# Compilers

## module of the course “Professional English”

Yulia Burkatovskaya

Department of Computer Engineering

Associate professor

# 4. Top-down parsing

Top-down parsing = depth-first search = leftmost derivation

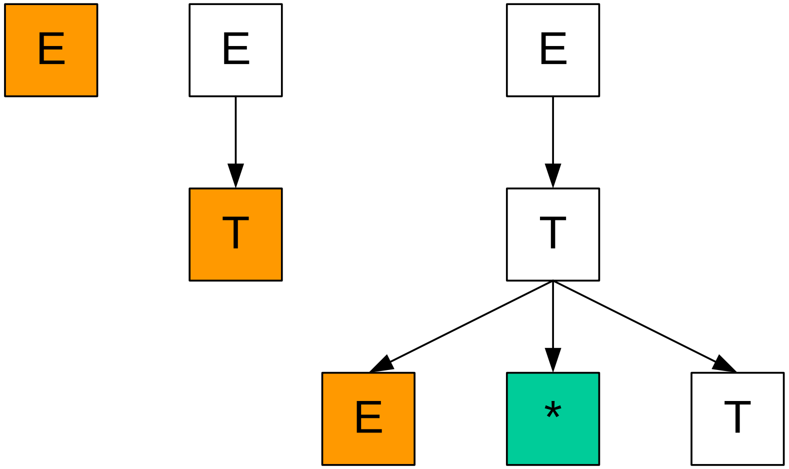
E → T

T → E+T

T → E*T

T → ID

T → (E)

