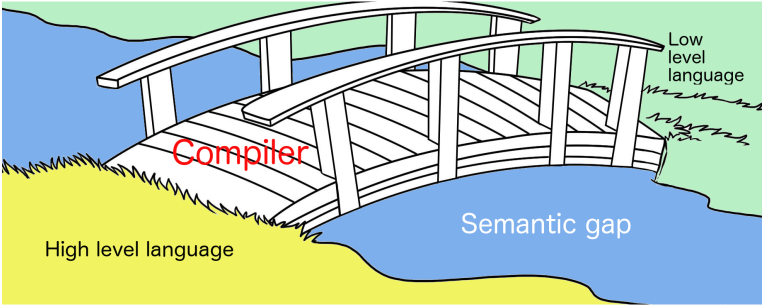
Compilers and computer architecture From strings to ASTs (2): context free grammars

Martin Berger [1]

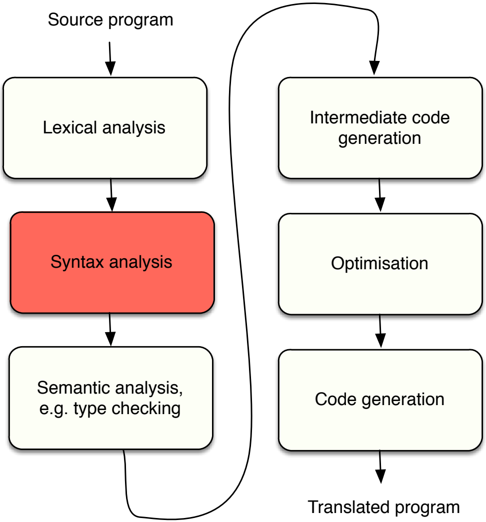
October 2019

[1] Email: `M.F.Berger@sussex.ac.uk`, Office hours: Wed 12-13 in Chi-2R312

Recall the function of compilers



Recall we are discussing parsing



Introduction

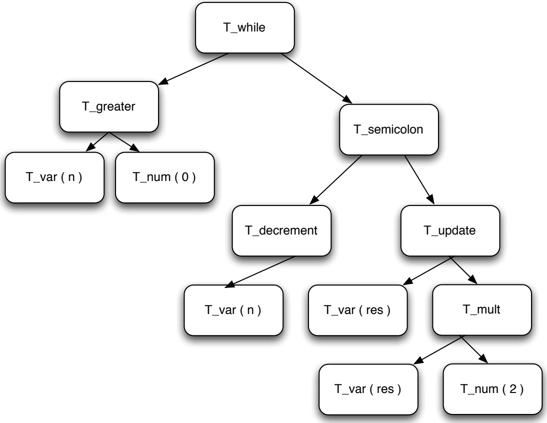
Introduction

Remember, we want to take a program given as a string and:

- Check if it's syntactically correct, e.g. is every opened bracket later closed?
- Produce an AST to facilitate **efficient** code generation.

Introduction

```
while( n > 0 ){
    n--;
    res *= 2; }
```



Introduction

We split that task into two phases, lexing and parsing. Lexing throws away some information (e.g. how many white-spaces) and prepares a token-list, which is used by the parser. The token-list simplifies the parser, because some detail is not important for syntactic correctness:

$$\texttt{if}\ x < 2 + 3\ \texttt{then}\ P\ \texttt{else}\ Q$$

is syntactically correct exactly when

$$\texttt{if}\ y < 111 + 222\ \texttt{then}\ P\ \texttt{else}\ Q$$

Introduction

The token-list simplifies the parser, because some detail is not important for syntactic correctness:

$$\texttt{if}\ x < 2 + 3\ \texttt{then}\ P\ \texttt{else}\ Q$$

is syntactically correct exactly when

$$\texttt{if}\ y < 111 + 222\ \texttt{then}\ P\ \texttt{else}\ Q$$

So from the point of view of the next stage (parsing), all we need to know is that the input is

```
T_if T_var T_less T_int T_plus T_int T_then ...
```

Of course we cannot throw away the names of variables etc completely, as the later stages (type-checking and code generation) need them. They are just irrelevant for syntax checking. We keep them and our token-lists are like this

```
T_if T_var ( "x" ) T_less T_int ( 2 ) T_plus ...
```

Two tasks of syntax analysis

Two tasks of syntax analysis

As with the lexical phase, we have to deal with two distinct tasks.

Two tasks of syntax analysis

As with the lexical phase, we have to deal with two distinct tasks.

- **Specifying** that the syntactically correct programs (token lists) are.

Two tasks of syntax analysis

As with the lexical phase, we have to deal with two distinct tasks.

- **Specifying** that the syntactically correct programs (token lists) are.
- **Checking** if an input program (token list) is syntactically correct according to the specification, and output a corresponding AST.

Two tasks of syntax analysis

As with the lexical phase, we have to deal with two distinct tasks.

- **Specifying** that the syntactically correct programs (token lists) are.
- **Checking** if an input program (token list) is syntactically correct according to the specification, and output a corresponding AST.

Let's deal with specification first. What are our options? How about using regular expressions for this purpose?

Two tasks of syntax analysis

As with the lexical phase, we have to deal with two distinct tasks.

- **Specifying** that the syntactically correct programs (token lists) are.
- **Checking** if an input program (token list) is syntactically correct according to the specification, and output a corresponding AST.

Let's deal with specification first. What are our options? How about using regular expressions for this purpose?

Alas not every language can be expressed in these formalisms. Example:

Two tasks of syntax analysis

As with the lexical phase, we have to deal with two distinct tasks.

- **Specifying** that the syntactically correct programs (token lists) are.
- **Checking** if an input program (token list) is syntactically correct according to the specification, and output a corresponding AST.

Let's deal with specification first. What are our options? How about using regular expressions for this purpose?

Alas not every language can be expressed in these formalisms. Example:

Alphabet = $\{'(', ')'\}$.

Two tasks of syntax analysis

As with the lexical phase, we have to deal with two distinct tasks.

- **Specifying** that the syntactically correct programs (token lists) are.
- **Checking** if an input program (token list) is syntactically correct according to the specification, and output a corresponding AST.

Let's deal with specification first. What are our options? How about using regular expressions for this purpose?

Alas not every language can be expressed in these formalisms. Example:

Alphabet = $\{'(', ')'\}$.

Language = all **balanced** parentheses $()$, $()()$, $(())$, $((()(()())()(())))$, ..., note: the empty string is balanced.

FSAs/REs can't count

FSAs/REs can't count

Let's analyse the situation a bit more. Why can we not describe the language of all balanced parentheses using REs or FSAs.

FSAs/REs can't count

Let's analyse the situation a bit more. Why can we not describe the language of all balanced parentheses using REs or FSAs.

Each FSA has only a fixed number (say n) of states. But what if we have more than n open brackets before we hit a closing bracket?

FSAs/REs can't count

Let's analyse the situation a bit more. Why can we not describe the language of all balanced parentheses using REs or FSAs.

Each FSA has only a fixed number (say n) of states. But what if we have more than n open brackets before we hit a closing bracket?

Since there are only n states, when we reach the n open bracket, we must have gone back to a state that we already visited earlier, say when we processed the i-th bracket with $i < n$. This means the automaton treats i as it does n, leading to confusion.

