Principles of Programming Languages
Lexical Analysis

Hien D. Nguyen, ph.D

University of Information Technology

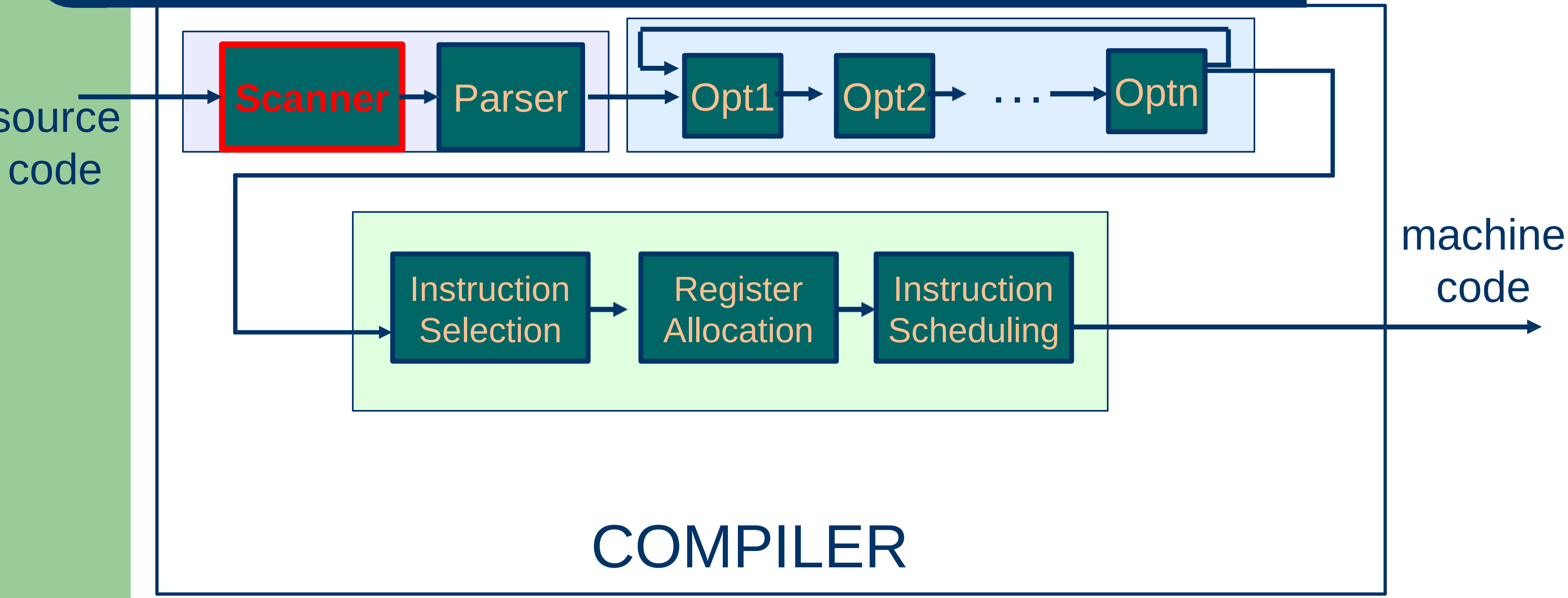
Lecture slides prepared by:

Quan Thanh Tho (qttho@hcmut.edu.vn)

Outline

- Lexical Analysis
- Token
- Regular Expression

The Big Picture



Lexical Analysis

- Lexical Analysis, also called “scanning” or “lexing”
- It does two things:
  - Transforms the input source string into a sequence of substrings
  - Classifies them according to their “role”
- The input is the source code
- The output is a list of tokens

```
if (x == y)
        z = 12;
else
        z = 7;
```

Lexical Analysis

- The output is a list of tokens

```
if (x == y)
        z = 12;
else
        z = 7;
```