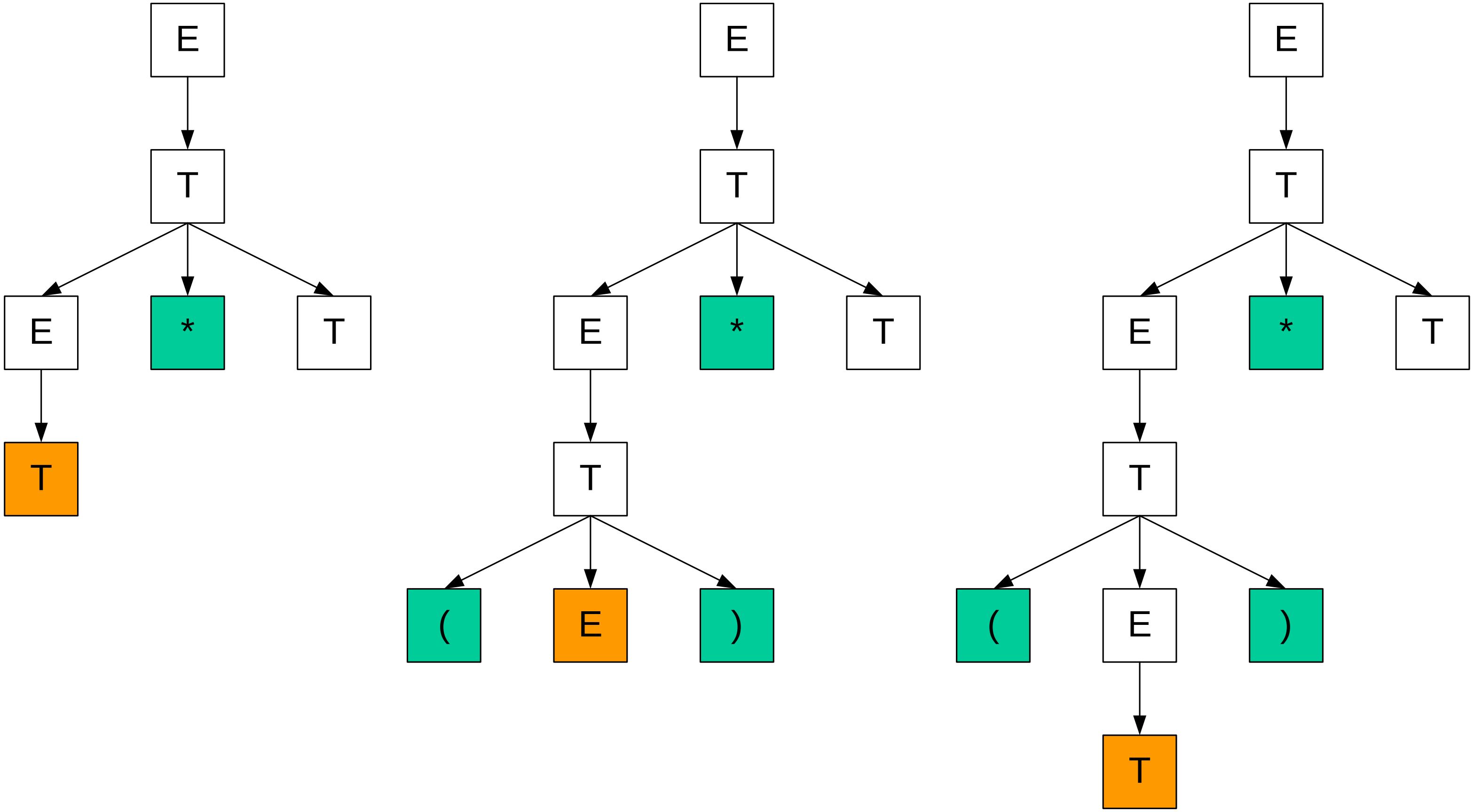


E → T → E*T → T*T → (E)*T → (T)*T → (E+T)*T → (T+T)*T

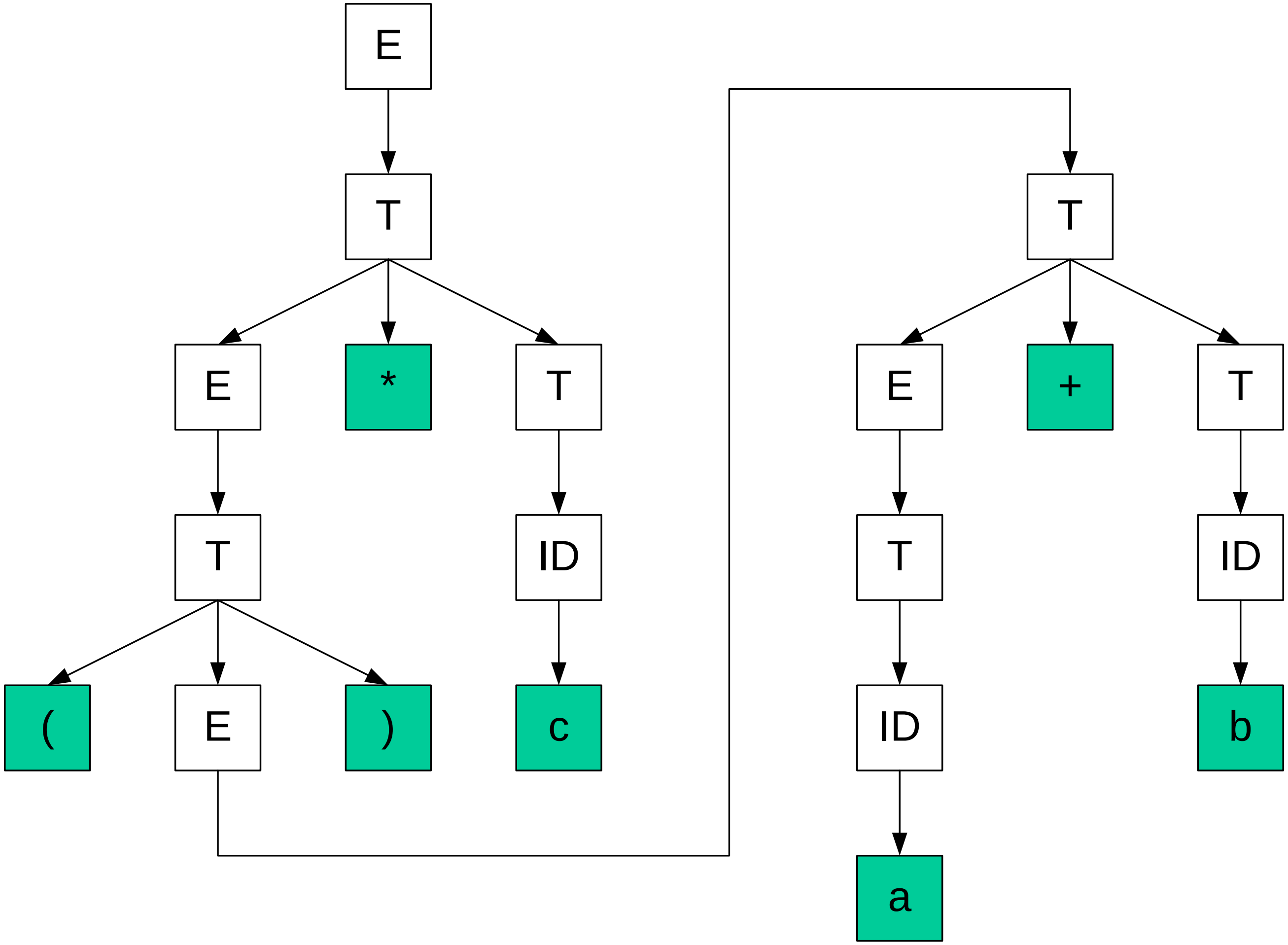
→ (ID+T)*T → (ID+ID)*ID → (a+ID)*T → (a+b)*T → (a+b)*ID

→ (a+b)*c

# Top-down parsing

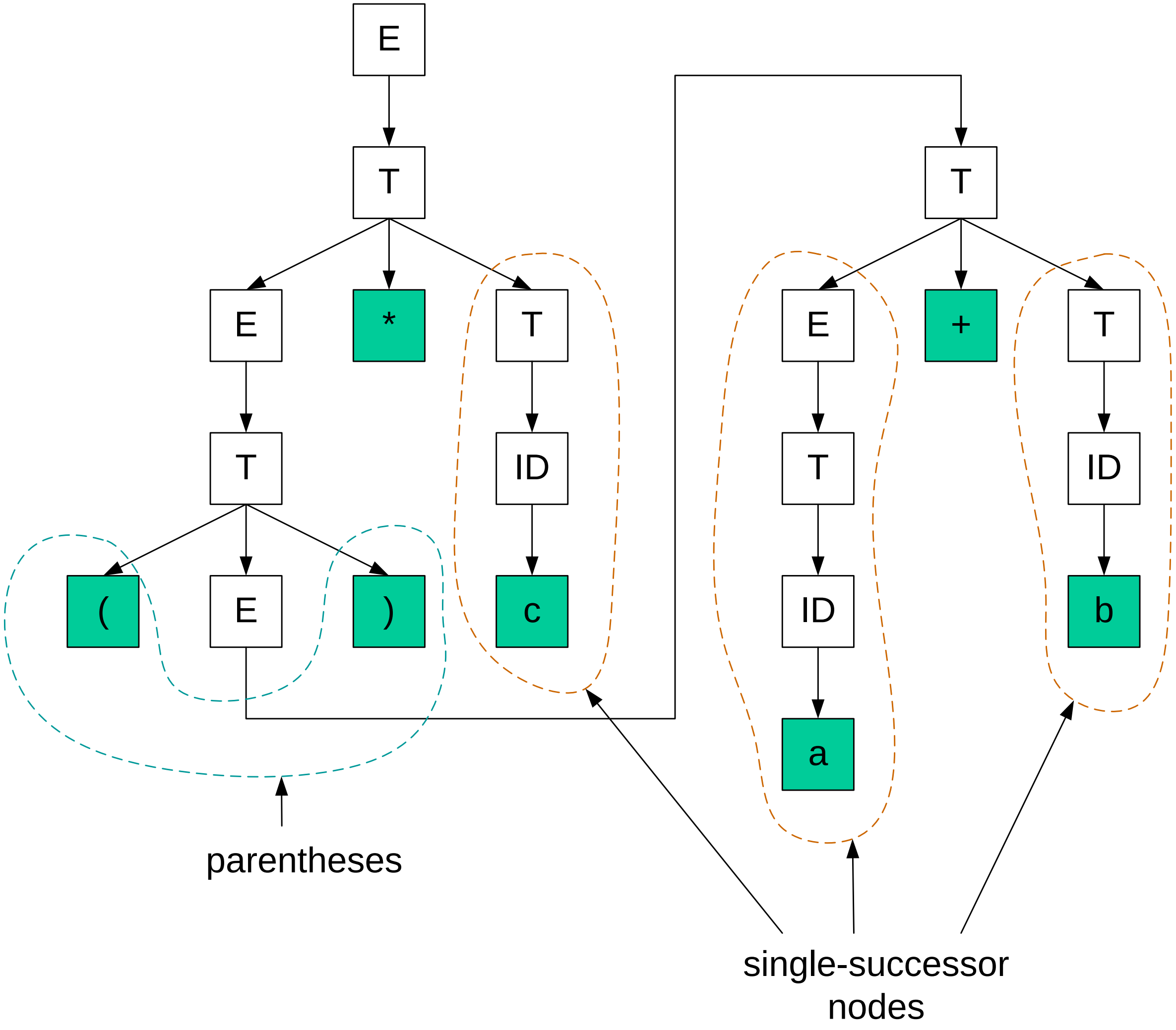


# Top-down parsing



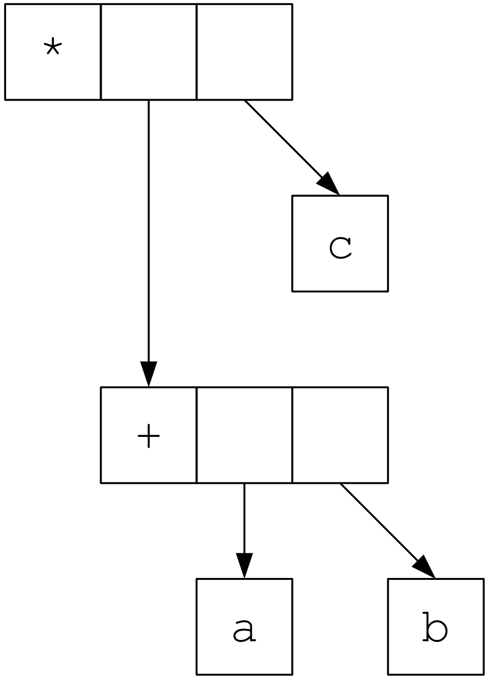
# Top-down parsing

Extra information:



# Top-down parsing

**Abstract syntax tree**

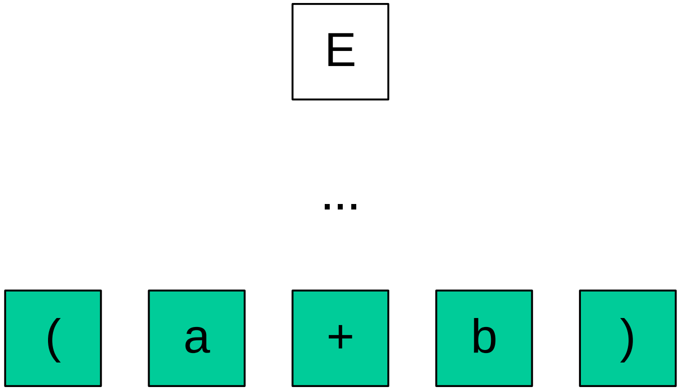


- Abstracts from the concrete grammar
- More compact and easy to use

# 4.1. Recursive-descent parsing

**Recursive-descent parsing:**

- from the top;
- from left to right;
- probably using backtracking.



