Construção de compiladores

Prof. Daniel Lucrédio
Departamento de Computação - UFSCar
1º semestre / 2015
Aula 4

Análise sintática descendente

Ou análise sintática top-down

Análise sintática descendente

- Vimos que existe duas formas de se 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

$$
\begin{aligned}
E &\rightarrow I \\
&\mid E + E \\
&\mid E * E \\
&\mid (E) \\
I &\rightarrow a \\
&\mid b \\
&\mid Ia \\
&\mid Ib \\
&\mid I0 \\
&\mid I1
\end{aligned}
$$

Análise sintática descendente

- Inferência recursiva
  - Dada uma cadeia (conjunto de símbolos terminais)
  - Do corpo para a cabeça
- Ex: a*(a+b00)
  - $a*(a+b00) \Leftarrow a*(a+I00) \Leftarrow a*(a+I0) \Leftarrow a*(a+I) \Leftarrow a*(a+E) \Leftarrow a*(I+E) \Leftarrow a*(E+E) \Leftarrow a*(E) \Leftarrow a*E \Leftarrow I*E \Leftarrow E*E \Leftarrow E$

Análise sintática descendente

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

- Método (algoritmo) que produz uma derivação mais à esquerda para uma cadeia da entrada
- O problema principal em cada passo é determinar qual produção aplicar
- Sendo que os tokens são lidos da esquerda para a direita
- Ex:
  - Entrada: a + b * c
  - Token atual = a
  - Símbolo inicial: E
  - Possíveis produções de E:
    - E + E
    - E * E
    - (E)

Qual escolher?

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

Análise sintática descendente

- Existem duas opções:
  - Com retrocesso (tentativa e erro)
  - Preditivo (tenta adivinhar)
- Imagine-se num labirinto de salas
  - Cada sala possui várias portas
  - Cada porta possui uma palavra mágica que a abre
  - Você possui uma lista de palavras, que só pode ser utilizada na ordem correta
- Em cada sala, você olha a próxima palavra da lista e precisa abrir a porta correta
  - Se não houver nenhuma porta com a próxima palavra, acabou a brincadeira
  - Mas pode haver mais de uma porta com a próxima palavra → não-determinismo

Análise sintática descendente

- Uma opção é escolher uma das portas arbitrariamente e tentar esse caminho até o fim
- Se der, deu
- Se não der, você precisa voltar até o ponto da escolha (retrocesso)
  - Por isso é bom deixar uma marca
  - Na verdade, várias marcas, uma para cada ponto de decisão

Análise sintática descendente

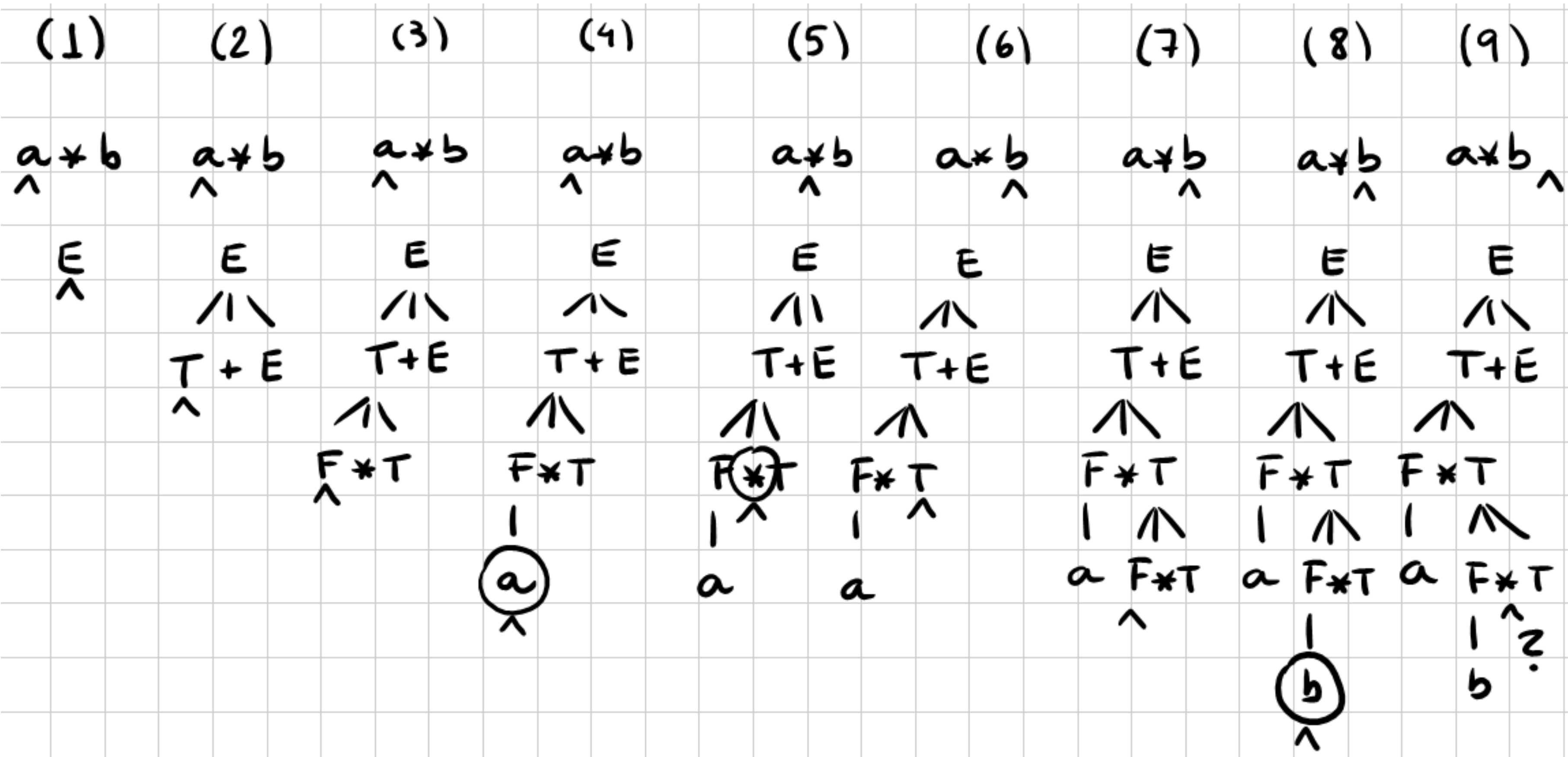
- Exemplo:

$E \rightarrow T + E \mid T$

$T \rightarrow F * T \mid F$

$F \rightarrow a \mid b \mid (E)$

Cadeia = **a * b**



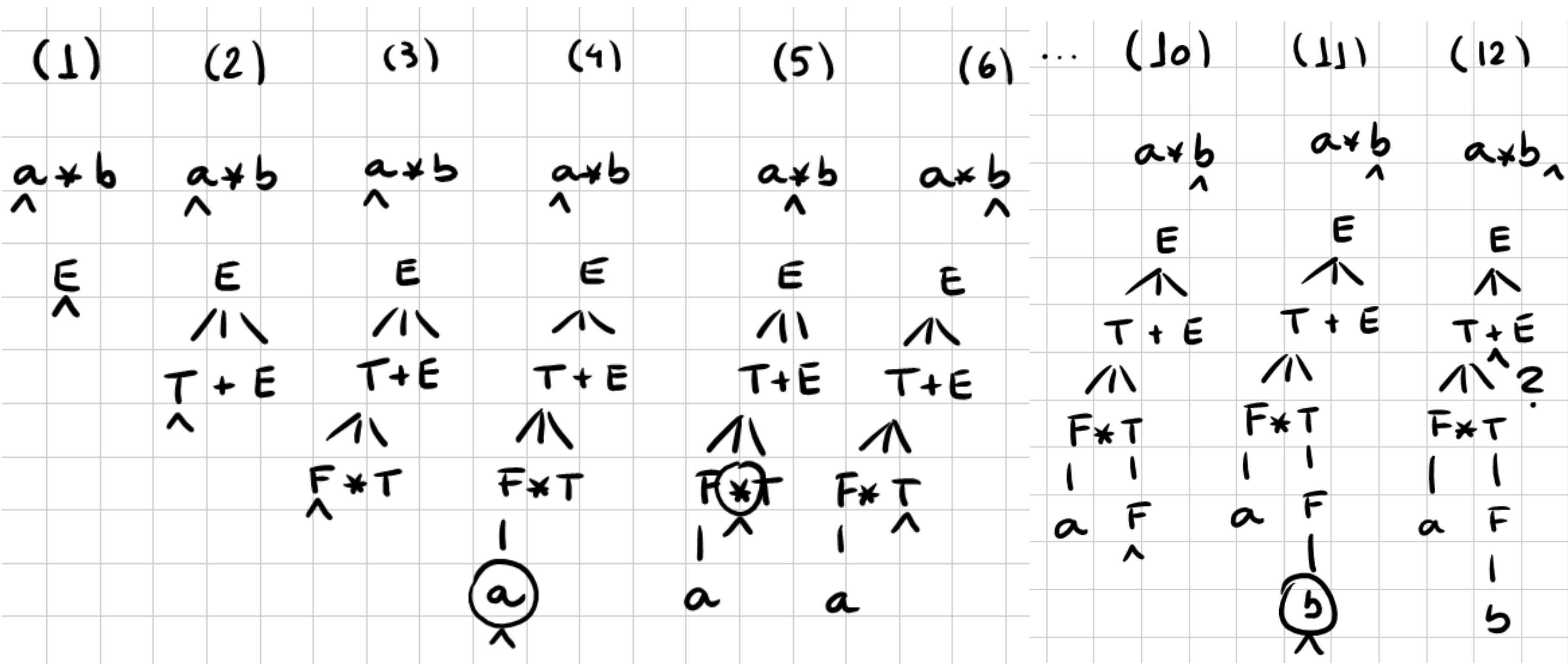
Retroceder: 6 ← 7 ← 8 ←

Análise sintática descendente

- Exemplo:

$$E \rightarrow \mathbf{T} + \mathbf{E} \mid \mathbf{T}$$
$$T \rightarrow \mathbf{F} * \mathbf{T} \mid \mathbf{F}$$
$$F \rightarrow \mathbf{a} \mid \mathbf{b} \mid (\mathbf{E})$$

