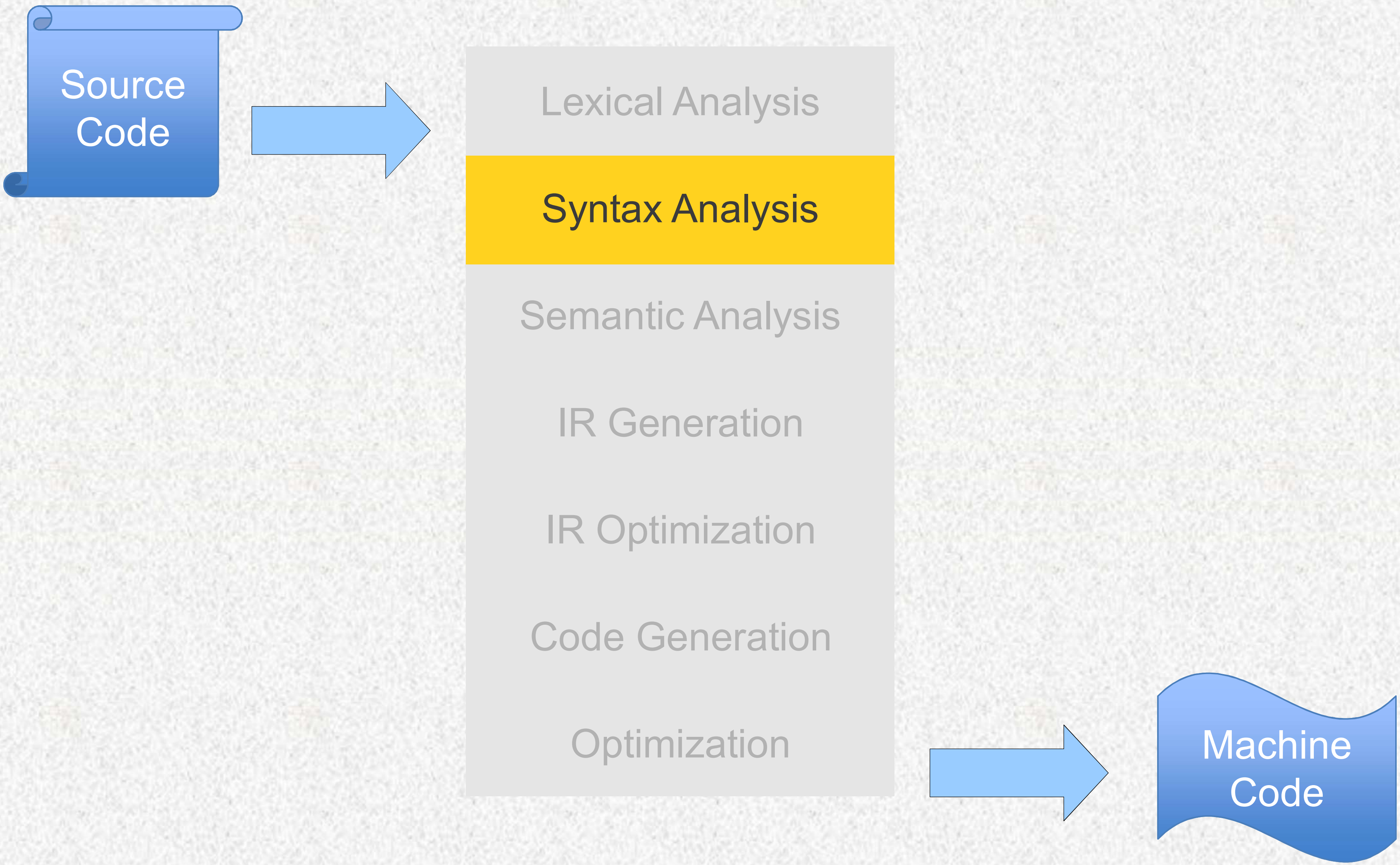
Lecture 5: Syntax Analysis

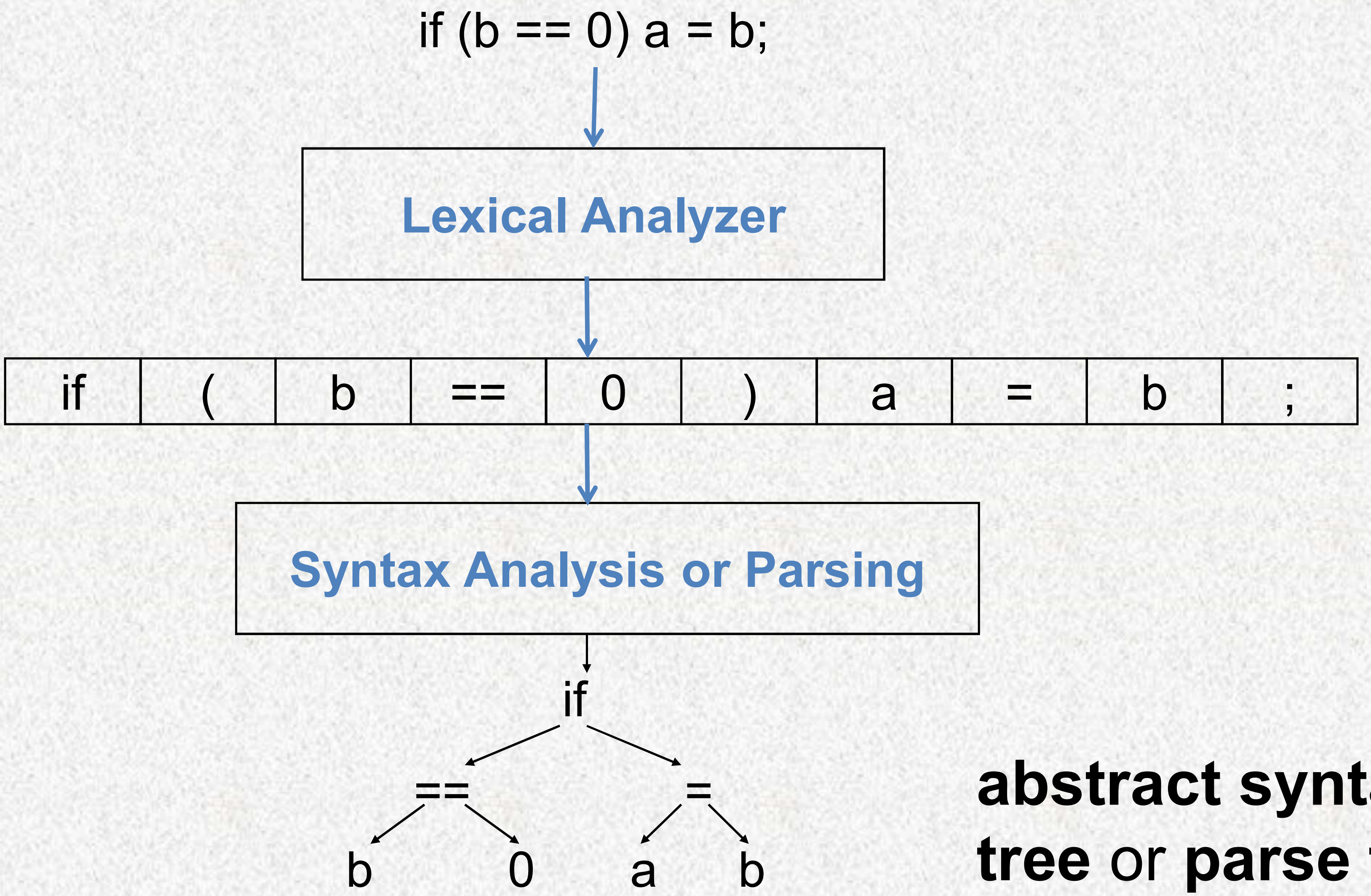
Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

Syntax Analysis

Where are we ?