FSAs/REs can't count

Let's analyse the situation a bit more. Why can we not describe the language of all balanced parentheses using REs or FSAs.

Each FSA has only a fixed number (say *n*) of states. But what if we have more than *n* open brackets before we hit a closing bracket?

Since there are only *n* states, when we reach the *n* open bracket, we must have gone back to a state that we already visited earlier, say when we processed the *i*-th bracket with $i < n$. This means the automaton treats *i* as it does *n*, leading to confusion.

Summary: **FSAs can't count**, and likewise for REs (why?).

Lack of expressivity of regular expressions & FSAs

Lack of expressivity of regular expressions & FSAs

Why is it a problem for syntax analysis in programming languages if REs and FSAs can't count?

Lack of expressivity of regular expressions & FSAs

Why is it a problem for syntax analysis in programming languages if REs and FSAs can't count?

Because programming languages contain many bracket-like constructs that can be nested, e.g.

```
begin ... end
do ... while
if ( ... ) then { ... } else { ... }
3 + ( 3 - (x + 6) )
```

Lack of expressivity of regular expressions & FSAs

Why is it a problem for syntax analysis in programming languages if REs and FSAs can't count?

Because programming languages contain many bracket-like constructs that can be nested, e.g.

```
begin ... end
do ... while
if ( ... ) then { ... } else { ... }
3 + ( 3 - (x + 6) )
```

But we must formalise the syntax of our language if we want to computer to process it. So we need a formalism that can 'count'.

Problem

Problem

What we are looking for is something like REs, but more powerful:

Problem

What we are looking for is something like REs, but more powerful:

$$\frac{\text{regular expression/FSA}}{\text{lexer}} = \frac{\text{???}}{\text{parser}}$$

Problem

What we are looking for is something like REs, but more powerful:

$$\frac{\text{regular expression/FSA}}{\text{lexer}} = \frac{\text{???}}{\text{parser}}$$

Let me introduce you to:

Problem

What we are looking for is something like REs, but more powerful:

$$\frac{\text{regular expression/FSA}}{\text{lexer}} = \frac{\text{???}}{\text{parser}}$$

Let me introduce you to: **context free grammars (CFGs).**

Context free grammars

Context free grammars

Programs have a naturally recursive and nested structure: A program is e.g.:

Context free grammars

Programs have a naturally recursive and nested structure: A program is e.g.:

- `if` P `then` Q `else` Q', where P, Q, Q' are programs.

Context free grammars

Programs have a naturally recursive and nested structure: A program is e.g.:

- `if` P `then` Q `else` Q', where P, Q, Q' are programs.
- `x :=` P, where P is a program.

Context free grammars

Programs have a naturally recursive and nested structure: A program is e.g.:

- `if` P `then` Q `else` Q', where P, Q, Q' are programs.
- `x :=` P, where P is a program.
- `begin x := 1; begin ... end; y := 2; end`

Context free grammars

Programs have a naturally recursive and nested structure: A program is e.g.:

- `if` P `then` Q `else` Q', where P, Q, Q' are programs.
- `x :=` P, where P is a program.
- `begin x := 1; begin ... end; y := 2; end`

CFGs are a generalisation of regular expression that is ideal for describing such recursive and nested structures.

Context free grammar

Context free grammar

A **context-free grammar** is a tuple $(A, V, Init, R)$ where

- A is a finite set called **alphabet**.
- V is a finite, non-empty set of **variables**.
- $A \cap V = \emptyset$.
- $Init \in V$ is the **initial variable**.
- R is the finite set of **reductions**, where each reduction in R is of the form (l, r) such that
  - l is a variable, i.e. $l \in V$.
  - r is a string (possibly empty) over the **new** alphabet $A \cup V$.

We usually write $l \to r$ for $(l, r) \in R$.

Context free grammar

A **context-free grammar** is a tuple $(A, V, Init, R)$ where

- A is a finite set called **alphabet**.
- V is a finite, non-empty set of **variables**.
- $A \cap V = \emptyset$.
- $Init \in V$ is the **initial variable**.
- R is the finite set of **reductions**, where each reduction in R is of the form (l, r) such that
  - l is a variable, i.e. $l \in V$.
  - r is a string (possibly empty) over the **new** alphabet $A \cup V$.

  We usually write $l \to r$ for $(l, r) \in R$.

Note that the alphabet are often also called **terminal symbols**, reductions are also called **reduction steps** or **transitions** or **productions**, some people say **non-terminal symbol** for variable, and the initial variable is also called **start symbol**.

Context free grammar

Context free grammar

Example:

- $A = \{a, b\}$.
- $V = \{S\}$.
- The initial variable is S.
- R contains only three reductions:

$$S \to a\ S\ b$$
$$S \to S\ S$$
$$S \to \epsilon$$

Recall that ϵ is the empty string.

Now the CFG is (A, V, S, R).

Context free grammar

Example:

- $A = \{a, b\}$.
- $V = \{S\}$.
- The initial variable is S.
- R contains only three reductions:

$$S \to a\ S\ b$$
$$S \to S\ S$$
$$S \to \epsilon$$

Recall that ϵ is the empty string.

Now the CFG is (A, V, S, R).

The language of balanced brackets with *a* being the open bracket, and *b* being the closed bracket!

Context free grammar

Example:

- $A = \{a, b\}$.
- $V = \{S\}$.
- The initial variable is S.
- R contains only three reductions:

$$S \to a\ S\ b$$
$$S \to S\ S$$
$$S \to \epsilon$$

Recall that ϵ is the empty string.

Now the CFG is (A, V, S, R).

The language of balanced brackets with a being the open bracket, and b being the closed bracket!

To make this intuition precise, we need to say precisely what the language of a CFG is.

The language accepted by a CFG

The language accepted by a CFG

The key idea is simple: **replace the variables according to the reductions.**

The language accepted by a CFG

The key idea is simple: **replace the variables according to the reductions.**

Given a string s over $A \cup V$, ie. the alphabet and variables, any occurrence of a variable T in s can be replaced by the string $r_1 ... r_n$, provided there is a reduction $T \rightarrow r_1 ... r_n$.

The language accepted by a CFG

The key idea is simple: **replace the variables according to the reductions.**

Given a string s over $A \cup V$, ie. the alphabet and variables, any occurrence of a variable T in s can be replaced by the string $r_1 ... r_n$, provided there is a reduction $T \to r_1 ... r_n$.

For example if we have a reduction

$$S \to a\ T\ b$$

The language accepted by a CFG

The key idea is simple: **replace the variables according to the reductions.**

Given a string s over $A \cup V$, ie. the alphabet and variables, any occurrence of a variable T in s can be replaced by the string $r_1 ... r_n$, provided there is a reduction $T \rightarrow r_1 ... r_n$.

For example if we have a reduction

$$S \rightarrow a\ T\ b$$

then we can rewrite the string

$$aaSbb$$

The language accepted by a CFG

The key idea is simple: **replace the variables according to the reductions.**

Given a string s over $A \cup V$, ie. the alphabet and variables, any occurrence of a variable T in s can be replaced by the string $r_1 ... r_n$, provided there is a reduction $T \rightarrow r_1 ... r_n$.

For example if we have a reduction

$$S \rightarrow a\ T\ b$$

then we can rewrite the string

$$aaSbb$$

to

$$aaaTbbb$$

The language accepted by a CFG

The language accepted by a CFG

How do we start this rewriting of variables?