Cadeia = **a * b**



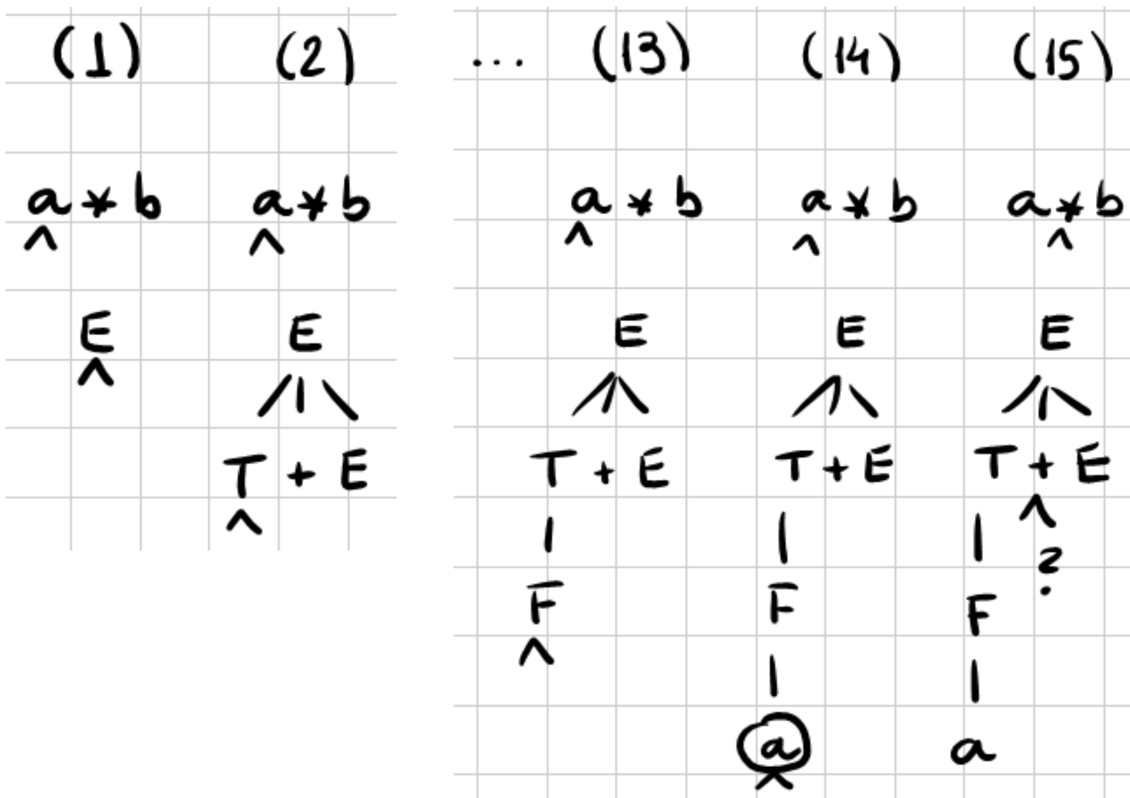
Retroceder: 2 ← 3 ← 4 ← 5 ← 6 ← 10 ← 11 ←

Análise sintática descendente

- Exemplo:

$$E \rightarrow T + E \mid T$$
$$T \rightarrow F * T \mid F$$
$$F \rightarrow a \mid b \mid (E)$$

Cadeia = **a * b**



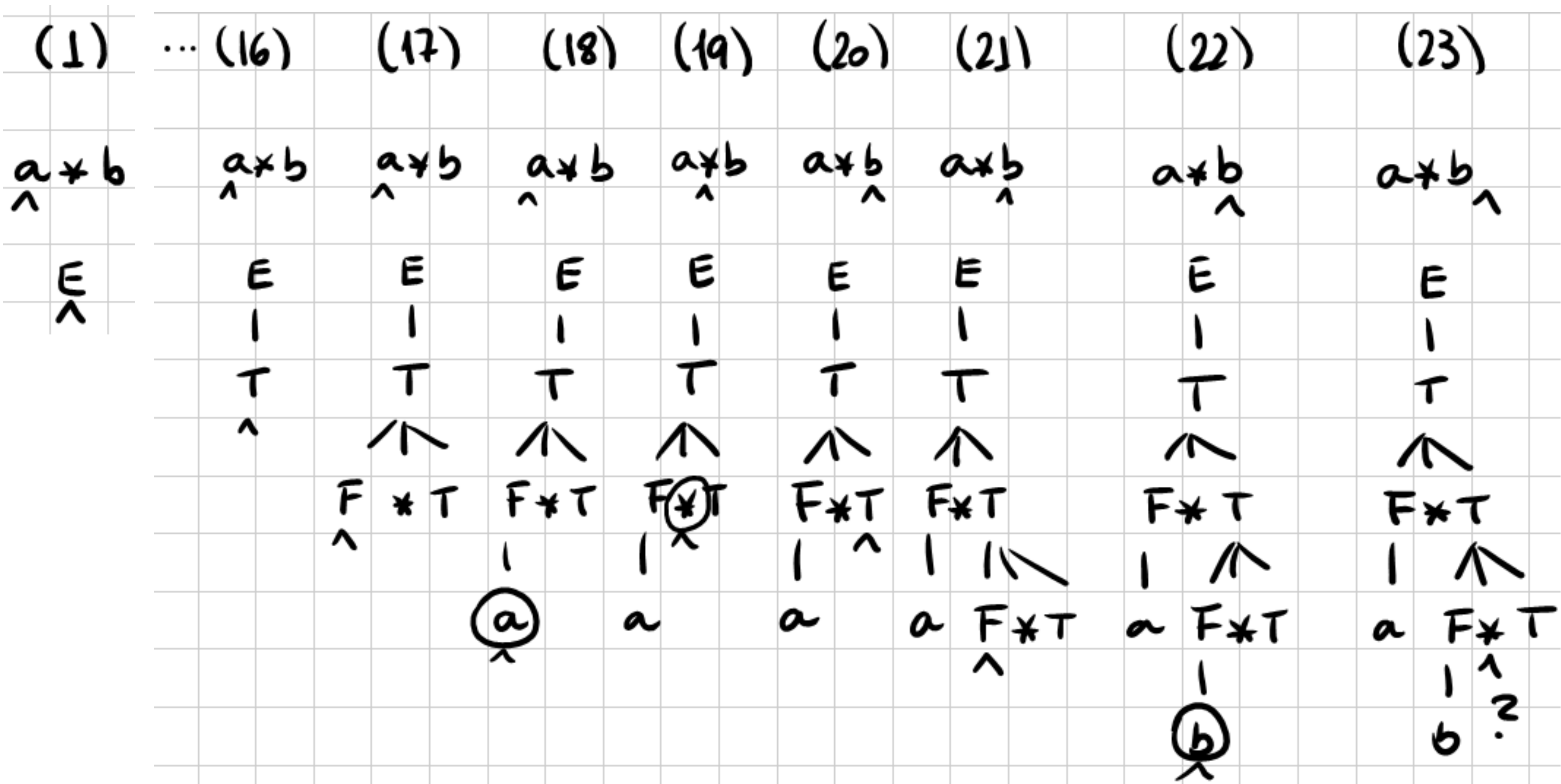
Retroceder: 1 ← 2 ← 13 ← 14 ←

Análise sintática descendente

- Exemplo:

$$E \rightarrow T + E \mid T$$
$$T \rightarrow F * T \mid F$$
$$F \rightarrow a \mid b \mid (E)$$

Cadeia = **a * b**



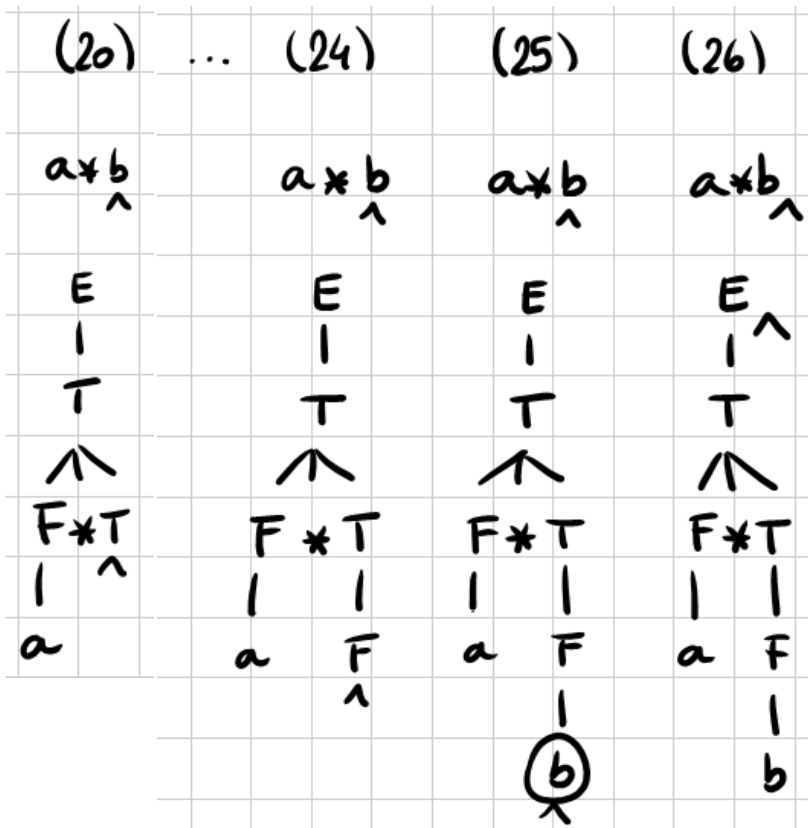
Retroceder: 20 ← 21 ← 22 ←

Análise sintática descendente

- Exemplo:

$$E \rightarrow T + E \mid T$$
$$T \rightarrow F * T \mid F$$
$$F \rightarrow a \mid b \mid (E)$$