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

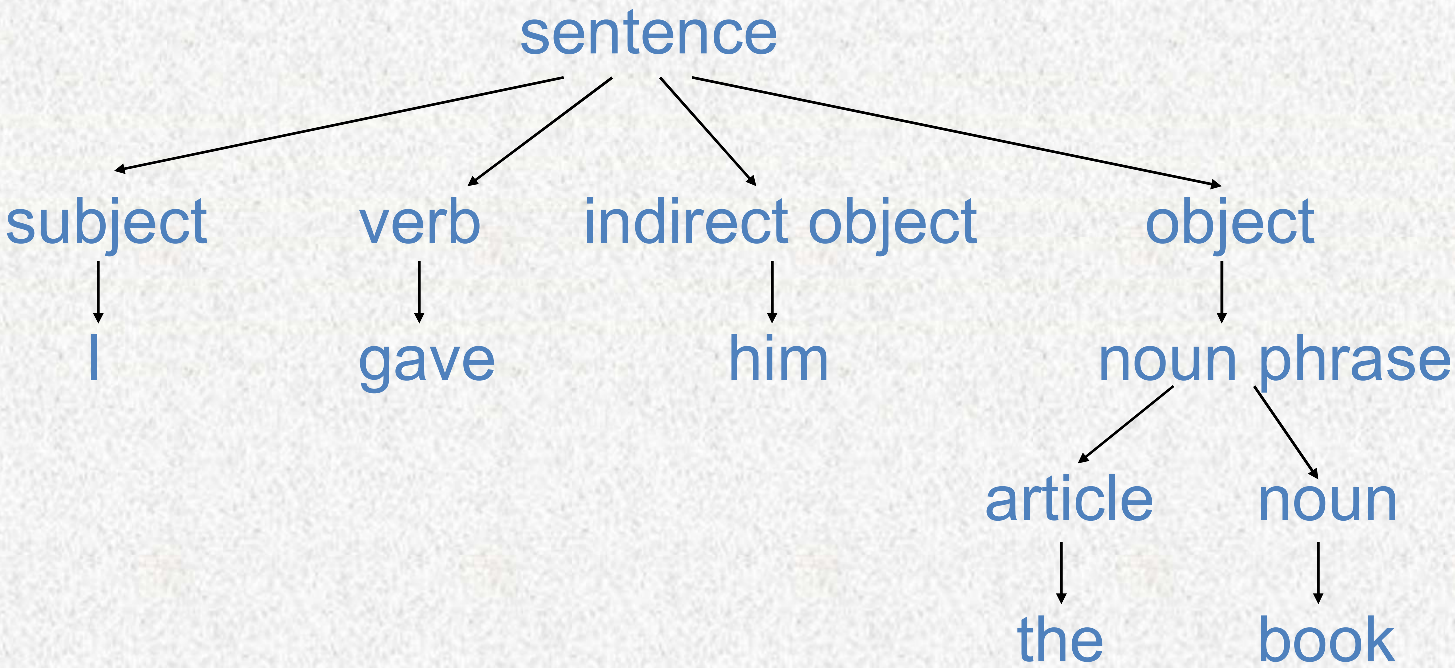
Where is Syntax Analysis Performed?



abstract syntax tree or **parse tree**

Parsing Analogy

- Syntax analysis for natural languages
  - Recognize whether a sentence is grammatically correct
  - Identify the function of each word