E → T

T → E+T

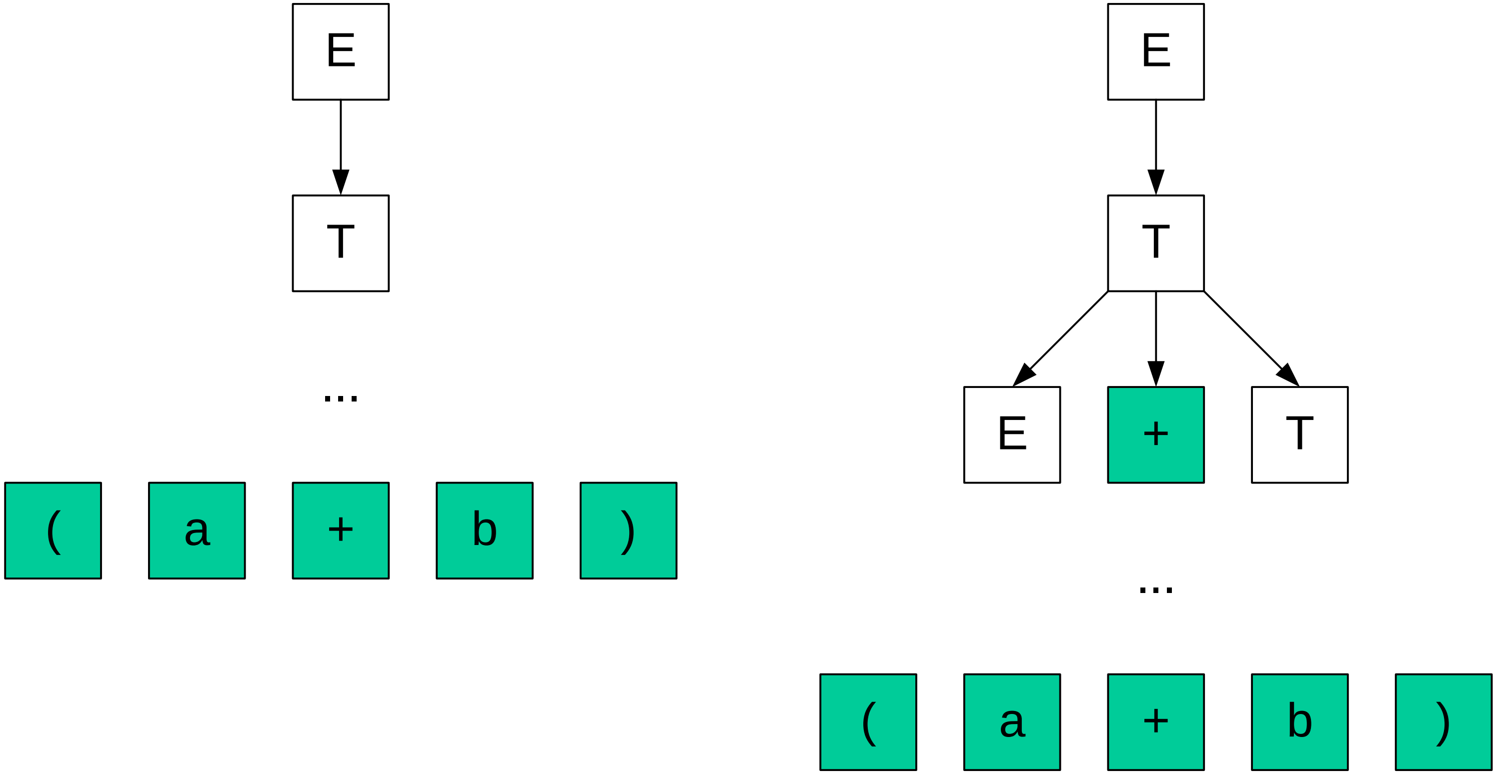
T → E*T

T → ID

T → (E)

