An Intuitive View of Lexical and Syntax Analysis

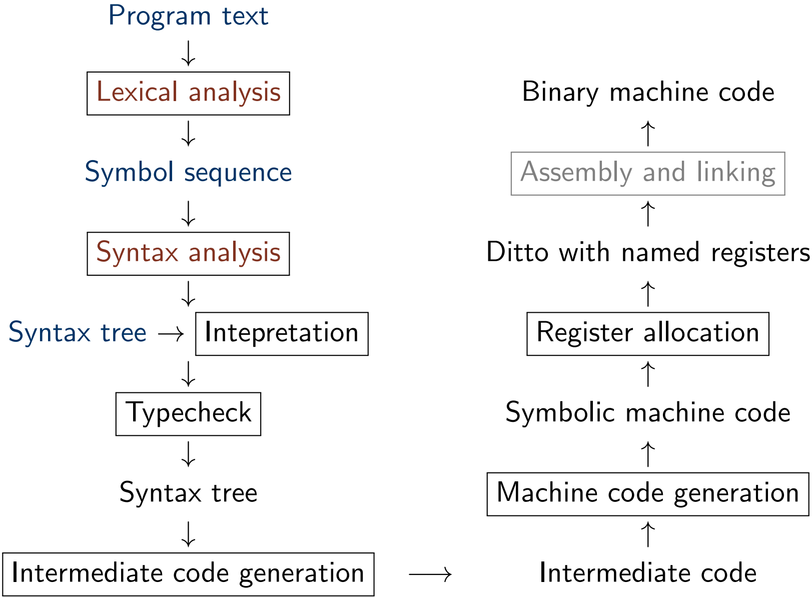
Cosmin E. Oancea
`cosmin.oancea@diku.dk`

Department of Computer Science (DIKU)
University of Copenhagen

February 2018 Compiler Lecture Slides/Notes

1. Intuition: Lexical and Syntactic Analysis
    - Lexical Analysis; Regular Expressions
    - Syntax Analysis; Context-Free Grammars

Structure of a Compiler



Program text

↓

Lexical analysis

↓

Symbol sequence

↓

Syntax analysis

↓

Syntax tree → Intepretation

↓

Typecheck

↓

Syntax tree

↓

Intermediate code generation ⟶ Intermediate code

↑

Machine code generation

↑

Symbolic machine code

↑

Register allocation

↑

Ditto with named registers

↑

Assembly and linking

↑

Binary machine code

1. Intuition: Lexical and Syntactic Analysis
    - Lexical Analysis; Regular Expressions
    - Syntax Analysis; Context-Free Grammars

Lexical Analysis

Lexical: relates to the words of the vocabulary of a language,
(as opposed to grammar, i.e., correct construction of sentences).

- "My mother coooookes dinner not."
- Lexical Analyzer, a.k.a. lexer, scanner or tokenizer, splits the input program, seen as a stream of characters, into a sequence of tokens.
- Tokens are the words of the (programming) language, e.g., keywords, numbers, comments, parenthesis, semicolon.

Compiler Phases

```
// My program
let result =
  let x = 10 :: 20 :: 0x30 :: []
  List.map (fun a -> 2 * 2 * a) x
```

- Input file also contains
- comments and meaningful formatting, which helps user only.
- Input file is read as a string, see below:

```
// My program\n let result =\n  let x = 10 :: 20 :: 0x30 :: []\n
List.map (fun a -> 2 * 2 * a) x
```

Compiler Phases

```
// My program
let result =
  let x = 10 :: 20 :: 0x30 :: []
  List.map (fun a -> 2 * 2 * a) x
```

- Input file also contains
- comments and meaningful formatting, which helps user only.
- Input file is read as a string, see below:

```
// My program\n let result =\n  let x = 10 :: 20 :: 0x30 :: []\n
List.map (fun a -> 2 * 2 * a) x
```

Lexical Analysis: transforms a character stream to a token sequence.

```
Keywd_Let, Id "result", Equal, Keywd_Let, Id "x", Equal, Int 10,
Op_Cons, Int 20, Op_Cons, Int 48, Op_Cons, LBracket, RBracket,
Id "List", Dot, Id "map", LParen, Keywd_fun Id "a", Arrow, Int 2, Multiply,
Int 2, Multiply, Id "a", RParen, Id "x"
```