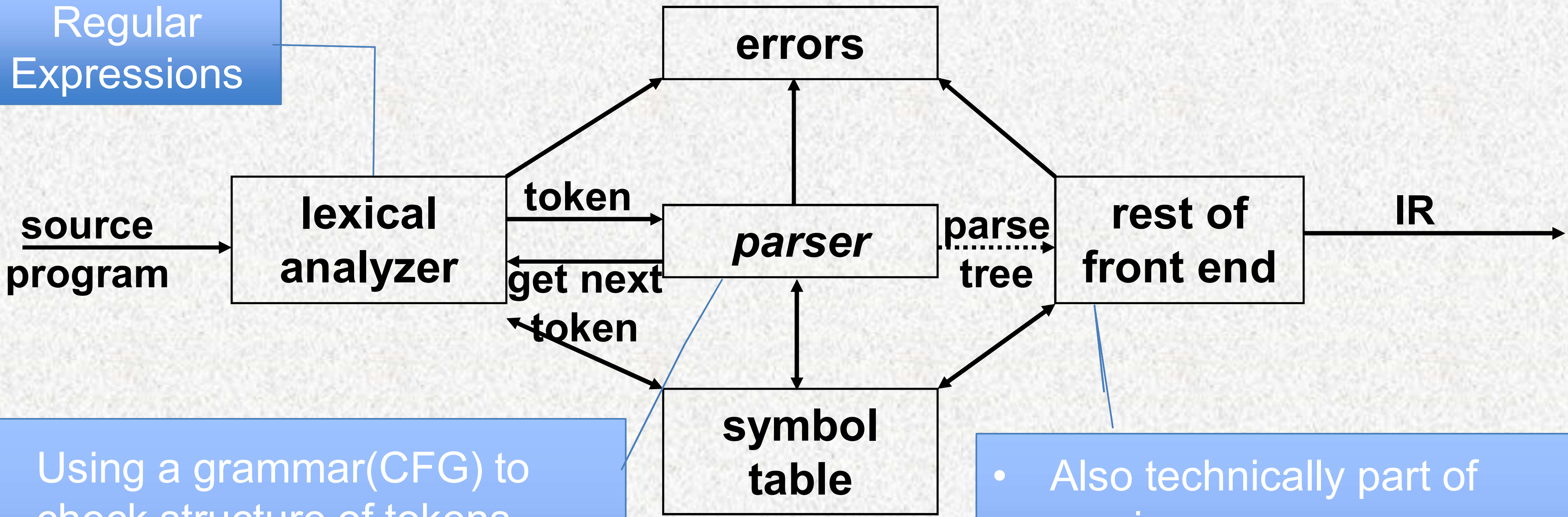


"I gave him the book"

Parsing During Compilation

- Parser works on a stream of tokens.
- The smallest item is a token.

Regular Expressions



- Using a grammar(CFG) to check structure of tokens
- Produce a parse tree
- Recognize correct syntax
- Syntactic errors and recovery
- Reprot errors

- Also technically part of parsing
- Includes augmenting info on tokens in source, type checking, semantic analysis

Error Processing

- Detecting errors
- Finding position at which they occur
- Clear / accurate presentation
- **Recover (pass over) to continue and find later errors**

Syntax Analysis Overview

- Goal – Determine if the input token stream **satisfies syntax** of the program
- What do we need to do this?
  - An expressive way to describe the syntax
  - A mechanism that determines if the input token stream satisfies the syntax description

Syntax Analysis Overview

For lexical analysis

- Regular expressions describe tokens
- Finite automata = mechanisms to generate tokens from input stream

For syntax analysis

- Concrete and Abstract Syntax Trees: formalisms for syntax analysis
- PushDown Automaton (PDA): top-down parsing, bottom-up parsing

Language Recognition Problem

- Let a *language* L be any set of some arbitrary objects s which will be dubbed "sentences."
  - "legal" or "grammatically correct" sentences of the language.
- Let the *language recognition problem* for L be:
  - Given a sentence s, is it a legal sentence of the language L?
    - That is, is $s \in L$?

Intro to Languages

- English grammar tells us if a given combination of words is a valid sentence.

The syntax of a sentence concerns its form while the semantics concerns its meaning.
e.g. the mouse wrote a poem

From a syntax point of view this is a valid sentence.

From a semantics point of view not so…perhaps in Disneyland

Natural languages (English, French, Portguese, etc) have very complex rules of syntax and not necessarily well-defined.

Formal Language

- An alphabet is a set Σ of symbols that act as letters.
- A language over Σ is a set of strings made from symbols in Σ.
- Formal language – is specified by well-defined set of rules of syntax
- We describe the sentences of a formal language using a grammar.

Grammars

- A formal *grammar G* is any compact, precise mathematical definition of a language *L*.
  - As opposed to just a raw listing of all of the language's legal sentences, or just examples of them.
- A grammar implies an algorithm that would generate all legal sentences of the language.
  - Often, it takes the form of a set of recursive definitions.
- A popular way to specify a grammar recursively is to specify it as a *phrase-structure grammar.*

Grammars (Semi-formal)

- Example: A grammar that generates a subset of the English language

$$\langle sentence \rangle \rightarrow \langle noun\_phrase \rangle \ \langle predicate \rangle$$

$$\langle noun\_phrase \rangle \rightarrow \langle article \rangle \ \langle noun \rangle$$

$$\langle predicate \rangle \rightarrow \langle verb \rangle$$

$$\langle article \rangle \rightarrow a$$

$$\langle article \rangle \rightarrow the$$

$$\langle noun \rangle \rightarrow boy$$

$$\langle noun \rangle \rightarrow dog$$

$$\langle verb \rangle \rightarrow runs$$

$$\langle verb \rangle \rightarrow sleeps$$

- A derivation of "the boy sleeps":

$$
\begin{aligned}
\langle sentence \rangle &\Rightarrow \langle noun\_phrase \rangle \ \langle predicate \rangle \\
&\Rightarrow \langle noun\_phrase \rangle \ \langle verb \rangle \\
&\Rightarrow \langle article \rangle \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow the \ \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow the \ \ boy \ \ \langle verb \rangle \\
&\Rightarrow the \ \ boy \ \ sleeps
\end{aligned}
$$

