CS480
Translators

Introduction to Lexical Analysis

Chap. 2

Odds and Ends

- Assignment #2 is posted
  - Please email me your teams, if not working alone
- Demo your Assignment #1

The Role of the Scanner…

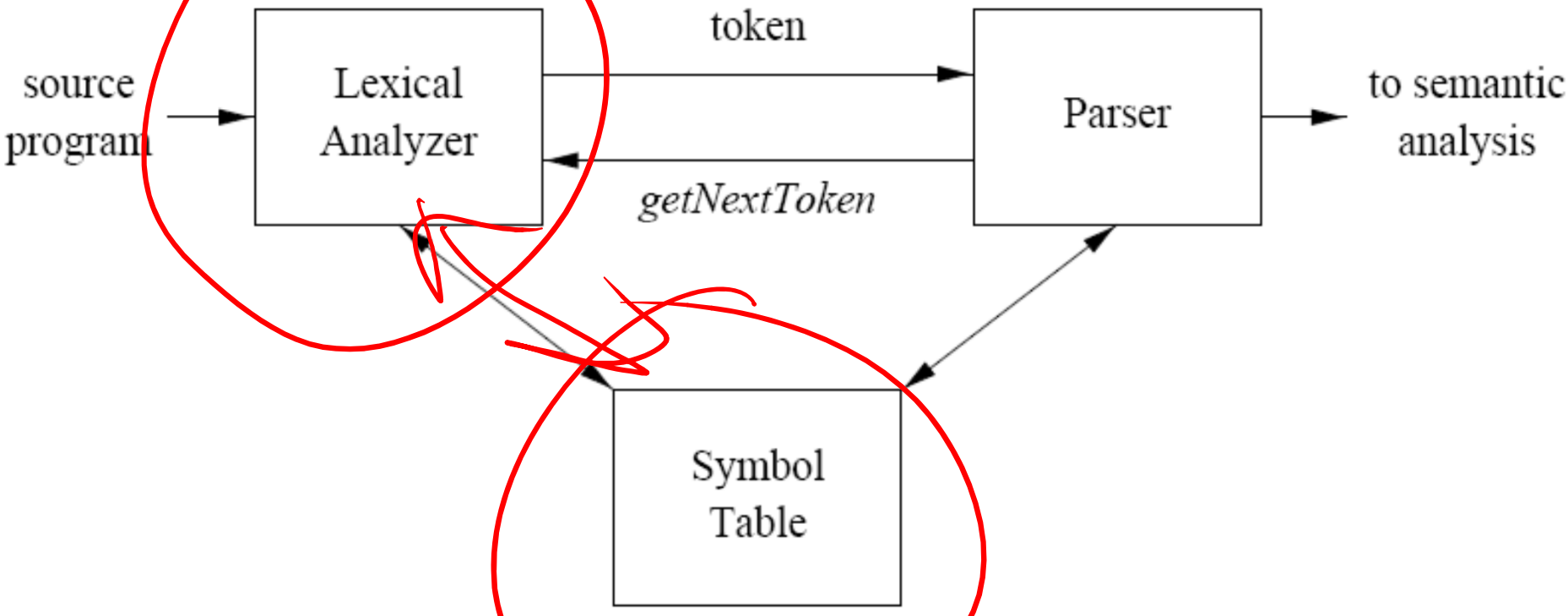
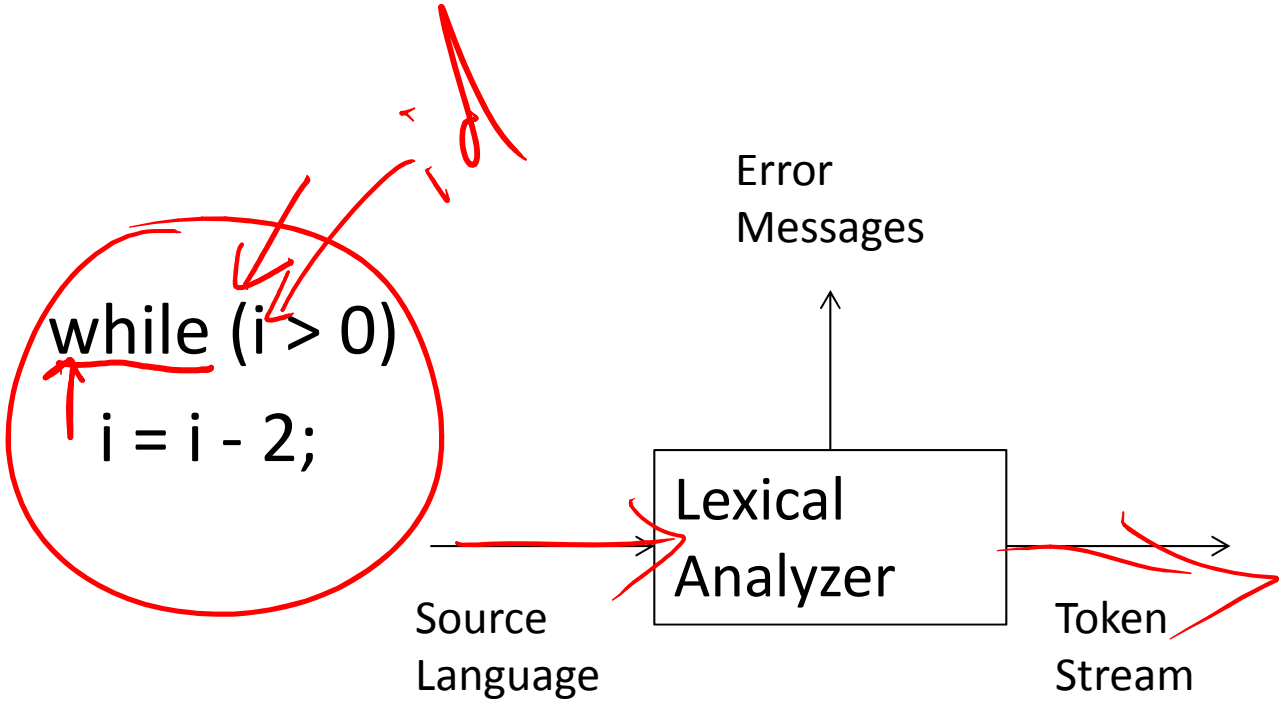