Compiler Phases

```
// My program
let result =
  let x = 10 :: 20 :: 0x30 :: []
  List.map (fun a -> 2 * 2 * a) x
```

- Input file also contains
- comments and meaningful formatting, which helps user only.
- Input file is read as a string, see below:

```
// My program\n let result =\n  let x = 10 :: 20 :: 0x30 :: []\n
List.map (fun a -> 2 * 2 * a) x
```

Lexical Analysis: transforms a character stream to a token sequence.

```
Keywd_Let, Id "result", Equal, Keywd_Let, Id "x", Equal, Int 10,
Op_Cons, Int 20, Op_Cons, Int 48, Op_Cons, LBracket, RBracket,
Id "List", Dot, Id "map", LParen, Keywd_fun Id "a", Arrow, Int 2, Multiply,
Int 2, Multiply, Id "a", RParen, Id "x"
```

- Tokens can be: (fixed) vocabulary words, e.g., keywords (`let`), built-in operators (∗, `::`), special symbols ({, }).
- Identifiers and Number Literals are **classes** of tokens, which are formed compositionally according tor certain rules.

Formalism

Definition (Formal Languages)

Let $\sum$ be an alphabet, i.e., a finite set of allowed characters.

- **A word** *over $\sum$ is a string of chars $w = a_1 a_2 \ldots a_n$, $a_i \in \sum$*
  $n = 0$ is allowed and results in the empty string, denoted ϵ.
  $\sum^*$ is the set of all words over $\sum$.
- **A language** *L over $\sum$ is a set of words over $\sum$, i.e., $L \subset \sum^*$.*

Examples over the alphabet of small latin letters:

- $\sum^*$ and $\emptyset$
- All C keywords: {`if`, `else`, `return`, `do`, `while`, `for`, ...}
- $\{a^n b^n\}, \forall n \geq 0$
- All palindromes: {`kayak`, `racecar`, `mellem`, `retter`}
- $\{a^n b^n c^n\}, \forall n \geq 0$

Languages

Aim of compiler's front end: decide whether a program respects the language rules.

Lexical analysis: decides whether the individual tokens are well formed, i.e., requires the implementation of a simple language.

Syntactical Analysis: decides whether the composition of tokens is well formed, i.e., more complex language that checks compliance to grammar rules.

Type Checker: verifies that the program complies with (some of) the language semantics.

Language Examples: Number Literals in C++

- Integers in decimal format: `234`, `0`, `8` but not `08` or `abc`!
- Integers in hexadecimal format: `0X123`, `0xcafe` but not `0X`, `0XG`!
- Floating point decimals: `0.` or `.345` or `123.45`.
- Scientific notation: `234E-45` or `0.E123` or `.234e+45`.

Language Examples: Number Literals in C++

- Integers in decimal format: `234`, `0`, `8` but not `08` or `abc`!
- Integers in hexadecimal format: `0X123`, `0xcafe` but not `0X`, `0XG`!
- Floating point decimals: `0.` or `.345` or `123.45`.
- Scientific notation: `234E-45` or `0.E123` or `.234e+45`.

- A decimal integer is either `0` or a sequence of digits (`0-9`) that does not start with `0`.
- A hexadecimal integer starts with `0x` or `0X` and is followed by one or more hexadecimal digits (`0-9` or `a-f` or `A-F`).

Language Examples: Number Literals in C++

- Integers in decimal format: `234`, `0`, `8` but not `08` or `abc`!
- Integers in hexadecimal format: `0X123`, `0xcafe` but not `0X`, `0XG`!
- Floating point decimals: `0.` or `.345` or `123.45`.
- Scientific notation: `234E-45` or `0.E123` or `.234e+45`.

- A decimal integer is either `0` or a sequence of digits (`0-9`) that does not start with `0`.
- A hexadecimal integer starts with `0x` or `0X` and is followed by one or more hexadecimal digits (`0-9` or `a-f` or `A-F`).
- Floating-point cts have a “mantissa,” [..][and] an “exponent,” [..]. The mantissa is as a sequence of digits followed by a period, followed by an optional sequence of digits[..]. The exponent, if present, specifies the magnitude[..] using e or E[..] followed by an optional sign (+ or -) and a sequence of digits. If an exponent is present, the trailing decimal point is unnecessary in whole numbers. http://msdn.microsoft.com/en-us/library/tfh6f0w2.aspx.

