CS251: Introduction to Language Processing

Syntax Analysis

Vishwesh Jatala

Department of CSE

Indian Institute of Technology Bhilai

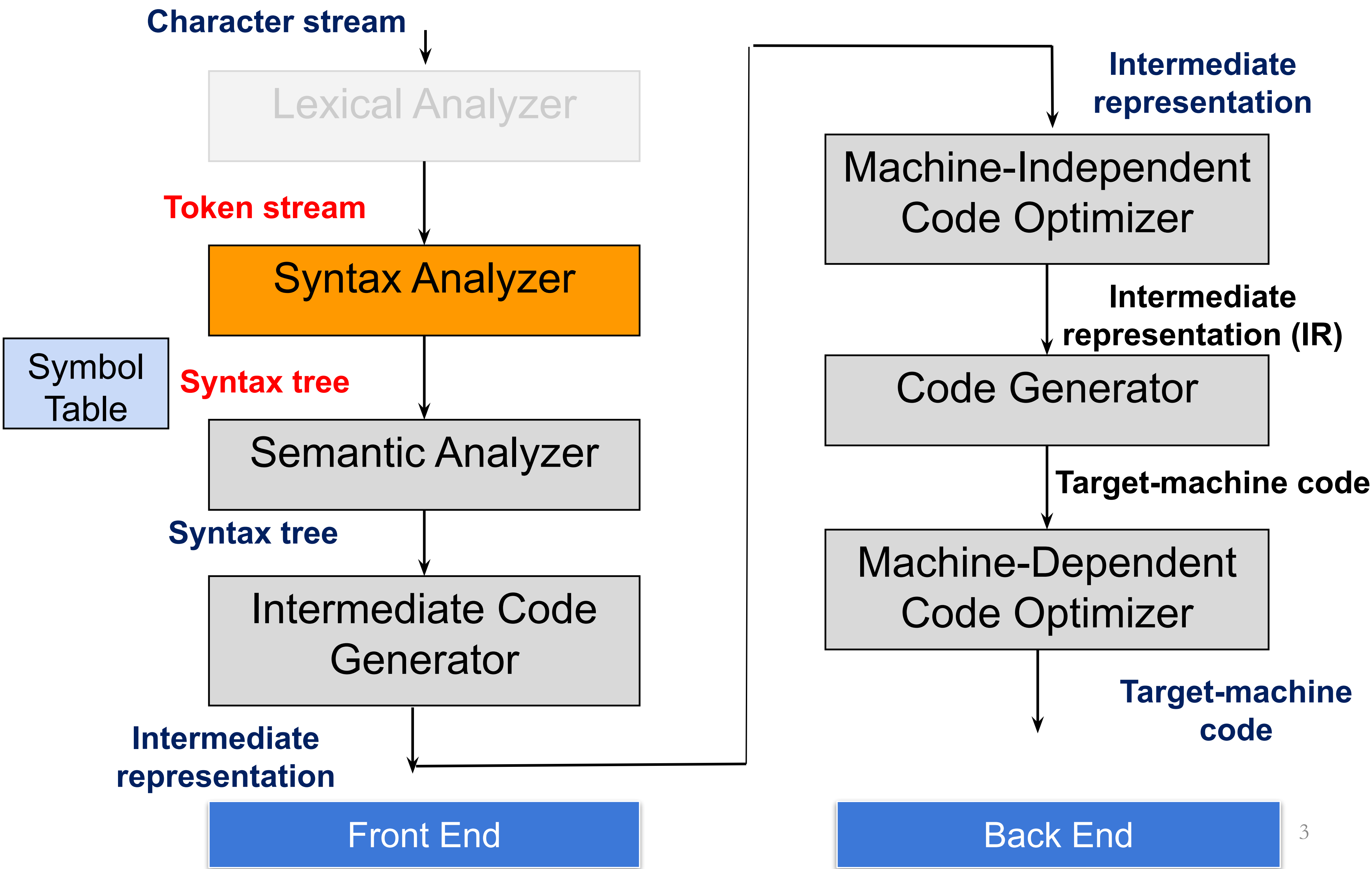
vishwesh@iitbhilai.ac.in

2023-24 M

Acknowledgement

- References for today's slides
  - *Stanford University:*
    - *https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/*
  - *Lecture notes of Prof. Amey Karkare (IIT Kanpur) and Late Prof. Sanjeev K Aggarwal (IIT Kanpur)*