The language accepted by a CFG

How do we start this rewriting of variables? With the initial variable.

The language accepted by a CFG

How do we start this rewriting of variables? With the initial variable.

When does this rewriting of variables stop?

The language accepted by a CFG

How do we start this rewriting of variables? With the initial variable.

When does this rewriting of variables stop? When the string we arrive at by rewriting in a finite number of steps from the initial variable contains no more variables.

The language accepted by a CFG

The language accepted by a CFG

Then: the **language** of a CFG is the set of all strings over the alphabet of the CFG that can be arrived at by rewriting from the initial variable.

The language accepted by a CFG

The language accepted by a CFG

Let's do this with the CFG for balanced brackets (A, V, S, R) where

- $A = \{(,)\}$.
- $V = \{S\}$.
- The initial variable is S.
- Reductions R are $S \rightarrow (\ S\ )$, $S \rightarrow SS$, and $S \rightarrow \epsilon$

The language accepted by a CFG

Let's do this with the CFG for balanced brackets (A, V, S, R) where

- $A = \{(,)\}$.
- $V = \{S\}$.
- The initial variable is S.
- Reductions R are $S \to (\ S\ )$, $S \to SS$, and $S \to \epsilon$

$$
\begin{aligned}
S &\to (S) \\
&\to (SS) \\
&\to ((S)S) \\
&\to (((S))S) \\
&\to (((S))SS) \\
&\to (((S))\epsilon S) = (((S))S) \\
&\to (((\epsilon))S) = ((())S) \\
&\to ((())\epsilon) = ((()))
\end{aligned}
$$

Question: Why / how can CFGs count?

Question: Why / how can CFGs count?

Why / how does the CFG (A, V, S, R) with

$$S \rightarrow (\ S \ )$$
$$S \rightarrow S \ S$$
$$S \rightarrow \epsilon$$

count?

Question: Why / how can CFGs count?

Why / how does the CFG (A, V, S, R) with

$$S \rightarrow (\ S \ )$$
$$S \rightarrow S \ S$$
$$S \rightarrow \epsilon$$

count?

Because only $S \rightarrow (\ S \ )$ introduces new brackets. But by construction it always introduces a closing bracket for each new open bracket.

The language accepted by a CFG: infinite reductions

The language accepted by a CFG: infinite reductions

Note that many CFGs allow infinite reductions:

The language accepted by a CFG: infinite reductions

Note that many CFGs allow infinite reductions: for example with the grammar the previous slide we can do this:

$$
\begin{aligned}
S &\rightarrow (S) \\
&\rightarrow ((S)) \\
&\rightarrow (((S))) \\
&\rightarrow ((((S)))) \\
&\rightarrow (((((S))))) \\
&\rightarrow ((((((S)))))) \\
&\vdots
\end{aligned}
$$

The language accepted by a CFG: infinite reductions

Note that many CFGs allow infinite reductions: for example with the grammar the previous slide we can do this:

$$\begin{aligned} S &\to (S) \\ &\to ((S)) \\ &\to (((S))) \\ &\to ((((S)))) \\ &\to (((((S))))) \\ &\to ((((((S)))))) \\ &\vdots \end{aligned}$$

Such infinite reductions don't affect the language of the grammar. Only sequences of rewrites that end in a string free from variables count towards the language.

The language accepted by a CFG

The language accepted by a CFG

If you like formal definitions ...

The language accepted by a CFG

If you like formal definitions ...

Given a fixed CFG $\mathcal{G} = (A, V, S, R)$. For arbitrary strings $\sigma, \sigma' \in (V \cup A)^*$ we define the *one-step reduction relation* $\Rightarrow$ which relates strings from $(V \cup A)^*$ as follows.
$\sigma \Rightarrow \sigma'$ if and only if:

- $\sigma = \sigma_1 l \sigma_2$ where $l \in V$, and σ_1, σ_2 are strings from $(V \cup A)^*$.
- There is a reduction $l \longrightarrow \gamma$ in R.
- $\sigma' = \sigma_1 \gamma \sigma_2$.

The *language accepted by* $\mathcal{G}$, written $\text{lang}(\mathcal{G})$ is given as follows.

$$\text{lang}(\mathcal{G}) \stackrel{def}{=} \{\gamma_n \mid S \to \gamma_1 \to \cdots \to \gamma_n, \text{ where } \gamma_n \in A^*\}$$

The sequence $S \to \gamma_1 \to \cdots \to \gamma_n$ is called **derivation.**

Note: only strings free from variables can be in $\text{lang}(\mathcal{G})$.

Example CFG

Example CFG

Consider the following CFG where `while`, `if`, `;` etc are elements of the alphabet, and M is a variable.

$$
\begin{array}{lcl}
M & \rightarrow & \texttt{while}\ M\ \texttt{do}\ M \\
M & \rightarrow & \texttt{if}\ M\ \texttt{then}\ M \\
M & \rightarrow & M; M \\
 & \vdots &
\end{array}
$$

Example CFG

Consider the following CFG where `while`, `if`, `;` etc are elements of the alphabet, and M is a variable.

$$\begin{array}{lcl} M & \rightarrow & \texttt{while}\ M\ \texttt{do}\ M \\ M & \rightarrow & \texttt{if}\ M\ \texttt{then}\ M \\ M & \rightarrow & M; M \\ & \vdots & \end{array}$$

If M is the starting variable, then we can derive

Example CFG

Consider the following CFG where `while`, `if`, `;` etc are elements of the alphabet, and M is a variable.

$$\begin{array}{lcl} M & \to & \texttt{while}\ M\ \texttt{do}\ M \\ M & \to & \texttt{if}\ M\ \texttt{then}\ M \\ M & \to & M; M \\ & \vdots & \end{array}$$

If M is the starting variable, then we can derive

$$M \to M; M$$

Example CFG

Consider the following CFG where `while`, `if`, `;` etc are elements of the alphabet, and M is a variable.

$$
\begin{array}{lcl}
M & \to & \texttt{while}\ M\ \texttt{do}\ M \\
M & \to & \texttt{if}\ M\ \texttt{then}\ M \\
M & \to & M; M \\
 & \vdots &
\end{array}
$$

If M is the starting variable, then we can derive

$$
\begin{array}{lcl}
M & \to & M; M \\
 & \to & M; \texttt{if}\ M\ \texttt{then}\ M
\end{array}
$$

Example CFG

Consider the following CFG where `while`, `if`, `;` etc are elements of the alphabet, and M is a variable.

$$
\begin{array}{lcl}
M & \rightarrow & \texttt{while}\ M\ \texttt{do}\ M \\
M & \rightarrow & \texttt{if}\ M\ \texttt{then}\ M \\
M & \rightarrow & M; M \\
 & \vdots &
\end{array}
$$

If M is the starting variable, then we can derive

$$
\begin{array}{lcl}
M & \rightarrow & M; M \\
 & \rightarrow & M; \texttt{if}\ M\ \texttt{then}\ M \\
 & \rightarrow & M; \texttt{if}\ M\ \texttt{then}\ \texttt{while}\ M\ \texttt{do}\ M \\
 & \vdots &
\end{array}
$$

Example CFG

Consider the following CFG where `while`, `if`, `;` etc are elements of the alphabet, and M is a variable.

$$\begin{array}{lcl} M & \rightarrow & \texttt{while}\ M\ \texttt{do}\ M \\ M & \rightarrow & \texttt{if}\ M\ \texttt{then}\ M \\ M & \rightarrow & M; M \\ & \vdots & \end{array}$$

If M is the starting variable, then we can derive

$$\begin{array}{lcl} M & \rightarrow & M; M \\ & \rightarrow & M; \texttt{if}\ M\ \texttt{then}\ M \\ & \rightarrow & M; \texttt{if}\ M\ \texttt{then}\ \texttt{while}\ M\ \texttt{do}\ M \\ & \vdots & \end{array}$$

We do this until we reach a string without variables.