Regular Expressions

We need a formal, compositional (and intuitive) description of what tokens are, *and* automatic implementation of the token language.

Definition (Regular Expressions)

The set $RE(\sum)$ *of regular expressions over alphabet* $\sum$ *is defined:*

- *Base Rules (Non Recursive):*
  - $\epsilon \in RE(\sum)$ *describes the lang consisting of only the empty string.*

Regular Expressions

We need a formal, compositional (and intuitive) description of what tokens are, *and* automatic implementation of the token language.

Definition (Regular Expressions)

The set RE($\sum$) of regular expressions over alphabet $\sum$ is defined:

- *Base Rules (Non Recursive):*
  - $\epsilon \in$ *RE*($\sum$) *describes the lang consisting of only the empty string.*
  - a $\in$ *RE*($\sum$) *for* a $\in \sum$ *describes the lang. of one-letter word* a*.*
- *Recursive Rules: for every* $\alpha, \beta \in$ *RE*($\sum$)

Regular Expressions

We need a formal, compositional (and intuitive) description of what tokens are, *and* automatic implementation of the token language.

Definition (Regular Expressions)

The set RE($\sum$) of regular expressions over alphabet $\sum$ is defined:

- *Base Rules (Non Recursive):*
  - $\epsilon \in$ *RE*($\sum$) *describes the lang consisting of only the empty string.*
  - $\mathtt{a} \in$ *RE*($\sum$) *for* $\mathtt{a} \in \sum$ *describes the lang. of one-letter word* $\mathtt{a}$.
- *Recursive Rules: for every* $\alpha, \beta \in$ *RE*($\sum$)
  - $\alpha \cdot \beta \in$ *RE*($\sum$), *two language sequence/concatenation in which the first word is described by* α, *the second word by* β.

Regular Expressions

We need a formal, compositional (and intuitive) description of what tokens are, *and* automatic implementation of the token language.

Definition (Regular Expressions)

The set $RE(\sum)$ of regular expressions over alphabet $\sum$ is defined:

- *Base Rules (Non Recursive):*
    - $\epsilon \in RE(\sum)$ *describes the lang consisting of only the empty string.*
    - $\mathtt{a} \in RE(\sum)$ *for* $\mathtt{a} \in \sum$ *describes the lang. of one-letter word* $\mathtt{a}$.
- *Recursive Rules: for every* $\alpha, \beta \in RE(\sum)$
    - $\alpha \cdot \beta \in RE(\sum)$, *two language sequence/concatenation in which the first word is described by* α, *the second word by* β.
    - $\alpha \mid \beta \in RE(\sum)$, *alternative/union: lang described by* α *OR* β.

Regular Expressions

We need a formal, compositional (and intuitive) description of what tokens are, *and* automatic implementation of the token language.

Definition (Regular Expressions)

The set $RE(\sum)$ of regular expressions over alphabet $\sum$ is defined:

- *Base Rules (Non Recursive):*
  - $\epsilon \in RE(\sum)$ *describes the lang consisting of only the empty string.*
  - $\mathtt{a} \in RE(\sum)$ *for* $\mathtt{a} \in \sum$ *describes the lang. of one-letter word* $\mathtt{a}$.
- *Recursive Rules: for every* $\alpha, \beta \in RE(\sum)$
  - $\alpha \cdot \beta \in RE(\sum)$, *two language sequence/concatenation in which the first word is described by* α, *the second word by* β.
  - $\alpha \mid \beta \in RE(\sum)$, *alternative/union: lang described by* α *OR* β.
  - $\alpha^* \in RE(\sum)$, *repetition: zero or more words described by* α.

- One may use parenthesis `(...)` for grouping regular expressions.
- Sequence binds tighter than alternative: $\mathtt{a|bc}^* = \mathtt{a|(b(c}^*\mathtt{))}$.

Demonstrating Regular-Expression Combinators

$\alpha \cdot \beta$ Assume the language of regular expression α and β are `L(`α`)={"a","b"}` and `L(`β`)={"c","d"}`, respectively.

