Top Down Parsing

TEACHING ASSISTANT: DAVID TRABISH

Examples

```
void f(int a) {
     if ((8)) {

     }
}
```

Examples

```
void f(int a) {
     if ((8)) {

     }
}
```

Valid

Examples

```
void f(int a) {
     if ((8))) {

     }
}
```

Examples

```
void f(int a) {
     if ((8))) {

     }
}
```

Invalid

Examples

```
if ((8)) {

}
```

Examples

```
if ((8)) {

}
```

Invalid

Examples

```
void f() {
        int a[];
}
```

Examples

```
void f() {
        int a[];
}
```

Invalid

Examples

```
void f() {
        int a[10.0];
}
```

Examples

```
void f() {
        int a[10.0];
}
```

Invalid

Examples

```
void f(int a[]) {

}
```

Examples

```
void f(int a[]) {

}
```

Valid

Examples

```
void f() {
        int i = 0;
        int j = 1;
        j + i;
}
```

Examples

```
void f() {
        int i = 0;
        int j = 1;
        j + i;
}
```

Valid

Examples

```
void f() {
        int i = 0;
        j + i;
        int j = 1;
}
```

Examples

```
void f() {
        int i = 0;
        j + i;
        int j = 1;
}
```

Valid

Language of Balanced Parentheses

Contains string of the form:

- 8, (1), (((0))), …

Disallowing:

- ((1), 8()

Language of Balanced Parentheses

Contains string of the form:

- 8, (1), (((0))), …

Disallowing:

- ((1), 8()

Is there a DFA/NFA that accepts the language?

Is there a regular expression the accepts the language?

Language of Balanced Parentheses

The language is **not regular**

- There is no DFA that accepts it

Proof:

- If it has a DFA, the we have d states
- Consider the input $((\ldots 77$ that has $d+1$ left parentheses
- Every time we read (, we need to change to a new state
    - We need to act differently if we saw 4 parentheses or 10
- But we have only d states…

Context Free Grammar

- A set of terminals T and a set of non-terminals V
- Production rules of the form
    - $A \rightarrow a_1 a_2 \dots a_n$
    - $A \in V, a_i \in T \cup V$
- Starting symbol S :
    - $S \rightarrow a_1 a_2 \dots a_n$

Context Free Grammar

Example:

- $S \rightarrow c$
- $S \rightarrow aSb$

Which words belong to this grammar?

Context Free Grammar

Example:

- $S \rightarrow c$
- $S \rightarrow aSb$

Which words belong to this grammar?

- $c, acb, aacbb, aaacbbb, \ldots$

Context Free Grammar

Does the language of **balanced parentheses** have a CFG?

Context Free Grammar

Does the language of **balanced parentheses** have a CFG?

- $S \rightarrow N$
- $S \rightarrow (S)$

Context Free Grammar: Questions

Are there languages which **have no CFG**?

Context Free Grammar: Questions

Are there languages which **have no CFG**? Yes

Context Free Grammar: Questions

Are there languages which **have no CFG**? Yes

Can we have **multiple** CFG’s describing the same language?

Context Free Grammar: Questions

Are there languages which **have no CFG**? Yes

Can we have **multiple** CFG’s describing the same language? Yes

Predictive Parser: Definition

Some languages has a predictive parser:

- We determine the production rule according to the current token
- We begin we the start symbol
    - From the top…

Predictive Parser: Example

The language of balanced parentheses:

- $S \to N$
- $S \to (S)$

has a predictive parser.

Predictive Parser: Example

```
void parse_S() {
  switch (token) {
  case N:
    parse_token(N);
    break;
  case L_PAREN:
    parse_token(L_PAREN);
    parse_S();
    parse_token(R_PAREN);
    break
  default:
    // error
  }
}
```

```
void parse_token(int expected) {
  if (token == expected) {
      token = lexer.next_token();
  } else {
    // error
  }
}
```

```
void parse() {
  parse_S();
  if (token != EOF)
    // error
}
```

Predictive Parser: Example

What happens for the input (7)?

Call trace:

- parse_S
  - parse_token // match with ‘(‘
  - parse_S
    - parse_token // match with ‘7‘
  - parse_token // match with ‘)‘

Predictive Parser: Example

What happens for the input ((7)?