Compiler Design



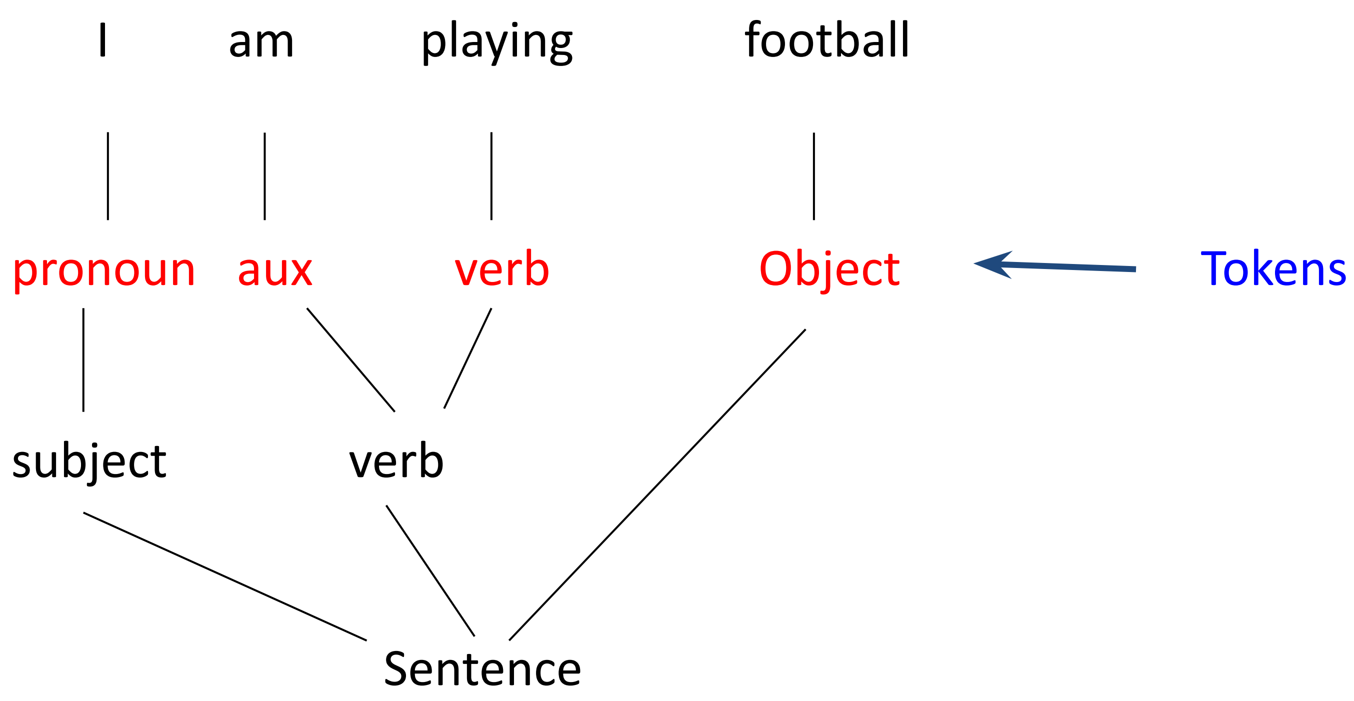
Example (English Lang)

- We understood the token (words), the next step is to understand the structure of the sentence
- The process is known as *syntax checking* or *parsing*

I	am	playing	football		
pronoun	aux	verb	Object	←	Tokens

Example (English Lang)

```
Sentence -> Subject Verb Object
Subject -> pronoun
Verb    -> aux Verb
```