Cadeia = **a * b**



Cadeia reconhecida

Análise sintática descendente

- Essa abordagem resolve o não-determinismo na base da força bruta!
- Vantagem é que dá pra sair de qualquer labirinto
  - Desde que tenha saída, claro!
  - E desde que não haja nenhum ciclo
    - Pois corre-se o risco de tentar a mesma porta várias vezes, já que as “marcas” de retrocesso só são vistas quando estamos voltando
    - É como se só pudéssemos marcar o verso da porta - veremos mais sobre isso depois
- Mas a desvantagem é a ineficiência
  - Vimos em LFA que força bruta = tempo exponencial

Análise sintática descendente

- Outra opção é “adivinhar” a porta correta
  - Abordagem preditiva
  - Mas como fazer isso?
  - O poder de oráculo (que discutimos em LFA) é apenas uma abstração matemática, não existe na prática
    - Que é o nosso caso!!
- Mas existem maneiras
  - 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 (por exemplo, atravessando no máximo 3 salas antes de voltar)
    - Na prática, é comum olhar apenas uma sala à frente

Análise sintática descendente

- Exemplo:

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

Cadeia = **a * b**

K = 1

a * b

E → T + E → F * T + E → a * T + E

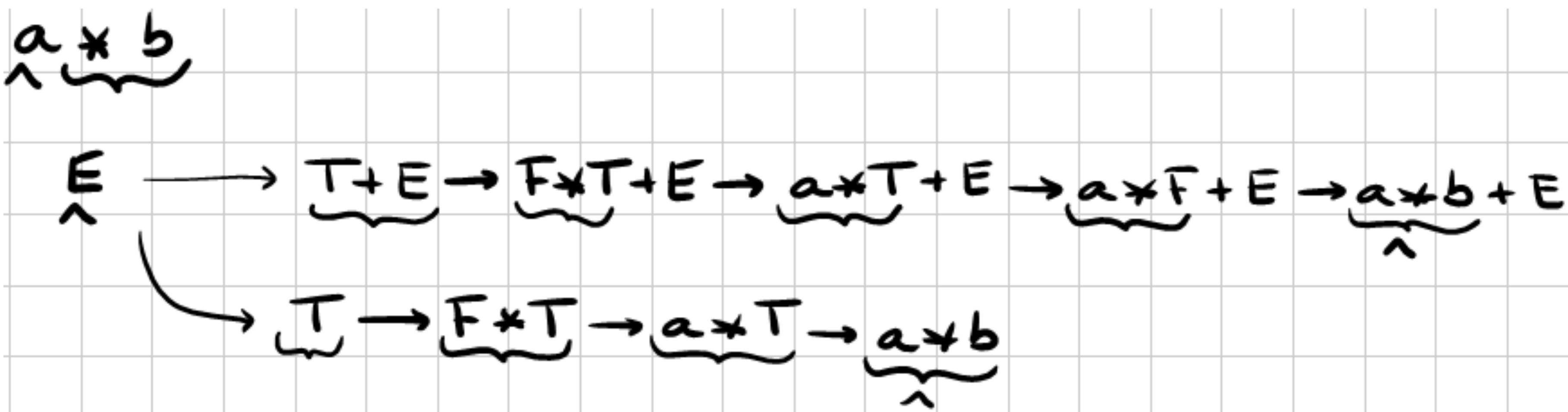
E → T → F * T → a * T

Olhando apenas um símbolo à frente não é possível resolver o não-determinismo em E

Análise sintática descendente

- Exemplo:

E → T + E | T
T → F * T | F
F → a | b | (E)
Cadeia = **a * b**
K = 2



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

Análise sintática descendente

- Exemplo:

`E → T + E | T`
`T → F * T | F`
`F → a | b | (E)`
Cadeia = `a * b`
`K = 3`

$a * b\ \varepsilon$

$E \rightarrow T+E \rightarrow F*T+E \rightarrow a*T+E \rightarrow a*F+E \rightarrow a*b+E$

$E \rightarrow T \rightarrow F*T\varepsilon \rightarrow a*T\varepsilon \rightarrow a*b\varepsilon$

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

Análise sintática descendente

- Exemplo:

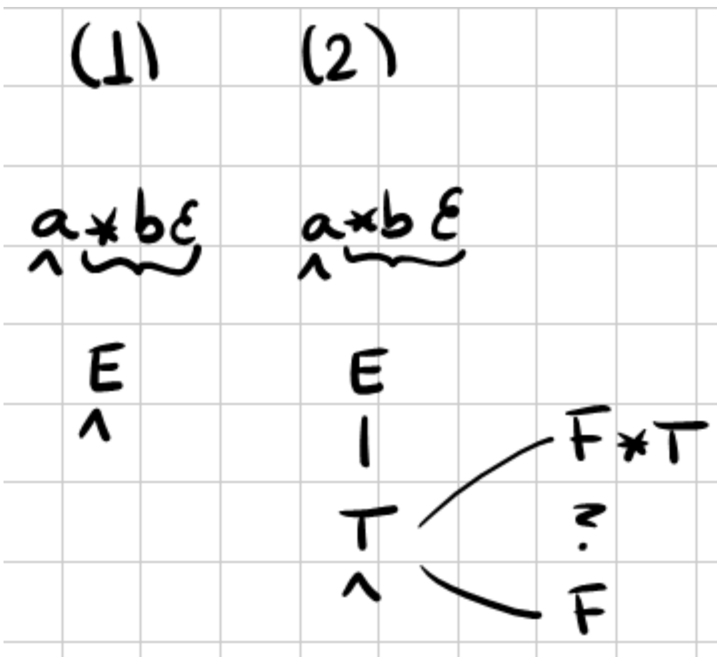
$E \rightarrow T + E \mid T$

$T \rightarrow F * T \mid F$

$F \rightarrow a \mid b \mid (E)$

Cadeia = **a * b**

K = 3

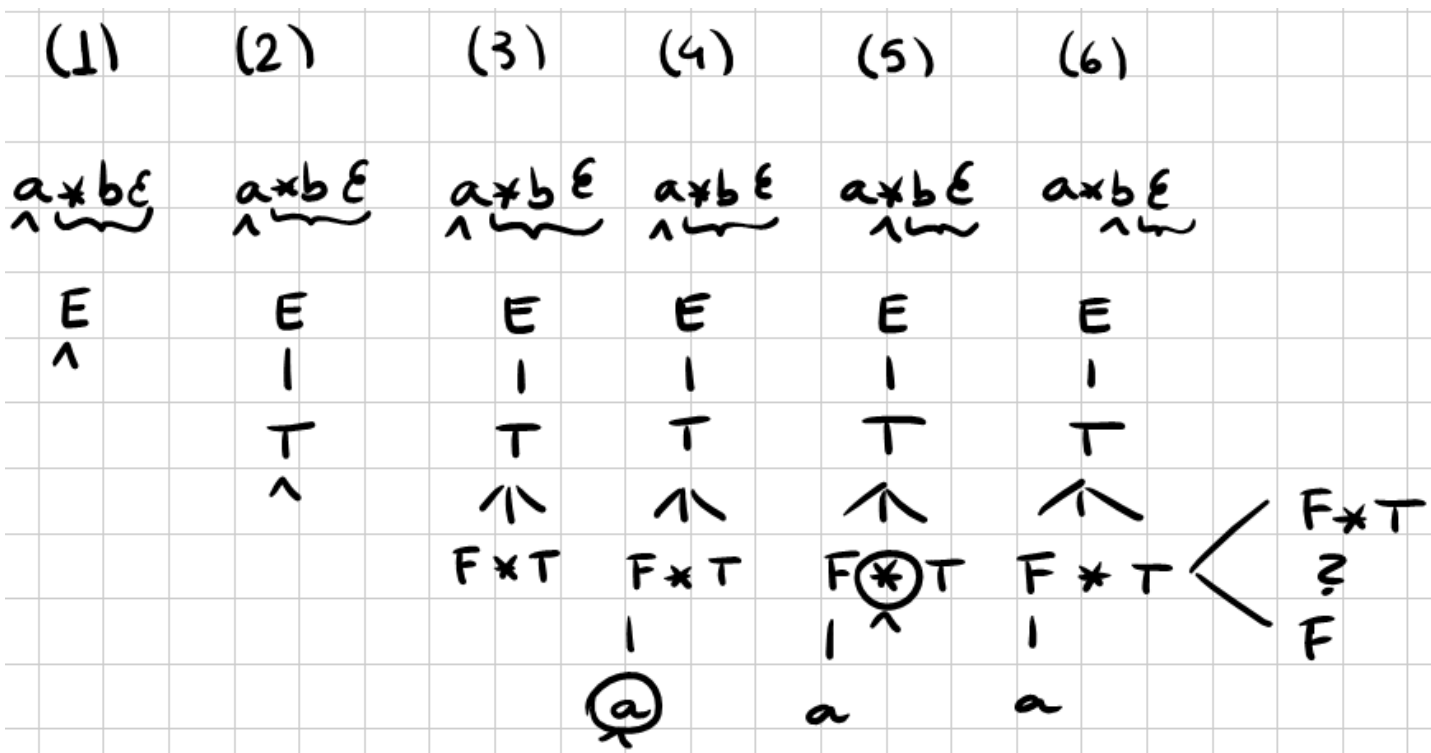


Olhando três símbolos à frente é possível resolver o não-determinismo em T (escolhendo T → F * T)

Análise sintática descendente

- Exemplo:

$E \rightarrow T + E \mid T$
$T \rightarrow F * T \mid F$
$F \rightarrow a \mid b \mid (E)$
Cadeia = $a * b$
$K = 3$