i	f		(	x	=	=	y	)	\n	\t	z		=		1	2	;	\n	e	l	s	e	\n	\t	z		=		7	;	\n

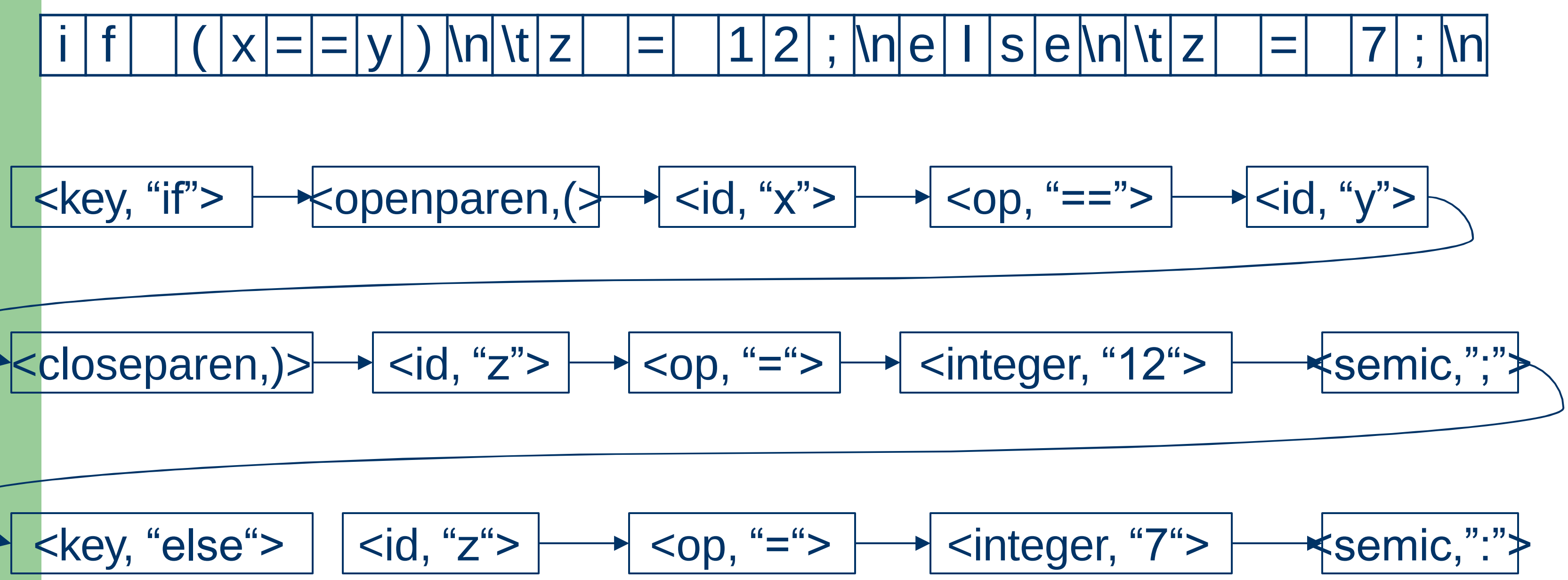
Tokens

- A token is a syntactic category
- Example tokens:
  - Identifier
  - Integer
  - Floating-point number
  - Keyword
  - etc.
- In English we'd talk about
  - Noun
  - Verb
  - Adjective
  - etc.

Lexeme

- A lexeme is the string that represents an instance of a token
- The set of all possible lexemes that can represent a token instance is described by a pattern
- For instance, we can decide that the pattern for an identifier is
  - A string of letters, numbers, or underscores, that starts with a capital letter

Lexing output



Lexicon output

- Note that the lexer removes non-essential characters
  - Spaces, tabs, linefeeds
  - And comments!
  - Typically a good idea for the lexer to allow arbitrary numbers of white spaces, tabs, and linefeeds

Regular Expressions

- To avoid the endless nesting of if-then-else to capture all types of possible tokens one needs a formalization of the lexing process
- If we have a good formalization, we could even generate the lexing code automatically!

Lexer Specification

- Question: How do we formalize the job a lexer has to do to recognize the tokens of a pecific language?
- Answer: We need a language!

- What's a language?
  - An alphabet (typically called ∑)
    - e.g., the ASCII characters
  - A subset of all the possible strings over ∑

- How to represent the language?
- It turns out that for all (reasonable) programming languages, the tokens can be described by a regular language

Describing Tokens

- The most popular way to describe tokens is to use regular expressions

- Regular expressions are just notations, which happen to be able to represent regular languages
  - A regular expression is a string (in a meta-language) that describes a pattern (in the token language)

- If A is a regular expression, then L(A) is the language represented by A

Describing Tokens

- Basic: L(“c”) = {“c”}
- Concatenation: L(AB) = {ab | a in L(A) and b in L(B)}
  - L(“i” “f”) = {“if”}
- Union: L(A|B) = {x | x in L(A) or x in L(B)}
  - L(“if”|”then”|”else”} = {“if”, “then”, “else”}
  - L((“0”|”1”) (“1”|”0”)} = {“00”, “01”, “10”, “11”}

Regular Expression Overview

Expression	Meaning
ε	empty pattern
a	Any pattern represented by ‘a’
ab	Strings with pattern ‘a’ followed by pattern ‘b’
a|b	Strings with pattern ‘a’ or pattern ‘b’
a^*	Zero or more occurrences of pattern ‘a’
a^+	One or more occurrences of pattern ‘a’
a^3	Exactly 3 occurrences of pattern ‘a’
a?	(a | ε)
.	Any single character (not very standard)

REs for Keywords

- It is easy to define a RE that describes all keywords

  Key = “if” | “else” | “for” | “while” | “int” | ..