ID → a | … | z

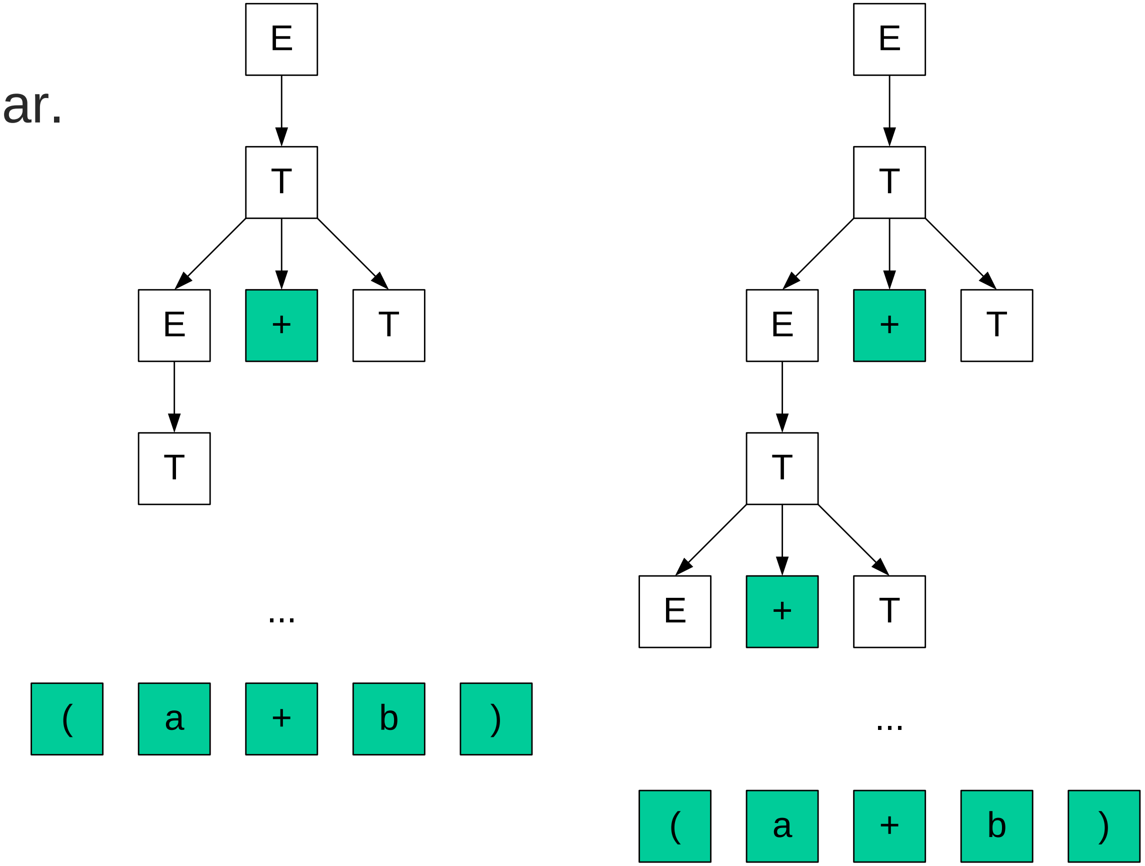
# Recursive-descent parsing



# Recursive-descent parsing

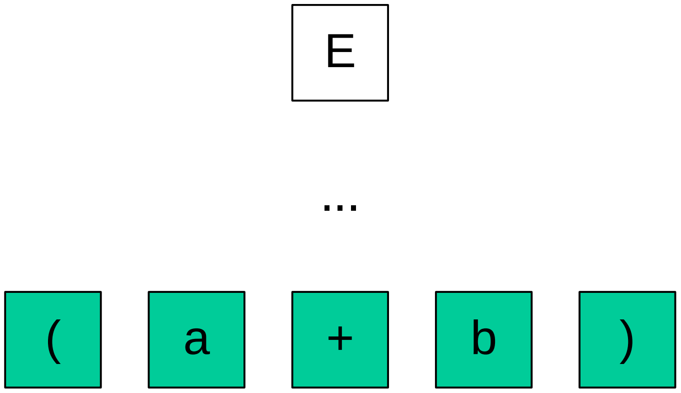
Loop!

Left-recursive grammar.

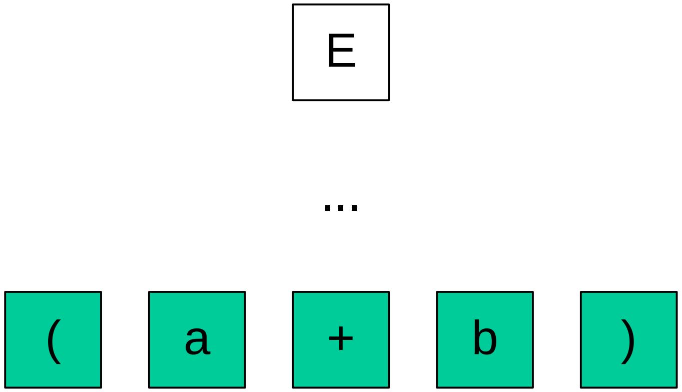


# Recursive-descent parsing

- E → TE’ | T
- E’ → +TE’ | +T
- T → FT’ | F
- T’ → *FT’ | *F
- F → (E) | ID
- ID → a | … | z



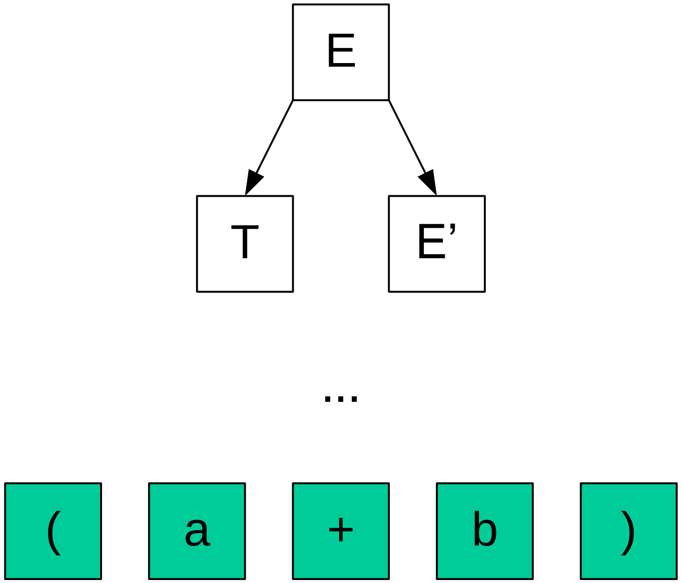
# Recursive-descent parsing



E

…

( a + b )



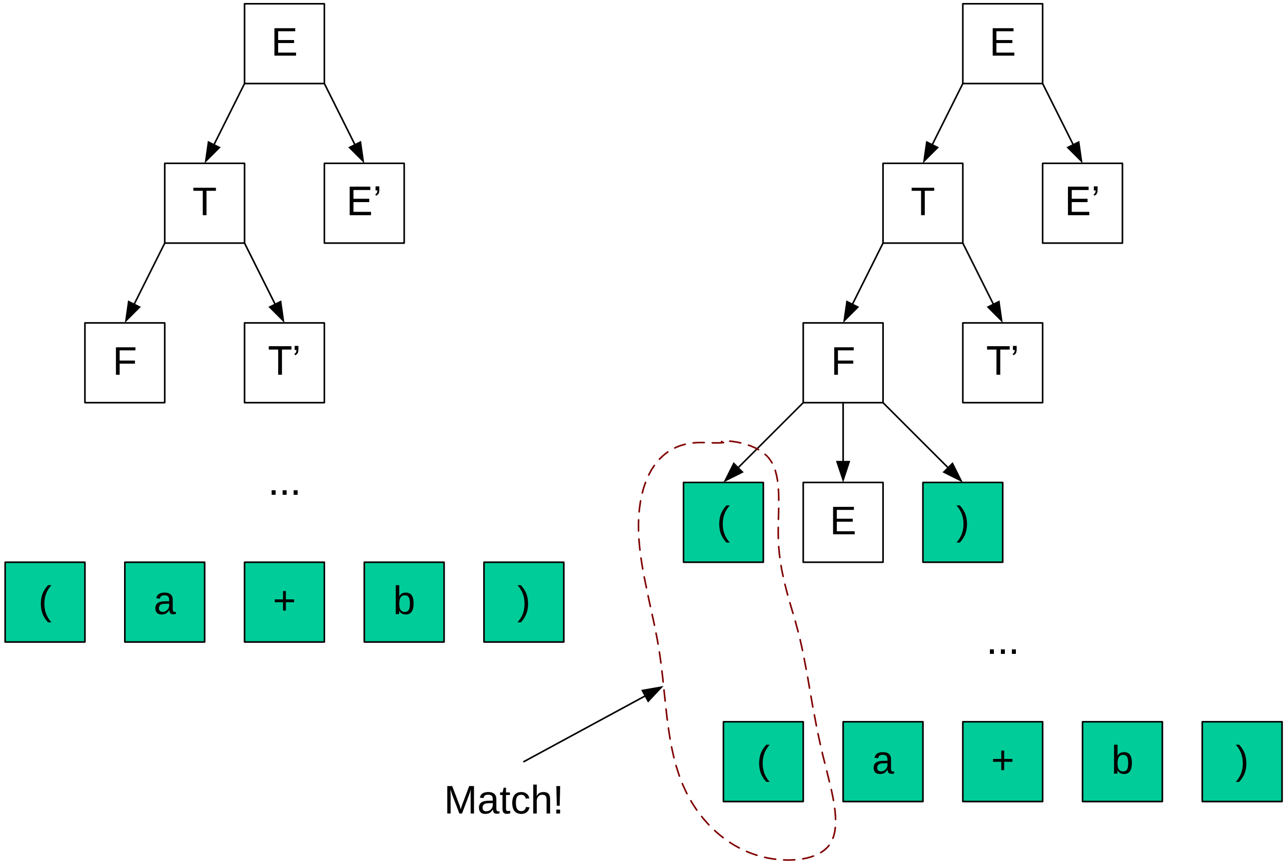
E

T E’