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

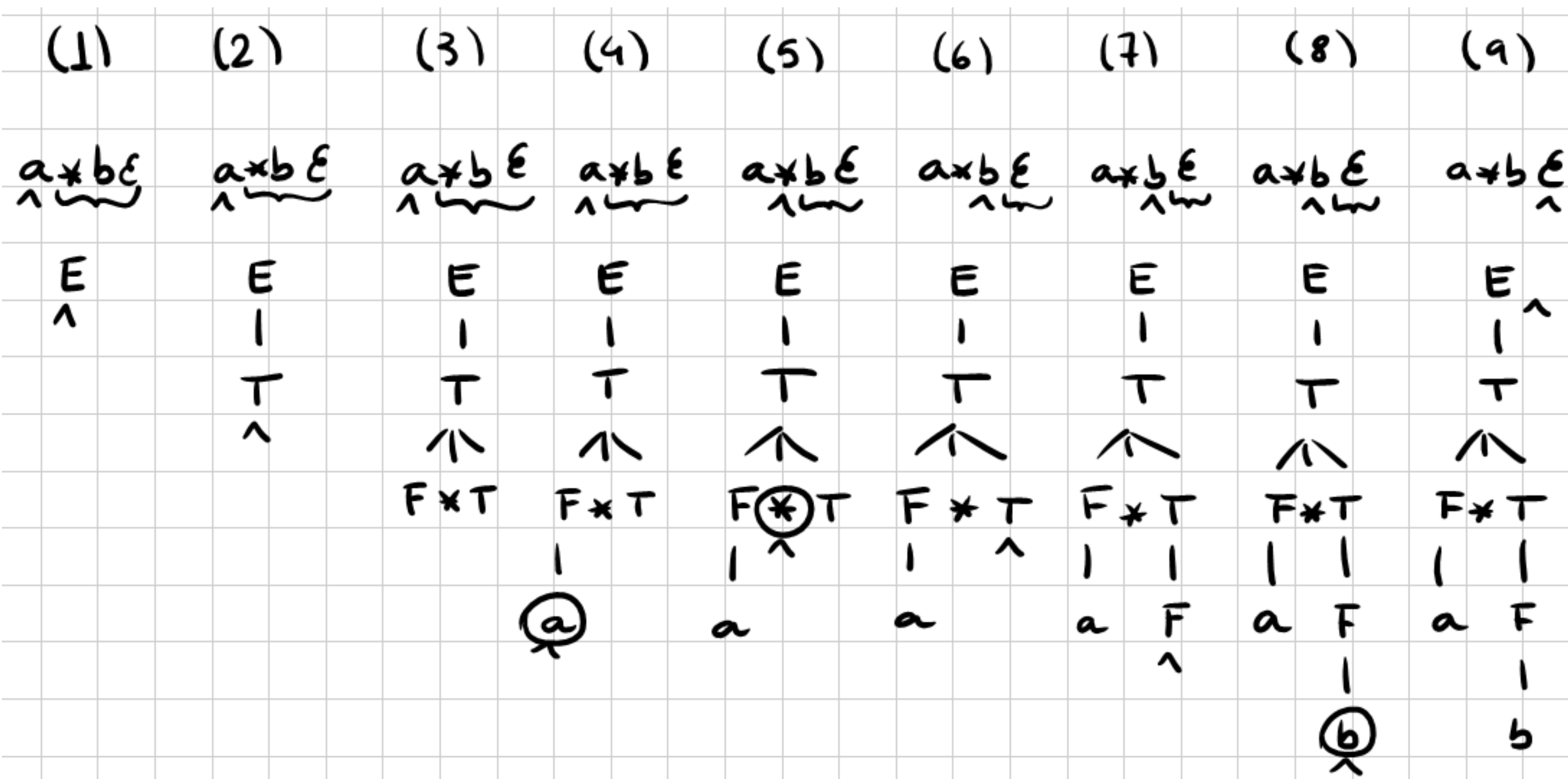
Análise sintática descendente

- Exemplo:

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

Cadeia = `a * b`

`K = 3`



Cadeia reconhecida

Análise sintática descendente

- A abordagem preditiva é obviamente mais eficiente
  - Ela não precisa tentar todos os caminhos
  - Em outras palavras, o tempo de execução não é exponencial
    - Como a força bruta da abordagem com retrocesso
- Porém, ela é mais cautelosa
- Não é todo labirinto que ela consegue encontrar a saída
  - Apenas alguns labirintos “especiais”
  - Com propriedades interessantes
    - Com pontos de não-determinismo limitados a 1, 2 ou k salas adjacentes
- Ou seja, 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-se somente k salas à frente

Análise sintática descendente

- Mas chega de labirintos, portas, palavras e listas
- Teste de aprendizado – faça a associação:

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

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(*)

Analisador sintático preditivo de descendência recursiva

Analisador sintático preditivo de descendência recursiva

- É o tipo mais simples
- É o preferido quando se constrói à mão
- Usa funções recursivas
  - Uma função para cada não-terminal
- Cada função é um espelho das regras de produção
  - Faz o "casamento" dos terminais
  - E chamadas para outros não-terminais

Analisador sintático preditivo de descendência recursiva

- Exemplo:

S → c A d

A → a b A | c

```
void match(token) {
// Testa se o símbolo atual
// casa com o token especificado
// Se sim, avança a leitura
// Se não, acusa erro
}

token prox() {
// Retorna o próximo token, mas
// avançar a leitura. Ou seja,
// dá apenas uma "olhadinha" à
// frente
}
```

```
void S() {
   match("c");
   A();
   match("d");
}
void A() {
   if(prox() == "a") {
      match("a");
      match("b");
      A();
   }
   else if(prox() == "c") {
      match("c");
   }
   else { // erro sintático
   }
}
```

Analisador sintático preditivo de descendência recursiva

- Vamos tentar implementar a linguagem ALGUMA

Analisador sintático preditivo de descendência recursiva

- Já vimos como fazer
  - Precisa apenas de algum raciocínio
  - Um pouco de lógica
  - E muita mão na massa
- Vamos agora estudar um pouco de teoria
  - Quem sabe não dá para automatizar esse processo?

Conjuntos primeiros e seguidores

- Funções associadas a uma gramática
  - Ajudam na construção de analisadores descendentes e ascendentes
  - Ajudam a escolher qual produção aplicar
  - Durante a recuperação em modo pânico, podem servir de tokens de sincronização
    - Mais sobre isso depois

Conjuntos primeiros e seguidores

- `primeiros(α)`
  - α é uma cadeia de **<u>símbolos gramaticais</u>**
    - **<u>Obs: não necessariamente uma cadeia de terminais!!</u>**
  - É o conjunto de terminais que começam as cadeias derivadas de α
- `seguidores(A)`
  - A é um não-terminal
  - 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 $S \Rightarrow \alpha A a \beta$
    - $\{a \mid S \Rightarrow \alpha A a \beta\}$
  - Símbolo especial $ (fim de cadeia)
  - Se A pode ser o símbolo mais à direita em alguma forma sentencial, $ está em seguidores(A)

Conjuntos primeiros e seguidores

- Gramática
  - $S \rightarrow A\ B$
  - $A \rightarrow a\ A \mid a$
  - $B \rightarrow b\ B \mid c$
- primeiros(S) = {a}
- primeiros(AB) = {a}
- primeiros(aA) = {a}
- primeiros(bB) = {b}
- primeiros(B) = {b,c}
- Obs: primeiros pode ser calculado para qualquer cadeia envolvendo terminais e/ou não-terminais

Conjuntos primeiros e seguidores

- Gramática
  - $S \rightarrow A\ B$
  - $A \rightarrow a\ A \mid a$
  - $B \rightarrow b\ B \mid c$
- seguidores(S) = {\$}
- seguidores(A) = {b,c}
- seguidores(B) = {\$}

Conjunto primeiros

- Para calcular `primeiros(X)`
  - Onde X é um único símbolo (terminal ou não-terminal)

1. Se X é um terminal, `primeiros(X)` = {X}
2. Se X é um não-terminal e $X \rightarrow Y_1Y_2...Y_k$ é uma produção ($k \geq 1$)
   - Insira "a" em `primeiros(X)` se, para algum i:
     - "a" está em $\texttt{primeiros}(Y_i)$ (obs: $a \neq \varepsilon$)
     - ε está em todos os $\texttt{primeiros}(Y_1)$,...,$\texttt{primeiros}(Y_{i-1})$, ou seja,
       - $Y_1...Y_{i-1} \Rightarrow^* \varepsilon$
   - Se ε está em $\texttt{primeiros}(Y_j)$ para todo j=1,2,...,k, então insira ε em `primeiros(X)`
3. Se $X \rightarrow \varepsilon$ é uma produção, então insira ε em `primeiros(X)`

Conjunto primeiros

- Para calcular `primeiros(X)`
  - Onde X é um único símbolo (terminal ou não-terminal)

1. Se X é um terminal, `primeiros(X)` = {X}
2. Se X é um não-terminal e $X \rightarrow Y_1Y_2...Y_k$ é uma produção ($k \geq 1$)
   - Insira "a" em `primeiros(X)` se, para algum i:
     - "a" está em `primeiros(`Y_i`)` (obs: $a \neq \varepsilon$)
     - ε está em todos os `primeiros(`Y_1`)`,...,`primeiros` $_1$`)`, ou seja,
       - $Y_1...Y_{i-1} \Rightarrow^* \varepsilon$
   - Se ε está em `primeiros(`Y_j`)` para todo … ε em `primeiros(X)`
3. Se $X \rightarrow \varepsilon$ é uma produção, então insira ε em

Significa que preciso olhar todas as produções de X

Conjunto primeiros

- Para calc
  - Onde X (terminal ou nã

1. Se X é um ter al, `primeiros(X)` = {X}
2. Se X é um não-terminal e X → $Y_1Y_2...Y_k$ é um
   - Insira “a” em `primeiros(X)` se, para algum i
     - “a” está em `primeiros(`Y_i`)` (obs: a ≠ ε)
     - ε está em todos os `primeiros(`Y_1`)`,...,`primeiros(`Y_{i-1}`)`, ou seja,
       - $Y_1...Y_{i-1} \Rightarrow^* \varepsilon$
   - Se ε está em `pri eiros(`Y_j para todo j=1,2,...,k, então insira ε em `primeiros(X)`
3. Se X → ε é uma p ção, então a ε em `primeiros(X)`

Todo mundo de primeiros(Y1) vai para primeiros(X) (Exceto ε)

Obs: veja que até agora o ε não é inserido, mesmo que ele apareça em alguns primeiros

Se primeiros(Y1) contém ε, então também insiro os primeiros(Y2) (Exceto ε)

Se primeiros(Y1) e primeiros(Y2) contém ε, então também insiro os primeiros(Y3) (Exceto ε)

E assim por diante...