Call trace:

- parse_S
  - parse_token // match with '('
  - parse_S
    - parse_token // match with '('
    - parse_S
      - parse_token // match with '7'
    - parse_token // match with ')'
  - parse_token // error, expecting ')'

Language of Balanced Parentheses 2

Find a CFG for a language with the 3 kinds of parentheses:

- (), [], {}

Contains string of the form:

- (([][]{}))[]
- [()]

Not allowing:

- (()){

Language of Balanced Parentheses 2

CFG definition:

- $S \rightarrow (S)S$
- $S \rightarrow [S]S$
- $S \rightarrow \{S\}S$
- $S \rightarrow \epsilon$

Language of Balanced Parentheses 2

```
void parse_S() {
  switch (token) {
  case L_PAREN:
    parse_S1();
    break;
  case L_BRACKET:
    parse_S2();
    break;
  case L_BRACE:
    parse_S3();
    break;
  default:
    break;
  }
}
```

```
void parse_S1() {
  parse_token(L_PAREN);
  parse_S();
  parse_token(R_PAREN);
  parse_S();
}
void parse_S2() {
  parse_token(L_BRACKET);
  parse_S();
  parse_token(R_BRACKET);
  parse_S();
}
void parse_S3() {
  parse_token(L_BRACE);
  parse_S();
  parse_token(R_BRACE);
  parse_S();
}
```

Calculator Language

A language with binary operators (+,-,*,/) and numbers:

- 1
- 1+1
- (1+1)*(7/2)
- 2+1-7

Calculator Language

A (possible) CFG for that language:

- $S \to N$
- $S \to S + S$
- $S \to S - S$
- $S \to S * S$
- $S \to S\ /\ S$
- $S \to (S)$

Calculator Language

A (possible) CFG for that language:

- $S \to N$
- $S \to S + S$
- $S \to S - S$
- $S \to S * S$
- $S \to S \,/\, S$
- $S \to (S)$

Will predictive parsing work here?

Left Recursion

There is no predictive parser which can handle the previous CFG

Why?

Left Recursion

There is no predictive parser which can handle the previous CFG

Why?

```
void parse_S() {
  switch (token) {
  case <...>:
    parse_S();
    parse_token(PLUS);
    parse_S();
    break;
  ...
}
```

Left Recursion

Why it happens?

In the rule $S \to S + S$:

- S itself appears on the **left side** of the alternative

If we still want a predictive parser

- Need to **eliminate** left recursion

Left Recursion Elimination

If we have:

- $X \rightarrow a$
- $X \rightarrow Xb$

Then the language contains:

- $a, ab, abb, abbb, \dots$

Define an alternative CFG:

- $X \rightarrow aY$
- $Y \rightarrow bY \mid \epsilon$

Left Recursion Elimination

In general, if we have:

- $X \rightarrow a_1 \mid a_2 \mid \ldots$
- $X \rightarrow Xb_1 \mid Xb_2 \mid \ \ldots$

We will rewrite as follows:

- $X \rightarrow a_1Y|a_2Y| \ldots$
- $Y \rightarrow b_1Y|b_2Y| \ldots | \ \epsilon$

Calculator Language

Before left recursion elimination:

- $S \to N$
- $S \to (S) \mid S + S \mid S - S \mid S \ * S \mid S \ / \ S$

What are our a_i, b_i ?

Calculator Language

Before left recursion elimination:

- $S \rightarrow N$
- $S \rightarrow (S) \mid S + S \mid S - S \mid S * S \mid S\ /\ S$

What are our a_i, b_i ?

- $a_1 = N, a_2 = (S)$
- $b_1 = +S, b_2 = -S, b_3 = * S, b_4 = /S$

Calculator Language

Before left recursion elimination:

- $S \to N$
- $S \to (S) \mid S + S \mid S - S \mid S * S \mid S \,/\, S$

The resulting CFG:

- $S \to NT \mid (S)T$
- $T \to +ST \mid -ST \mid * ST \mid /ST \mid \epsilon$

LL(1)

Definitions:

- A grammar that has a predictive parser is called LL(1)
- A language that has LL(1) grammar is called LL(1)

CFG vs Language

- A language may hove more the one CFG
- We might have a language which 2 CFG’s where:
  - One of them is LL(1)
  - The other one isn’t…

Derivation Tree

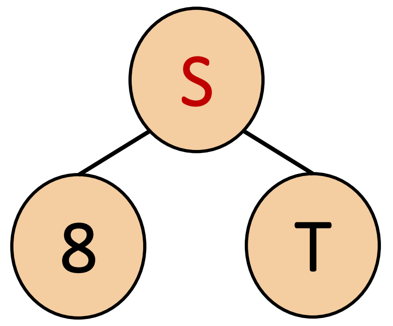
Which rules are applied for the expression 8 * 4 + 3?