- A derivation of “a dog runs”:

$$
\begin{aligned}
\langle sentence \rangle &\Rightarrow \langle noun\_phrase \rangle \ \langle predicate \rangle \\
&\Rightarrow \langle noun\_phrase \rangle \ \langle verb \rangle \\
&\Rightarrow \langle article \rangle \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow a \ \langle noun \rangle \ \langle verb \rangle \\
&\Rightarrow a \ \ dog \ \langle verb \rangle \\
&\Rightarrow a \ \ dog \ \ runs
\end{aligned}
$$

- Language of the grammar:

L = { "a boy runs",
"a boy sleeps",
"the boy runs",
"the boy sleeps",
"a dog runs",
"a dog sleeps",
"the dog runs",
"the dog sleeps" }

Notation

- $$\langle noun \rangle \rightarrow boy$$
  $$\langle noun \rangle \rightarrow dog$$

Variable
or
Non-terminal

Symbols of
the vocabulary

Production
rule

Terminal
Symbols of
the vocabulary

Phrase-Structure Grammars

- A *phrase-structure grammar* (abbr. PSG) $G = (V,T,S,P)$ is a 4-tuple, in which:
  - V is a vocabulary (set of symbols)
    - The "template vocabulary" of the language.
  - $T \subseteq V$ is a set of symbols called *terminals*
    - Actual symbols of the language.
  - $N :\equiv V - T$ is a set of special "symbols" called *nonterminals*. (Representing concepts like "noun")
  - $S \in N$ is a special nonterminal, the *start symbol.*
    - in our example the start symbol was "sentence".
  - P is a set of *productions* (to be defined).
    - Rules for substituting one sentence fragment for another
    - Every production rule must contain at least one nonterminal on its left side.

Phrase-structure Grammar

► **EXAMPLE:**

- Let $G = (V, T, S, P)$,
- where $V = \{a, b, A, B, S\}$
- $T = \{a, b\}$,
- S is a start symbol
- $P = \{S \rightarrow ABa, A \rightarrow BB, B \rightarrow ab, A \rightarrow Bb\}$.

What sentences can be generated with this grammar?

Derivation

- Let G=(V,T,S,P) be a phrase-structure grammar.
- Let $w_0=lz_0r$ (the concatenation of l, z_0, and r) $w_1=lz_1r$ be strings over V.
- If $z_0 \rightarrow z_1$ is a production of G we say that w1 is directly derivable from w0 and we write $w_o \Rightarrow w_1$.
- If $w_0, w_1, \ldots, w_n$ are strings over V such that $w_0 \Rightarrow w_1, w_1 \Rightarrow w_2, \ldots, w_{n-1} \Rightarrow w_n$, then we say that w_n is derivable from w_0, and write $w_0 \Rightarrow^* w_n$.
- The sequence of steps used to obtain w_n from w_o is called a derivation.

Language

- Let G(V,T,S,P) be a phrase-structure grammar. The
- language generated by G (or the language of G)
- denoted by L(G) , is the set of all strings of terminals
- that are derivable from the starting state S.
- $L(G)= \{w \in T^* \mid S \Rightarrow^* w\}$

Language L(G)

► **EXAMPLE:**

- Let $G = (V, T, S, P)$, where $V = \{a, b, A, S\}$, $T = \{a, b\}$, S is a start symbol and $P = \{S \rightarrow aA, S \rightarrow b, A \rightarrow aa\}$.

- The language of this grammar is given by L (G) = $\{b, aaa\}$;

1. we can derive aA from using $S \rightarrow aA,$ and then derive aaa using $A \rightarrow aa.$

2. We can also derive b using $S \rightarrow b.$

- Language of the grammar with the productions:

$$S \rightarrow aSb, S \rightarrow \varepsilon$$

$$L = \{a^n b^n : n \geq 0\}$$

Types of Grammars - Chomsky hierarchy of languages

- Type 2: Context-Free PSG:
  - All before fragments have length 1 and are nonterminals: P： $A \to \beta$， where $A \in N$， $\beta \in V^*$。

