CSL302: Compiler Design

Syntax Analysis

Vishwesh Jatala

Assistant Professor

Department of CSE

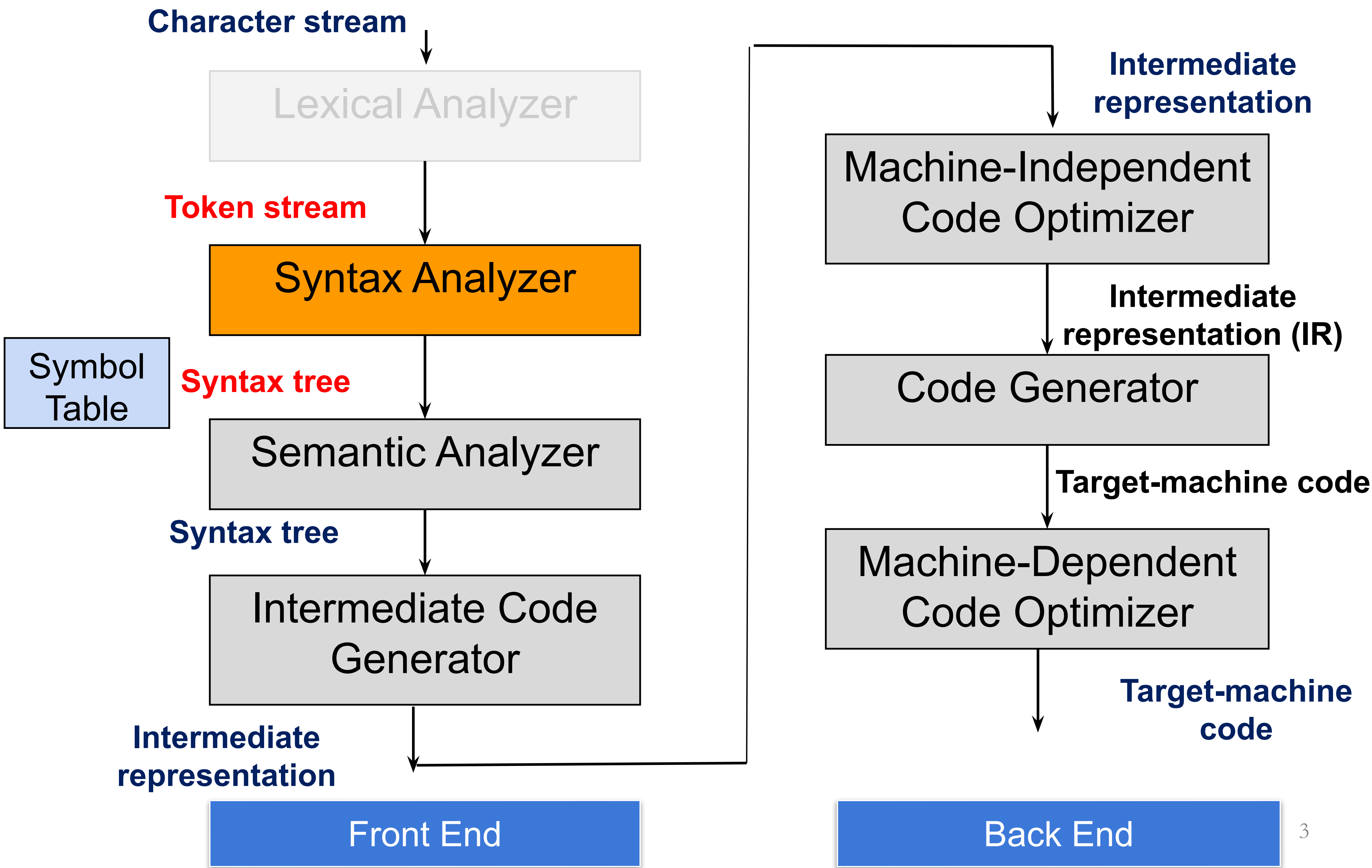
Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