- $S \rightarrow NT$
- $S \rightarrow (S)T$
- $T \rightarrow +ST$
- $T \rightarrow -ST$
- $T \rightarrow * ST$
- $T \rightarrow /ST$
- $T \rightarrow \epsilon$

Derivation Tree

Which rules are applied for the expression 8 * 4 + 3?

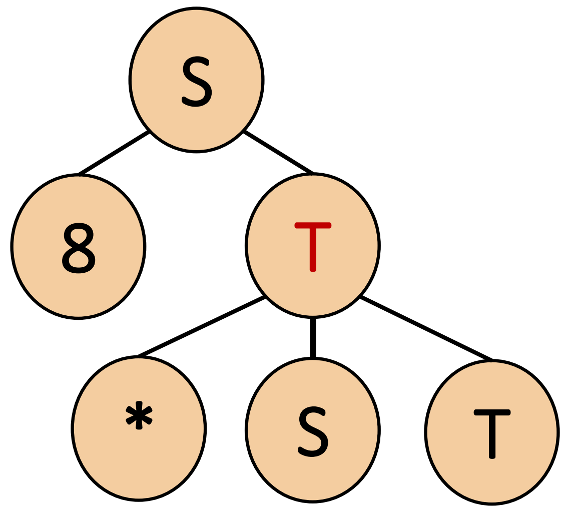
- $S \rightarrow NT$
- $S \rightarrow (S)T$
- $T \rightarrow +ST$
- $T \rightarrow -ST$
- $T \rightarrow * ST$
- $T \rightarrow /ST$
- $T \rightarrow \epsilon$



Derivation Tree

Which rules are applied for the expression 8 * 4 + 3?

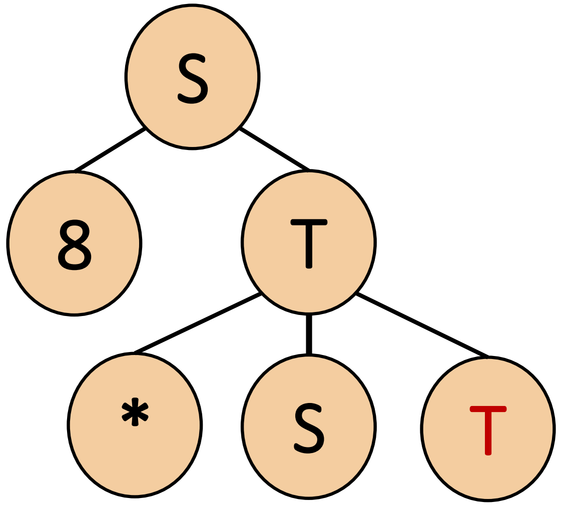
- $S \rightarrow NT$
- $S \rightarrow (S)T$
- $T \rightarrow +ST$
- $T \rightarrow -ST$
- $T \rightarrow *ST$
- $T \rightarrow /ST$
- $T \rightarrow \epsilon$



Derivation Tree

Which rules are applied for the expression 8 * 4 + 3?

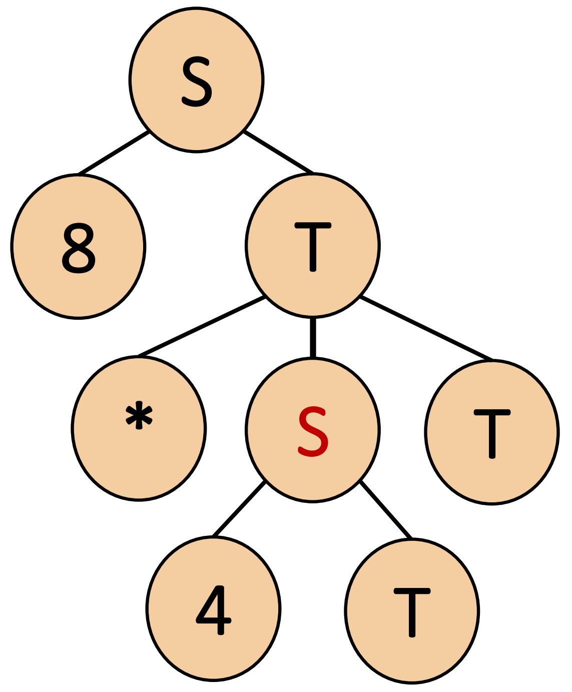
- $S \rightarrow NT$
- $S \rightarrow (S)T$
- $T \rightarrow +ST$
- $T \rightarrow -ST$
- $T \rightarrow * ST$
- $T \rightarrow /ST$
- $T \rightarrow \epsilon$



Derivation Tree

Which rules are applied for the expression 8 * 4 + 3?