Then `L(`$\alpha \cdot \beta$`)=`

Demonstrating Regular-Expression Combinators

$\alpha \cdot \beta$ Assume the language of regular expression α and β are `L(`α`)={"a","b"}` and `L(`β`)={"c","d"}`, respectively.

Then `L(`$\alpha \cdot \beta$`)={"ac", "ad", "bc", "bd"}`.

When matching keywords, `if` is the concatenation of two regular expressions: `i` and `f`.

α^* Assume the language of regular expression α is `L(`α`)={"a","b"}`.

Then

`L(`α^*`)=`

Demonstrating Regular-Expression Combinators

$\alpha \cdot \beta$ Assume the language of regular expression α and β are `L(`α`)={"a","b"}` and `L(`β`)={"c","d"}`, respectively.

Then `L(`$\alpha \cdot \beta$`)={"ac", "ad", "bc", "bd"}`.

When matching keywords, `if` is the concatenation of two regular expressions: `i` and `f`.

α^* Assume the language of regular expression α is `L(`α`)={"a","b"}`.

Then

`L(`α^*`)={"","a","b","aa","ab","ba","bb","aaa",...}`.

Examples: Integers and Variable Names in C++

- Integers in decimal format: `234`, `0`, `8` but not `08` or `abc`!
- Integers in hexadecimal format: `0X123`, `0xcafe` but not `0X`, `0XG`!
- A variable name consists of letters, digits and underscore, and it must begin with a letter or underscore.

Examples: Integers and Variable Names in C++

- Integers in decimal format: `234`, `0`, `8` but not `08` or `abc`!
- Integers in hexadecimal format: `0X123`, `0xcafe` but not `0X`, `0XG`!
- A variable name consists of letters, digits and underscore, and it must begin with a letter or underscore.

- Integers in decimal format:
  `(1|2|...|9)(0|1|2|...|9)* | 0`
  Shorthand via character range (`[-]`): `[1-9][0-9]* | 0`
- Integers in hexadecimal format:
  `0 (x|X) [0-9a-fA-F][0-9a-fA-F]*`
  Shorthand via at least one (`+`): `0 (x|X) [0-9a-fA-F]+`.
- Variable names: `[a-zA-Z_][a-zA-Z_0-9]*`

Useful Abbreviations for Regular Expressions

- Character Sets: `[`$a_1a_2\dots a_n$`]` `:=` `(`a_1 `|` a_2 `|` `...` `|` a_n`)`, i.e., one of a_1, a_2, $\dots$, $a_n \in \sum$.
- Negation: `[^` $a_1a_2\dots$`]` describes any a $\in \sum \setminus \{a_1, a_2, \dots, a_n\}$.
- Character Ranges: `[`a_1`-`a_n`]` `:=` `(`a_1 `|` a_2 `|` a_n`)`, where $\{a_i\}$ is ordered, i.e., one character in the range between a_1 and a_n.
- Optional Parts: $\alpha?$ `:=` $(\alpha|\epsilon)$ `for` $\alpha \in RE(\sum)$, optionally a string described by α.
- Repeated Parts: α^+ `:=` $(\alpha\alpha^*)$ `for` $\alpha \in RE(\sum)$, at least ONE string describing α (but possibly more).

Properties of Regular Expression Combinators

- | is associative: `(r|s)|t = r|(s|t) = r|s|t`
- | is commutative: `s|t = t|s`
- | is idempotent: `s|s = s`
- Also, by definition, `s? = s|`ϵ

- $\cdot$ is associative: `(rs)t = r(st) = rst`
- ϵ neutral element for $\cdot$: `s`ϵ `=` ϵ`s = s`
- $\cdot$ distributes over |: `r(s|t) = rs|rt` and `(r|s)t = rt|st`.

- $*$ is idempotent: $(s^*)^* = s^*$.
- Also, $s^*s^* = s^*$ and $ss^* = s^+ = s^*s$ by definition!

1. Intuition: Lexical and Syntactic Analysis
    - Lexical Analysis; Regular Expressions
    - Syntax Analysis; Context-Free Grammars

Syntax Analysis (Parsing)

Relates to the correct construction of sentences, i.e., grammar.