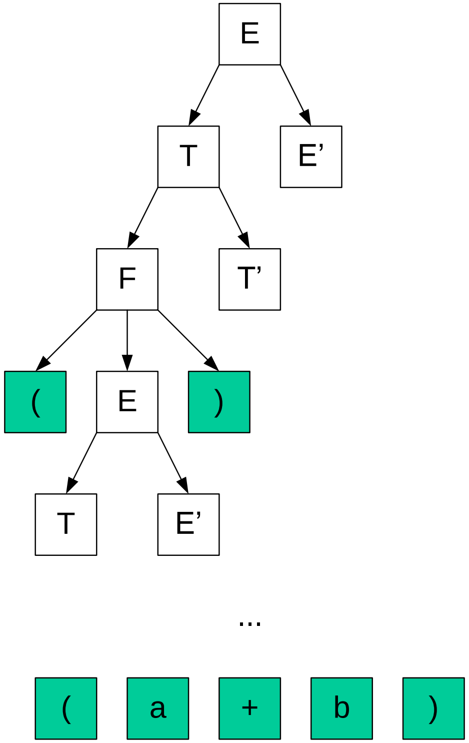
…

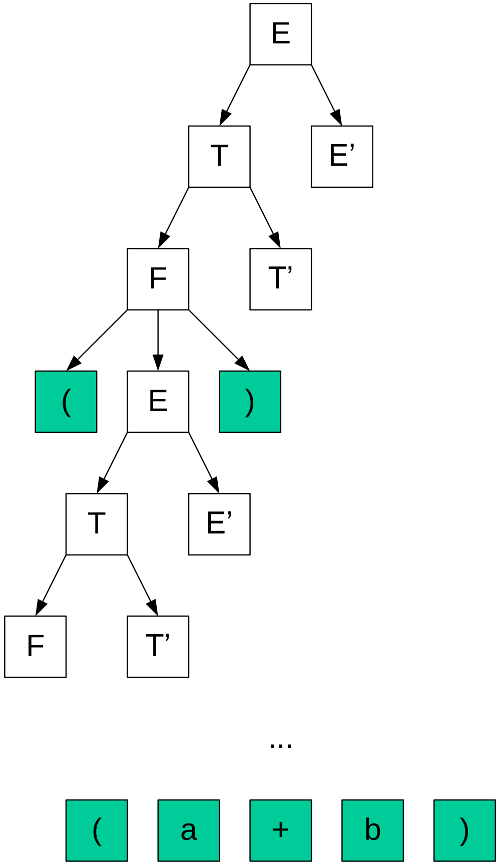
( a + b )