Conjunto primeiros

- Para calcular `primeiros(X)`
  - Onde X é um único símbolo (terminal ou não-terminal)

1. Se X é um terminal, `primeiros(X)` = {X}
2. Se X é um não-terminal e $X \rightarrow Y_1Y_2...Y_k$ é uma produção ($k \geq 1$)
   - Insira "a" em `primeiros(X)` se, para algum i:
     - "a" está em $\texttt{primeiros}(Y_i)$ (obs: $a \neq \varepsilon$)
     - ε está em todos os $\texttt{primeiros}(Y_1)$,...,$\texttt{primeiros}(Y_{i-1})$, ou seja,
       - $Y_1...Y_{i-1} \Rightarrow^* \varepsilon$
   - Se ε está em $\texttt{primeiros}(Y_j)$ para todo j=1,2,...,k, então insira ε em `primeiros(X)`
3. Se $X \rightarrow \varepsilon$ é uma produção, então insira ε em `primeiros(X)`

Agora sim: se ε aparece sozinho no lado direito, ou se está em TODOS os primeiros de TODOS os símbolos à direita, então ele entra!

Conjunto primeiros

- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \varepsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \varepsilon$
- $F \rightarrow (E) \mid id$
- $X \rightarrow E'T$
- $Y \rightarrow T'E'$

```
primeiros(E)={(,id}
primeiros(T)={(,id}
primeiros(F)={(,id}
primeiros("(")={(}
primeiros(id)={id}
primeiros(E')={+,ε}
primeiros(+)={+}
primeiros(T')={*,ε}
primeiros(*)={*}
primeiros(X)={+,(,id}
primeiros(Y)={*,+,ε}
```

Conjunto primeiros

- Na verdade nosso interesse é calcular o conjunto primeiros de uma cadeia de símbolos gramaticais
  - Para ajudar na análise sintática
- Para calcular `primeiros` $(X_1X_2...X_n)$
  - Adicione a `primeiros` $(X_1X_2...X_n)$ todos os símbolos (exceto ε) de `primeiros` (X_1)
  - Se ε está em `primeiros` (X_1), adicione também todos os símbolos (exceto ε) de `primeiros` (X_2)
  - Se ε está em `primeiros` (X_1) e `primeiros` (X_2), adicione também todos os símbolos (exceto ε) de `primeiros` (X_3)
  - Se ε está em `primeiros` de TODOS os Xi, então adicione ε

Conjunto primeiros

- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \varepsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \varepsilon$
- $F \rightarrow (E) \mid id$
- $X \rightarrow E'T$
- $Y \rightarrow T'E'$

```
primeiros(E)={(,id}
primeiros(T)={(,id}
primeiros(F)={(,id}
primeiros("(")={(}
primeiros(id)={id}
primeiros(E')={+,ε}
primeiros(+)={+}
primeiros(T')={*,ε}
primeiros(*)={*}
primeiros(X)={+,(,id}
primeiros(Y)={*,+,ε}
```

primeiros(TE')={(,id}
primeiros(+TE')={+}
primeiros(FT')={(,id}
primeiros(*FT')={*}
primeiros("("E")")={(}
primeiros(E'T)={+,(,id}
primeiros(T'E')={*,+,ε}

Conjunto seguidores

- Para calcular o conjunto `seguidores`

1. Adicione $ a `seguidores(S)`
   - S é o símbolo inicial
2. Se existir uma produção A → αBβ, adicione os `primeiros(β)` a `seguidores(B)` (exceto ε)
3. Se existir uma produção A → αB, ou uma produção A → αBβ onde `primeiros(β)` contém ε, adicione os `seguidores(A)` a `seguidores(B)`

Conjunto seguidores

- 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)={*,+,$,)}

Exercício

- Construa os conjuntos primeiros e seguidores para a seguinte gramática

```
expr → expr soma termo | termo
soma → + | -
termo → termo mult fator | fator
mult → *
fator → ( expr ) | número
```

Resposta

```
primeiros(expr)={(,número}
primeiros(soma)={+,-}
primeiros(termo)={(,número}
primeiros(mult)={*}
primeiros(fator)={(,número}
primeiros(expr soma termo)={(,número}
primeiros(termo mult fator)={(,número}
primeiros("(" expr ")")={(}

seguidores(expr)={$,),+,-}
seguidores(soma)={(,número}
seguidores(termo)={$,),+,-,*}
seguidores(mult)={(,número}
seguidores(fator)={$,),+,-,*}
```

Exercício

- Construa os conjuntos primeiros e seguidores para a seguinte gramática

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

Resposta

```
primeiros(declaracao)={'outra','if'}
primeiros(if-decl)={'if'}
primeiros(else-parte)={'else',ε}
primeiros(exp)={'0','1'}

seguidores(declaracao)={$,'else'}
seguidores(if-decl)={$,'else'}
seguidores(else-parte)={$,'else'}
seguidores(exp)={')'}
```

Gramáticas LL

Gramáticas LL(k)

- LL(k) = classe de gramáticas / técnica de análise sintática
  - Left-to-right = da esquerda para a direita
  - Leftmost derivation = derivação mais à esquerda
  - k = símbolos à frente necessários para a previsão correta
- k > 1 geralmente acarreta em baixa eficiência
  - Por isso, estudaremos inicialmente o caso LL(1)
- LL(1) é bastante rica para a maioria das construções de linguagens de programação
  - Mas exige cuidado e trabalho
  - Por exemplo: é preciso remover ambiguidade / recursão à esquerda

Gramáticas LL(1)

- Uma gramática G é LL(1) sse: para duas produções distintas G
  - $A \rightarrow \alpha \mid \beta$:

1. Para um terminal "a", tanto α quanto β não derivam cadeias começando com "a".
2. No máximo um dos dois, α ou β, pode derivar a cadeia vazia
3. Se $\beta \Rightarrow^* \varepsilon$, então α não deriva nenhuma cadeia começando com um terminal em seguidores(A).
   - Se $\alpha \Rightarrow^* \varepsilon$, então β não deriva nenhuma cadeia começando com um terminal em seguidores(A).

Gramáticas LL(1)

> primeiros(α) e primeiros(β) são disjuntos

- Uma gramática G é LL(1) sse: para c
  produções distintas G
  - $A \rightarrow \alpha \mid \beta$:

1. Para um terminal "a", tanto α quanto β não derivam cadeias começando com "a".
2. No máximo um dos dois, α ou β, pode derivar a cadeia vazia
3. Se $\beta \Rightarrow^* \varepsilon$, então α não deriva nenhuma cadeia começando com um terminal em seguidores(A).
   - Se $\alpha \Rightarrow^* \varepsilon$, então β não deriva nenhuma cadeia começando com um terminal em seguidores(A).

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

<u>FatorRel → '(' ExprRel ')' | Expr '<' Expr</u>

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) ss... s G

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

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

...al "a", tanto α ... β não ...derivam cadeias começando ...m "a".

No máximo um dos dois, α ou β, pode derivar a cadeia vazia

3 Se β ⇒* ε, então α não deriva nenhuma cadeia começando com um terminal em seguidores(A).

- Se α ⇒* ε, então β não deriva nenhuma cadeia começando com um terminal em seguidores(A).

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
```

<u>then-parte → comandoThen | listaComandos</u>

```
comandoThen → 'soma' | ε
exp → '0' | '1'
```

primeiros(comandoThen)={'soma',ε}

primeiros(listaComandos)={'if','outra',<u>';'</u>,'{'}

seguidores(then-parte)={'else',<u>';'</u>}

Gramáticas LL(1)

- Na prática:

```
Entrada = outra; outra; ; if (1) ; outra ; outra;

void ifDecl() {
  match('if'); match('(');
  exp();
  match(')');
  thenParte();
}
void thenParte() {
  if(proximo == 'soma') comandoThen();
  if(proximo == ';') {
     listaComandos();
     // mas também poderia parar por aqui
  }
}
```

Gramáticas LL(1)

- Estas condições garantem que sempre é possível escolher a produção apropriada olhando-se somente 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
- Implementação é simples: basta um if (ou switch)

Gramáticas LL(1)

- Em um analisador sintático preditivo de descendência recursiva
  - Essas condições facilitam a implementação da escolha da alternativa
- No exemplo anterior de implementação manual, fizemos a “predição” usando apenas raciocínio
  - Mas é possível utilizar um algoritmo
  - Baseado nas condições das gramáticas LL(1)

Tabela LL(1)

- Tabela de predição
  - Array bidimensional M[A,a]

	Terminais e $
Não-terminais	

Produção a ser escolhida

Tabela LL(1)

- Idéia geral
  - A produção $A \rightarrow \alpha$ é escolhida se o próximo símbolo de entrada "a" estiver em `primeiros(α)`
  - Se $\alpha = \varepsilon$ ou $\alpha \overset{*}{\Rightarrow} \varepsilon$, escolhemos $A \rightarrow \alpha$ se o próximo símbolo de entrada "a" estiver em `seguidores(A)` ou se o \$ foi alcançado e \$ está em `seguidores(A)`
- Objetivo
  - Colocar na tabela as possibilidades de escolha de produção, dados:
    - Um símbolo de entrada (terminal), que representa o próximo símbolo a ser lido
    - A produção sendo atualmente expandida (não-terminal), que representa qual símbolo precisa ser substituído no processo de derivação

Tabela LL(1)

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

Tabela LL(1)

- Exemplo:

- 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)={*,+,$,)}
```

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	

Tabela LL(1)

- Exercício

```
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)

	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

Algoritmo LL(1)

- É fácil imaginar como implementar o algoritmo LL(1) em um analisador sintático de descendência recursiva
  1. Monte a tabela LL(1) para a gramática
  2. O seguinte pseudocódigo ilustra a estratégia, para cada não-terminal T

T → T1 T2 T3 | T4 T5 T6 | ...