Figure 3.1: Interactions between the lexical analyzer and the parser

Mini-Translator



T_WHILE
T_LPAREN
T_IDENTIFIER
T_LESSTHAN
T_INTCONSTANT
T_RPAREN
T_IDENTIFIER
T_EQUALS
T_MINUS
T_INTCONSTANT
T_SEMICOLON

What's new in this grammar?

expr	→	*expr* + *term*	{ print('+') }
	|	*expr* − *term*	{ print('−') }
	|	*term*	
term	→	*term* * *factor*	{ print('*') }
	|	*term* / *factor*	{ print('/') }
	|	*factor*	
factor	→	(*expr*)	
	|	**num**	{ print(**num**.*value*) }
	|	**id**	{ print(**id**.*lexeme*) }

Figure 2.28: Actions for translating into postfix notation

The Scanner

```
for ( ; ; peek = next input character ) {
        if ( peek is a blank or a tab ) do nothing;
        else if ( peek is a newline ) line = line+1;
        else break;
}
```

Figure 2.29: Skipping white space

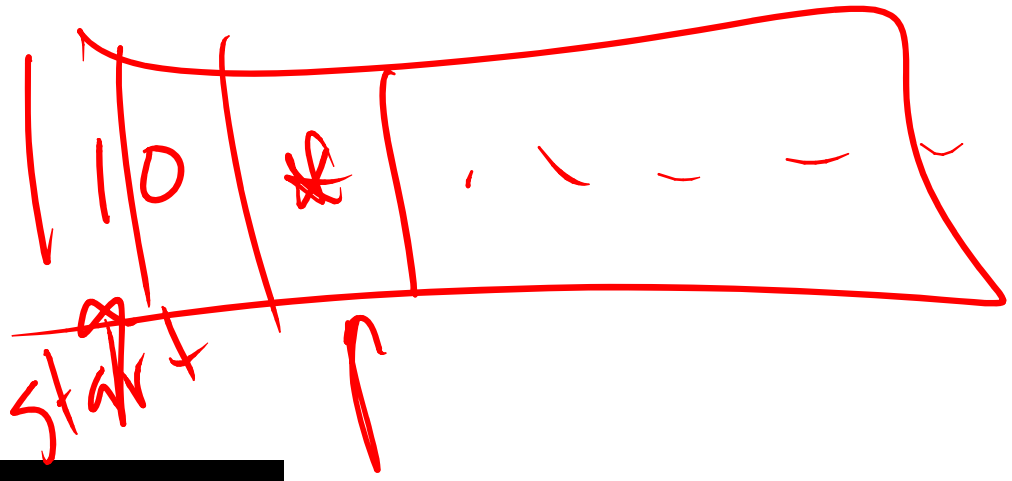
- What is the purpose of line?
- What is the purpose of peek?