Some conventions regarding CFGs

Some conventions regarding CFGs

Here is a collection of conventions for making CFGs more readable. You will find them a lot when programming languages are discussed.

Some conventions regarding CFGs

Here is a collection of conventions for making CFGs more readable. You will find them a lot when programming languages are discussed.

Variables are CAPITALISED, the alphabet is lower case (or vice versa).

Some conventions regarding CFGs

Here is a collection of conventions for making CFGs more readable. You will find them a lot when programming languages are discussed.

Variables are CAPITALISED, the alphabet is lower case (or vice versa).

Variables are in **BOLD**, the alphabet is not (or vice versa).

Some conventions regarding CFGs

Here is a collection of conventions for making CFGs more readable. You will find them a lot when programming languages are discussed.

Variables are CAPITALISED, the alphabet is lower case (or vice versa).

Variables are in **BOLD**, the alphabet is not (or vice versa).

Variables are written in ⟨angle-brackets⟩, the alphabet isn't.

Some conventions regarding CFGs

Some conventions regarding CFGs

Instead of multiple reductions from the same variable, like

$$N \to r_1$$
$$N \to r_2$$
$$N \to r_3$$

we write

$$N \to r_1 \mid r_2 \mid r_3$$

Some conventions regarding CFGs

Instead of multiple reductions from the same variable, like

$$N \to r_1$$
$$N \to r_2$$
$$N \to r_3$$

we write

$$N \to r_1 \mid r_2 \mid r_3$$

Instead of

$$P \to \texttt{if}\ P\ \texttt{then}\ P \mid \texttt{while}\ P\ \texttt{do}\ P$$

We often write

$$P, Q \to \texttt{if}\ P\ \texttt{then}\ Q \mid \texttt{while}\ P\ \texttt{do}\ Q$$

Some conventions regarding CFGs

Instead of multiple reductions from the same variable, like

$$N \to r_1$$
$$N \to r_2$$
$$N \to r_3$$

we write

$$N \to r_1 \mid r_2 \mid r_3$$

Instead of

$$P \to \texttt{if}\ P\ \texttt{then}\ P \mid \texttt{while}\ P\ \texttt{do}\ P$$

We often write

$$P, Q \to \texttt{if}\ P\ \texttt{then}\ Q \mid \texttt{while}\ P\ \texttt{do}\ Q$$

Finally, many write $::=$ instead of $\to$.

Simple arithmetic expressions

Simple arithmetic expressions

Let's do another example. Grammar:

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

Simple arithmetic expressions

Let's do another example. Grammar:

$$E \quad \to \quad E + E \;|\; E * E \;|\; (E) \;|\; 0 \;|\; 1 \;|\; \ldots$$

The language contains:

Simple arithmetic expressions

Let's do another example. Grammar:

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

The language contains:

- 7

Simple arithmetic expressions

Let's do another example. Grammar:

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

The language contains:

- 7
- $7 * 4$

Simple arithmetic expressions

Let's do another example. Grammar:

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

The language contains:

- 7
- $7 * 4$
- $7 * 4 + 222$

Simple arithmetic expressions

Let's do another example. Grammar:

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

The language contains:

- 7
- $7 * 4$
- $7 * 4 + 222$
- $7 * (4 + 222)$ ...

A well-known context free grammar

A well-known context free grammar

$$R \quad \rightarrow \quad \emptyset \mid \epsilon \mid {'c'} \mid R + R \mid RR \mid R^* \mid (R)$$

A well-known context free grammar

$$R \quad \rightarrow \quad \emptyset \mid \epsilon \mid {'c'} \mid R + R \mid RR \mid R^* \mid (R)$$

What's this?

A well-known context free grammar

$$R \quad \rightarrow \quad \emptyset \mid \epsilon \mid {'c'} \mid R + R \mid RR \mid R^* \mid (R)$$

What's this?

(The syntax of) regular expressions can be described by a CFG (but not by an RE)!

REs vs CFGs

REs vs CFGs

Since regular expressions are a special case of CFGs, could we not do both, lexing and parsing, using only CFGs?

REs vs CFGs

Since regular expressions are a special case of CFGs, could we not do both, lexing and parsing, using only CFGs?

In principle yes, but lexing based on REs (and FSAs) is simpler and faster!

Example: Java grammar

Let's look at the CFG for a real language:

Example: Java grammar

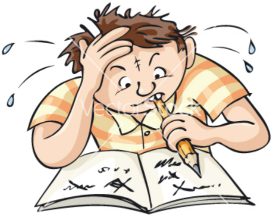
Let's look at the CFG for a real language:

`https://docs.oracle.com/javase/specs/jls/se13/html/jls-19.html`

Example: Java grammar

Let's look at the CFG for a real language:

`https://docs.oracle.com/javase/specs/jls/se13/html/jls-19.html`



CFGs, what's next?

CFGs, what's next?

Recall we were looking for this:

CFGs, what's next?

Recall we were looking for this:

$$\frac{\text{regular expression/FSA}}{\text{FSA}} = \frac{\text{???}}{\text{parser}}$$

CFGs, what's next?

Recall we were looking for this:

$$\frac{\text{regular expression/FSA}}{\text{FSA}} = \frac{\text{???}}{\text{parser}}$$

And the answer was CFGs. But what is a parser?

CFGs, what's next?

Recall we were looking for this:

$$\frac{\text{regular expression/FSA}}{\text{FSA}} = \frac{\text{???}}{\text{parser}}$$

And the answer was CFGs. But what is a parser?

$$\frac{\text{regular expression/FSA}}{\text{FSA}} = \frac{\mathit{CFG}}{\text{???}}$$

CFGs, what's next?

CFGs, what's next?

CFGs allow us to specify the grammar for programming languages. But that's not all we want. We also want:

CFGs, what's next?

CFGs allow us to specify the grammar for programming languages. But that's not all we want. We also want:

- An algorithm that decided whether a given token list is in the language of the grammar or not.

CFGs, what's next?

CFGs allow us to specify the grammar for programming languages. But that's not all we want. We also want:

- An algorithm that decided whether a given token list is in the language of the grammar or not.
- An algorithm that converts the list of tokens (if valid) into an AST.

CFGs, what's next?

CFGs allow us to specify the grammar for programming languages. But that's not all we want. We also want:

- An algorithm that decided whether a given token list is in the language of the grammar or not.
- An algorithm that converts the list of tokens (if valid) into an AST.

The key idea to solving both problems in one go is the **parse tree**.

