Intro to Compiler Design

Análise Léxica

CS 1622

Implementation of Lexical Analysis

Simple Scanner by Hand

```
token_t scan_token( FILE *fp ) {
    int c = fgetc(fp);
    if(c=='*') {
        return TOKEN_MULTIPLY;
    } else if(c=='!') {
        char d = fgetc(fp);
        if(d=='=') {
            return TOKEN_NOT_EQUAL;
        } else {
            ungetc(d,fp);
            return TOKEN_NOT;
        }
    } else if(isalpha(c)) {
        do {
           char d = fgetc(fp);
        } while(isalnum(d));
        ungetc(d,fp);
        return TOKEN_IDENTIFIER;
    } else if ( . . . ) {
        . . .
    }
}
```

Simple Scanner using Flex

token.h:

```
typedef enum {
        TOKEN_EOF=0,
        TOKEN_WHILE,
        TOKEN_ADD,
        TOKEN_IDENT,
        TOKEN_NUMBER,
        TOKEN_ERROR
} token_t;
```

Simple Scanner using Flex (2)

scanner.flex:

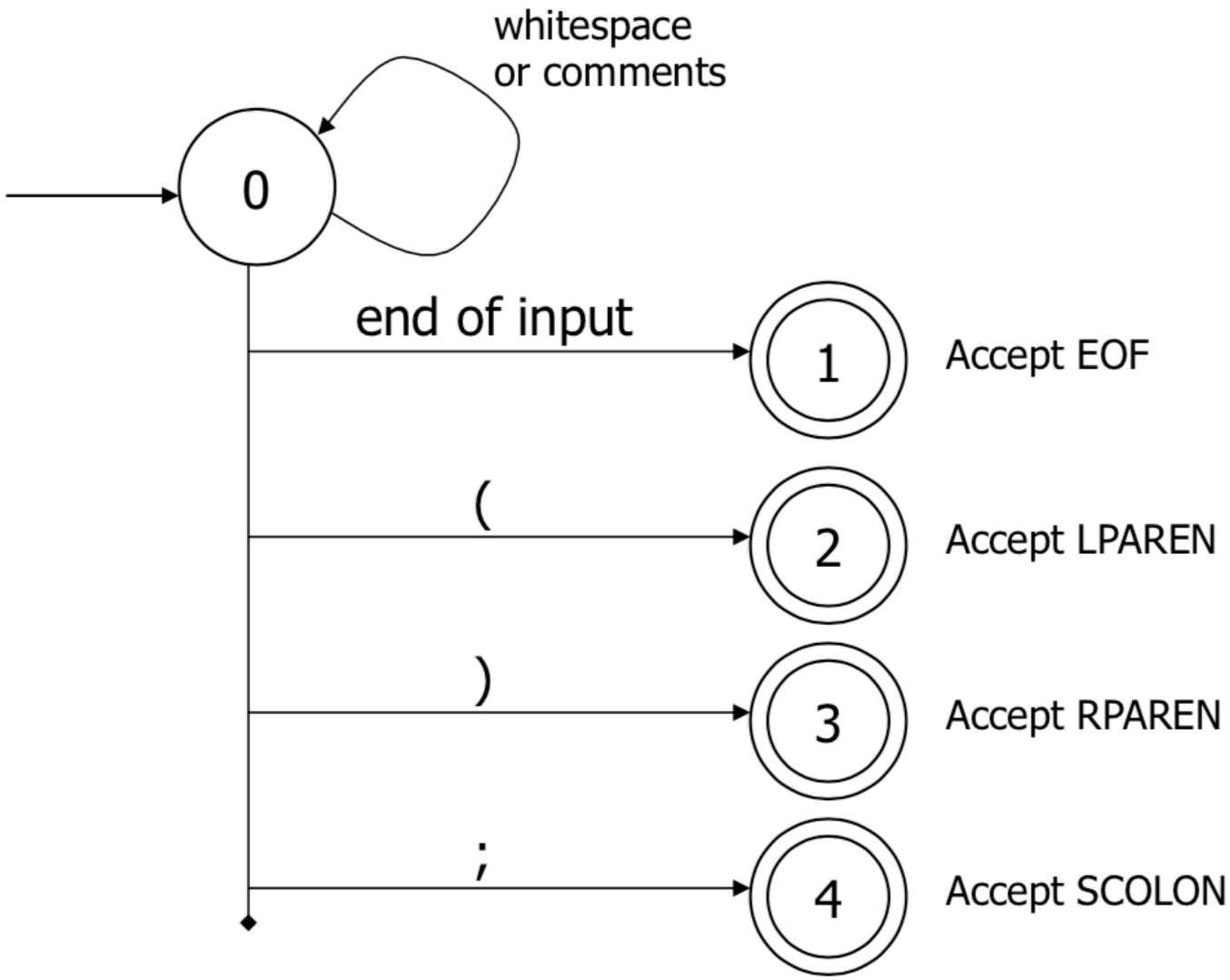
```
%{
#include "token.h"
%}
DIGIT  [0-9]
LETTER [a-zA-Z]
%%
(" "|\t|\n)   /* skip whitespace */
\+            { return TOKEN_ADD; }
while         { return TOKEN_WHILE; }
{LETTER}+     { return TOKEN_IDENT; }
{DIGIT}+      { return TOKEN_NUMBER; }
.             { return TOKEN_ERROR; }
%%
int yywrap() { return 1; }
```

