Parsing preditivo (top-down)

Ideia: olhando apenas para o próximo token no input decide-se qual a regra (derivação) a fazer sem nunca necessitar de fazer backtracking.

- É possível apenas se a gramática não for
  - recursiva à esquerda
  - e estiver factorizada à esquerda

- Uma gramática é recursiva à esquerda se existir um símbolo não terminal A tal que

$$A \overset{+}{\Rightarrow} A\alpha$$

para uma string α.

- Exemplo:

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

É recursiva à esquerda: $E \Rightarrow E + \underbrace{T}_{\alpha}$

A mesma coisa para T.

Como eliminar a recursividade à esquerda?

Intuição: o que é E?

resposta: $T + \dots + T$

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

Fazendo a mesma coisa para T obtém-se

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

↳ esta gramática é equivalente à gramática inicial (i.e, geram a mesma linguagem)

↳ não tem recursividade à esquerda, mas é mais complicada (menos intuitiva) que a inicial.

- Este truque resolve a maioria das situações (as imediatas, obtidas numa só derivação) mas há uma técnica mais geral. Ver detalhes no livro do dragão.

- Exemplo de recursividade à esq. não imediata:

$$S \to Aa \mid b$$
$$A \to Ac \mid Sd \mid \varepsilon$$

$$S \Rightarrow Aa \Rightarrow Sda$$

Factorização à esquerda

- Se um símbolo não terminal tiver mais de uma regra que comece com o mesmo símbolo, a gramática não está factorizada à esquerda.

- É problemático porque o parser preditivo não sabe qual das alternativas escolher.

- Exemplo: $E \rightarrow T + E \mid T$

(ambas começam com T)