Reading Ahead

- Read the next char, it is an “i”
- Could be int, if, or an identifier, so read next char, “f”
- Could be if, could still be an identifier, so read next char, “(”
- Oops, we’ve gone too far, push back “(”

Buffers

- Why is this important? – efficiency
- Ways to implement:
  - Two pointers into buffer (start_char, look_ahead)
  - Push back buffer (peek)



The Lexical Analyzer

```
if ( peek holds a digit ) {
        v = 0;
        do {
            v = v * 10 + integer value of digit peek;
            peek = next input character;
        } while ( peek holds a digit );
        return token ⟨num, v⟩;
}
```

Figure 2.30: Grouping digits into integers

Keywords vs. Identifiers

- count = count + increment;

<id, “count”> <=> <id, “count”> <+> <id, “increment”> <;>

- How do we know count is an id vs. keyword?
- Why use a hash table?
- What is in the hash table?

How to distinguish words?

```
if ( peek holds a letter ) {
        collect letters or digits into a buffer b;
        s = string formed from the characters in b;
        w = token returned by words.get(s);
        if ( w is not null ) return w;
        else {
                Enter the key-value pair (s, ⟨id, s⟩) into words
                return token ⟨id, s⟩;
        }
}
```

Figure 2.31: Distinguishing keywords from identifiers

Tokens

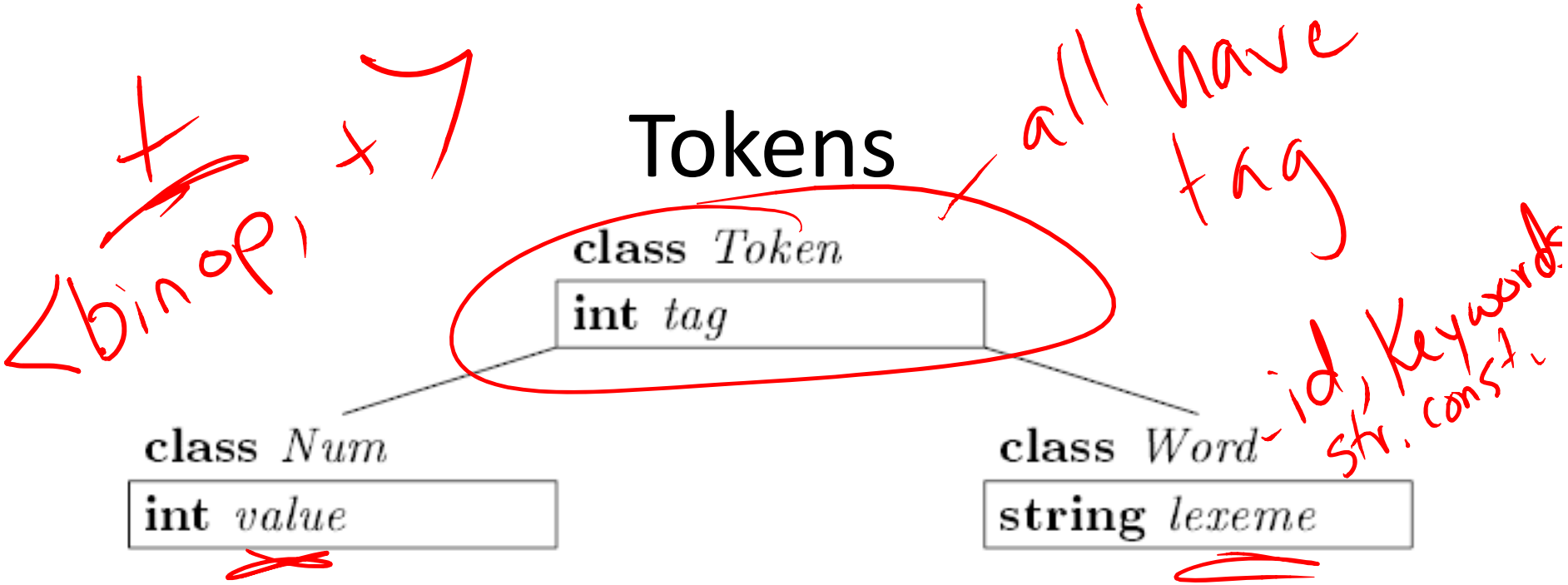


Figure 2.32: Class *Token* and subclasses *Num* and *Word*

```
public class Token {
    public final int tag;
    public Token(int t) { tag = t; }
}
```

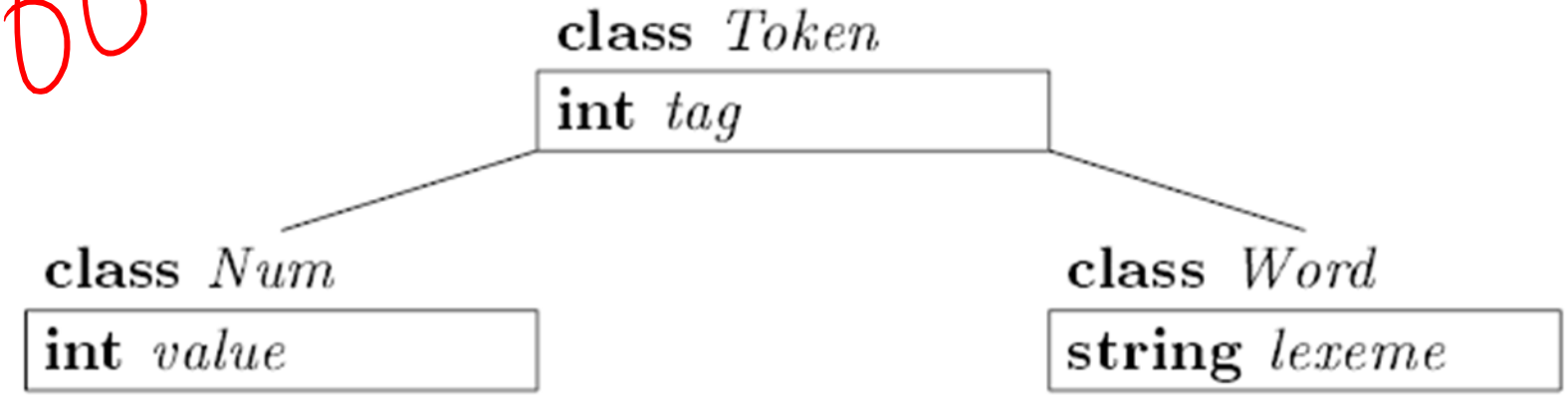
Numbers vs. Words

```
1) package lexer;                          // File Num.java
2) public class Num extends Token {
3)    public final int value;
4)    public Num(int v) { super(Tag.NUM); value = v; }
5) }
```

```
1) package lexer;                          // File Word.java
2) public class Word extends Token {
3)    public final String lexeme;
4)    public Word(int t, String s) {
5)       super(t); lexeme = new String(s);
6)    }
7) }
```

Figure 2.33: Subclasses Num and Word of Token

Token Data Structures




```
struct token_t {
  int tag;
  union {
    char *lexeme;
    int value;
  } val;
};
```