CFG vs AST

CFG vs AST

Here is a grammar that you were asked to write an AST for in the tutorials.

```
P   ::=   x := e | if0 e then P else P
     |    whileGt0 e do P | P; P

e   ::=   e + e | e - e | e * e | (e) | e % e
     |     x | 0 | 1 | 2 | ...
```

CFG vs AST

Here's a plausible definition of ASTs for the language: Syntax.java

CFG vs AST

Do you notice something?

CFG vs AST

Do you notice something?

Looks very similar. The CFG is (almost?) a description of data type for the AST.

CFG vs AST

Do you notice something?

Looks very similar. The CFG is (almost?) a description of data type for the AST.

This is no coincidence, and we will use this similarity to construct the AST as we parse, in that we will see the parsing process as a tree. How?

Derivations and parse trees

Derivations and parse trees

Recall that a derivation in a CFG (A, V, I, R) is a sequence

$$I \to t_1 \to t_2 \to \cdots \to t_n$$

where t_n is free from variables, and each step is goverend by a reduction from R.

Derivations and parse trees

Recall that a derivation in a CFG (A, V, I, R) is a sequence

$$I \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n$$

where t_n is free from variables, and each step is goverend by a reduction from R.

We can drawn each derivation as a tree, called **parse tree.** The parse tree tells us how the input token list 'fits into' the grammar, e.g. which reduction we applied and when to 'consume' the input.

Derivations and parse trees

Recall that a derivation in a CFG (A, V, I, R) is a sequence

$$I \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n$$

where t_n is free from variables, and each step is goverend by a reduction from R.

We can drawn each derivation as a tree, called **parse tree.** The parse tree tells us how the input token list 'fits into' the grammar, e.g. which reduction we applied and when to 'consume' the input.

- The start symbol I is the tree's root.

Derivations and parse trees

Recall that a derivation in a CFG (A, V, I, R) is a sequence

$$I \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n$$

where t_n is free from variables, and each step is goverend by a reduction from R.

We can drawn each derivation as a tree, called **parse tree.** The parse tree tells us how the input token list 'fits into' the grammar, e.g. which reduction we applied and when to 'consume' the input.

- The start symbol I is the tree's root.
- For each reduction $X \rightarrow \langle y_1, ..., y_n \rangle$ we add all the y_i as children to node X.

I.e. nodes in the tree are elements of $A \cup V$.

Derivations and parse trees

Recall that a derivation in a CFG (A, V, I, R) is a sequence

$$I \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n$$

where t_n is free from variables, and each step is goverend by a reduction from R.

We can drawn each derivation as a tree, called **parse tree.** The parse tree tells us how the input token list 'fits into' the grammar, e.g. which reduction we applied and when to 'consume' the input.

- The start symbol I is the tree's root.
- For each reduction $X \rightarrow \langle y_1, ..., y_n \rangle$ we add all the y_i as children to node X.

I.e. nodes in the tree are elements of $A \cup V$. Let's look at an example.

Example parse tree

Example parse tree

Recall our CFG for arithmetic expressions: Let's do another example. Grammar:

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

Example parse tree

Recall our CFG for arithmetic expressions: Let's do another example. Grammar:

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

Let's say we have the string "4 * 3 + 17". Let's parse this string and build the corresponding parse tree.

Example parse tree

E

Example parse tree

$$E \quad \rightarrow \quad E + E$$

Example parse tree

$$
\begin{aligned}
E &\rightarrow E + E \\
  &\rightarrow E * E + E
\end{aligned}
$$

Example parse tree

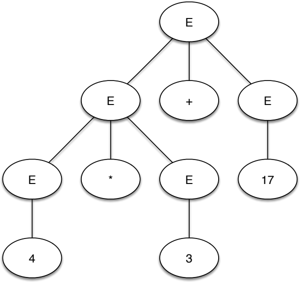
$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E \\
&\rightarrow 4 * E + E
\end{aligned}
$$

Example parse tree

$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E \\
&\rightarrow 4 * E + E \\
&\rightarrow 4 * E + 17
\end{aligned}
$$

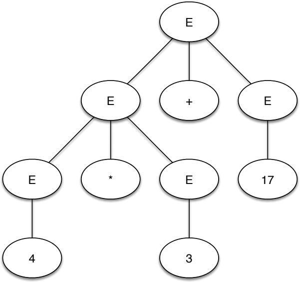
Example parse tree

$$
\begin{aligned}
E & \rightarrow E + E \\
& \rightarrow E * E + E \\
& \rightarrow 4 * E + E \\
& \rightarrow 4 * E + 17 \\
& \rightarrow 4 * 3 + 17
\end{aligned}
$$



Example parse tree

$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E \\
&\rightarrow 4 * E + E \\
&\rightarrow 4 * E + 17 \\
&\rightarrow 4 * 3 + 17
\end{aligned}
$$



Let's do this in detail on the board.

Derivations and parse trees

Derivations and parse trees

The following is important about parse trees.

Derivations and parse trees

The following is important about parse trees.

- Terminal symbols are at the leaves of the tree.

Derivations and parse trees

The following is important about parse trees.

- Terminal symbols are at the leaves of the tree.
- Variables symbols are at the non-leave nodes.

Derivations and parse trees

The following is important about parse trees.

- Terminal symbols are at the leaves of the tree.
- Variables symbols are at the non-leave nodes.
- An **in-order** traversal of the tree returns the input string.

Derivations and parse trees

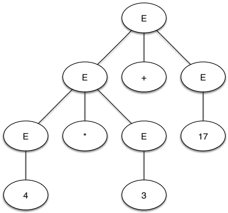
The following is important about parse trees.

- Terminal symbols are at the leaves of the tree.
- Variables symbols are at the non-leave nodes.
- An **in-order** traversal of the tree returns the input string.
- The parse tree reveals bracketing structure explicitly.

Left- vs rightmost derivation

Left- vs rightmost derivation

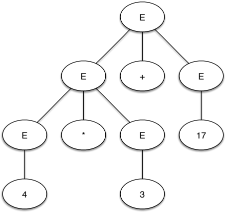
BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



E

Left- vs rightmost derivation

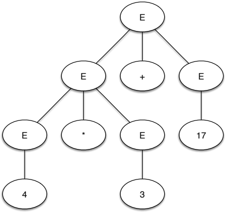
BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



$$E \rightarrow E + E$$

Left- vs rightmost derivation

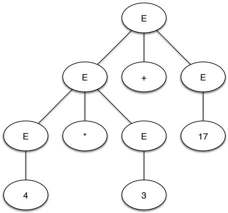
BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E
\end{aligned}
$$

Left- vs rightmost derivation

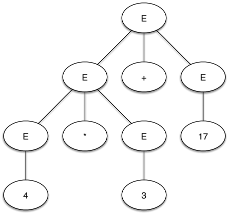
BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E \\
&\rightarrow 4 * E + E
\end{aligned}
$$

Left- vs rightmost derivation

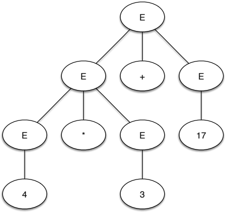
BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



$$\begin{aligned} E &\rightarrow E + E \\ &\rightarrow E * E + E \\ &\rightarrow 4 * E + E \\ &\rightarrow 4 * E + 17 \end{aligned}$$