$E \rightarrow T\ X$

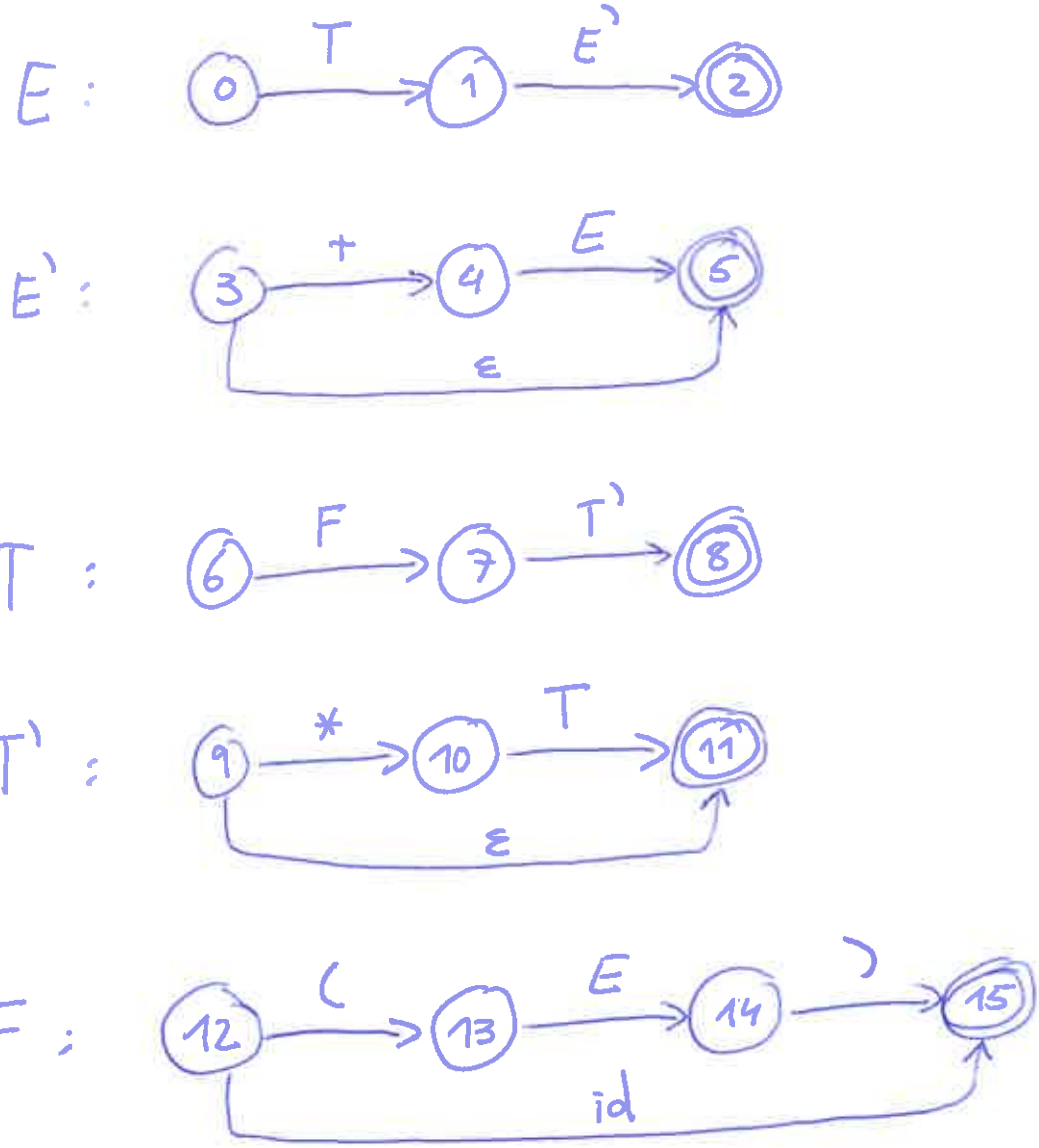
$X \rightarrow + E \mid \varepsilon$

- Após eliminar a recursividade à esquerda e de garantirmos que a gramática está factorizada à esquerda, podemos fazer um parser preditivo.

$$
\begin{aligned}
E &\to TE' \\
E' &\to +E \mid \varepsilon \\
T &\to FT' \\
T' &\to *T \mid \varepsilon \\
F &\to (E) \mid id
\end{aligned}
$$

- Podemos usar um diagrama de transição como um plano para construir o parser preditivo.

- Para cada não terminal A fazer:
  - Criar um estado inicial e final (de retorno)
  - Para cada regra $A \to X_1 X_2 \dots X_n$ criar um caminho do estado inicial até ao estado final com os arcos etiquetados com $X_1, X_2, \dots, X_n$.



Interpretação do diagrama

- É diferente de um autómato de uma Exp. Reg.
- Os arcos aqui têm etiquetas (labels) que podem ser símbolos terminais ou não terminais.

(cont.)

- As transições nos símbolos terminais (tokens) avançam o input ou dão erro.
- As transições nos símbolos não terminais correspondem a chamadas de funções.
- O parser começa no estado inicial do símbolo inicial. (Estado 0 no exemplo.)
- Se estivermos num estado s com um arco etiquetado pelo token a para o estado t, e se o próximo token no input for a, então o parser vai para o estado t e avança o input para o próximo token.

Input: a...
↑

Se o arco estiver etiquetado com o não terminal A, o parser vai para o estado inicial do símbolo A sem avançar o input.

quando chegar ao estado final de A, vai imediatamente para o estado t, tendo efectivamente "lido" algo que foi derivado de A.

- Se houver um arco de s para t etiquetado por ε, o parser vai para o estado t sem avançar o input.

- Apenas seguimos ε se não for possível fazer "match" do token que está no input.

```
Parser () {
    currToken = scan();
    E();
    if ( currToken == EOT )
        print "Accepted"
    else
        print "Not Accepted"
}
```

```
E() {
    T();
    E'();
}
```

```
T() {
    F();
    T'();
}
```

```
E'() {
    if ( currToken == PLUS ) {
        match (PLUS);
        E();
    }
}
```