# Recursive-descent parsing

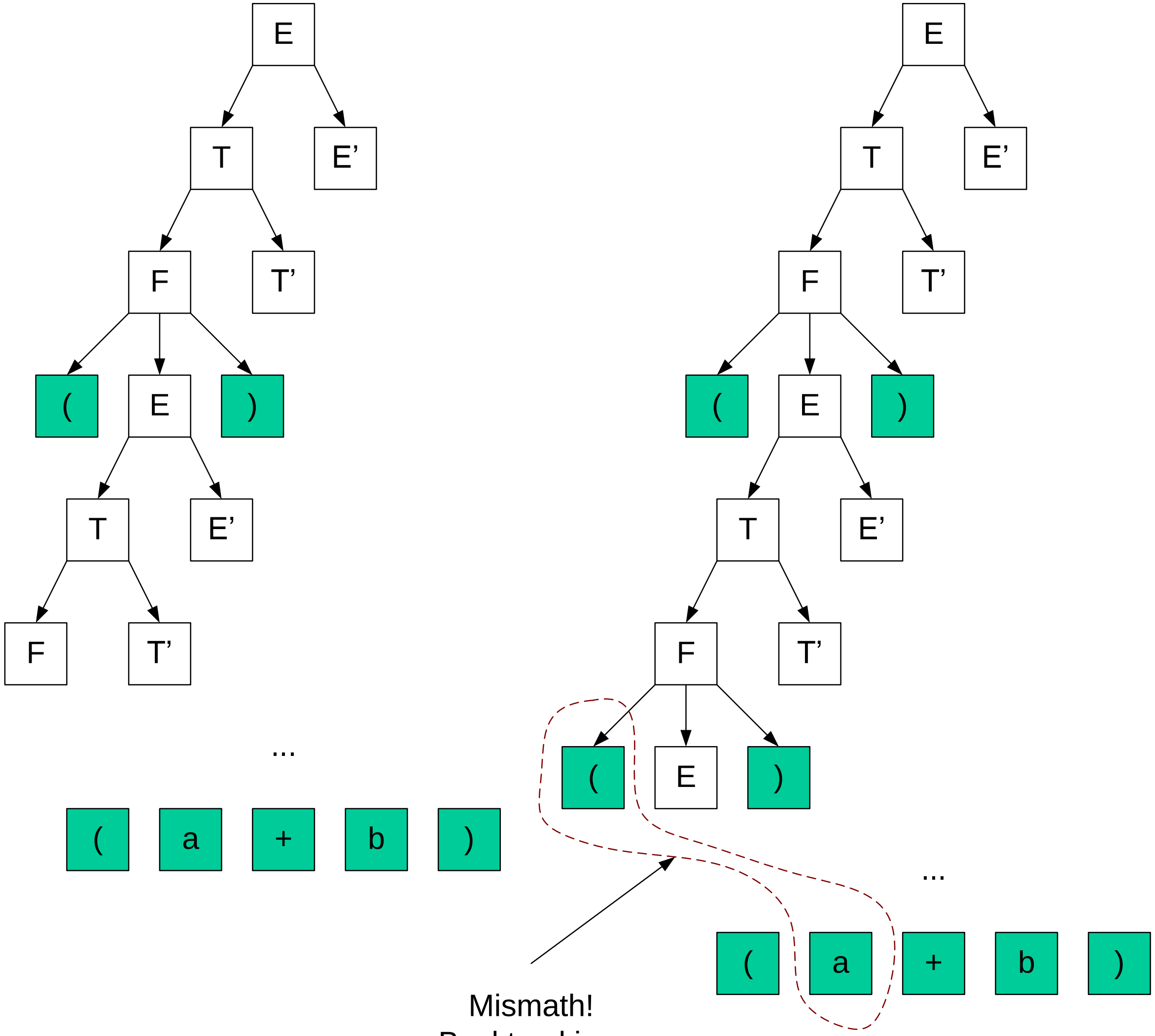


# Recursive-descent parsing



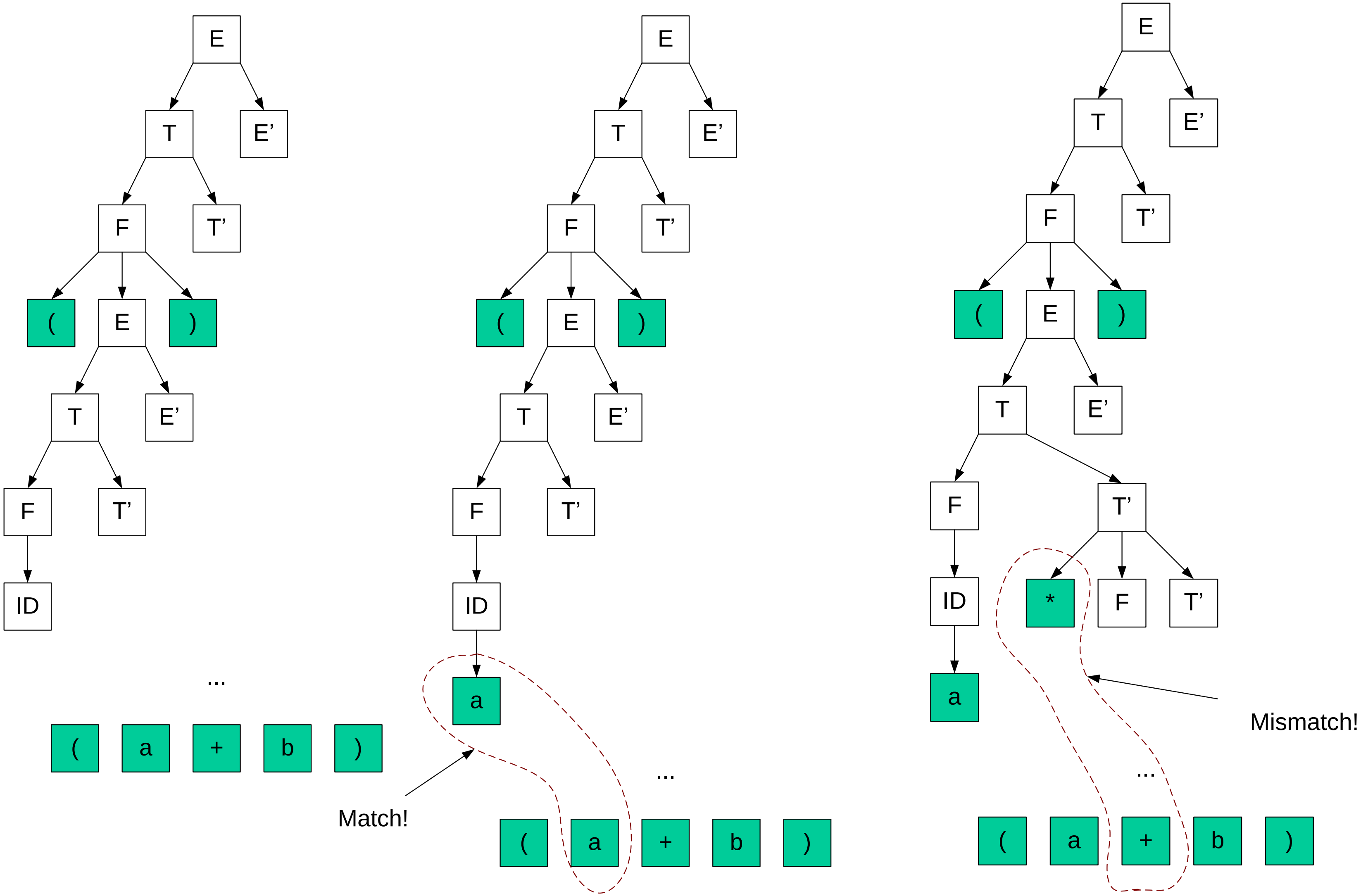


# Recursive-descent parsing



Mismath!
Backtracking.

# Recursive-descent parsing



# 4.2. FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G.

- During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol.
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

# FIRST and FOLLOW

- Define **FIRST($\alpha$),** where a is any string of grammar symbols, to be the set of terminals that begin strings derived from $\alpha$. If $\alpha \rightarrow \ldots \rightarrow \varepsilon$, then $\varepsilon \in \text{FIRST}(\alpha)$.
- Define **FOLLOW(A),** for nonterminal A, to be the set of terminals *a* that can appear immediately to the right of A in some sentential form; that is, the set of terminals *a* such that there exists a derivation of the form $S \rightarrow \ldots \rightarrow \alpha A a \beta$, for some $\alpha$ and $\beta$. Note that there may have been symbols between A and a, at some time during the derivation, but if so, they derived $\varepsilon$ and disappeared. If A can be the rightmost symbol in some sentential form, then $\$ \in$ FOLLOW(A).

# FIRST and FOLLOW

- If X is a terminal, then FIRST(X) = {X}.
- If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k \in P$, $a \in \mathrm{FIRST}(Y_1)$, then place a in FIRST(X). if for some i, $a \in \mathrm{FIRST}(Y_i)$, and $\varepsilon \in \mathrm{FIRST}(Y_1), \ldots,$ $\varepsilon \in \mathrm{FIRST}(Y_{i-1})$, that is, $Y_1 Y_2 \ldots Y_k \rightarrow \varepsilon$. If $\varepsilon \in \mathrm{FIRST}(Y_1), \ldots,$ $\varepsilon \in \mathrm{FIRST}(Y_k)$, then add ε to FIRST(X). For example, everything in FIRST(Y1) is included into FIRST(X). If $Y_1$ does not derive ε, then we add nothing more to FIRST(X), but if $Y_1 \rightarrow \varepsilon \in P$, then we add $\mathrm{FIRST}(Y_2)$ , and so on.
- If $X \rightarrow \varepsilon \in P$, then add ε to FIRST(X).

# FIRST and FOLLOW

Now, we can compute FIRST for any string $X_1X_2\ldots X_n$ as follows.

- Add to FIRST($X_1X_2\ldots X_n$) all non-ε symbols of FIRST($X_1$) .
- Also add the non-ε symbols of FIRST($X_2$), if $\varepsilon \in$FIRST($X_1$); the non-ε symbols of FIRST($X_2$), if $\varepsilon \in$FIRST($X_1$) and $\varepsilon \in$FIRST($X_2$); and so on.
- Finally, add ε to FIRST($X_1X_2\ldots X_n$) if, for all i, $\varepsilon \in$FIRST($X_i$).

# FIRST and FOLLOW

- E → TE’ | T
- E’ → +TE’ | +T
- T → FT’ | F
- T’ → *FT’ | *F
- F → (E) | ID
- ID → a | … | z

- FIRST(ID)={a,…,z}
- FIRST(F)={a,…,z,(}
- FIRST(T’)={*}
- FIRST(T)={a,…,z,(}
- FIRST(E’)={+}
- FIRST(E)={a,…,z,(}

# FIRST and FOLLOW

- Place \$ in FOLLOW(S), where *S* is the start symbol, and \$ is the input right endmarker.
- If $A \rightarrow \alpha B\beta \in P$, then everything in FIRST(β) except ε is in FOLLOW(B).
- If $A \rightarrow \alpha B \in P$, or $A \rightarrow \alpha B\beta \in P$, where $\varepsilon \in$ FIRST(β), then everything in FOLLOW(A) is in FOLLOW(B) .