Example (Programming Lang)

```
while (ip < z)
    ++ip;
```

w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

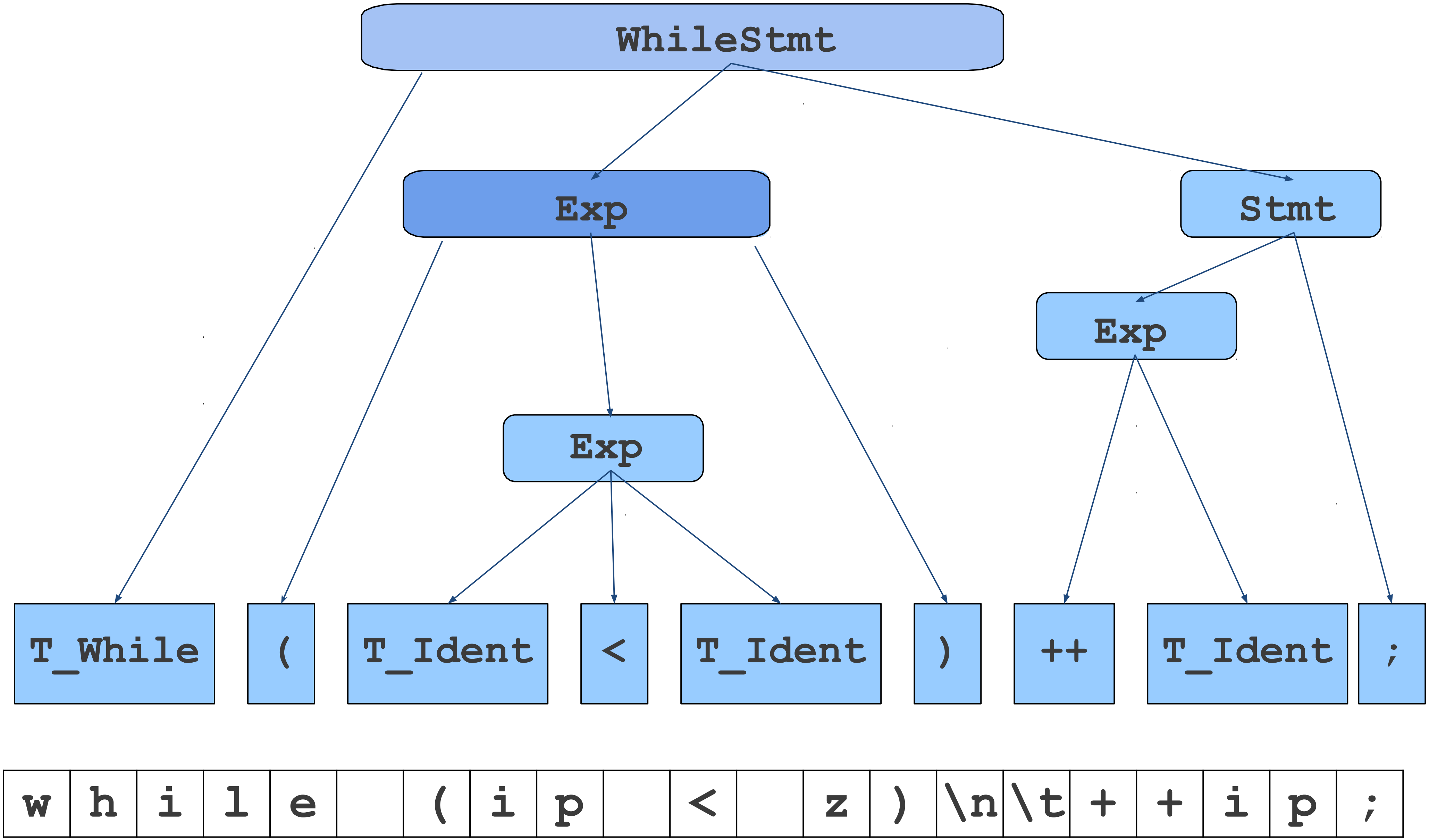
```
while (ip < z)
    ++ip;
```

```
WhileStmt -> T_WHILE (Exp) Stmt
Stmt -> Exp;
Exp -> (Exp)
Exp -> Exp < Exp
Exp -> T_Ident
......
```

T_While	(	T_Ident	<	T_Ident	)	++	T_Ident	;

w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```


WhileStmt
Exp
Stmt
Exp
Exp
T_While
(
T_Ident
<
T_Ident
)
++
T_Ident
;
w h i l e (i p < z) \n \t + + i p ;

```
while (ip < z)
++ip;
```

What is Syntax Analysis?

- After lexical analysis (scanning), we have a series of tokens.
- In syntax analysis (or parsing), we want to interpret what those tokens mean.
- Goal:
  - Recover the *structure* described by that series of tokens
  - Report *errors* if those tokens do not properly encode a structure.

Outline

- Today: Formalisms for syntax analysis.
  - Context-Free Grammars Derivations
  - Ambiguity

- Next Week: Parsing algorithms.
  - Top-Down Parsing
  - Bottom-Up Parsing

The Limits of Regular Languages

- In lexical analysis, we used regular expressions to define each token.
- Unfortunately, regular expressions are (usually) too weak to define programming languages.
  - A finite automata may repeat states, however, it cannot remember the number of times it has been to a particular state
  - 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.
- We need a more powerful formalism.
  - Context Free Languages (CFL)

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.
- CFGs are best explained by example...

Context-Free Grammars

- Formally, a context-free grammar is a collection of four objects:
  - A set of nonterminal symbols (or variables),
  - A set of terminal symbols,
  - A set of production rules saying how each nonterminal can be converted by a string of terminals and nonterminals,
  - and

  A start symbol that begins the derivation.

