CS419 Compiler Construction

A Simple One-Pass Compiler [Chapter 2]

Lecture 3

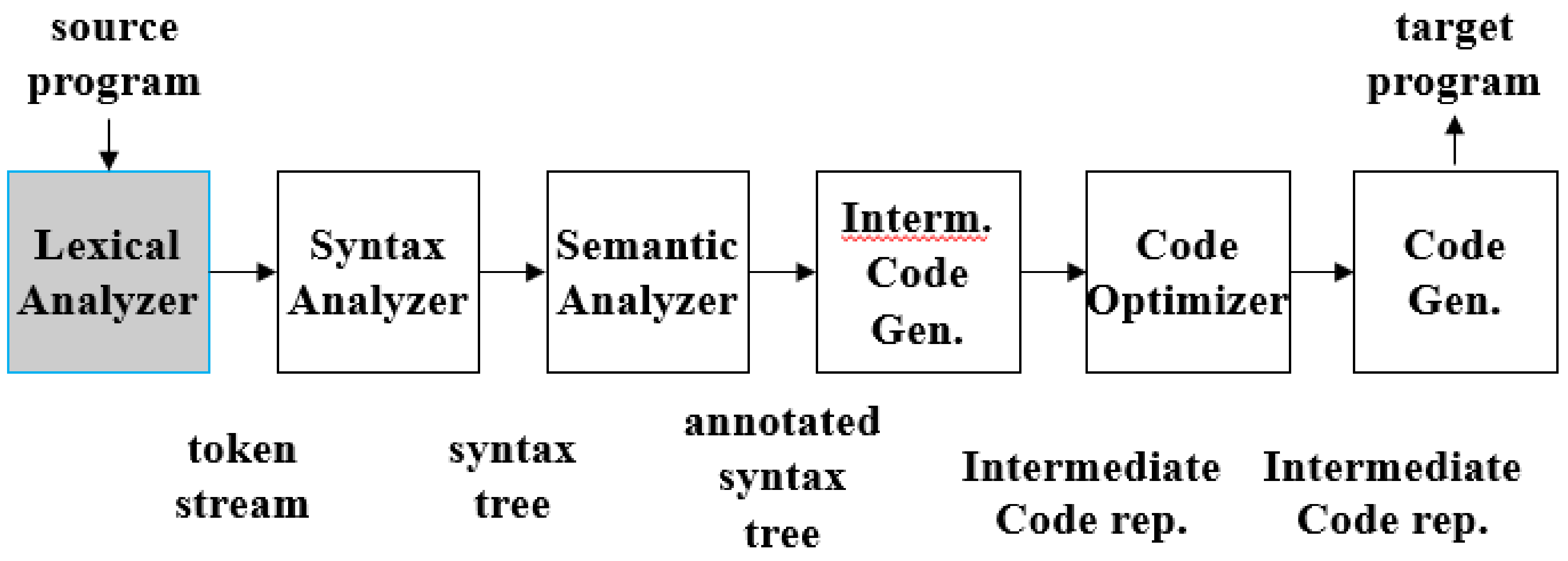
Edited by Dr. Ismail Hababeh

Adapted from slides by Dr. Mohammad Daoud

German Jordanian University

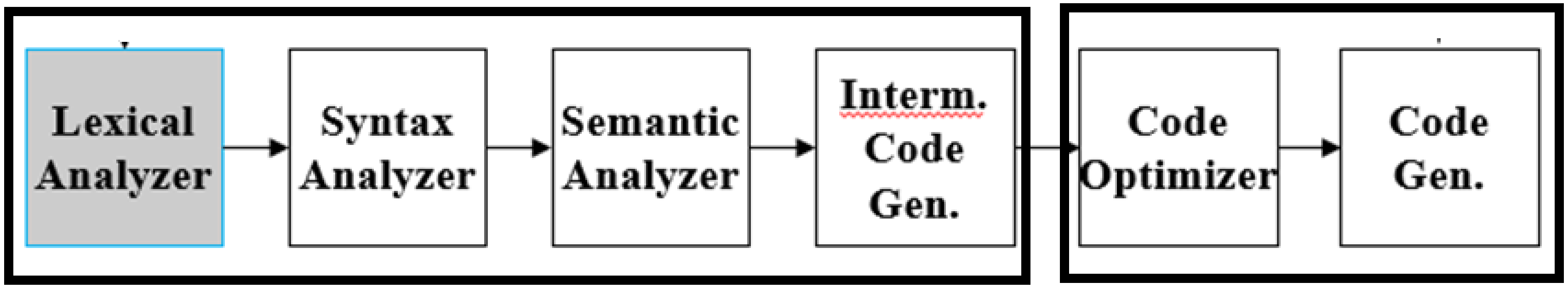
Originally from slides by Dr. Robert A. Engelen