OR

```
struct token_t {
  int tag;
  void *val;
};
```

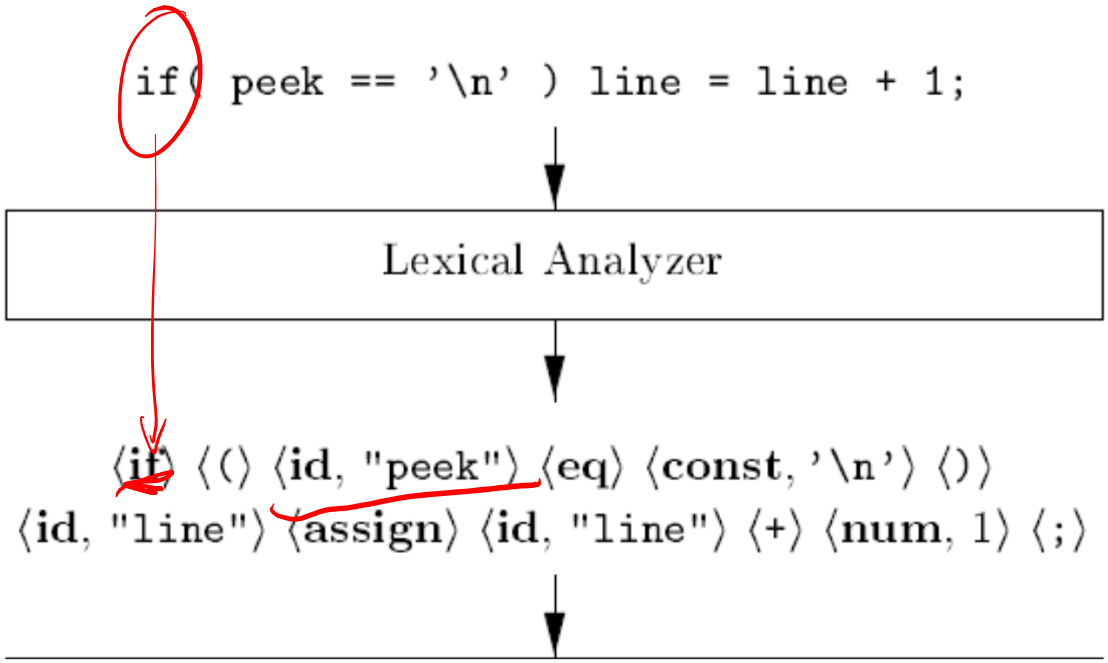
```
1)  package lexer;                          // File Lexer.java
2)  import java.io.*; import java.util.*;
3)  public class Lexer {
4)     public int line = 1;
5)     private char peek = ' ';
6)     private Hashtable words = new Hashtable();
7)     void reserve(Word t) { words.put(t.lexeme, t); }
8)     public Lexer() {
9)        reserve( new Word(Tag.TRUE,  "true")  );
10)       reserve( new Word(Tag.FALSE, "false") );
11)    }
12)    public Token scan() throws IOException {
13)       for( ; ; peek = (char)System.in.read() ) {
14)          if( peek == ' ' || peek == '\t' ) continue;
15)          else if( peek == '\n' ) line = line + 1;
16)          else break;
17)       }
          /* continues in Fig. 2.35 */
```

Figure 2.34: Code for a lexical analyzer, part 1 of 2

```
18)         if( Character.isDigit(peek) ) {
19)             int v = 0;
20)             do {
21)                 v = 10*v + Character.digit(peek, 10);
22)                 peek = (char)System.in.read();
23)             } while( Character.isDigit(peek) );
24)             return new Num(v);
25)         }
26)         if( Character.isLetter(peek) ) {
27)             StringBuffer b = new StringBuffer();
28)             do {
29)                 b.append(peek);
30)                 peek = (char)System.in.read();
31)             } while( Character.isLetterOrDigit(peek) );
32)             String s = b.toString();
33)             Word w = (Word)words.get(s);
34)             if( w != null ) return w;
35)             w = new Word(Tag.ID, s);
36)             words.put(s, w);
37)             return w;
38)         }
39)         Token t = new Token(peek);
40)         peek = ' ';
41)         return t;
42)     }
43) }
```


Figure 2.35: Code for a lexical analyzer, part 2 of 2

Job of a Tokenizer…



Reading/Assignment

- Milestone 2
- Read Chap. 2.6 - 2.7 and Chap. 3