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

$$\begin{aligned}
E &\to \texttt{int} \\
E &\to E \text{ Op } E \\
E &\to \texttt{(}E\texttt{)} \\
\text{Op} &\to \texttt{+} \\
\text{Op} &\to \texttt{-} \\
\text{Op} &\to \texttt{*} \\
\text{Op} &\to \texttt{/}
\end{aligned}$$

`int / int`

$$\begin{aligned}
& E \\
\Rightarrow\ & E \text{ Op } E \\
\Rightarrow\ & E \text{ Op } \texttt{int} \\
\Rightarrow\ & \texttt{int} \text{ Op } \texttt{int} \\
\Rightarrow\ & \texttt{int / int}
\end{aligned}$$

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

E → int
E → E Op E
E → (E)
Op → +
Op → -
Op → *
Op → /

int * (int + int)

E
⇒ E Op E
⇒ E Op (E)
⇒ E Op (E Op E)
⇒ E * (E Op E)
⇒ int * (E Op E)
⇒ int * (int Op E)
⇒ int * (int Op int)
⇒ int * (int + int)

A Notational Shorthand

$E \rightarrow \texttt{int}$

$E \rightarrow E\ Op\ E$

$E \rightarrow \texttt{(}E\texttt{)}$

$Op \rightarrow \texttt{+}$

$Op \rightarrow \texttt{-}$

$Op \rightarrow \texttt{*}$

$Op \rightarrow \texttt{/}$

A Notational Shorthand

$E \rightarrow$ **`int`** $|$ E Op E $|$ **`(`**E**`)`**

$Op \rightarrow$ **`+`** $|$ **`-`** $|$ **`*`** $|$ **`/`**

Small Exercise

- Write a CFG for expression in C language.

CFGs for Programming Languages

```
EXPR    →  identifier
         |  constant
         |  EXPR + EXPR
         |  EXPR - EXPR
         |  EXPR * EXPR
         |  ...
```

Small Exercise

- Write a CFG for Statement in C language.
  - Statement can be Simple stmt, If-else stmt, While stmt, do-while

CFGs for Programming Languages

```
STMT  →  EXPR;
      |  if (EXPR) BLOCK
      |  while (EXPR) BLOCK
      |  do BLOCK while (EXPR);
      |  BLOCK
      |  ...

EXPR  →  identifier
      |  constant
      |  EXPR + EXPR
      |  EXPR – EXPR
      |  EXPR * EXPR
      |  ...
```

CFGs for Programming Languages

```
BLOCK  → STMT
       | { STMTS }

STMTS  → ε
       | STMT STMTS

STMT   → EXPR;
       | if (EXPR) BLOCK
       | while (EXPR) BLOCK
       | do BLOCK while (EXPR);
       | BLOCK
       | …

EXPR   → identifier
       | constant
       | EXPR + EXPR
       | EXPR – EXPR
       | EXPR * EXPR
       | …
```

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

$$A \rightarrow \omega$$

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

$$At$$

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

$$B \rightarrow aAt\omega$$

Derivations

$$
\begin{aligned}
& E \\
\Rightarrow\ & E\ Op\ E \\
\Rightarrow\ & E\ Op\ (E) \\
\Rightarrow\ & E\ Op\ (E\ Op\ E) \\
\Rightarrow\ & E\ *\ (E\ Op\ E) \\
\Rightarrow\ & \texttt{int}\ *\ (E\ Op\ E) \\
\Rightarrow\ & \texttt{int}\ *\ (\texttt{int}\ Op\ E) \\
\Rightarrow\ & \texttt{int}\ *\ (\texttt{int}\ Op\ \texttt{int}) \\
\Rightarrow\ & \texttt{int}\ *\ (\texttt{int}\ +\ \texttt{int})
\end{aligned}
$$

- This sequence of steps is called a **derivation**.
- A string $\alpha A\omega$ **yields** string $\alpha\gamma\omega$ iff $A \rightarrow \gamma$ is a production.
- If α yields β, we write $\alpha \Rightarrow \beta$.
- We say that α **derives** β iff there is a sequence of strings where

$$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \beta$$

- If α derives β, we write $\alpha \Rightarrow^* \beta$.

Derivations

- A leftmost derivation is a derivation in which each step expands the leftmost nonterminal.
- A rightmost derivation is a derivation in which each step expands the rightmost nonterminal.

Leftmost Derivation

$E \rightarrow$ `int` | E Op E | `(`E`)`

Op $\rightarrow$ `+` | `-` | `*` | `/`

$\Rightarrow$ `int * (int + int)`

E

$\Rightarrow$ E Op E

$\Rightarrow$ `int` Op E

$\Rightarrow$ `int *` E

$\Rightarrow$ `int * (`E`)`

$\Rightarrow$ `int * (`E Op E`)`

$\Rightarrow$ `int * (int` Op E`)`

$\Rightarrow$ `int * (int +` E`)`

$\Rightarrow$ `int * (int + int)`

Summary

- Syntax analysis (**parsing**) extracts the structure from the tokens produced by the scanner.
- Process of determination whether a string can be generated by a grammar
- Languages are usually specified by **context-free grammars** (**CFG**s).
- A **parse tree** shows how a string can be **derived** from a grammar.