```
void T() {
   Token atual = obterTokenAtual();
   int alt = M[T,atual];
   if(alt == 1) { T1(); T2(); T3(); }
   else if(alt == 2) {T4(); T5(); T6(); }
   ...
}
```

Algoritmo LL(1)

- Exercício
  - Analise a implementação “ad hoc” apresentada na aula, e implemente um método que faz a predição das produções a serem utilizadas, com base na tabela LL(1)
  - Obs: não vai ser cobrado, é apenas para você praticar e entender o conceito
  - Se algum dia tiver que implementar um analisador sintático profissional, você vai agradecer por ter feito esse exercício

Analisador sintático preditivo sem recursividade

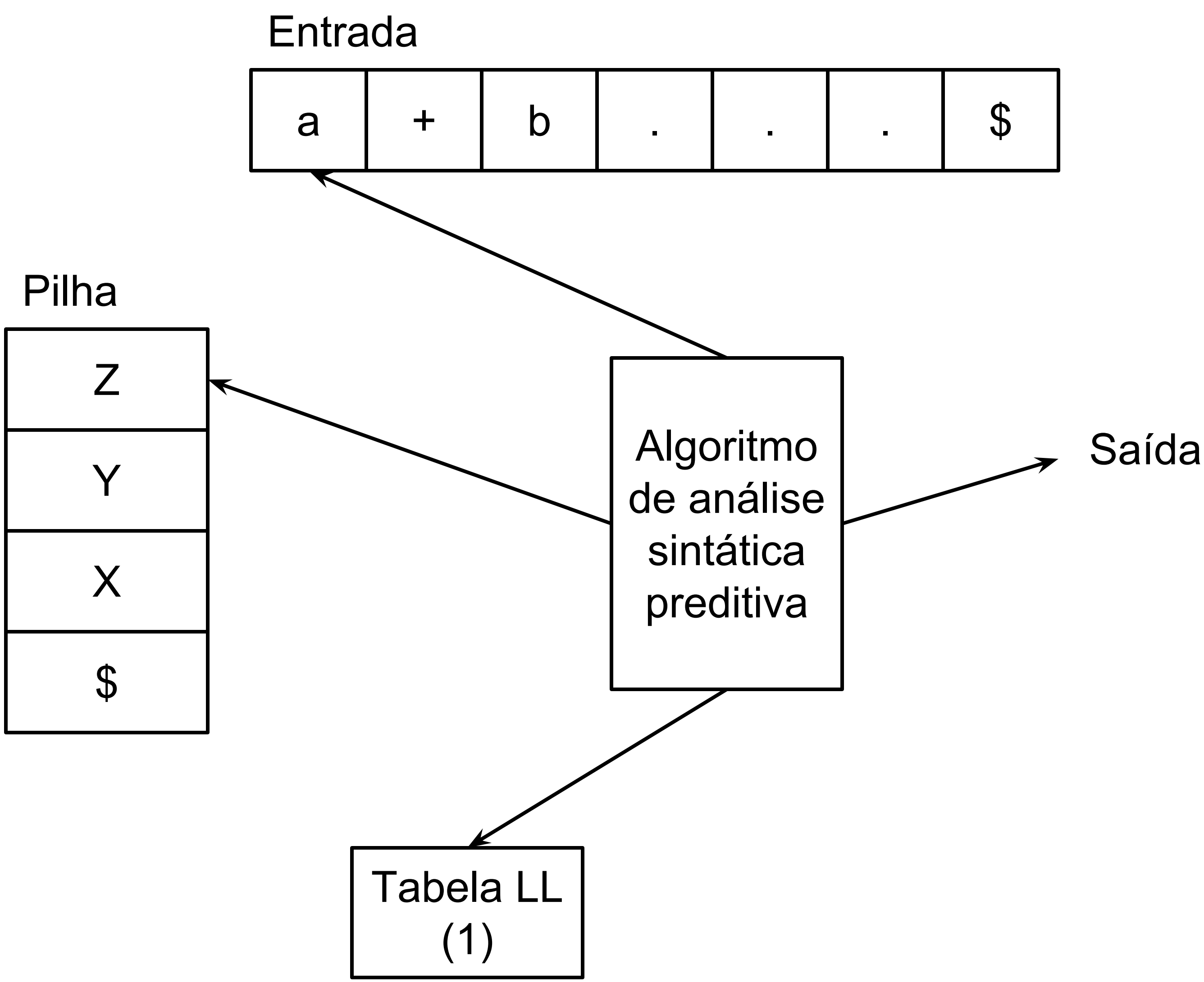
Lembrando um pouco de LFA

- 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
  - Autômato com pilha
- Mas acabamos de implementar um analisador que não usa pilha!!!
  - Aí é que você se engana!
  - Recursividade e pilha são conceitos similares
    - Cada chamada recursiva equivale a um empilhamento
    - Cada retorno de chamada recursiva equivale a um desempilhamento

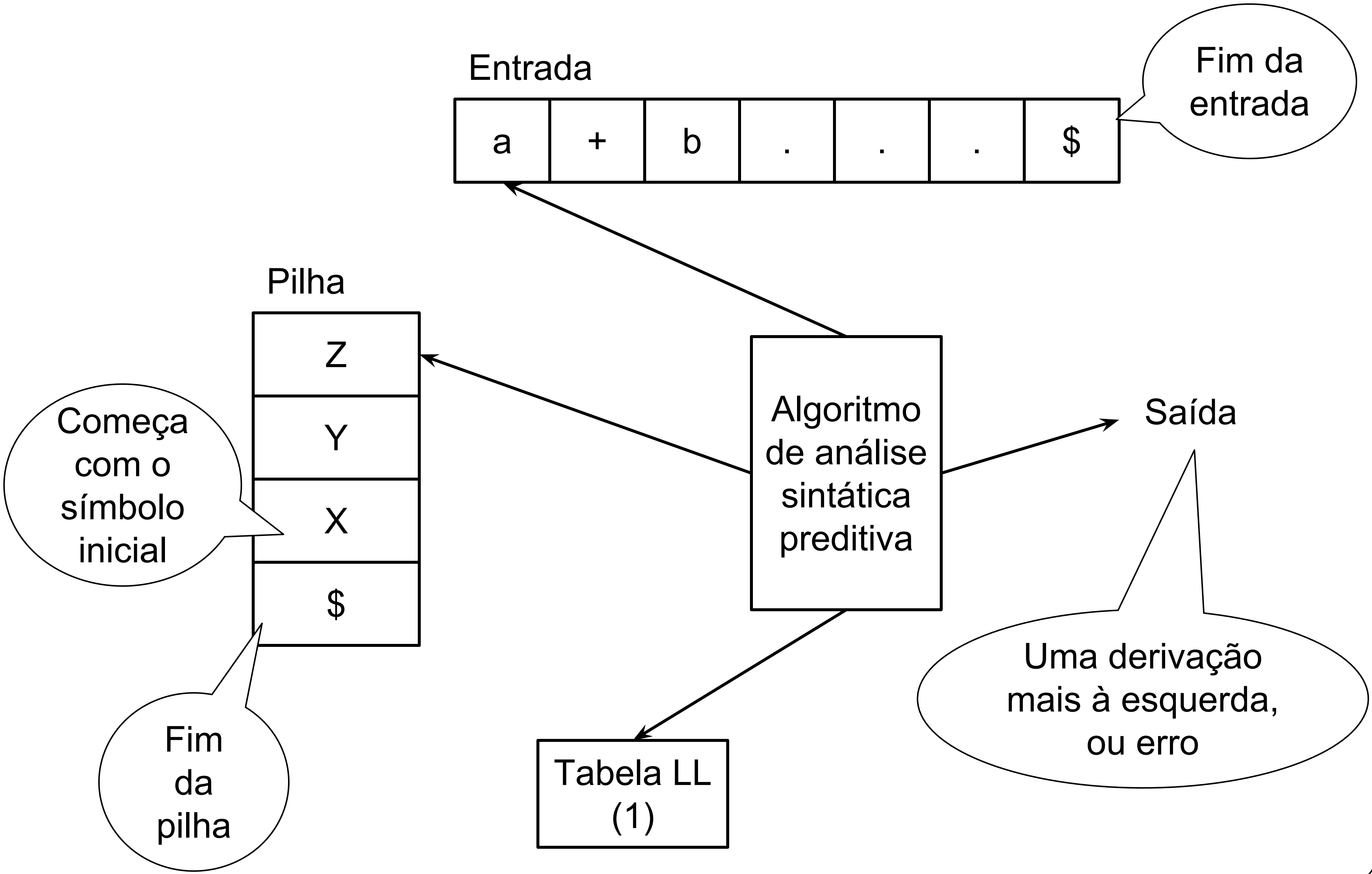
Análise sintática preditiva sem recursão

- Veremos agora um modelo baseado no uso de uma pilha
  - É exatamente o mesmo algoritmo que a versão recursiva
  - Mas aqui a sequência de derivação fica explícita
  - É um bom exercício de compreensão do processo de análise LL(1)

Análise sintática preditiva sem recursão



Análise sintática preditiva sem recursão



Algoritmo de análise sintática preditiva

