha não vazia
4.       a = w[ip]
5.       se(X == a) desempilhar e avançar ip
6.       senão se(X é terminal) erro
7.       senão se(M[X,a] é vazio) erro
8.       senão se(M[X,a] = X→Y1Y2...Yk {
9.          imprimir "X→Y1Y2...Yk"
10.         desempilhar
11.         empilhar Yk,Yk-1,...,Y1 // nessa ordem
12.      }
13.      X = topo da pilha
14.   }
```

Análise sintática preditiva sem recursão

- Exercício
  - Entrada = id + id * id $

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

	+	*	(	)	id	$
E			E→TE’		E→TE’	
E’	E’→+TE’			E’→ε		E’→ε
T			T→FT’		T→FT’	
T’	T’→ε	T’→*FT’		T’→ε		T’→ε
F			F→(E)		F→id	

Exercício

Casamento	Pilha	Entrada	Ação
	E$	**id**+id*id$	E→TE′
	TE′$	**id**+id*id$	T→FT′
	FT′E′$	**id**+id*id$	F→id
	idT′E′$	**id**+id*id$	match
id	**T′**E′$	**+**id*id$	T′→ε
	E′$	**+**id*id$	E′→+TE′
	+TE′$	**+**id*id$	match
id**+**	**T**E′$	**id***id$	T→FT′
	FT′E′$	**id***id$	F→id
	idT′E′$	**id***id$	match
id+**id**	**T′**E′$	***** id$	T′→*FT′
	***** FT′E′$	***** id$	match
id+id****	**F**T′E′$	**id**$	F→id
	idT′E′$	**id**$	match
id+id***id**	**T′**E′$	**$**	T′→ε
	E′$	**$**	E′→ε
	$	**$**	OK

LL(k) e LL(*)

LL(1)

- A grande vantagem do algoritmo LL(1) é a sua facilidade no entendimento/depuração!
  - É mais fácil seguir a execução em um processo de derivação (quando comparado com a inferência)
- A grande desvantagem são os não-determinismos
  - Ambiguidade
  - Fatoração à esquerda

LL(k)

- Estendendo o algoritmo LL(1) para LL(k), onde k > 1
  - Ao invés de `primeiros` e `seguidores`, teríamos $\texttt{primeiros}_k$ e $\texttt{seguidores}_k$
  - Teríamos uma LL(k), construída exatamente da mesma maneira
- A diferença é que, ao tentar prever qual regra usar, é preciso olhar (e comparar) k símbolos adiante

LL(k)

- Exemplo:

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

Essa gramática não é LL(1), mas é LL(2)

É possível gerar o seguinte analisador preditivo de descendência recursiva

```
void stat() {
  if ( LA(1)==ID&&LA(2)==EQUALS ) { // PREDICT
    match(ID);                      // MATCH
    match(EQUALS);
    expr();
  }
  else if ( LA(1)==ID&&LA(2)==COLON ) { // PREDICT
    match(ID);                          // MATCH
    match(COLON);
    stat();
  }
  else «error»;
}
```

LL(k)

- Mas, na prática k>1 não é muito interessante
  - A tabela seria (exponencialmente) grande, com muitas colunas para cobrir todas as combinações de k símbolos à frente
  - A tabela precisaria considerar diferentes contextos para os conjuntos seguidores
  - Muitas vezes, se uma gramática não é LL(1), ela provavelmente não é LL(k) também
    - Ex: recursividade à esquerda independe de k

LL(k)

- Outro exemplo de uma gramática que não é LL(k)
  - Nenhum k fixo pode resolver o problema de decidir qual regra usar

```
method
    : type ID '(' args ')' ';'           // E.g., "int f(int x,int y);"
    | type ID '(' args ')' '{' body '}' // E.g., "int f(int z) {...}"
    ;
type: 'void' | 'int' ;
args: arg (',' arg)* ; // E.g., "int x, int y, int z, ..."
arg : 'int' ID ;
body: ... ;
```

  - Solução (menos legível, ruim para adicionar semântica)