Left- vs rightmost derivation

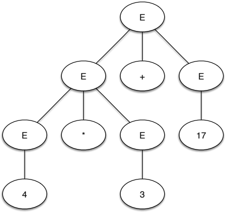
BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



$$
\begin{aligned}
E &\rightarrow E + E \qquad\qquad E \\
&\rightarrow E * E + E \\
&\rightarrow 4 * E + E \\
&\rightarrow 4 * E + 17 \\
&\rightarrow 4 * 3 + 17
\end{aligned}
$$

Left- vs rightmost derivation

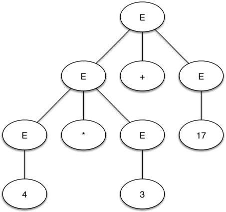
BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E \\
&\rightarrow 4 * E + E \\
&\rightarrow 4 * E + 17 \\
&\rightarrow 4 * 3 + 17
\end{aligned}
\qquad
E \rightarrow E + E
$$

Left- vs rightmost derivation

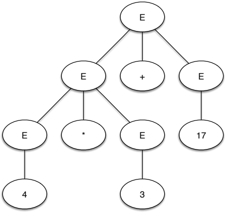
BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



$$
\begin{array}{lll}
E & \rightarrow & E + E \\
  & \rightarrow & E * E + E \\
  & \rightarrow & 4 * E + E \\
  & \rightarrow & 4 * E + 17 \\
  & \rightarrow & 4 * 3 + 17
\end{array}
\qquad
\begin{array}{lll}
E & \rightarrow & E + E \\
  & \rightarrow & E * E + E
\end{array}
$$

Left- vs rightmost derivation

BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:

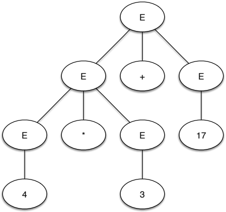


$$\begin{aligned} E &\rightarrow E + E \\ &\rightarrow E * E + E \\ &\rightarrow 4 * E + E \\ &\rightarrow 4 * E + 17 \\ &\rightarrow 4 * 3 + 17 \end{aligned}$$

$$\begin{aligned} E &\rightarrow E + E \\ &\rightarrow E * E + E \\ &\rightarrow E * 3 + E \end{aligned}$$

Left- vs rightmost derivation

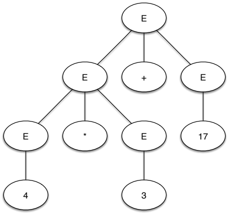
BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E \\
&\rightarrow 4 * E + E \\
&\rightarrow 4 * E + 17 \\
&\rightarrow 4 * 3 + 17
\end{aligned}
\qquad
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E \\
&\rightarrow E * 3 + E \\
&\rightarrow E * 3 + 17
\end{aligned}
$$

Left- vs rightmost derivation

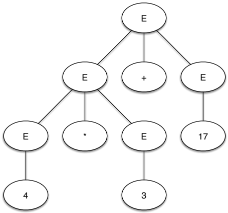
BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



$$\begin{aligned} E &\rightarrow E + E \\ &\rightarrow E * E + E \\ &\rightarrow 4 * E + E \\ &\rightarrow 4 * E + 17 \\ &\rightarrow 4 * 3 + 17 \end{aligned} \qquad \begin{aligned} E &\rightarrow E + E \\ &\rightarrow E * E + E \\ &\rightarrow E * 3 + E \\ &\rightarrow E * 3 + 17 \\ &\rightarrow 4 * 3 + 17 \end{aligned}$$

Left- vs rightmost derivation

BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:

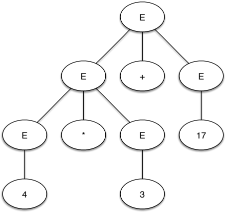


$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E \\
&\rightarrow 4 * E + E \\
&\rightarrow 4 * E + 17 \\
&\rightarrow 4 * 3 + 17
\end{aligned}
\qquad
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E \\
&\rightarrow E * 3 + E \\
&\rightarrow E * 3 + 17 \\
&\rightarrow 4 * 3 + 17
\end{aligned}
$$

Canonical choices:

Left- vs rightmost derivation

BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



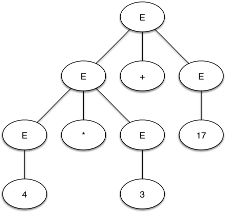
$$
\begin{array}{lll}
E & \rightarrow & E + E \\
  & \rightarrow & E * E + E \\
  & \rightarrow & 4 * E + E \\
  & \rightarrow & 4 * E + 17 \\
  & \rightarrow & 4 * 3 + 17
\end{array}
\qquad
\begin{array}{lll}
E & \rightarrow & E + E \\
  & \rightarrow & E * E + E \\
  & \rightarrow & E * 3 + E \\
  & \rightarrow & E * 3 + 17 \\
  & \rightarrow & 4 * 3 + 17
\end{array}
$$

Canonical choices:

- **Left-most:** always replace the left-most variable.

Left- vs rightmost derivation

BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



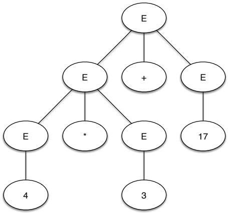
$$
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E \\
&\rightarrow 4 * E + E \\
&\rightarrow 4 * E + 17 \\
&\rightarrow 4 * 3 + 17
\end{aligned}
\qquad
\begin{aligned}
E &\rightarrow E + E \\
&\rightarrow E * E + E \\
&\rightarrow E * 3 + E \\
&\rightarrow E * 3 + 17 \\
&\rightarrow 4 * 3 + 17
\end{aligned}
$$

Canonical choices:

- **Left-most:** always replace the left-most variable.
- **Right-most:** always replace the right-most variable.

Left- vs rightmost derivation

BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



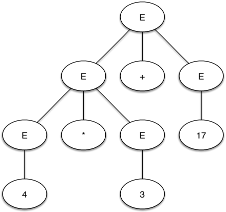
$$\begin{array}{rcl} E & \rightarrow & E + E \\ & \rightarrow & E * E + E \\ & \rightarrow & 4 * E + E \\ & \rightarrow & 4 * E + 17 \\ & \rightarrow & 4 * 3 + 17 \end{array} \qquad \begin{array}{rcl} E & \rightarrow & E + E \\ & \rightarrow & E * E + E \\ & \rightarrow & E * 3 + E \\ & \rightarrow & E * 3 + 17 \\ & \rightarrow & 4 * 3 + 17 \end{array}$$

Canonical choices:

- **Left-most:** always replace the left-most variable.
- **Right-most:** always replace the right-most variable.
- NB the examples above were neither left-nor right-most!

Left- vs rightmost derivation

BUT ... usually there are many derivations for a chosen string, giving the same parse tree. For example:



$$\begin{aligned} E &\rightarrow E + E \\ &\rightarrow E * E + E \\ &\rightarrow 4 * E + E \\ &\rightarrow 4 * E + 17 \\ &\rightarrow 4 * 3 + 17 \end{aligned} \qquad \begin{aligned} E &\rightarrow E + E \\ &\rightarrow E * E + E \\ &\rightarrow E * 3 + E \\ &\rightarrow E * 3 + 17 \\ &\rightarrow 4 * 3 + 17 \end{aligned}$$

Canonical choices:

- **Left-most:** always replace the left-most variable.
- **Right-most:** always replace the right-most variable.
- NB the examples above were neither left-nor right-most!