Compiler Phases



Grouping Compiler Phases into Passes

- The activities from several phases could be grouped into one *pass*
- Example:
  - The lexical analysis, syntax analysis, semantic analysis, and intermediate code are grouped as the front-end pass
  - Code generation is the back-end pass



Front-End Pass

Our focus

Back-end Pass

Compiler Phases – Front-End Pass

Phase	Output	Sample
Programming	Source string	**`A=B+60`**
Lexical analyzer (Scanner) Drops down spaces and creates token strings for non-spaces	Token strings	**`'A', '=', 'B', '+', '60'`** And *symbol table* for identifiers

Syntax analyzer (Parser)
creates a tree representation that shows the grammatical structure of the token string
Gives the poriority of the parts depending on the languege (Organizing the priority)
	Pars tree
	(each interior node represents an operation and the children of the node represent the arguments of the operation)
		`    =`
		`  /   \`
		`A       +`
		`      /   \`
		`    B      60`

Semantic analyzer
constructs an abstract syntax tree with attributes.
Syntax Directed Translator
Translates the abstract syntax tree into an array of actions
	Abstract Syntax Tree with attributes.
	Array of actions.
	checking the data types to unify the data types.
		`    =`
		`  /   \`
		`A       +`
		`      /   \`
		`    B     inttofloat`
		`              |`
		`              60`

Intermediate code generator
generates a machine-like representation.
	Three-address code	**`t1 = inttofloat(60)`**
		`t2 = B + t1`
		`A = t2`