```
method
    :   type ID '(' args ')' (';' | '{' body '}')
    ;
```

LL(k)

- Em resumo, LL(1) era o melhor que podíamos fazer de forma automática
- A alternativa é construir o analisador “na mão” ou partir para a análise LR
  - Que veremos a seguir
- Mas analisadores LR são mais complexos de implementar/utilizar em outros aspectos
  - Apesar de as gramáticas ficarem mais legíveis
  - E construir à mão é trabalhoso
- Felizmente, existe uma técnica chamada LL(*), que une o melhor dos mundos LL e LR
  - Foi desenvolvida há alguns anos, e não consta nos livros clássicos de compiladores (Dragão, Louden)

LL(*)

- Outro exemplo

```
def : modifier* classDef     // E.g., public class T {...}
    | modifier* interfaceDef // E.g., interface U {...}
    ;
```

- Podemos fatorar à esquerda (ruim)

```
def : modifiers* (classDef|interfaceDef) ;
```

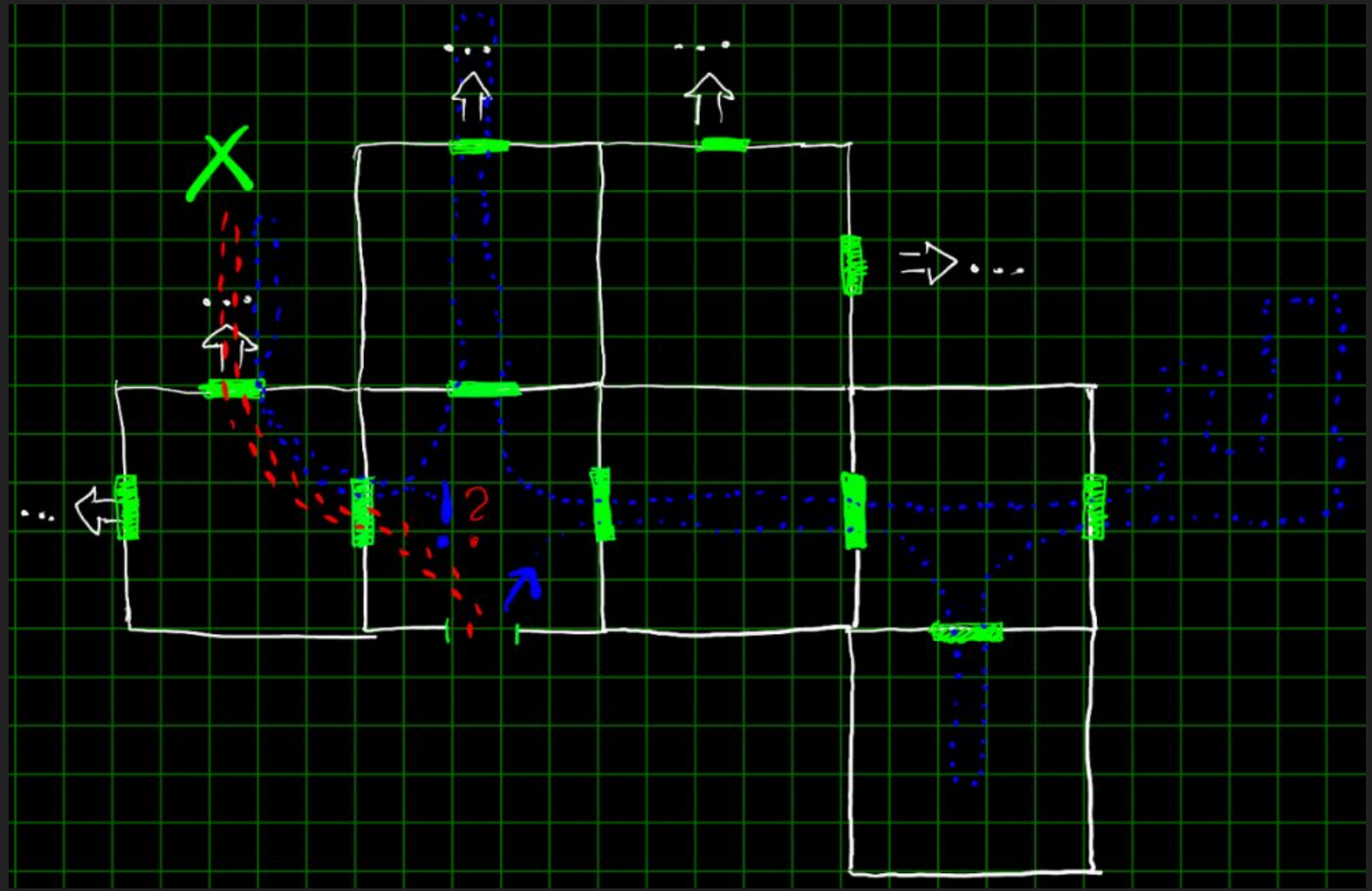
- Outra opção: método de busca feito à mão (findAhead)