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



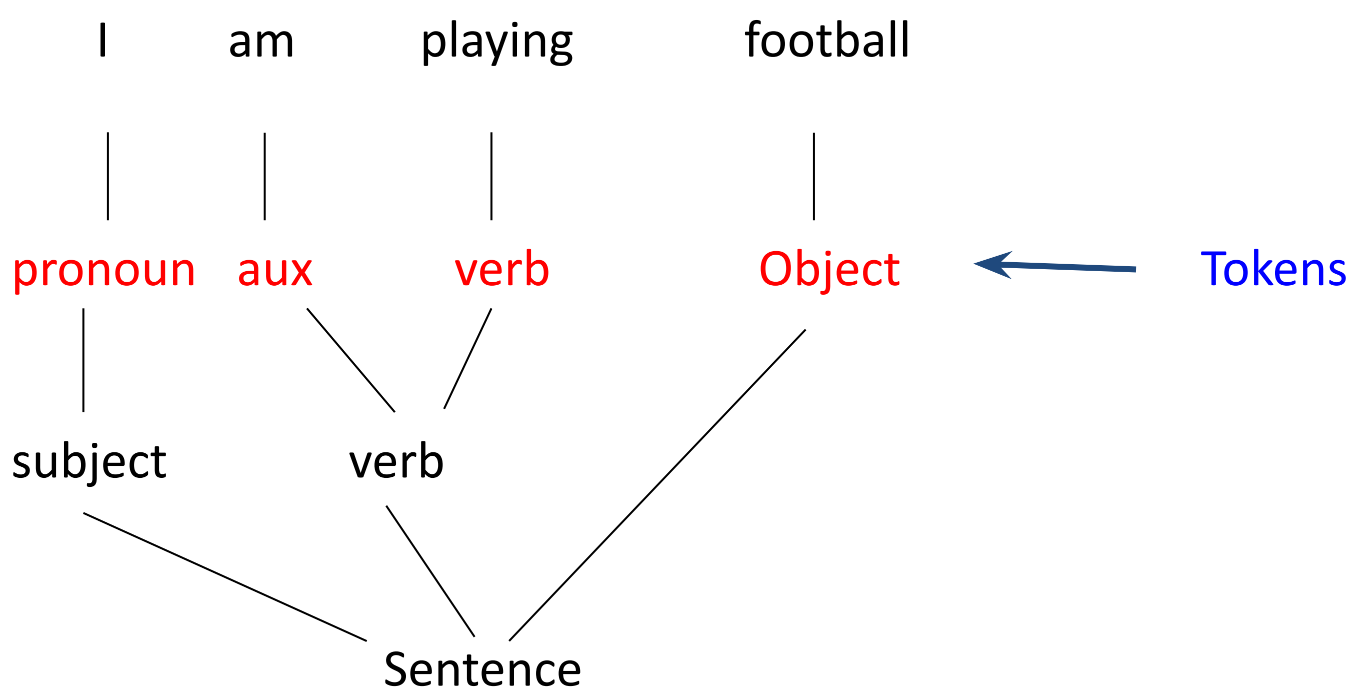
Example (English Lang)

- 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	;