# FIRST and FOLLOW

- E → TE’ | T
- E’ → +TE’ | +T
- T → FT’ | F
- T’ → *FT’ | *F
- F → (E) | ID
- ID → a | … | z

- FIRST(ID)={a,…,z}
- FIRST(F)={a,…,z,(}
- FIRST(T’)={*}
- FIRST(T)={a,…,z,(}
- FIRST(E’)={+}
- FIRST(E)={a,…,z,(}
- FOLLOW(E)={$,)}
- FOLLOW(T)={+,$,)}
- FOLLOW(F)={*,+,$,)}
- FOLLOW(E’)={$,)}
- FOLLOW(T’)={+,$,)}
- FOLLOW(ID)={*,+,$,)}

# FIRST and FOLLOW

- E → TE’
- E’ → +TE’ | ε
- T → FT’
- T’ → *FT’ | ε
- F → (E) | ID
- ID → a | … | z

- FIRST(ID)={a,…,z}
- FIRST(F)={a,…,z,(}
- FIRST(T’)={ε,*}
- FIRST(T)={a,…,z,(}
- FIRST(E’)={ε,+}
- FIRST(E)={a,…,z,(}
- FOLLOW(E)={$,)}
- FOLLOW(T)={+,$,)}
- FOLLOW(F)={*,+,$,)}
- FOLLOW(E’)={$,)}
- FOLLOW(T’)={+,$,)}
- FOLLOW(ID)={*,+,$,)}

# 4.3. Predictive parsing table

For each production A → α of the grammar, do the following:

- For each terminal x∈FIRST(α), add A → α to M[A, x];
- If ε∈FIRST(α), then for each terminal y∈FOLLOW(A), add the production A→α to M[A,y]. If ε∈FIRST(α), and $∈FOLLOW(A), add A → α to M[A, $] as well.

If, after performing the above, there is no production at all in the cell M[A,x], then set M[A,x] to **error** (which we normally represent by an empty entry in the table).

# Predictive parsing table

- E → TE’ | T
- FIRST(T)={a,…,z,(}
- FIRST(E’)={+}
- FIRST(TE’)={a,…,z,(}

Add E → TE’  and E → T to:

- M[E,a]
- …
- M[E,z]
- M[E,(]

- E → TE’
- FIRST(T)={a,…,z,(}
- FIRST(E’)={ε,+}
- FIRST(TE’)={a,…,z,(}

Add E → TE’ to:

- M[E,a]
- …
- M[E,z]
- M[E,(]

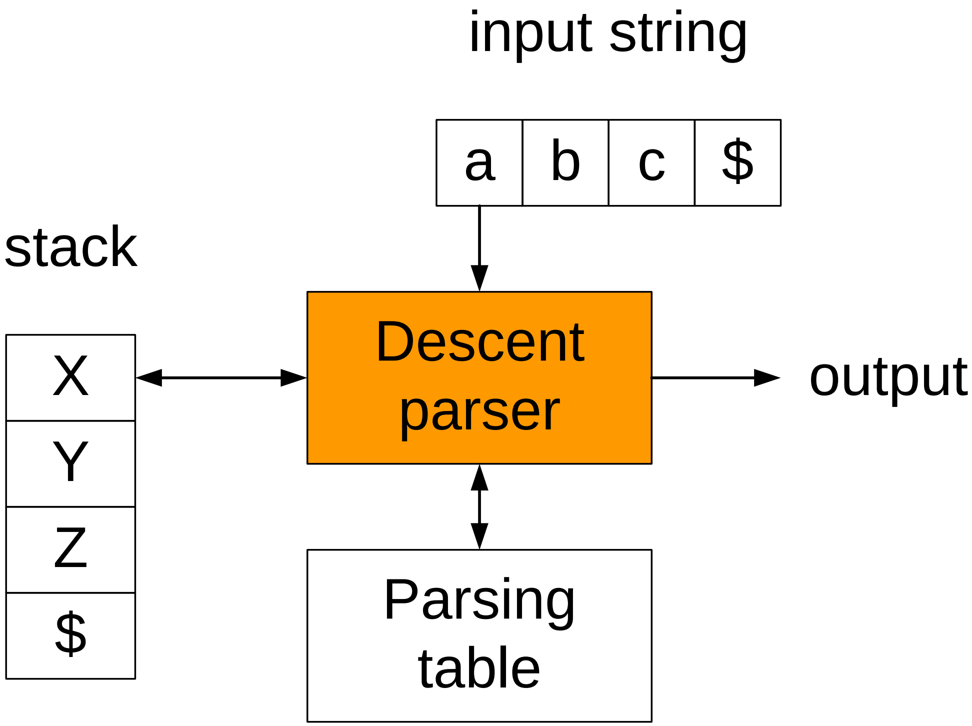
# Predictive parsing table

| | a | … | z | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|---|---|
| **E** | E→TE'<br>E→ T | | E→TE'<br>E→ T | | | E→ TE'<br>E→ T | | |
| **E'** | | | | E'→ +TE'<br>E'→ +T | | | | |
| **T** | T→ FT'<br>T→ F | | T→ FT'<br>T→ F | | | T→ FT'<br>T→ F | | |
| **T'** | | | | | T'→ *FT'<br>T'→ *F | | | |
| **F** | F→ ID | | F→ ID | | | F→(E) | | |
| **ID** | ID →a | | ID →z | | | | | |

# Predictive parsing table

| | **a** | **…** | **z** | **+** | ***** | **(** | **)** | **$** |
|---|---|---|---|---|---|---|---|---|
| **E** | E→TE’ | | E→TE’ | | | E→ TE’ | | |
| **E’** | | | | E’→ +TE’ | | | E’→ ε | E’→ ε |
| **T** | T→ FT’ | | T→ FT’ | | | T→ FT’ | | |
| **T’** | | | | | T’→ *FT’ | | T’→ ε | T’→ ε |
| **F** | F→ ID | | F→ ID | | | F→(E) | | |
| **ID** | ID →a | | ID →z | | | | | |

# 4.4. Non-recursive-descent parsing



# Non-recursive-descent parsing

- X – a symbol at the top of the stack, $ – the bottom marker (initially there is $S in the stack, S – the start symbol of the grammar);
- a – an input symbol ($ – the right position marker);
- M[A,a] – the  parsing table, A is a non-terminal, a is a terminal or $.

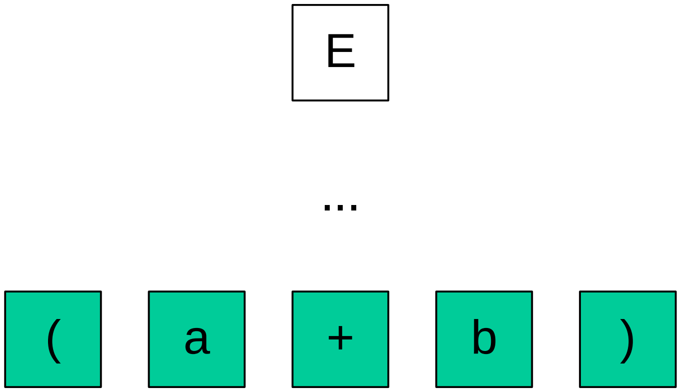
# Non-recursive-descent parsing

The descent parser observes X and a:

- if X=a=$ then parsing is successfully completed;
- if X=a≠$ then the parser removed X from the top of the stack and moves to the next input symbol;
- If X is a non-terminal then the row M[X,a] of the parsing table is considered. If M[X,a] = X→Z1…Zk then X is replaced by Zk…Z1 at the stack (Z1 is at the top). If M[X,a] = error then an error recovery program is called.
- If X is a terminal and X≠a then an error recovery program is called.