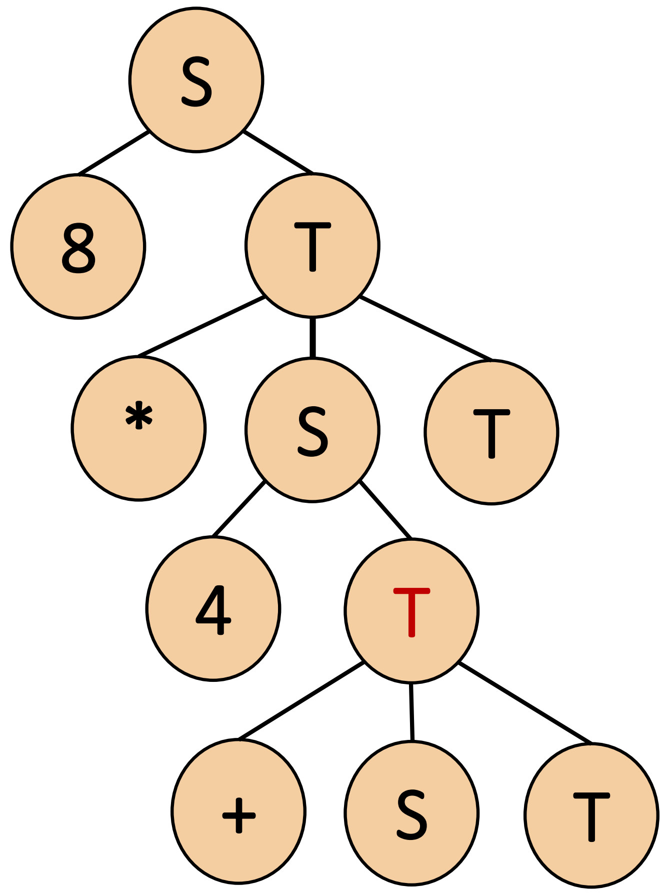
- $S \rightarrow NT$
- $S \rightarrow (S)T$
- $T \rightarrow +ST$
- $T \rightarrow -ST$
- $T \rightarrow * ST$
- $T \rightarrow /ST$
- $T \rightarrow \epsilon$



Derivation Tree

Which rules are applied for the expression 8 * 4 + 3?

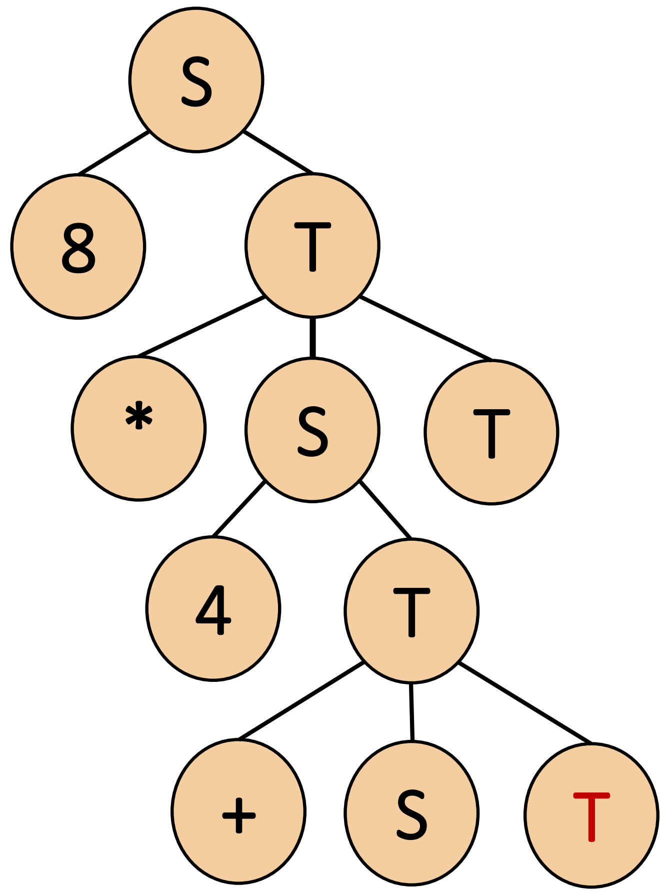
- $S \rightarrow NT$
- $S \rightarrow (S)T$
- $T \rightarrow +ST$
- $T \rightarrow -ST$
- $T \rightarrow * ST$
- $T \rightarrow /ST$
- $T \rightarrow \epsilon$