- These can be split in groups if needed

  Keyword = “if” | “else” | “for” | …

  Type = “int” | “double” | “long” | …

RE for Numbers

- Straightforward representation for integers
  - digits = “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9”
  - integer = digits^{+}
- Typically, regular expression systems allow the use of ‘-’ for ranges, sometimes with ‘[‘ and ‘]’
  - digits = 0-9

RE for Numbers

- Floating point numbers are much more complicated
  - .12e-12
  - 312.00001E+12
  - 4
- Here is one attempt
  - (digit$^+$ “.”? | digits* (“.” digit$^+$)) ((“E”|”e”)(“+”|”-”|ε) digit+)))?
- Note the difference between meta-character and language-characters
  - “+” versus +, “-” versus -, “(“ versus (, etc.
- Often books/documentations use different fonts for each level of language

RE for Identifiers

- Here is a typical description
  - letter = a-z | A-Z
  - ident = letter (letter | digit | “_”)*
    - Starts with a letter
    - Has any number of letter or digit or “_” afterwards
- In C: ident = (letter | “_”) (letter | digit | “_”)*

RE for Phone Numbers

- Simple RE
  - digit = 0-9
  - area = $digit^3$
  - exchange = $digit^3$
  - local = $digit^4$
  - phonenumber = “(“ area “)” “ “? exchange (“-”|” “) local

Now What?

- Now we have a nice way to formalize each token (which is a set of possible strings)
- Each token is described by a RE
  - And hopefully we have made sure that our REs are correct
  - Easier than writing the lexer from scratch
  - But still requires that one be careful
- Question: How do we use these REs to parse the input source code and generate the token stream?

Example

- Say we have the following tokens (described by a RE, and thus a natural NFA, and thus a DFA):
  - TOKEN_IF: “if”
  - TOKEN_IDENT: letter (letter | “_”)+
  - TOKEN_NUMBER: (digit)+
  - TOKEN_COMPARE: “==“
  - TOKEN_ASSIGN: “=“
- This is a very small set of tokens for a tiny language
- The language assumes that tokens are all separated by spaces
- Let’s see what happens on the following input:

i	f		i	f	0		=	=		c		x		=		2	x	3

Example

i	f		i	f	0		=	=		c		x		=		2	x	3

- If there had be no syntax error, the lexer would have emitted:
  - <TOKEN_IF, “if”>
  - <TOKEN_ID, “if0”>
  - <TOKEN_COMPARE, “==“>
  - <TOKEN_ID, “c”>
  - <TOKEN_ID, “x”>
  - <TOKEN_ASSIGN, “=“>
  - <TOKEN_NUMBER,”23”>

Lexer Generation

- A lot of of the lexing process is really mechanical once on has defined the REs (Contrast with the horrible if-then-else nesting of the "by hand" lexer!)
- So there are "lexer generators" available
  - They take as input a list of token specifications
    - token name
    - regular expression
  - They produce a piece of code that is the lexer for these tokens
- Well-known examples of such generators are lex and flex
- With these tools, a complicated lexer for a full language can be developed in a few hours

Exercises

- Find RE for $a^n b^m$: $n+m$ is even

Solution

$(aa)^*(ab+\varepsilon)(bb)^*$

- $(aa)^*$: Matches zero or more occurrences of "aa". This ensures that n, the number of "a"s, is even.
- $(ab+\varepsilon)$: Matches either "ab" (where n is odd and m is even) or the empty string (ε) (where both n and m are even).
- $(bb)^*$: Matches zero or more occurrences of "bb". This ensures that m, the number of "b"s, is even.

- Find RE for $a^n b^m$: $n \geq 1$, $m \geq 1$

Solution

a+b+

Further Exercises

```
int a= 2;
float b = 2.0;
if (a>=5|| b<5)
{
    a ++;
   b +=3.5;
}
```

Token Set

d: [0-9]
ID: (‘a’-’z’|’A’-’Z’|’_’)(‘a’-’z’|’A’-’Z’|’0’-’9’|’_’)*
N: d+
F: N’.’N
ADD: ‘+’
RELOP: ‘<‘|’<=‘|’>’|’>=‘
EQLOP: ‘==‘|’!=‘
OR: ‘||’
INT: “int”
FLOAT: “float”
BRACE: ‘{‘|’}’
ASSIGN: ‘=‘
SEMI: ‘;’