In parsing we usually use either left- or rightmost derivations to construct a parse tree.

Left- vs rightmost derivation

Question: do left- and rightmost derivations lead to the same parse tree?

Left- vs rightmost derivation

Question: do left- and rightmost derivations lead to the same parse tree?

Answer: For a context-free grammar: YES. It really doesn't matter in what order variables are rewritten. Why?

Left- vs rightmost derivation

Question: do left- and rightmost derivations lead to the same parse tree?

Answer: For a context-free grammar: YES. It really doesn't matter in what order variables are rewritten. Why? Because the rest of the string is unaffected by rewriting a variable, so we can modify the order.

Which rule to choose?

Which rule to choose?

Alas there is a second degree of freedom:

Which rule to choose?

Alas there is a second degree of freedom: which rule to choose?

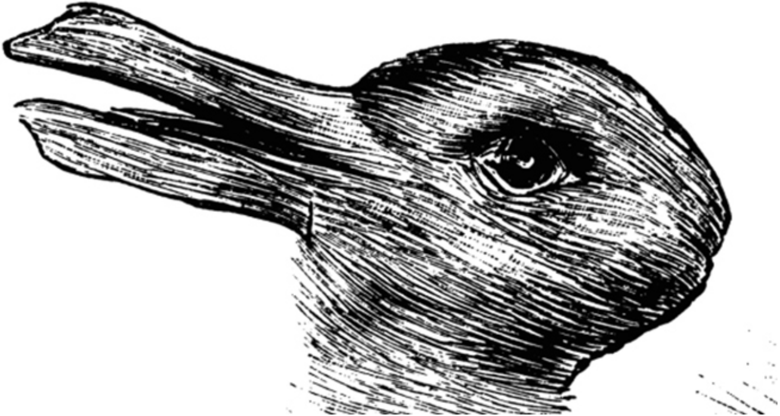
Which rule to choose?

Alas there is a second degree of freedom: which rule to choose?

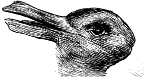
In constrast: it can make a big difference which **rule** we apply when rewriting a variable

This leads to an important subject: **ambiguity.**

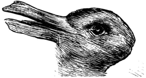
Ambiguity



Ambiguity



Ambiguity

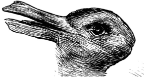


In the grammar

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid ...$$

the string 4 * 3 + 17 has **two** distinct parse trees!

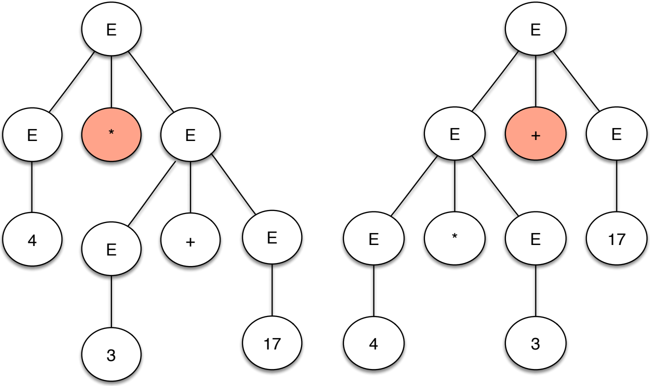
Ambiguity



In the grammar

$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

the string 4 * 3 + 17 has **two** distinct parse trees!

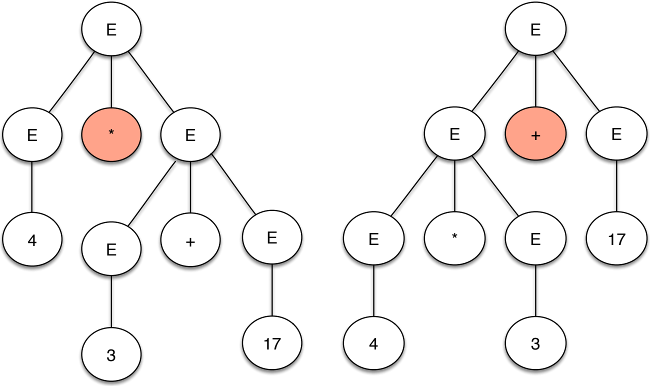


Ambiguity

In the grammar

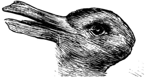
$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

the string 4 * 3 + 17 has **two** distinct parse trees!



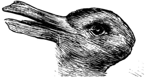
A CFG with this property is called **ambiguous**.

Ambiguity



More precisely: a context-free grammar is **ambiguous** if there is a string in the language of the grammar that has more than one parse tree.

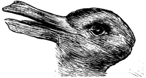
Ambiguity



More precisely: a context-free grammar is **ambiguous** if there is a string in the language of the grammar that has more than one parse tree.

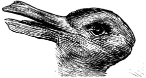
Note that this has **nothing** to do with left- vs right-derivation. Each of the ambiguous parse trees has a left- and a right-derivation.

Ambiguity



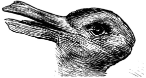
We also have ambiguity in natural language, e.g.

Ambiguity



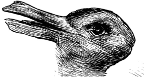
We also have ambiguity in natural language, e.g.

Ambiguity



Ambiguity is programming languages is bad, because it leaves the meaning of a program unclear, e.g. the compiler should generate different code for $1 + 2 * 3$ when it's uderstood as $(1 + 2) * 3$ than for $1 + (2 * 3)$.

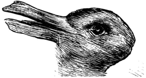
Ambiguity



Ambiguity is programming languages is bad, because it leaves the meaning of a program unclear, e.g. the compiler should generate different code for $1 + 2 * 3$ when it's uderstood as $(1 + 2) * 3$ than for $1 + (2 * 3)$.

Can we automatically check whether a grammar is ambigouous?

Ambiguity

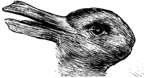


Ambiguity is programming languages is bad, because it leaves the meaning of a program unclear, e.g. the compiler should generate different code for $1 + 2 * 3$ when it's uderstood as $(1 + 2) * 3$ than for $1 + (2 * 3)$.

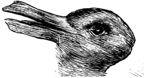
Can we automatically check whether a grammar is ambigouous?

Bad news: ambiguity of grammars is undecidable, i.e. no algorithm can exist that takes as input a CFG and returns "Ambiguous" or "Not ambiguous" correctly for all CFGs.

Ambiguity

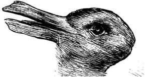


Ambiguity



There are several ways to deal with ambiguity.

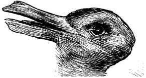
Ambiguity



There are several ways to deal with ambiguity.

- **Parser returns all possible parse trees**, leaving choice to later compiler phases. Example: combinator parsers often do this, Earley parser. Downside: kicks can down the road ... need to disambiguate later (i.e. doesn't really solve the problem), and does too much work if some of the parse trees are later discarded.

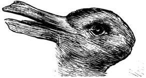
Ambiguity



There are several ways to deal with ambiguity.

- **Parser returns all possible parse trees**, leaving choice to later compiler phases. Example: combinator parsers often do this, Earley parser. Downside: kicks can down the road ... need to disambiguate later (i.e. doesn't really solve the problem), and does too much work if some of the parse trees are later discarded.
- **Use non-ambiguous grammar**. Easier said than done ...