Derivation Tree

Which rules are applied for the expression 8 * 4 + 3?

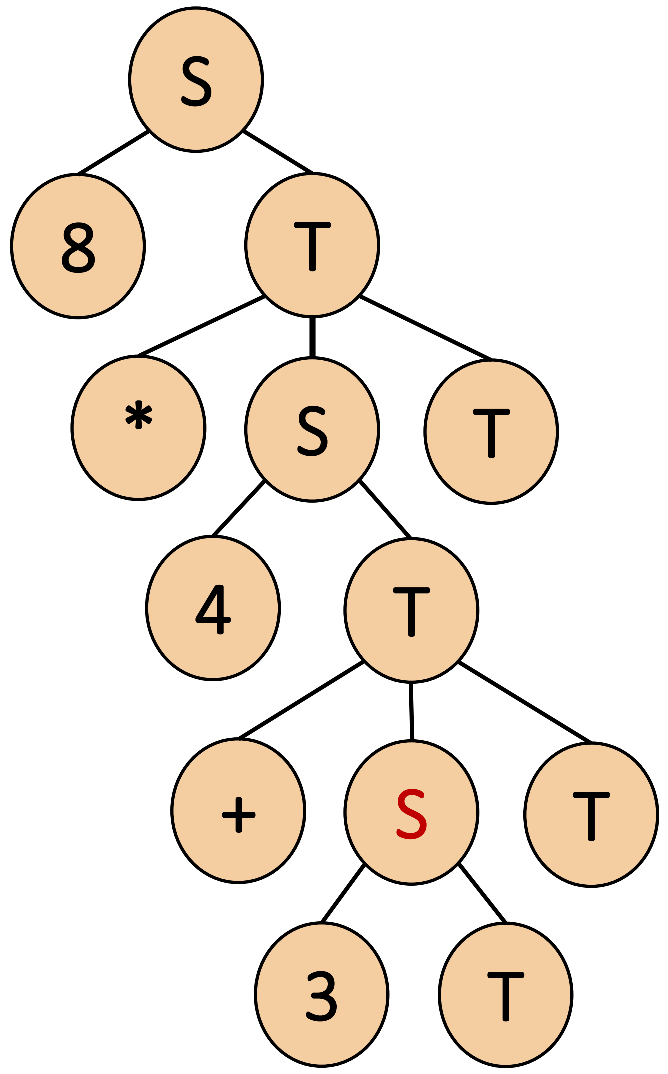
- $S \rightarrow NT$
- $S \rightarrow (S)T$
- $T \rightarrow +ST$
- $T \rightarrow -ST$
- $T \rightarrow *\,ST$
- $T \rightarrow /ST$
- $T \rightarrow \epsilon$



Derivation Tree

Which rules are applied for the expression 8 * 4 + 3?