```
def : {findAhead(CLASS_TOKEN)}?     modifier* classDef
    | {findAhead(INTERFACE_TOKEN)}? modifier* interfaceDef
```

LL(*)

- O método findAhead serve apenas para a predição
  - Ele inspeciona símbolos à frente, em busca de um determinado símbolo de decisão
  - Portanto, é um método leve e eficiente
- O número de símbolos inspecionado é variável
  - k=*, por isso LL(*)

Voltando à analogia do labirinto: É como se pudéssemos mandar um macaco treinado na frente, para encontrar o caminho rapidamente


X
2

LL(*)

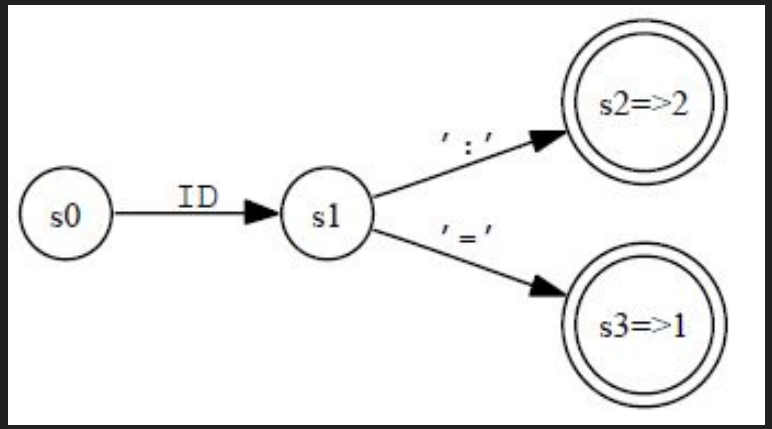
A técnica LL(*) consiste essencialmente em gerar automaticamente o método lookAhead com base na definição da gramática

LL(*)

- Voltando ao exemplo

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

- Nesse caso, a decisão consiste em encontrar um ponto de divergência entre as duas regras
  - Normalmente é um único token que fica alguns símbolos à frente
  - Podemos ver como um DFA!



LL(*)

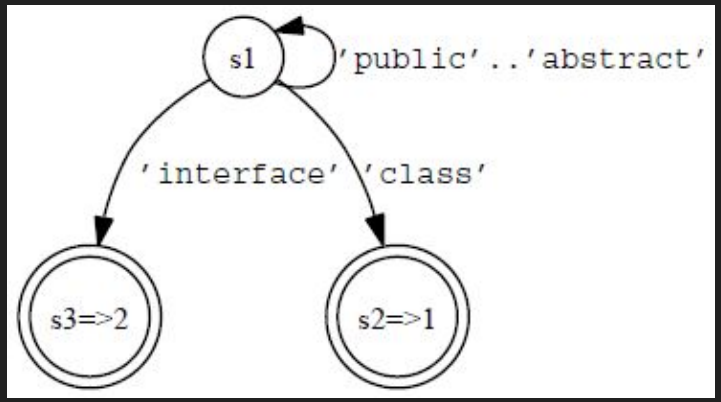
- Faz sentido então pensarmos em um DFA de decisão
  - Um modelo mais simples, não chega a ser uma gramática livre de contexto
  - De fato, é uma gramática regular
  - Serve apenas para tomar a decisão
- Importante:
  - Em análise LL(k), com k fixo, o DFA é sempre ACÍCLICO!!!
  - Ou seja, o macaco não passa por uma mesma sala mais do que uma única vez
  - LL(*) utiliza um DFA que pode conter ciclos

LL(*)

- Observe o seguinte exemplo:

```
def : modifier* classDef
    | modifier* interfaceDef
    ;
```

- O DFA de decisão seria:

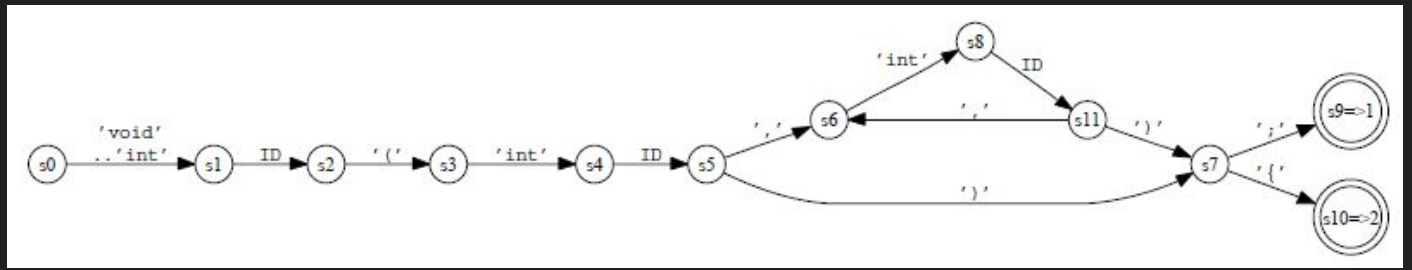


LL(*)

- Outro exemplo:

```
method
    : type ID '(' args ')' ';'           // E.g., "int f(int x,int y);"
    | type ID '(' args ')' '{' body '}' // E.g., "int f(int z) {...}"
    ;
type: 'void' | 'int' ;
args: arg (',' arg)* ; // E.g., "int x, int y, int z, ..."
arg : 'int' ID ;
body: ... ;
```

- O DFA ficaria:



LL(*)

- Ou seja, LL(*) é uma técnica poderosa, com grande poder de reconhecimento
  - Não precisa fatorar à esquerda, o que leva a gramáticas mais intuitivas
  - Permite a geração de analisadores de descendência recursiva
    - Mais legíveis e fáceis de compreender (quando comparado com analisadores LR – que veremos a seguir)
    - Facilita a inserção de ações semânticas (veremos mais adiante)

ALL(*) - Adaptive LL(*)

- Geração do DFA de decisão em tempo de execução
  - Múltiplos subparsers em cada ponto de dúvida
  - Fase de especulação
  - Em caso de ambiguidade: predicado semântico ou "a regra que aparecer primeiro"
- Uso da pilha de execução do parser
  - Para realizar a previsão de acordo com o contexto anterior
- Resultado: qualquer gramática não-recursiva à esquerda pode ser analisada

ALL(*)

- Além disso:
  - Reescrita automática de regras - remoção da recursividade direta à esquerda
  - Muitas otimizações - permite recursividade no léxico!
- Resultado:
  - Quase qualquer gramática "roda"!

Resumo

1. Analisadores descendentes - derivação
2. Esforço é conseguir 100% de predição / eficiência
    a. Sem abrir mão da facilidade de uso e desenvolvimento
3. Técnicas automatizadas
4. Evolução até a mais recente

Simples e eficiente
Porém com restrições na gramática que dificultavam bastante seu uso e abrangência

LL(1) => LL(k) => LL(*) => ALL(*)

Simples e eficiente
Praticamente qualquer gramática pode ser utilizada sem necessidade de reescrita

Fim