Compiler Phases – Back-End Pass

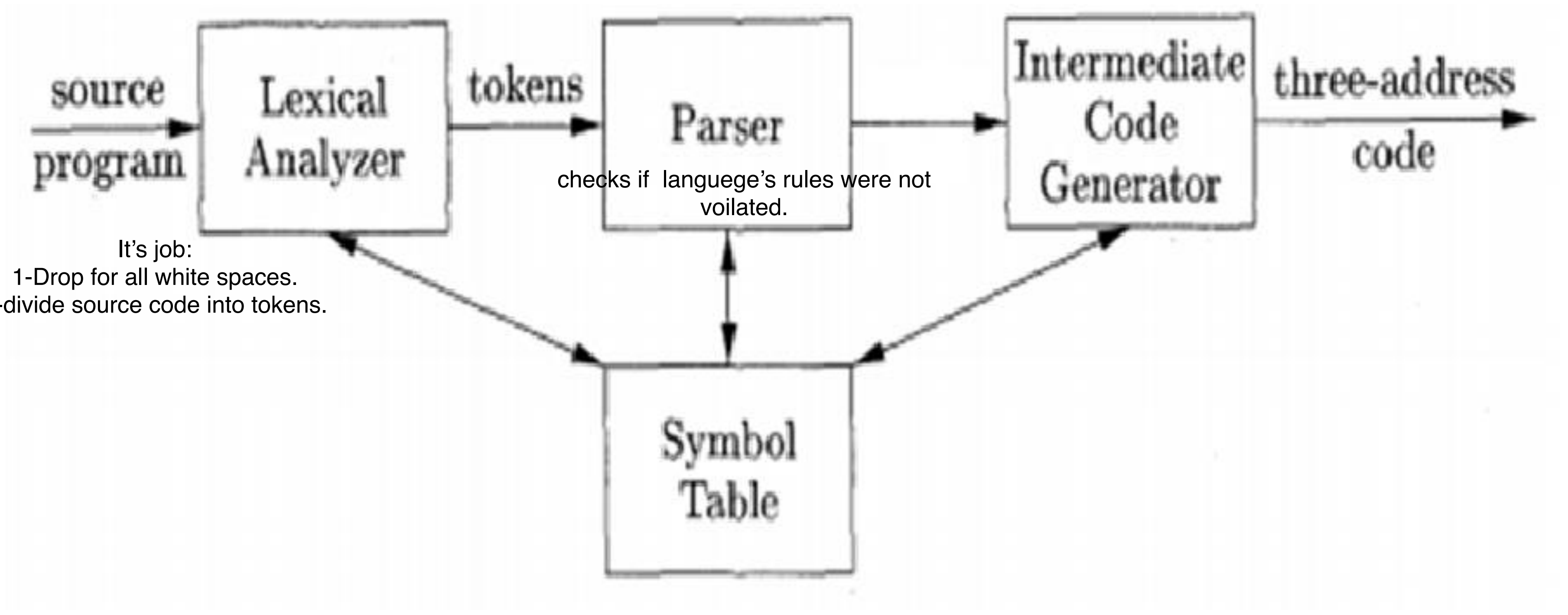
Phase	Output	Sample

Code optimization (optional)
improves the intermediate code to make it faster, shorter, etc…
	Three-address code, quads, or RTL (Register Transfer Level)	`t1 = inttofloat(60)`
		`A = B + t1`

Code generation
maps the intermediate code into the target language
	Assembly code	`MOVF  #60.0,r1`
		`ADDF r1,r2`
		`MOVF  r2,A`

Peephole optimizer
changing the small set of instructions to an equivalent set that has better performance
	Assembly code	`ADDF #60.0,r2`
		`MOVF  r2,A`
		ADDF: add float

A model of a compiler front-end pass



checks if languege's rules were not voilated.

It's job:
1-Drop for all white spaces.
2-divide source code into tokens.

A table that holds the whole varaibles. so it can be used mutliple times
at compile time the symbol table is allways checked

Compiler-Construction Tools

- Software development tools are available to implement one or more compiler phases
  - Scanner generator (lexical analyzer): finds the source code tokens.
  - Parser generator (syntax and Semantics analyzers): describes the input to a source code program.

Compiler-Construction Tools

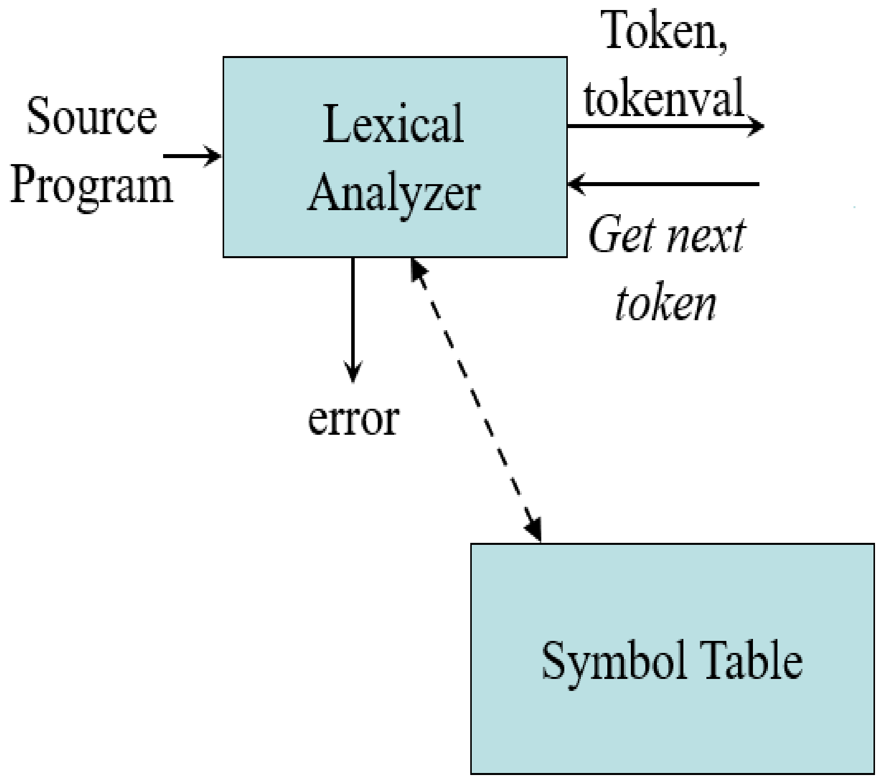
- Syntax-directed translation engine translates source code strings into a sequence of actions by attaching each action to one rule of a grammar.
- Automatic code generator: a program that enables to generate the target code automatically.
- Dataflow engine: helps in improving the efficiency of the compiler code generation.

Lexical Analyzer (Scanner Generator) Tasks

- Typical tasks of the lexical analyzer:
  - Remove white spaces and comments
  - Encode constants as tokens
  - Recognize keywords
  - Recognize identifiers and store identifier names in a global symbol table

The Role of Lexical Analyzer

- Reads the input characters of the source program, group them into lexemes.
- Produce a sequence of tokens for lexemes.
- Interact with the symbol table
  - Inserts lexeme into the symbol table for new identifier
  - Retrieves information about existing lexemes from the symbol table.