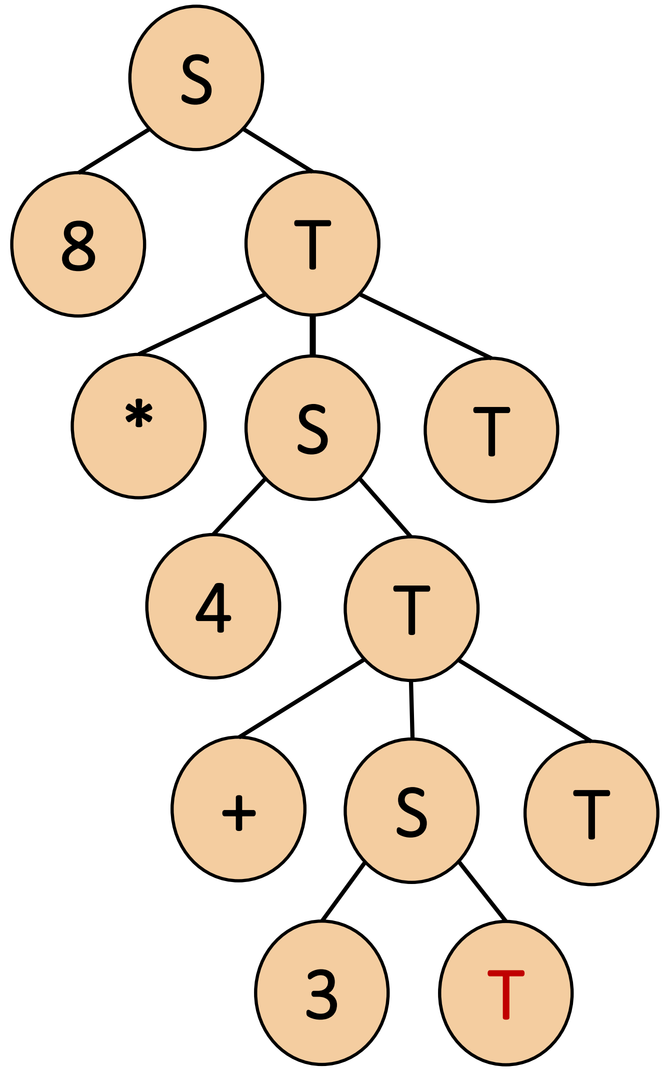
- $S \rightarrow NT$
- $S \rightarrow (S)T$
- $T \rightarrow +ST$
- $T \rightarrow -ST$
- $T \rightarrow *ST$
- $T \rightarrow /ST$
- $T \rightarrow \epsilon$



Derivation Tree

Which rules are applied for the expression 8 * 4 + 3?

- $S \rightarrow NT$
- $S \rightarrow (S)T$
- $T \rightarrow +ST$
- $T \rightarrow -ST$
- $T \rightarrow * ST$
- $T \rightarrow /ST$
- $T \rightarrow \epsilon$



Operator Precedence

Our CFG does not contain information about **operator precedence!**

- The expression 8 * 4 + 3 is interpreted as 8 * (4 + 3)
- We need to find another grammar…

Operator Precedence

A CFG with **operator precedence:**

- $S \rightarrow S + T \mid S - T \mid T$
- $T \rightarrow T * F \mid T \,/\, F \mid F$
- $F \rightarrow N \mid (S)$

Operator Precedence

A CFG with **operator precedence:**

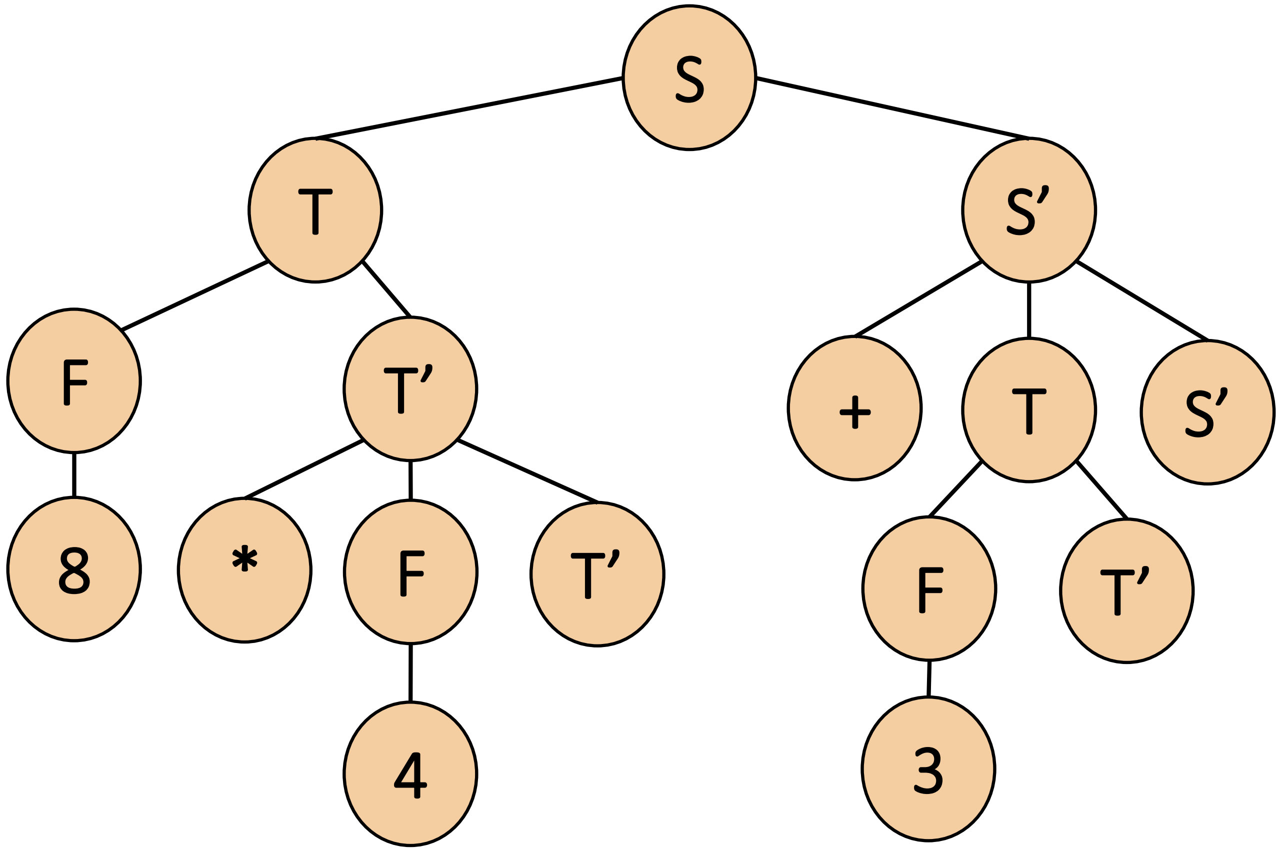
- $S \rightarrow S + T \mid S - T \mid T$
- $T \rightarrow T * F \mid T \mathbin{/} F \mid F$
- $F \rightarrow N \mid (S)$