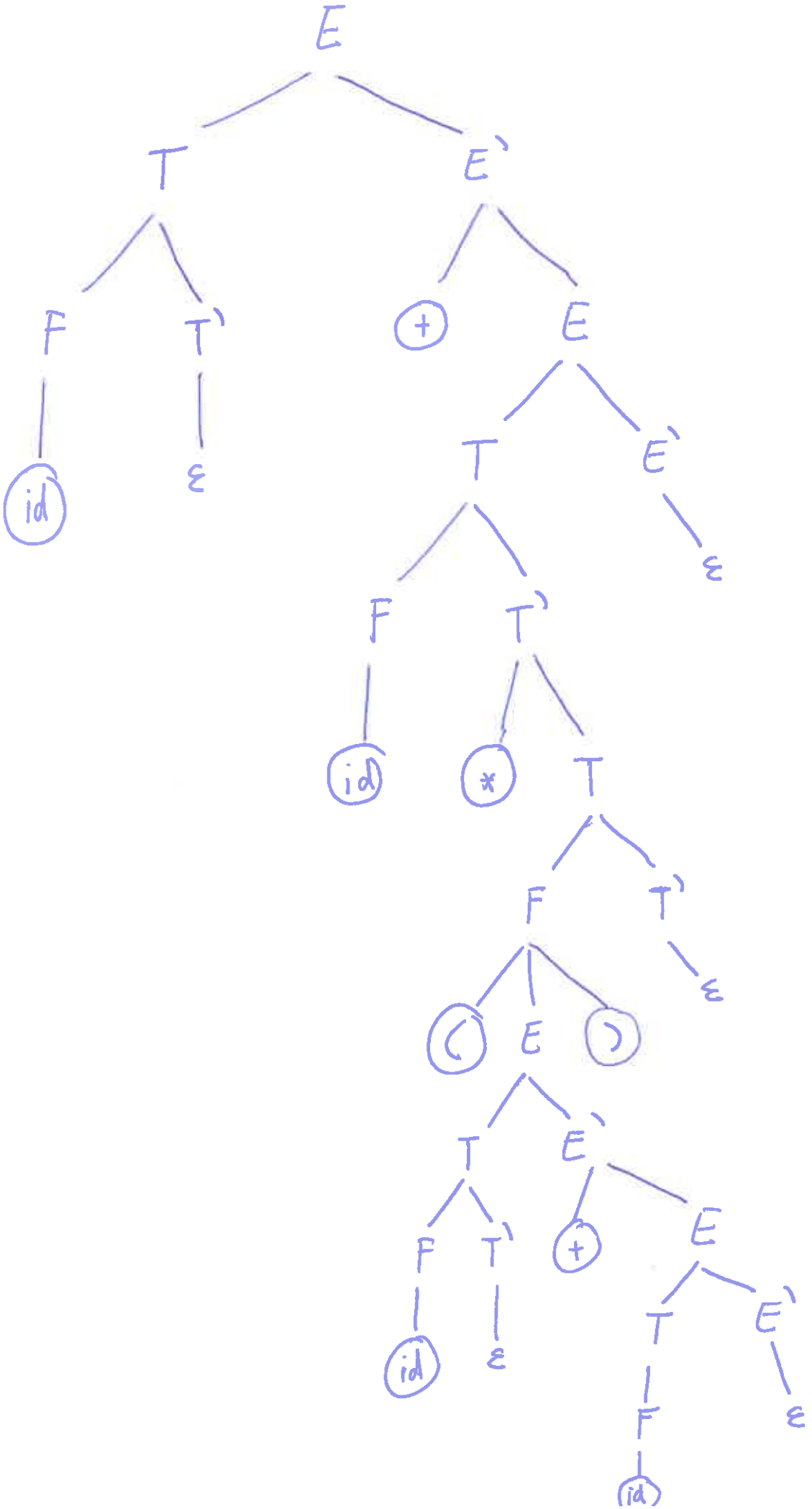
```
T'() {
    if ( currToken == TIMES ) {
        match (TIMES);
        T();
    }
}
```

```
F() {
    switch ( currToken) {
        case LPAREN: match (LPAREN);
                     E();
                     match (RPAREN);
                     break;
        case ID :    match (ID);
                     break;
        default :    print "Error: ( or id expected";
    }
}
```

```
match ( TOKEN tok ) {
    if ( curToken == tok )
        curToken = scan();
    else
        print "Error: token" + tok + "expected";
}
```

Experimentem programar isto em C ou Java. Para simplificar façam id ser um caracter único i. Usem $ como EOT.

Sample input: $i + i * (i + i)$ $


E
T
E`
F
T`
+
E
id
ε
T
E`
F
T`
ε
id
*
T
F
T`
ε
(
E
)
T
E`
F
T`
+
E
id
ε
T
E`
F
ε
id