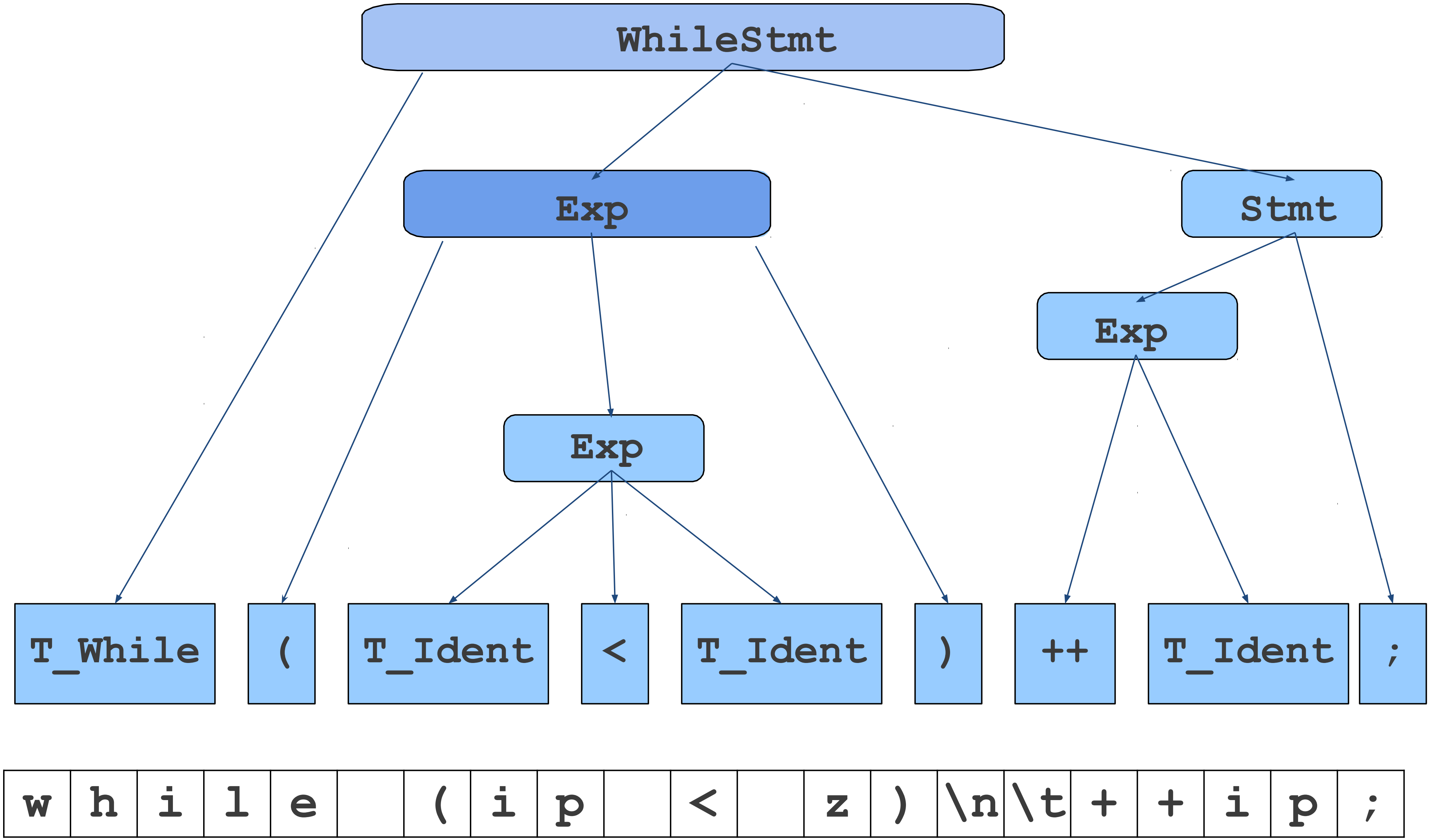
```
while (ip < z)
    ++ip;
```

```
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

- Next 2 Lectures: 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

- Make an attempt to write a CFG to express all legal arithmetic expressions using addition, subtraction, multiplication, and division.

Arithmetic Expressions

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

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

`int / int`

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

Arithmetic Expressions

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

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

`int * (int + int)`

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

A Notational Shorthand

$E \rightarrow$ `int`

$E \rightarrow E\ Op\ E$

$E \rightarrow$ `(`E`)`

$Op \rightarrow$ `+`

$Op \rightarrow$ `-`

$Op \rightarrow$ `*`

$Op \rightarrow$ `/`

A Notational Shorthand

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

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

Small Exercise-1

- Write a CFG for expression in C language, involving identifiers and constants. Operators are +,-,/,*..etc

CFGs for Programming Languages

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

Small Exercise-2

- Write a CFG for C language for the following
  - Statement -- which is a single statement containing an expression followed by semicolon
  - Block -- single statement (or) group of statements enclosed within { }

CFGs for Programming Languages

$$
\begin{array}{lcl}
\text{STMT} & \rightarrow & \texttt{EXPR;} \\
\text{EXPR} & \rightarrow & \texttt{identifier} \\
 & \mid & \texttt{constant} \\
 & \mid & \text{EXPR} \ \texttt{+} \ \text{EXPR} \\
 & \mid & \text{EXPR} \ \texttt{-} \ \text{EXPR} \\
 & \mid & \text{EXPR} \ \texttt{*} \ \text{EXPR} \\
 & \mid & \ldots \\
\text{BLOCK} & \rightarrow & \text{STMT} \\
 & \mid & \texttt{\{} \ \text{STMTS} \ \texttt{\}} \\
\text{STMTS} & \rightarrow & \varepsilon \\
 & \mid & \text{STMT STMTS}
\end{array}
$$

Small Exercise-3

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

CFGs for Programming Languages

$$
\begin{array}{lll}
\text{STMT} & \rightarrow & \text{EXPR}\ \texttt{;} \\
 & | & \texttt{if (}\text{EXPR}\texttt{)}\ \text{BLOCK} \\
 & | & \texttt{while (}\text{EXPR}\texttt{)}\ \text{BLOCK} \\
 & | & \texttt{do}\ \text{BLOCK}\ \texttt{while (}\text{EXPR}\texttt{);} \\
 & | & \text{BLOCK} \\
 & | & \ldots \\
\text{EXPR} & \rightarrow & \texttt{identifier} \\
 & | & \texttt{constant} \\
 & | & \text{EXPR}\ \texttt{+}\ \text{EXPR} \\
 & | & \text{EXPR}\ \texttt{-}\ \text{EXPR} \\
 & | & \text{EXPR}\ \texttt{*}\ \text{EXPR} \\
 & | & \ldots
\end{array}
$$

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