After eliminating **left recursion**:

- $S \rightarrow TS'$
- $S' \rightarrow +TS' \mid -TS' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid /FT' \mid \epsilon$
- $F \rightarrow N \mid (S)$

Derivation Tree

With the new CFG, the derivation tree for 8 * 4 + 3:



Left Factoring

Left recursion was an issue, are there other issues?

What about the following grammar:

- $E \rightarrow if\ (E)\ then\ E$
- $E \rightarrow if\ (E)\ then\ E\ else\ E$
- $E \rightarrow int$

Left Factoring

Rewrite the original CFG:

- $E \rightarrow if\ (E)\ then\ E$
- $E \rightarrow if\ (E)\ then\ E\ else\ E$
- $E \rightarrow int$

To the following:

- $E \rightarrow if\ (E)\ then\ EX$
- $X \rightarrow \epsilon$
- $X \rightarrow else\ E$
- $E \rightarrow int$

Nullable Rules

Consider the following grammar:

- $S \rightarrow Tab$
- $T \rightarrow a \mid \epsilon$

No left recursion, no left factoring…

But can we build a predictive parser for it?

Nullable Rules

Consider the following grammar:

- $S \to Tab$
- $T \to a \mid \epsilon$

No left recursion, no left factoring…

But can we build a predictive parser for it?

No!

Nullable Rules

Consider the following grammar:

- $S \to Tab$
- $T \to a \mid \epsilon$

If the first symbol is a, we can't predict the right rule:

- If we choose $T \to a$, then it will fail to parse the input ab
- If we choose $T \to \epsilon$, then it will fail to parse the input aab

Nullable Rules

We can substitute T with it’s possible alternatives.

The original grammar:

- $S \to Tab$
- $T \to a \mid \epsilon$

After substitution:

- $S \to ab$
- $S \to aab$

Are we done?

Nullable Rules

We need to perform left factoring:

- $S \rightarrow ab$
- $S \rightarrow aab$

After left factoring:

- $S \rightarrow aX$
- $X \rightarrow b \mid ab$

LL(1) Parsing is not always possible

The following grammar can’t be fixed:

- $S \rightarrow A$
- $S \rightarrow B$
- $A \rightarrow aAb$
- $A \rightarrow \epsilon$
- $B \rightarrow aBbb$
- $B \rightarrow \epsilon$

LL(1) Parsing: is it always desirable?

Grammars of real languages are overloaded with

- Left recursion
- Left factoring
- Nullable rules

Even if we can fix it, the resulting grammar may be unreadable…