# Non-recursive-descent parsing

- E → TE’ | T
- E’ → +TE’ | +T
- T → FT’ | F
- T’ → *FT’ | *F
- F → (E) | ID
- ID → a | … | z



E

...

( a + b )

# Non-recursive-descent parsing

| | **a** | **…** | **z** | **+** | ***** | **(** | **)** | **$** |
|---|---|---|---|---|---|---|---|---|
| **E** | E→TE’<br>E→ T | | E→TE’<br>E→ T | | | E→ TE’<br>E→ T | | |
| **E’** | | | | E’→ +TE’<br>E’→ +T | | | | |
| **T** | T→ FT’<br>T→ F | | T→ FT’<br>T→ F | | | T→ FT’<br>T→ F | | |
| **T’** | | | | | T’→ *FT’<br>T’→ *F | | | |
| **F** | F→ ID | | F→ ID | | | F→(E) | | |
| **ID** | ID →a | | ID →z | | | | | |

# Non-recursive-descent parsing

| Stack | Input | Output |
|---|---|---|
| E$ | (a+b)$ | |
| T$ | (a+b)$ | E→T |
| F$ | (a+b)$ | T→F |
| (E)$ | (a+b)$ | F →(E) |
| E)$ | a+b)$ | |
| TE’)$ | a+b)$ | E →TE’ |
| FE’)$ | a+b)$ | T →F |
| IDE’)$ | a+b)$ | F →ID |

| Stack | Input | Output |
|---|---|---|
| aE’)$ | a+b)$ | ID →a |
| E’)$ | +b)$ | |
| +T)$ | +b)$ | E’ →+T |
| T)$ | b)$ | |
| ID)$ | b)$ | T →ID |
| b)$ | b)$ | ID →b |
| )$ | )$ | |
| $ | $ | |

# 4.5. LL(1) grammars

- L – scanning the input from left to right;
- L – producing a leftmost derivation;
- 1 – using one input symbol of lookahead at each step to make parsing action decisions.

- For every LL(1) grammar, each parsing-table entry uniquely identifies a production or signals an error.
- **No backtracking!**
- Although left recursion elimination and left factoring are easy to do, there are some grammars for which no amount of alteration will produce an LL(1) grammar.

# LL(1) grammars

**LL(1) grammar**

- E → TE’
- E’ → +TE’ | ε
- T → FT’
- T’ → *FT’ | ε
- F → (E) | ID
- ID → a | … | z

**For all A→α | β:**

- For no terminal x do both α and β derive strings beginning with x (FIRST(α)∩FIRST(β)=Ø);
- At most one of α and β can derive the empty string;
- If β derives ε, then α does not derive any string beginning with a terminal in FOLLOW (A).

# LL(1) grammars

| | **a** | **…** | **z** | **+** | ***** | **(** | **)** | **$** |
|---|---|---|---|---|---|---|---|---|
| **E** | E→TE’ | | E→TE’ | | | E→ TE’ | | |
| **E’** | | | | E’→ +TE’ | | | E’→ ε | E’→ ε |
| **T** | T→ FT’ | | T→ FT’ | | | T→ FT’ | | |
| **T’** | | | | | T’→ *FT’ | | T’→ ε | T’→ ε |
| **F** | F→ ID | | F→ ID | | | F→(E) | | |
| **ID** | ID →a | | ID →z | | | | | |

# LL(1) grammars

Not LL(1) grammar

- ST → if EXPR then ST ST’ | a
- ST’ → else ST | ε
- EXPR → b

- FIRST(EXPR)={b}
- FIRST(ST’)={else, ε}
- FIRST(ST)={if,a}
- FOLLOW(ST)={$,else}
- FOLLOW(ST’)={$,else}
- FOLLOW(EXPR)={b,then}

# LL(1) grammars

| | **a** | **b** | **if** | **else** | **$** |
|---|---|---|---|---|---|
| **ST** | ST→a | | ST → if EXPR then ST ST’ | | |
| **ST’** | | | | **ST’ → else ST**<br>**ST ’ → ε** | ST ’ → ε |
| **EXPR** | | EXPR → b | | | |

# 4.6. Error recovery in predictive parsing

An **error** is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal **A** is on top of the stack, **x** is the next input symbol, and **M[A,a]** is **error** (i.e., the parsing-table entry is empty).

- **Panic Mode**

Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice**.**

# Error recovery in predictive parsing

**Some heuristics are as follows:**

- As a starting point, place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.
- It is not enough to use FOLLOW(A) as the synchronizing set for *A.* For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal representing expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.

# Error recovery in predictive parsing

- If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.
- If a nonterminal can generate the empty string, then the production deriving ε can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
- If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

# Error recovery in predictive parsing

- E → TE’
- E’ → +TE’ | ε
- T → FT’
- T’ → *FT’ | ε
- F → (E) | ID
- ID → a | … | z

- FIRST(ID)={a,…,z}
- FIRST(F)={a,…,z,(}
- FIRST(T’)={ε,*}
- FIRST(T)={a,…,z,(}
- FIRST(E’)={ε,+}
- FIRST(E)={a,…,z,(}
- FOLLOW(E)={$,)}
- FOLLOW(T)={+,$,)}
- FOLLOW(F)={*,+,$,)}
- FOLLOW(E’)={$,)}
- FOLLOW(T’)={+,$,)}
- FOLLOW(ID)={*,+,$,)}

# Error recovery in predictive parsing

| | **a** | **…** | **z** | **+** | ***** | **(** | **)** | **$** |
|---|---|---|---|---|---|---|---|---|
| **E** | E→TE’ | | E→TE’ | | | E→ TE’ | Synch | Synch |
| **E’** | | | | E’→ +TE’ | | | E’→ ε | E’→ ε |
| **T** | T→ FT’ | | T→ FT’ | Synch | | T→ FT’ | Synch | Synch |
| **T’** | | | | | T’→ *FT’ | | T’→ ε | T’→ ε |
| **F** | F→ ID | | F→ ID | Synch | Synch | F→(E) | Synch | Synch |
| **ID** | ID →a | | ID →z | Synch | Synch | | Synch | Synch |

# Error recovery in predictive parsing

- Here **"synch"** indicating synchronizing tokens obtained from the FOLLOW set of the nonterminal.
- If the parser looks up entry M[A,x] and finds that it is blank, then the input symbol x is skipped.
- If the entry is "synch," then the nonterminal on top of the stack is popped in an attempt to resume parsing.
- If a token on top of the stack does not match the input symbol, then we pop the token from the stack, as mentioned above.

# Error recovery in predictive parsing

| Stack | Input | Output |
|---|---|---|
| E$ | )a*+b$ | synch |
| E$ | a*+b$ | E → TE’ |
| TE’$ | a*+b$ | T → FT’ |
| FT’E’$ | a*+b$ | F → ID |
| IDT’E’$ | a*+b$ | ID → a |
| aT’E’$ | a*+b$ | |
| T’E’$ | *+b$ | T’→ *FT’ |
| *FT’E’$ | *+b$ | |

| Stack | Input | Output |
|---|---|---|
| FT’E’$ | +b$ | synch |
| FT’E’$ | b$ | F → ID |
| IDT’E’$ | b$ | ID → b |
| bT’E’$ | b$ | |
| T’E’$ | $ | T’→ ε |
| E’$ | $ | E’→ ε |
| $ | $ | |
| | | |