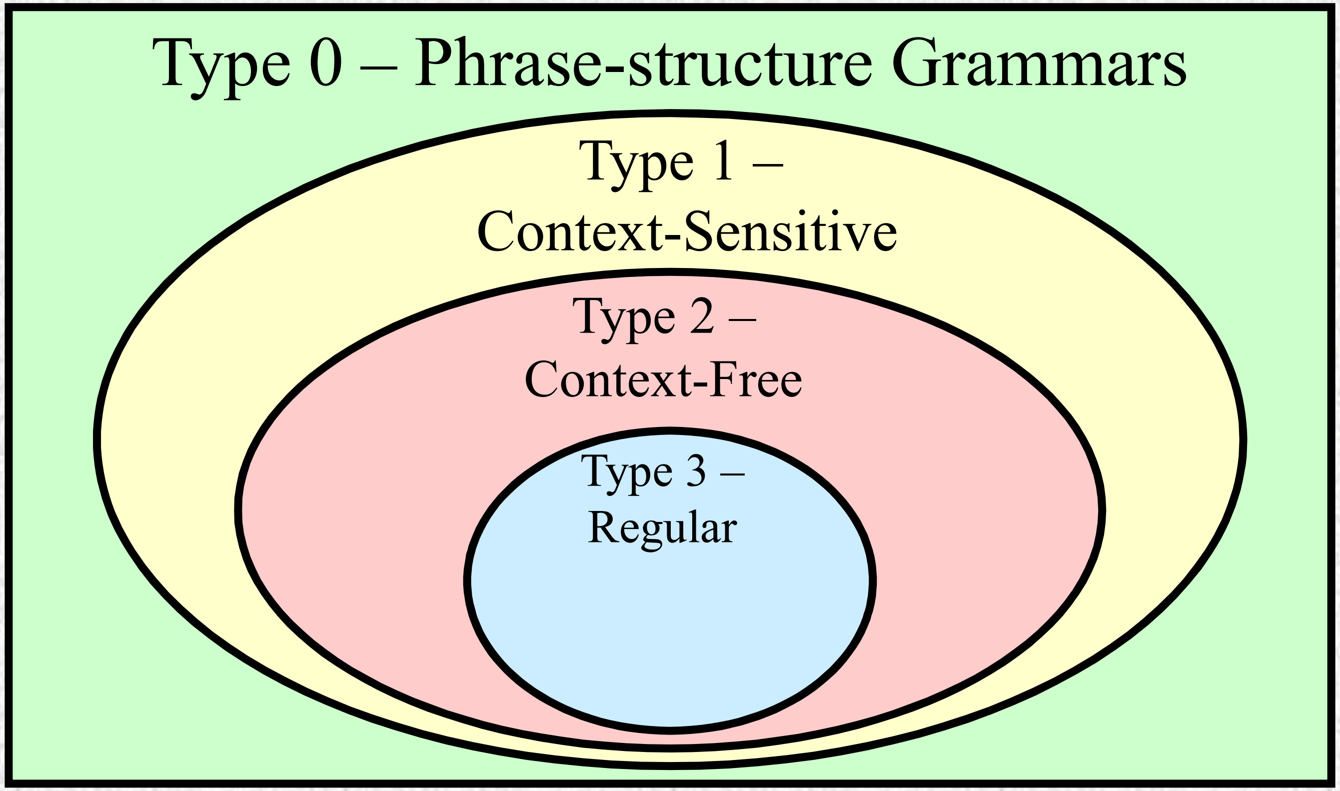
- Type 3: Regular PSGs:
  - All before fragments have length 1 and nonterminals
  - All after fragments are either single terminals, or a pair of a terminal followed by a nonterminal.

    either $A \to \alpha B$， $A \to \alpha$ or, $A \to B\alpha$， $A \to \alpha$

    where A， $B \in N$， $\alpha \in T^*$。

Types of Grammars - Chomsky hierarchy of languages

- Venn Diagram of Grammar Types:



The Limits of Regular Languages

- When scanning, we used **regular expressions** to define each token.
- Unfortunately, regular expressions are (usually) too weak to define programming languages.
  - Cannot define a regular expression matching all expressions with properly balanced parentheses.
  - Cannot define a regular expression matching all functions with properly nested block structure (blocks, expressions, statements)

We need a more powerful formalism.

Context Free Grammars

- A context-free grammar (or CFG) is a formalism for defining languages.
- Can define the context-free languages, a strict superset of the the regular languages.