1. Checks that grammar is respected, otherwise syntax error, and

2. Arranges tokens into a syntax tree reflecting the text structure: leaves are tokens, which if read from left to right results in the original text!

```
mother
cokes
dinner
My.
```

syntax error

```
My dinner
cokes
mother.
```

semantic error

Essential tool and theory used are *Context-Free Grammars*:
a notation suitable for human understanding that can be transformed into an efficient implementation.

Context-Free Grammar (CFG) Definition

1. a set of *terminals* $\sum$ – the language alphabet, e.g., the set of tokens produced by lexer. (Convention: use small letters.)
2. a set of *non-terminals* N, denoting sets of recursively defined strings.
3. *a start symbol* $S \in N$, denoting the lang defined by the grammar.
4. a set P of productions of form $Y \to X_1 \ldots X_n$, where $Y \in N$ is a (single) non-terminal, and $X_i \in (\sum \cup N), \forall i$ can be a terminal or non-terminal. Each production describes some of the strings of the corresponding non-terminal Y.

G: $S \to aS$
$S \to \epsilon$

Context-Free Grammar (CFG) Definition

1. a set of *terminals* $\sum$ – the language alphabet, e.g., the set of tokens produced by lexer. (Convention: use small letters.)
2. a set of *non-terminals* N, denoting sets of recursively defined strings.
3. *a start symbol* $S \in N$, denoting the lang defined by the grammar.
4. a set P of productions of form $Y \to X_1 \ldots X_n$, where $Y \in N$ is a (single) non-terminal, and $X_i \in (\sum \cup N), \forall i$ can be a terminal or non-terminal. Each production describes some of the strings of the corresponding non-terminal Y.

G: $S \to aS$
$S \to \epsilon$
regular-expression language a^*

G: $S \to aSb$
$S \to \epsilon$

Context-Free Grammar (CFG) Definition

1. a set of *terminals* $\sum$ – the language alphabet, e.g., the set of tokens produced by lexer. (Convention: use small letters.)
2. a set of *non-terminals* N, denoting sets of recursively defined strings.
3. *a start symbol* $S \in N$, denoting the lang defined by the grammar.
4. a set P of productions of form $Y \rightarrow X_1 \ldots X_n$, where $Y \in N$ is a (single) non-terminal, and $X_i \in (\sum \cup N), \forall i$ can be a terminal or non-terminal. Each production describes some of the strings of the corresponding non-terminal Y.

G: $S \rightarrow aS$
$S \rightarrow \epsilon$
regular-expression language a^*

G: $S \rightarrow aSb$
$S \rightarrow \epsilon$
describes language $\{a^n b^n, \forall n \geq 0\}$

G: $S \rightarrow aSa \mid bSb \mid \ldots$
$S \rightarrow a \mid b \mid \ldots \mid \epsilon$

Context-Free Grammar (CFG) Definition

1. a set of *terminals* $\sum$ – the language alphabet, e.g., the set of tokens produced by lexer. (Convention: use small letters.)
2. a set of *non-terminals* N, denoting sets of recursively defined strings.
3. *a start symbol* $S \in N$, denoting the lang defined by the grammar.
4. a set P of productions of form $Y \rightarrow X_1 \ldots X_n$, where $Y \in N$ is a (single) non-terminal, and $X_i \in (\sum \cup N), \forall i$ can be a terminal or non-terminal. Each production describes some of the strings of the corresponding non-terminal Y.

G: $S \rightarrow aS$
$S \rightarrow \epsilon$
regular-expression language a^*

G: $S \rightarrow aSb$
$S \rightarrow \epsilon$
describes language $\{a^n b^n, \forall n \geq 0\}$

G: $S \rightarrow aSa \mid bSb \mid \ldots$
$S \rightarrow a \mid b \mid \ldots \mid \epsilon$
describes palyndromes, e.g., *abba*, *babab*.

The latter two languages cannot be described with regular expressions.