```
Condições iniciais:
   entrada = w$
   símbolo S no topo da pilha, sobre $
Algoritmo:
      ip = primeiro símbolo de w
      X = topo da pilha // inicialmente, X = S
      enquanto(X != S) { // pilha não vazia
         a = w[ip]
         se(X == a) desempilhar e avançar ip
         senão se(X é terminal) erro
         senão se(M[X,a] é vazio) erro
         senão se(M[X,a] = X→Y1Y2...Yk {
            imprimir "X→Y1Y2...Yk"
            desempilhar
            empilhar Yk,Yk-1,...,Y1 // nessa ordem
         }
         X = topo da pilha
      }
```

Análise sintática preditiva sem recursão

- Exercício
- Entrada = id + id * id $

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

	+	*****	**(**	**)**	**id**	**$**
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'	$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$		$E' \rightarrow \varepsilon$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$		$T' \rightarrow \varepsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	

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

Recuperação de erros

Recuperação de erros

- Ao se deparar com erros sintáticos, o que fazer?
  - Relatar os erros o quanto antes, de modo claro e preciso
  - (opcional) corrigir erros mais simples (ex: ponto-e-vírgula faltando)
  - Continuar a análise
  - Evitar o problema da cascata de erros
  - Evitar laços infinitos na tentativa de recuperação de erros
  - Não atrasar muito o processamento de programas corretos
- Tarefa difícil
- Na maioria das vezes, feita de forma ad hoc

Recuperação de erros

- Há duas situações de erro na técnica LL apresentada:
  1. A pilha tem um terminal no topo, que é diferente do próximo símbolo de entrada
  2. A entrada na tabela LL(1) para o próximo símbolo de entrada e não-terminal no topo da pilha é vazia

Recuperação de erros

- Não vamos entrar em muitos detalhes, pois são técnicas muito avançadas
  - Quem precisar implementar um compilador profissional pode posteriormente se aprofundar nesse tópico
- Mas veremos duas estratégias básicas
  - Modo pânico
  - Recuperação em nível de frase

Recuperação de erros – modo pânico

- Estratégia:
  - Uma vez encontrado um erro, ignorar símbolos até um ponto de sincronismo
  - A partir do qual se sabe continuar
- Ex:

Pilha	Entrada	Ação
if-decl$	**if**(*2>3) then outra$	
if expr then cmd else cmd$	**if**(*2>3) then outra$	
expr then cmd else cmd$	**(***2>3) then outra$	
(expr) then cmd else cmd$	**(***2>3) then outra$	
expr) then cmd else cmd$	*****2>3) then outra$	Pular até achar um ")" e desempilha o símbolo atual
) then cmd else cmd$	**)** then outra$	Sincronismo
...	...	...

Recuperação de erros – modo pânico

- A eficácia depende da escolha dos tokens de sincronização
- Evita loop infinito
- Porém, pode descartar muitos tokens
  - Nos piores casos, pode descartar quase o programa inteiro

Recuperação de erros – modo pânico

- Heurísticas para escolha dos tokens de sincronização

1. Todos os símbolos seguidores de um não-terminal são tokens de sincronização para esse terminal
    - Não é suficiente. Por exemplo, “;” é seguidor de muitos não-terminais, o que pode levar ao descarte de muitos tokens desnecessariamente
2. Utilizar a hierarquia
    - Programa > blocos > comandos > expressões
    - Usar palavras-chave dos não-terminais mais altos na hierarquia como tokens de sincronização para os não-terminais mais baixos na hierarquia
    - Ex: “if”, “while”, “for” são tokens de sincronização para o não-terminal “Expr”

Recuperação de erros – modo pânico

3. Se um terminal no topo da pilha não puder casar com o terminal da entrada
   - Solução simples: fazer de conta que o terminal estava lá e continuar a análise
   - Emitir uma mensagem (ou não)
   - Ex: HTML

```
<html>
<head>
</head>
<body>
<table>
   <tr><td>celula 1</tr>
   <tr><td>celula2</td></tr>
</table>
</body>
```

Arquivo HTML original

```
▼<html>
   <head></head>
  ▼<body>
    ▼<table>
      ▼<tbody>
        ▼<tr>
            <td>celula 1</td>
          </tr>
        ▼<tr>
            <td>celula2</td>
          </tr>
        </tbody>
      </table>
    </body>
</html>
```

Elemento sendo inspecionado no Google Chrome

Recuperação de erros em nível de frase

- Consiste em preencher as entradas vazias da tabela com ponteiros para rotinas de tratamento de erro
- Essas rotinas podem modificar, inserir ou remover símbolos da entrada e emitir as mensagens apropriadas
- As rotinas implementam maneiras de recuperação de erro específicas para a linguagem
  - Em contraste com o modo pânico, que sempre ignora tokens até achar um de sincronização
- Por exemplo: ao encontrar
  - while(2 > 3)) { comandos... }
- O analisador pode desconsiderar o segundo parêntese e tentar continuar
  - IMPORTANTE: "desconsiderar" significa relatar e continuar tentando, e não tratar o programa como se estivesse correto!
  - Em outras palavras, o programa não deve "compilar" com sucesso!

LL(k) e LL(*)

LL(k)

- A grande vantagem do algoritmo LL(1) é a sua possibilidade de automação!
  - É possível construir analisadores à mão, claro
  - Mas estamos interessados em poupar trabalho (tempo=dinheiro)
- A princípio, é simples estender 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
  - Essa tabela seria (exponencialmente) grande, com muitas colunas, para cobrir todas as combinações de k símbolos à frente
  - Segundo, a tabela LL(k) como vimos não expressa realmente todo o poder da análise LL(k). Isto porque, para k>1, a tabela precisaria considerar diferentes contextos para os conjuntos seguidores
  - Terceiro, se uma gramática não é LL(1), ela provavelmente não é LL(k) também
    - Exemplo: 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 ad hoc

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

LL(*)

- O grande problema dos analisadores LL não é o reconhecimento em si
  - Que é fácil:
    - Não-terminal = naoTerminal()
    - Terminal = match(Terminal)
- O problema é determinar (predizer) corretamente a regra a ser utilizada
- O método findAhead auxilia na predição
  - Ele inspeciona símbolos à frente, em busca de um determinado símbolo
  - O número de símbolos a ser inspecionado é variável
    - k=*, por isso LL(*)

LL(*)

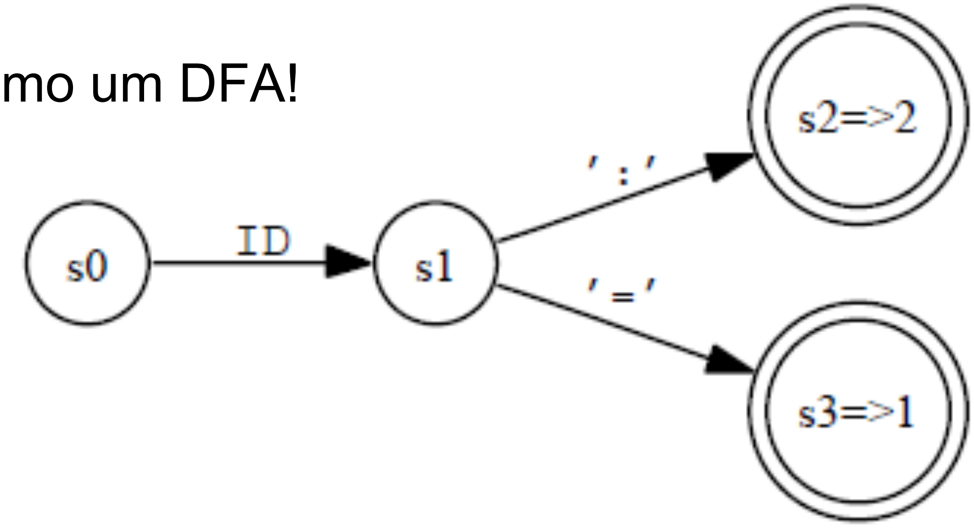
- Voltemos à analogia do labirinto
  - É como se pudéssemos mandar um macaco treinado em busca de uma pista
  - Não precisamos efetivamente ir adiante (e voltar)
    - O que é custoso (retrocesso/backtracking)
- A técnica LL(*) consiste em gerar automaticamente o método lookAhead
  - Com base na definição da gramática
- Vamos então estudar como isso é feito

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(*)

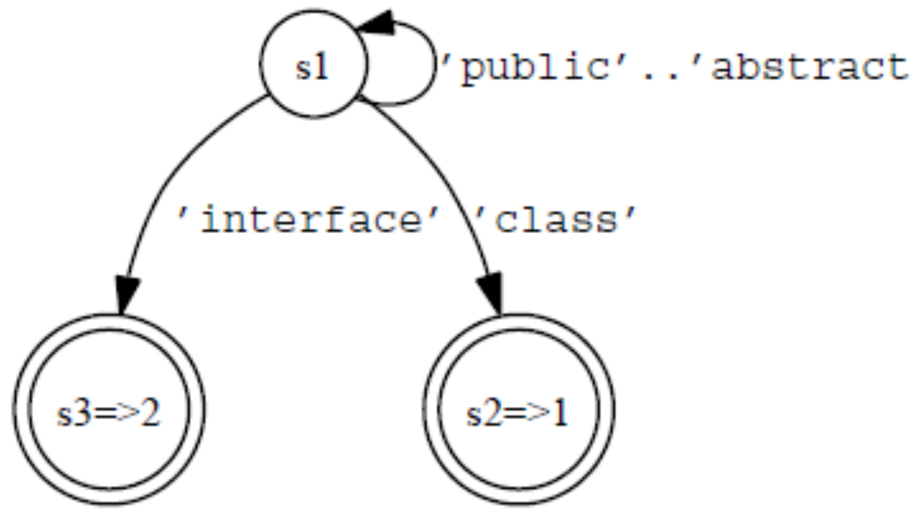
- Faz sentido então pensarmos em um DFA de decisão
  - Um modelo mais simples, não chega a ser uma gramática livre de contexto
    - De fato, é uma gramática regular
  - Serve apenas para tomar a decisão
- Importante:
  - Em análise LL(k), com k fixo, o DFA é sempre ACÍCLICO!!!
    - Ou seja, o macaco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

- Possível código para predição

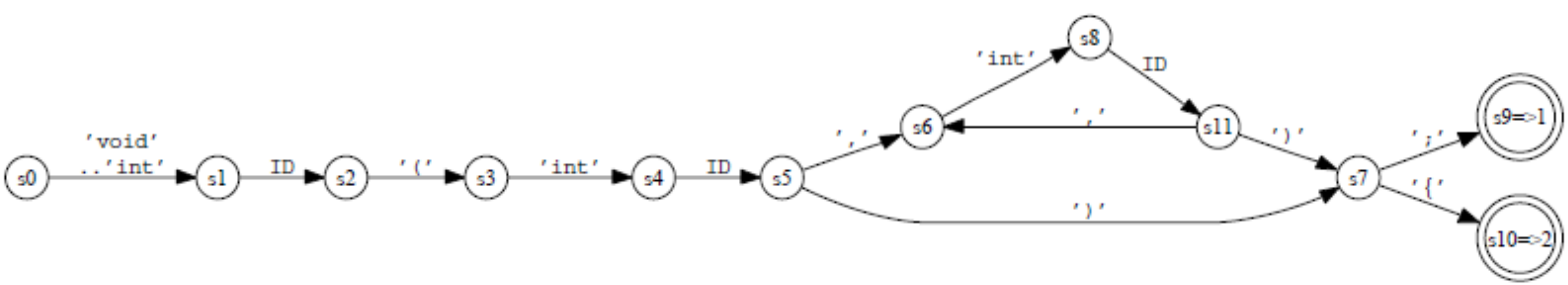
```
void def() {
  int alt=0;
  while (LA(1) in modifier) consume(); // scan past modifiers
  if ( LA(1)==CLASS ) alt=1;           // 'class'?
  else if ( LA(1)==INTERFACE ) alt=2;  // 'interface'?
  switch (alt) {
    case 1 : ...
    case 2 : ...
    default : error;
  }
}
```

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:



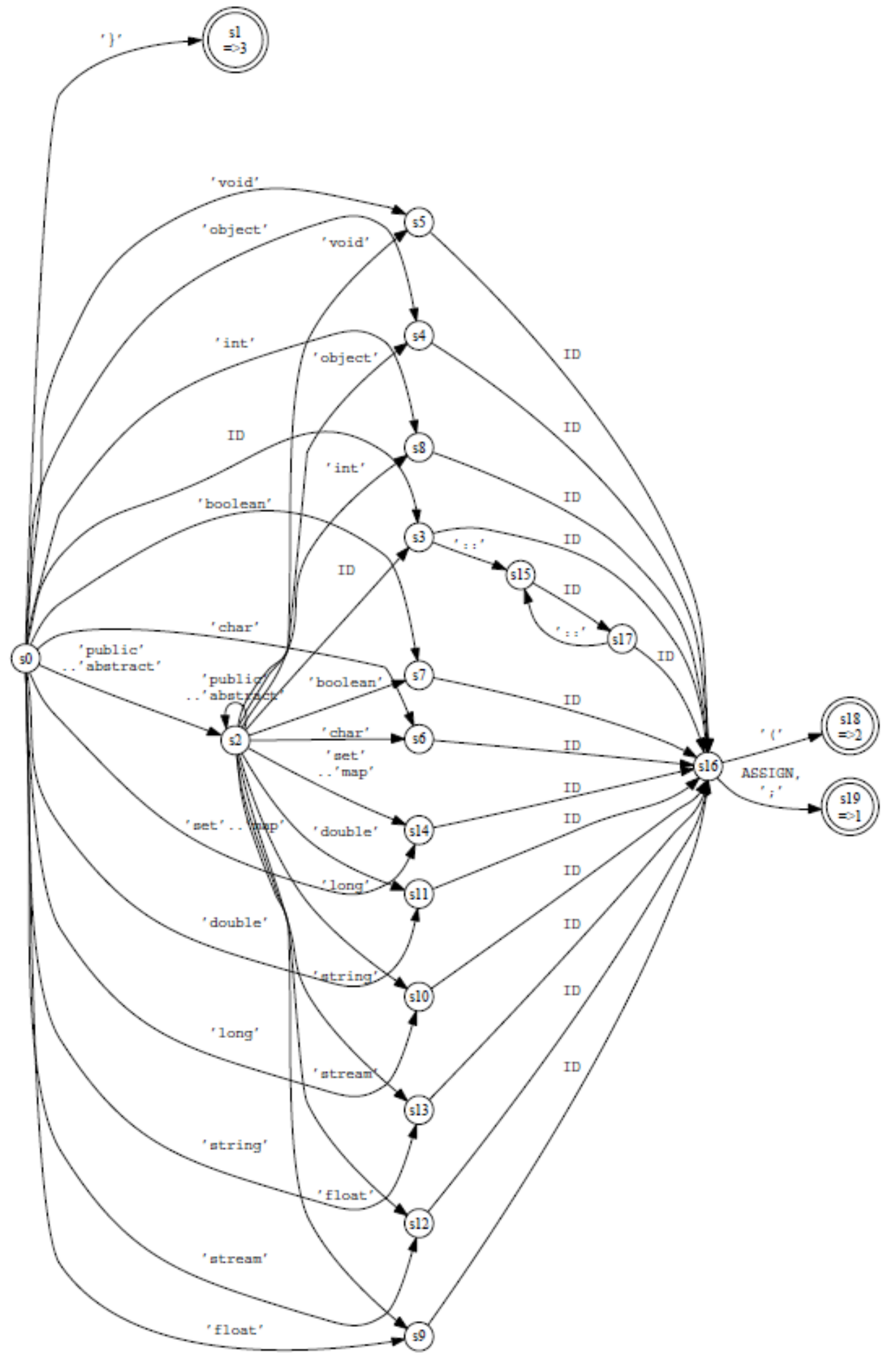
LL(*)

- Mais um exemplo