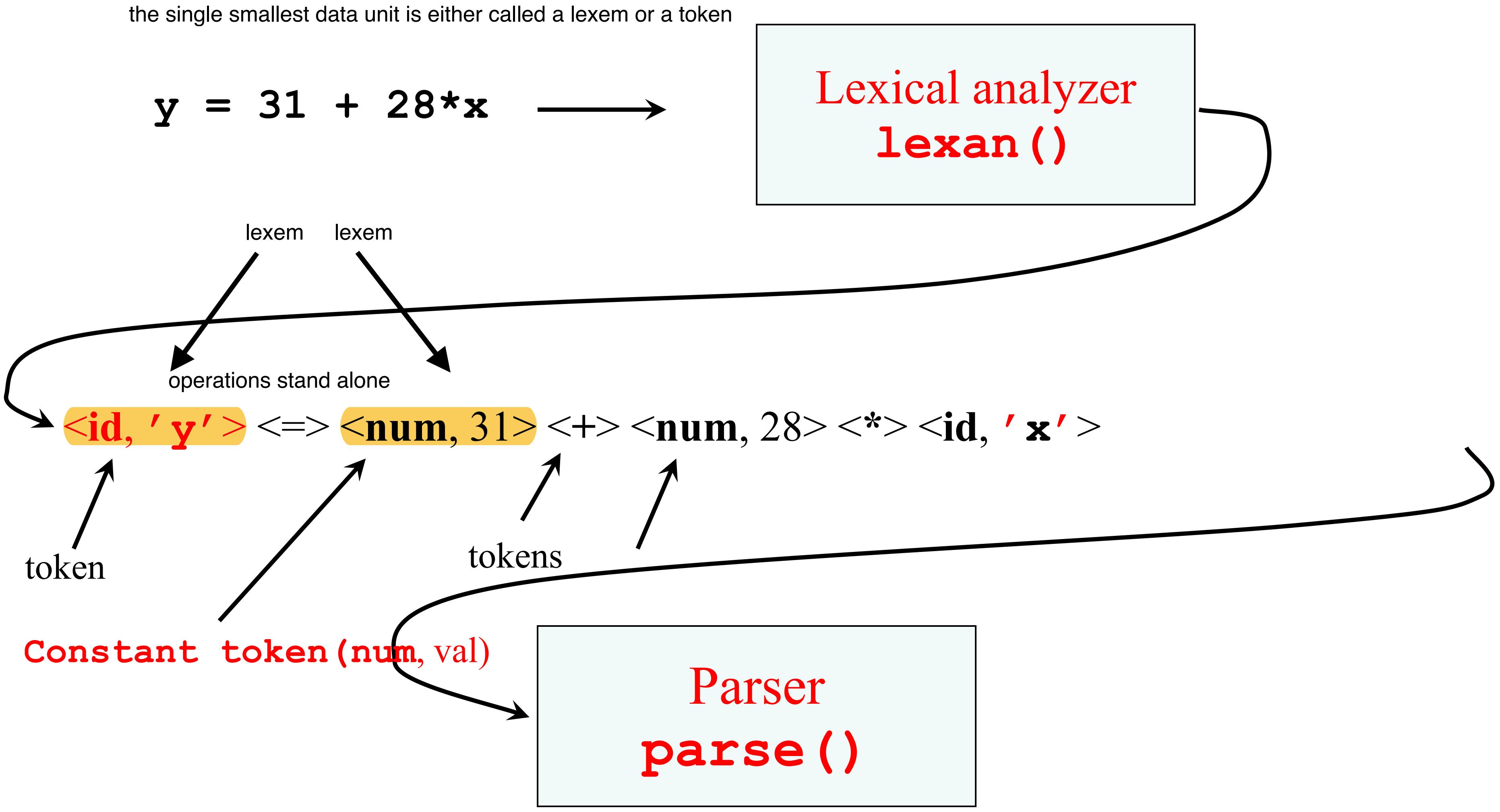


Lexemes,Tokens and Patterns

- *Lexemes* are the specific character strings in the source program that form a token A part of the tokens (menaingful smallest unit in the source code)
  - Example: **`abc, y, 31`**
- A *token* is the smallest element of a program that has meaningful to the compiler
  - A pair consists of a token name and an optional attribute (for operands)
  - Examples: <**id,** pointer to symbol-table entry>, <**num**, integer value >, <=>
- *Patterns* are rules describing the set of lexemes belonging to a token
  - Example: "*letter followed by letter or digits*"