Example: Deriving Words

Nonteminals recursively refer to each other
(cannot do that with regular expressions):

$$\begin{array}{llllll}
\text{G: } S \to aSB & (1) & & & & \\
\quad S \to \epsilon & (2) & \text{G: } S \to aSB & \mid \epsilon & & S = \{a\cdot x\cdot y \mid x \in S,\ y \in B\} \cup \{\epsilon\} \\
\quad B \to Bb & (3) & \quad B \to Bb & \mid b & & B = \{x \cdot b \mid x \in B\} \cup \{b\} \\
\quad B \to b & (4) & & & &
\end{array}$$

- Words of the language can be constructed by
- starting with the start symbol S, and
- successively replacing nonterminals with right-hand sides.

$$S \ \Rightarrow^1\ \underline{aSB}\ \Rightarrow^1\ a\underline{aSB}B\ \Rightarrow^4\ aaS\underline{b}B\ \Rightarrow^1\ aa\underline{aSB}bB$$

Example: Deriving Words

Nonteminals recursively refer to each other
(cannot do that with regular expressions):

$$
\begin{array}{lll}
\text{G: } S \to aSB & (1) & \\
\phantom{\text{G: }} S \to \epsilon & (2) & \text{G: } S \to aSB \mid \epsilon \quad S = \{a \cdot x \cdot y \mid x \in S,\ y \in B\} \cup \{\epsilon\} \\
\phantom{\text{G: }} B \to Bb & (3) & \phantom{\text{G: }} B \to Bb \quad \mid b \quad B = \{x \cdot b \mid x \in B\} \cup \{b\} \\
\phantom{\text{G: }} B \to b & (4) &
\end{array}
$$

- Words of the language can be constructed by
- starting with the start symbol S, and
- successively replacing nonterminals with right-hand sides.

$$
\begin{array}{l}
S \Rightarrow^1 \underline{aSB} \Rightarrow^1 a\underline{aSB}B \Rightarrow^4 aaS\underline{b}B \Rightarrow^1 aa\underline{aSB}bB \\
\quad \Rightarrow^1 aaa\underline{\ }BbB \Rightarrow^3 aaa\underline{Bb}bB \Rightarrow^4 aaaBbb\underline{b} \Rightarrow^4 aaa\underline{b}bbb.
\end{array}
$$

Definition: Derivation Relation

Let $G = (\sum, N, S, P)$ be a grammar.
The derivation relation $\Rightarrow$ on $(\sum \cup N)^*$ is defined as:

- for a nonterminal $X \in N$ and a production $(X \rightarrow \beta) \in P$, $\alpha_1 X \alpha_2 \;\Rightarrow\; \alpha_1 \beta \alpha_2$, for all $\alpha_1, \alpha_2 \in (\sum \cup N)^*$
- describes one derivation step using one of the productions.
- Production can be numbered with the grammar-rule number.

G: $S \rightarrow aSB$ (1)
$S \rightarrow \epsilon$ (2)
$B \rightarrow Bb$ (3)
$B \rightarrow b$ (4)

$$S \;\Rightarrow^1\; \underline{aSB} \;\Rightarrow^1\; a\underline{aSB}B \;\Rightarrow^2\; aa\underline{\ }BB$$

Definition: Derivation Relation

Let $G = (\sum, N, S, P)$ be a grammar.
The derivation relation $\Rightarrow$ on $(\sum \cup N)^*$ is defined as:

- for a nonterminal $X \in N$ and a production $(X \rightarrow \beta) \in P$, $\alpha_1 X \alpha_2 \;\Rightarrow\; \alpha_1 \beta \alpha_2$, for all $\alpha_1, \alpha_2 \in (\sum \cup N)^*$
- describes one derivation step using one of the productions.
- Production can be numbered with the grammar-rule number.

G: $S \rightarrow aSB$ (1)
$S \rightarrow \epsilon$ (2)
$B \rightarrow Bb$ (3)
$B \rightarrow b$ (4)

$$S \Rightarrow^1 \underline{aSB} \Rightarrow^1 a\underline{aSB}B \Rightarrow^2 aa\underline{\;}BB$$
$$\Rightarrow^3 aa\underline{Bb}B \Rightarrow^4 aa\underline{b}bB \Rightarrow^4 aabb\underline{b}.$$

- Here we have used leftmost derivation, i.e., always expanded the leftmost terminal first. Could also use right-most derivation.
- $aaabbbb$ and $aabbb \in L(G)$.