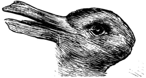
Ambiguity



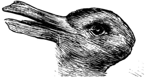
There are several ways to deal with ambiguity.

- **Parser returns all possible parse trees**, leaving choice to later compiler phases. Example: combinator parsers often do this, Earley parser. Downside: kicks can down the road ... need to disambiguate later (i.e. doesn't really solve the problem), and does too much work if some of the parse trees are later discarded.
- **Use non-ambiguous grammar**. Easier said than done ...
- **Rewriting the grammar to remove ambiguity**. For example by enforcing precedence that * binds more tightly than +. We look at this now.

Ambiguity: grammar rewriting



Ambiguity: grammar rewriting

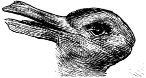


The problem with

$$E \quad \to \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

is that addition and multiplication have the same status. But in our everyday understanding, we think of $a * b + c$ as meaning $(a * b) + c$. Moreover, we evaluate $a + b + c$ as $(a + b) + c$. But there's nothing in the naive grammar that ensures this.

Ambiguity: grammar rewriting



The problem with

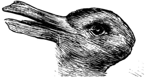
$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

is that addition and multiplication have the same status. But in our everyday understanding, we think of $a * b + c$ as meaning $(a * b) + c$. Moreover, we evaluate $a + b + c$ as $(a + b) + c$. But there's nothing in the naive grammar that ensures this.

Let's bake these preferences into the grammar.

Ambiguity: grammar rewriting

Ambiguity: grammar rewriting



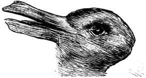
Let's rewrite

$$E \rightarrow E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

to

$$\begin{aligned} E &\rightarrow F + E \mid F \\ F &\rightarrow N * F \mid N \mid (E) * F \mid (E) \\ N &\rightarrow 0 \mid 1 \mid \ldots \end{aligned}$$

Ambiguity: grammar rewriting



Let's rewrite

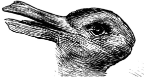
$$E \quad \rightarrow \quad E + E \mid E * E \mid (E) \mid 0 \mid 1 \mid \ldots$$

to

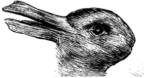
$$\begin{array}{lcl} E & \rightarrow & F + E \mid F \\ F & \rightarrow & N * F \mid N \mid (E) * F \mid (E) \\ N & \rightarrow & 0 \mid 1 \mid \ldots \end{array}$$

Examples in class.

If-Then ambiguity



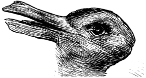
If-Then ambiguity



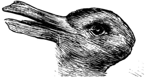
Here is a problem that often arises when specifying programming languages.

$$
\begin{array}{lcl}
M & \rightarrow & \texttt{if}\ M\ \texttt{then}\ M \\
  & \mid & \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M \\
  & \mid & \ldots
\end{array}
$$

If-Then ambiguity



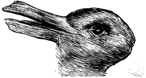
If-Then ambiguity



Now we can find two distinct parse trees for

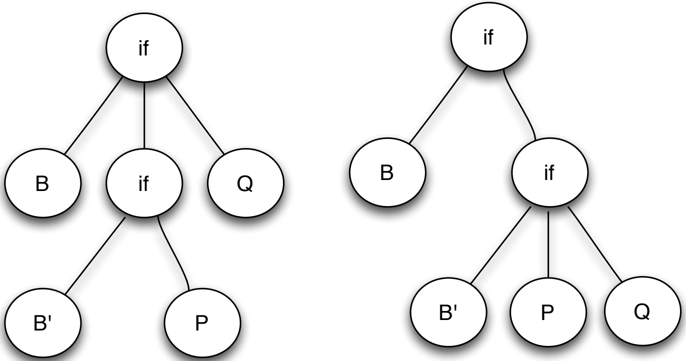
```
if B then if B′ then P else Q
```

If-Then ambiguity



Now we can find two distinct parse trees for

```
if B then if B′ then P else Q
```



If-Then ambiguity

If-Then ambiguity

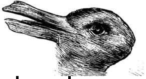
We solved the */+ ambiguity by giving * precedence. At the level of grammar that meant we had + coming 'first' in the grammar.

If-Then ambiguity

We solved the */+ ambiguity by giving * precedence. At the level of grammar that meant we had + coming 'first' in the grammar.

Let's do this for the if-then ambiguity by saying:

If-Then ambiguity



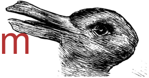
We solved the */+ ambiguity by giving * precedence. At the level of grammar that meant we had + coming 'first' in the grammar.

Let's do this for the if-then ambiguity by saying:

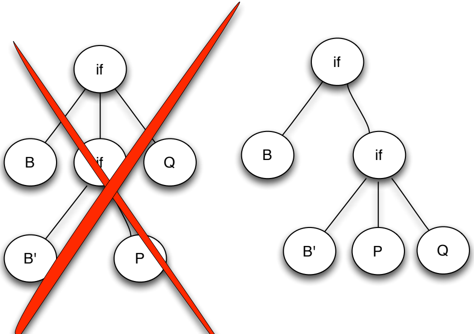
`else` always closes the nearest **unclosed** `if`, so `if-then-else` has priority over `if-then`.

M	$\rightarrow$	ITE	only if-then-else
	$\mid$	$BOTH$	both if-then and if-then-else
	$\mid$	...	
ITE	$\rightarrow$	`if` M `then` ITE `else` ITE	
	$\mid$	...	other reductions
$BOTH$	$\rightarrow$	`if` M `then` M	
	$\mid$	`if` M `then` ITE `else` $BOTH$	no other reductions

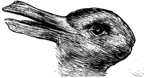
If-Then ambiguity, aka the dangling-else problem



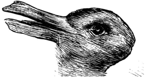
M	$\rightarrow$	ITE	only if-then-else
	$\mid$	$BOTH$	both if-then and if-then-else
ITE	$\rightarrow$	`if` M `then` ITE `else` ITE	
	$\mid$	...	other reductions
$BOTH$	$\rightarrow$	`if` M `then` M	
	$\mid$	`if` M `then` ITE `else` $BOTH$	no other reductions



Ambiguity: general algorithm?

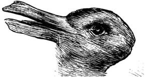


Ambiguity: general algorithm?



Alas there is no algorithm that can rewrite all ambiguous CFGs into unambiguous CFGs with the same language, since some CFGs are **inherently ambiguous**, meaning they are only recongnised by ambiguous CFGs.

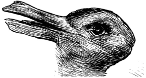
Ambiguity: general algorithm?



Alas there is no algorithm that can rewrite all ambiguous CFGs into unambiguous CFGs with the same language, since some CFGs are **inherently ambiguous**, meaning they are only recongnised by ambiguous CFGs.

Fortunately, such languages are esoteric and not relevant for programming languages. For languages relevant in programming, it is generally straightforward to produce an unambiguous CFG.

Ambiguity: general algorithm?



Alas there is no algorithm that can rewrite all ambiguous CFGs into unambiguous CFGs with the same language, since some CFGs are **inherently ambiguous**, meaning they are only recongnised by ambiguous CFGs.

Fortunately, such languages are esoteric and not relevant for programming languages. For languages relevant in programming, it is generally straightforward to produce an unambiguous CFG.

I will not ask you in the exam to convert an ambiguous CFG into an unambiguous CFG. You should just know what ambiguity means in parsing and why it is a problem.