Tokens Structures

Token defined as <identifier | number, value> or <operation | assignment>


the single smallest data unit is either called a lexem or a token
y = 31 + 28*x
Lexical analyzer
lexan()
lexem
lexem
operations stand alone
<id, 'y'> <=> <num, 31> <+> <num, 28> <*> <id, 'x'>
token
tokens
Constant token(num, val)
Parser
parse()

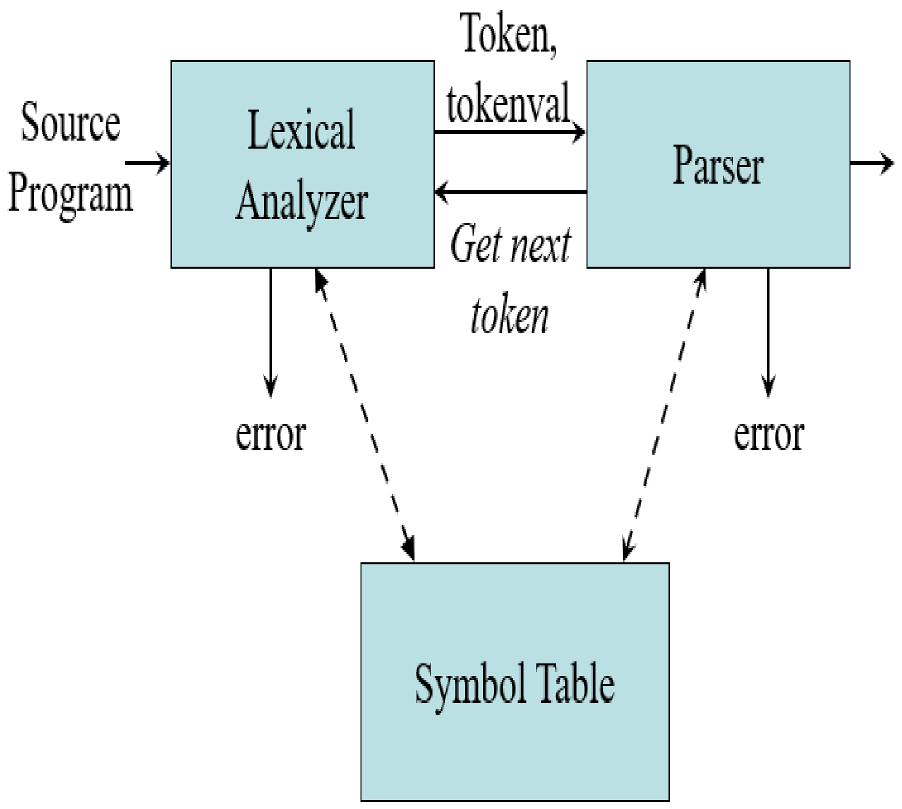
Lexemes - Example

to explain to the lixical analyzer what rules

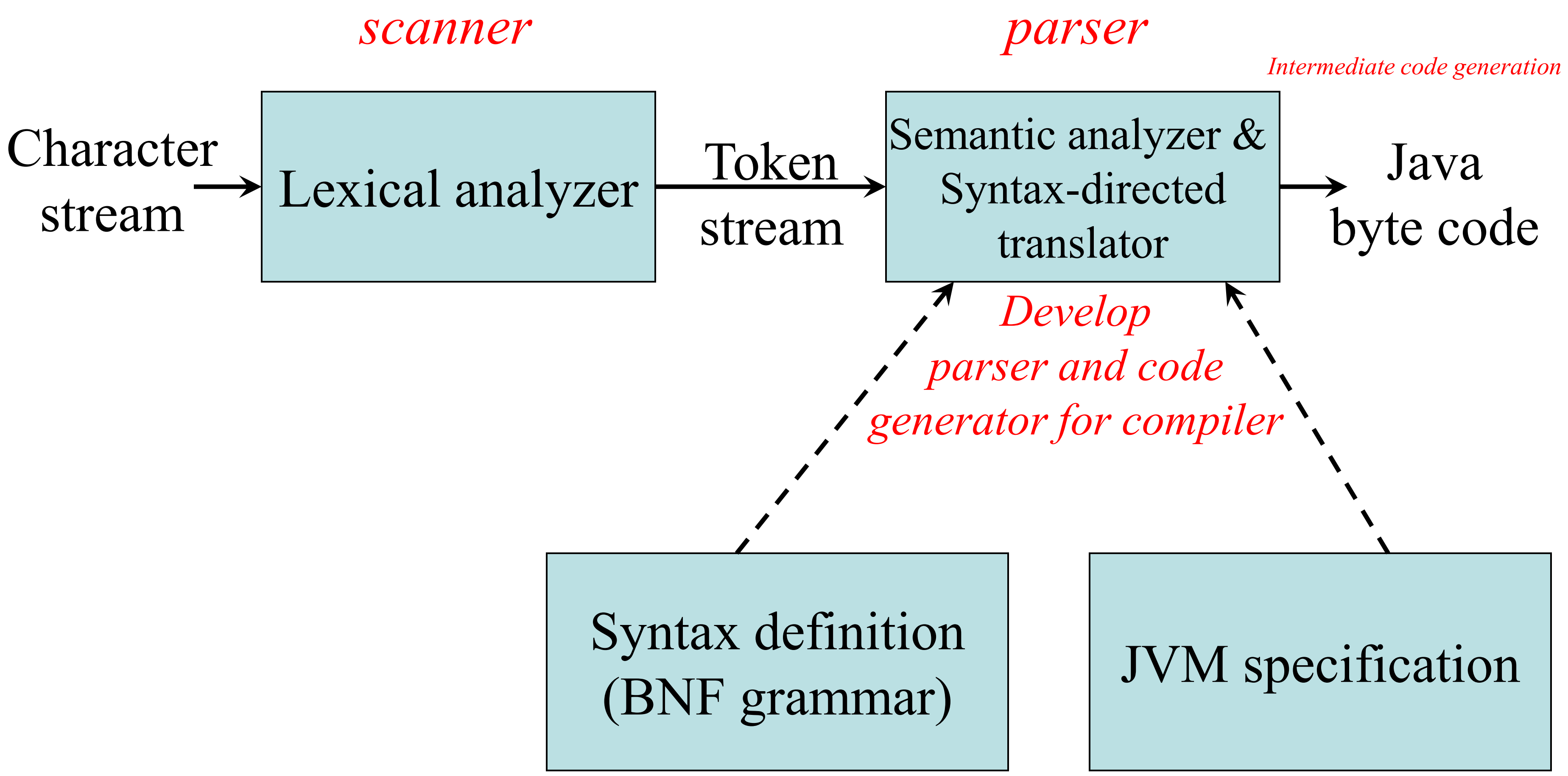
Token	Description	Example
if	Characters i,f	if
id Variable	Letter followed by letters or digits	Pi, score, D2
num	Any numeric constant	3.12159, 0 , 6.2
string	Any set of characters surrounded by " "	"CS419"
comparison	Less than, greater than, …	<=, !=

Interaction of the Lexical Analyzer with the Parser

- The parser call the lexical analyzer to get next token
  - The analyzer reads input characters
  - Identifies the next lexeme.
  - Identifies the next token, which is returned to the parser.
- The lexical analyzer correlates errors generated by the compiler with the source code.



The Enhanced Front-End pass Structure


scanner
parser
Intermediate code generation
Character stream
Lexical analyzer
Token stream
Semantic analyzer & Syntax-directed translator
Java byte code
Develop parser and code generator for compiler
Syntax definition (BNF grammar)
JVM specification

Separated Lexical Analyzer and Parser Phases

- Why the analysis phase in compiler is separated into lexical analyzer and parser?
- Simplifies the design of the compiler
  - Parser does not deal with comments + whitespaces
- Provides efficient implementation
  - Apply systematic techniques that focus on implementing lexical analyzer
  - Speed up the compiler by using stream buffering methods to scan input
- Improves compiler portability
  - Input-device-specific individualities can be restricted to the lexical analyzer

Building Simple Java Compiler

- The main objective of this course is to develop a simple Java compiler that performs the tasks of the front-end pass of a compiler, specifically:
  - Lexical analysis
  - Parsing
  - Intermediate code generation

The Project Outline

- Building a simple Java compiler using:
  - – Lexical Analysis
  - – Syntax Analysis
    - Syntax definition
    - Syntax-directed translation
  - – Parsing (Semantics Analysis)
  - – Symbol Tables
  - – Intermediate Code Generation