Transitive Derivation Relation Definition

Let $G = (\sum, N, S, P)$ be a grammar and $\Rightarrow$ its derivation relation. The transitive derivation relation is defined as:

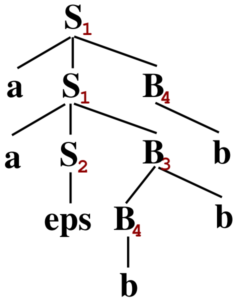
- $\alpha \Rightarrow^* \alpha$, for $\alpha \in (\sum \cup N)^*$, derived in 0 steps,
- for $\alpha, \beta \in (\sum \cup N)^*$, $\alpha \Rightarrow^* \beta$ *iff* there exists $\gamma \in (\sum \cup N)^*$ such that $\alpha \Rightarrow \gamma$, and $\gamma \Rightarrow^* \beta$, i.e., derived in at least one step.

The Language of a Grammar consists of all the words that can be obtained via the transitive derivation relation:
$L(G) = \{w \in \sum^* \mid S \Rightarrow^* w\}$.

For example *aaabbbb* and *aabbb* $\in L(G)$,
because $S \Rightarrow^* aaabbbb$ and $S \Rightarrow^* aabbb$.

Syntax Trees

G: $S \rightarrow aSB$ (1)
$S \rightarrow \epsilon$ (2)
$B \rightarrow Bb$ (3)
$B \rightarrow b$ (4)



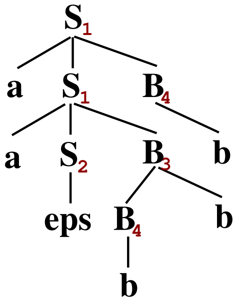
Syntax trees describe the "structure" of the derivation (independent of the order in which nonterminals have been chosen to be derived).

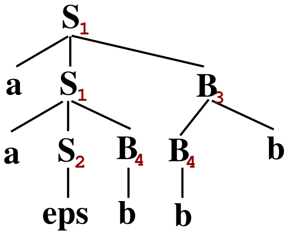
Leftmost derivation always derives the leftmost nonterminal first, and corresponds to a *depth-first, left-to-right tree traversal*:

$$S \Rightarrow^1 \underline{aSB} \Rightarrow^1 a\underline{aSB}B \Rightarrow^2 aa\_BB \Rightarrow^3 aa\underline{Bb}B \Rightarrow^4 aa\underline{b}bB \Rightarrow^4 aabb\underline{b}.$$

Syntax Trees & Ambiguous Grammars

G: $S \rightarrow aSB$ (1)
$S \rightarrow \epsilon$ (2)
$B \rightarrow Bb$ (3)
$B \rightarrow b$ (4)





Syntax trees describe the “structure” of the derivation (independent of the order in which nonterminals have been chosen to be derived).

The grammar is said to be ambiguous if there exists a word that can be derived in two ways corresponding to different syntax trees.

$S \Rightarrow^1 \underline{aSB} \Rightarrow^1 a\underline{aSB}B \Rightarrow^2 aa\underline{\ }BB \Rightarrow^3 aa\underline{Bb}B \Rightarrow^4 aa\underline{b}bB \Rightarrow^4 aabb\underline{b}.$

$S \Rightarrow^1 \underline{aSB} \Rightarrow^1 a\underline{aSB}B \Rightarrow^2 aa\underline{\ }BB \Rightarrow^4 aa\underline{b}B \Rightarrow^3 aab\underline{bB} \Rightarrow^4 aabb\underline{b}.$

Handling/Removing Grammar Ambiguity

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \\ E & \rightarrow & E * E \mid E \;/\; E \\ E & \rightarrow & a \mid (E) \end{array}$$

- *Precedence and Associativity* guide decision:
- ambiguity resolved by parsing directives,
- or by rewriting the grammar.

What are the problems:

- Ambiguous derivation of $a - a - a$

Handling/Removing Grammar Ambiguity

$$\begin{array}{lll} E & \rightarrow & E + E \mid E - E \\ E & \rightarrow & E * E \mid E \;/\; E \\ E & \rightarrow & a \mid (E) \end{array}$$

- *Precedence and Associativity* guide decision:
- ambiguity resolved by parsing directives,
- or by rewriting the grammar.