```
interfaceDef
    :   'interface' ID ('extends' classname)?
        '{'
        (   variableDefinition
        |   methodDefinition
        )*
        '}'
    ;
```

LL(*)

- DFA de decisão para a gramática do slide anterior



LL(*)

- O DFA de decisão pode ser arbitrariamente complexo
  - Não tem importância, a execução de um DFA é rápida
  - Além disso, o DFA está apenas tentanto predizer a regra
    - Não está executando ações, fazendo chamadas recursivas, o que torna o analisador sintático mais “pesado”
    - É a diferença entre mandar um macaco e ir você mesmo
- Mas eventualmente, ele chegará a um ponto de decisão
  - Um único token que diferencia entre as regras

LL(*)

- Formalmente, a técnica LL(*) usa um DFA que:
  - Reconhece a linguagem de decisão
    - Que deve ser uma linguagem regular!!
  - Não possui estados "mortos"
  - Possui pelo menos um estado de aceitação para cada regra alternativa
- Em outras palavras:
  - Dada uma gramática livre de contexto
  - Sempre que houver mais de uma alternativa
  - Gera-se uma expressão regular que representa a linguagem de decisão para cada alternativa
    - E constrói-se um DFA de decisão

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)

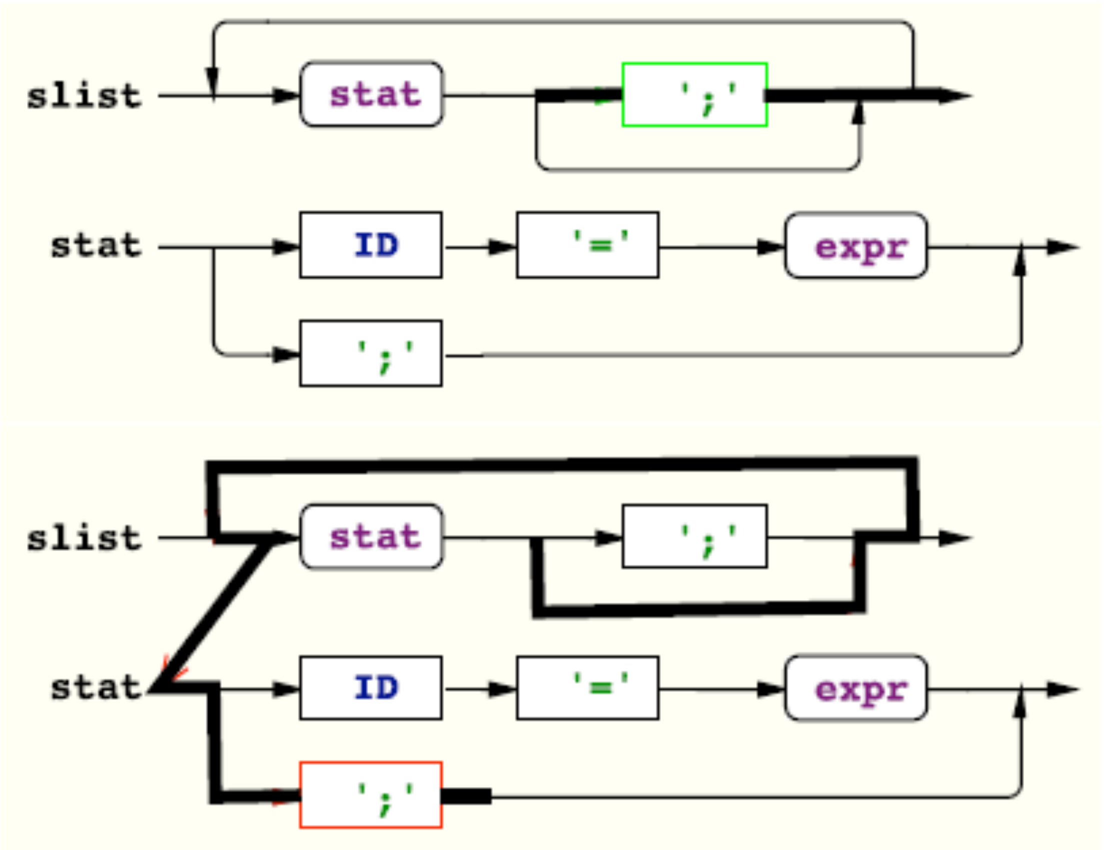
Gramáticas não-LL(*)

- Mas as gramáticas LL(*) têm limitações
- Duas categorias
  - Incompatibilidade com análise LL ou LL(*)
    - Recursividade à esquerda não é tolerada
    - Linguagem de predição não é regular
    - Normalmente problemas de uso da recursividade
      - Operadores EBNF (*, +) podem evitar estes problemas
  - Existência de não-determinismo no DFA de decisão
    - Ambiguidades na gramática

Gramáticas não-LL(*)

- Exemplo

```
slist
    :   ( stat ';'? )+
    ;
stat:   ID '=' expr
    |   ';'
    ;
```



LL(*)

- Em resumo, a técnica é bastante poderosa
  - E bastante utilizada
  - Graças ao ANTLR
    - Gerador de analisadores baseado na técnica LL(*)
    - De fato, foi o seu criador quem desenvolveu a técnica
- Veremos mais sobre o ANTLR a seguir

ANTLR

ANTLR

- ANTLR é um gerador baseado na técnica LL(*)
- Gera analisadores sintáticos preditivos de descendência recursiva
  - Facilidade de manutenção/legibilidade
- Suporte a várias linguagens
- Suporte a retrocesso, predicados sintáticos/semânticos
- Suporte para geração de árvores de sintaxe abstrata
- Demonstração

Fim