Context-Free Grammars

- Inherently **recursive** structures of a programming language are defined by a context-free grammar.
- In a context-free grammar, we have:
  - **A finite set of terminals (in our case, this will be the set of tokens)**
  - **A finite set of non-terminals (syntactic-variables)**
  - **A finite set of productions rules in the following form**
    - **$A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)**
  - **A start symbol (one of the non-terminal symbol)**

Example Grammar

$$
\begin{aligned}
expr &\rightarrow expr\ \ op\ \ expr \\
expr &\rightarrow (\ expr\ ) \\
expr &\rightarrow -\ expr \\
expr &\rightarrow id \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow * \\
op &\rightarrow /
\end{aligned}
$$

Black : Nonterminal

Blue : Terminal

expr **:** ***Start Symbol***

8 Production rules

Terminology

- L(G) is *the* *language* of G (the language generated by G) which is a set of sentences.
- A ***sentence*** of L(G) is a string of terminal symbols of G.
- If S is the start symbol of G then

  ω is a sentence of L(G) if $S \stackrel{+}{\Rightarrow} \omega$ where ω is a string of terminals of G.
- A language that can be generated by a grammar is said to be a **context-free language.**
- If G is a context-free grammar, L(G) is a *context-free language*.
- Two grammars are *equivalent* if they produce the same language.
- $S \Rightarrow^* \alpha$
  - If α contains non-terminals, it is called as a ***sentential form*** of G.
  - If α does not contain non-terminals, it is called as a ***sentence*** of G.

Terminology

EX. $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$

id * id **is a sentence**

Here's the derivation:

$exp \Rightarrow exp\ \mathrm{op}\ exp \Rightarrow exp * exp \Rightarrow \mathrm{id} * exp \Rightarrow \mathrm{id} * \mathrm{id}$

Sentential forms

Sentence

$exp \Rightarrow^* id * id$

Some CFG Notation

- Capital letters at the beginning of the alphabet will represent nonterminals.
  - i.e. A, B, C, D
- Lowercase letters at the end of the alphabet will represent terminals.
  - i.e. **t**, **u**, **v**, **w**
- Lowercase Greek letters will represent arbitrary strings of terminals and nonterminals.
  - i.e. α, γ, ω

Examples

- We might write an arbitrary production as

$$A \to \omega$$

- We might write a string of a nonterminal followed by a terminal as

$$A\mathtt{t}$$

- We might write an arbitrary production containing a nonterminal followed by a terminal as

$$B \to \alpha A\mathtt{t}\omega$$

Derivations

- The central idea here is that a production is treated as a **rewriting rule** in which the non-terminal on the left is replaced by the string on the right side of the production.
- $E \Rightarrow E+E$ E+E derives from E
  - we can replace E by E+E
  - to able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.

- $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$
- A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E.
- In general a derivation step is
- $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

A Notational Shorthand

$expr \rightarrow expr\ \ op\ \ expr$

$expr \rightarrow (\ expr\ )$

$expr \rightarrow -\ expr$

$expr \rightarrow \mathbf{id}$

$op \rightarrow +$

$op \rightarrow -$

$op \rightarrow *$

$op \rightarrow /$

$$
\begin{aligned}
expr \rightarrow\ & expr\ \ op\ \ expr \\
&\mid (\ expr\ ) \\
&\mid -\ expr \\
&\mid \mathbf{id} \\
op \rightarrow\ & + \mid - \mid * \mid /
\end{aligned}
$$

Black : Nonterminal

Blue : Terminal

expr* : *Start Symbol

CFG for Programming Language

program	$\rightarrow$	stmt-sequence
stmt-sequence	$\rightarrow$	stmt-sequence ; statement
	|	statement
Statement	$\rightarrow$	if-stmt
	|	repeat-stmt
	|	assign-stmt
	|	read-stmt
	|	write-stmt
if-stmt	$\rightarrow$	**if** exp **then** stmt-sequence **end**
	|	**if** exp **then** stmt-sequence **else** stmt-sequence **end**

Other Derivation Concepts

- At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.
- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

Left-Most and Right-Most Derivations

- Left-Most Derivation

  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

- Right-Most Derivation (called *canonical derivation*

  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

- We will see that the ***top-down parsers*** try to find the ***left-most derivation*** of the given source program.
- We will see that the ***bottom-up parsers*** try to find the ***right-most derivation*** of the given source program in the reverse order.