What are the problems:

- Ambiguous derivation of $a - a - a$ can be resolved by fixing *a left-associative* derivation: $(a - a) - a$.
- Ambiguous derivation of $a + a * a$

Handling/Removing Grammar Ambiguity

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \\ E & \rightarrow & E * E \mid E \;/\; E \\ E & \rightarrow & a \mid (E) \end{array}$$

- *Precedence and Associativity* guide decision:
- ambiguity resolved by parsing directives,
- or by rewriting the grammar.

What are the problems:

- Ambiguous derivation of $a - a - a$ can be resolved by fixing *a left-associative* derivation: $(a - a) - a$.
- Ambiguous derivation of $a + a * a$ can be resolved by setting *the precedence* of $*$ higher than $+$: $a + (a * a)$.

Defining/Resolving Operator Precedence

- Introduce precedence levels to set operator priorities
- for example precedence of $*$ and $/$ over (higher than) $+$ and $-$,
- and more precedence levels can be added, e.g., exponentiation.

Defining/Resolving Operator Precedence

- Introduce precedence levels to set operator priorities
- for example precedence of $*$ and $/$ over (higher than) $+$ and $-$,
- and more precedence levels can be added, e.g., exponentiation.

At grammar level: this can be accomplished by introducing one nonterminal for each level of precedence:

$$\begin{aligned} E &\rightarrow E + E \mid E - E \\ E &\rightarrow E * E \mid E \,/\, E \\ E &\rightarrow a \mid (E) \end{aligned}$$

$$\begin{aligned} E &\rightarrow E + E \mid E - E \mid T \\ T &\rightarrow T * T \mid T \,/\, T \\ T &\rightarrow a \mid (E) \end{aligned}$$

Defining/Resolving Operator Associativity

A binary operator is called:

- *left associative* if expression $x \oplus y \oplus z$ should be evaluated from left to right: $(x \oplus y) \oplus z$
- *right associative* if expression $x \oplus y \oplus z$ should be evaluated from right to left: $x \oplus (y \oplus z)$
- *non-associative* if expressions such as $x \oplus y \oplus z$ are disallowed,
- associative if both left-to-right and right-to-left evaluations lead to the same result.

Examples:

- left associative operators: $-$ and $/$,
- right associative operators

Defining/Resolving Operator Associativity

A binary operator is called:

- *left associative* if expression $x \oplus y \oplus z$ should be evaluated from left to right: $(x \oplus y) \oplus z$
- *right associative* if expression $x \oplus y \oplus z$ should be evaluated from right to left: $x \oplus (y \oplus z)$
- *non-associative* if expressions such as $x \oplus y \oplus z$ are disallowed,
- associative if both left-to-right and right-to-left evaluations lead to the same result.

Examples:

- left associative operators: $-$ and $/$,
- right associative operators: exponentiation, assignment.

Establishing Intended Associativity

- Can be declared in the parser file via directives
- when operators are associative use same associativity as comparable operators,
- cannot mix left- and right-associative operators at the same precedence level.

Establishing Intended Associativity

- Can be declared in the parser file via directives
- when operators are associative use same associativity as comparable operators,
- cannot mix left- and right-associative operators at the same precedence level.

At grammar level: this can be accomplished by introducing new nonterminals that establish explicitly operator's associativity :

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \mid T \\ T & \rightarrow & T * T \mid T \,/\, T \\ T & \rightarrow & a \mid (E) \end{array} \qquad \begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T \,/\, F \mid F \\ F & \rightarrow & a \mid (E) \end{array}$$

Establishing Intended Associativity

- Can be declared in the parser file via directives
- when operators are associative use same associativity as comparable operators,
- cannot mix left- and right-associative operators at the same precedence level.

At grammar level: this can be accomplished by introducing new nonterminals that establish explicitly operator's associativity :

$$\begin{array}{lcl} E & \rightarrow & E + E \mid E - E \mid T \\ T & \rightarrow & T * T \mid T \,/\, T \\ T & \rightarrow & a \mid (E) \end{array} \qquad \begin{array}{lcl} E & \rightarrow & E + T \mid E - T \mid T \\ T & \rightarrow & T * F \mid T \,/\, F \mid F \\ F & \rightarrow & a \mid (E) \end{array}$$

- Left associative ⇒ Left-recursive grammar production.
- Right associative ⇒ Right-recursive grammar production.