Simple Scanner using Flex (3)

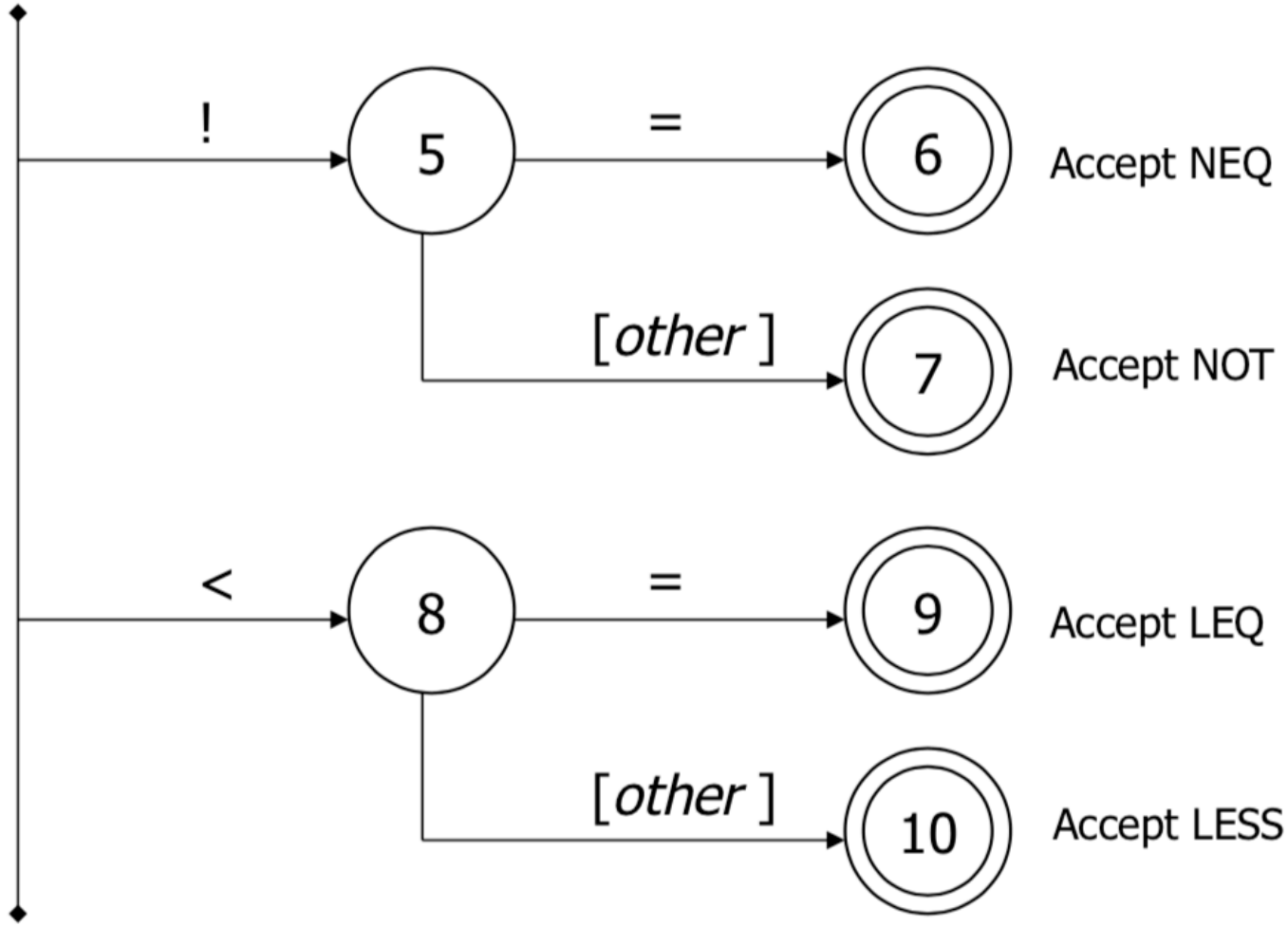
main.c

```
#include "token.h"
#include <stdio.h>

extern FILE *yyin;
extern int yylex();
extern char *yytext;

int main() {
    yyin = fopen("program.c","r");
    if(!yyin) {
        printf("could not open program.c!\n");
        return 1;
    }

    while(1) {
        token_t t = yylex();
        if(t==TOKEN_EOF) break;
        printf("token: %d  text: %s\n",t,yytext);
    }
}
```

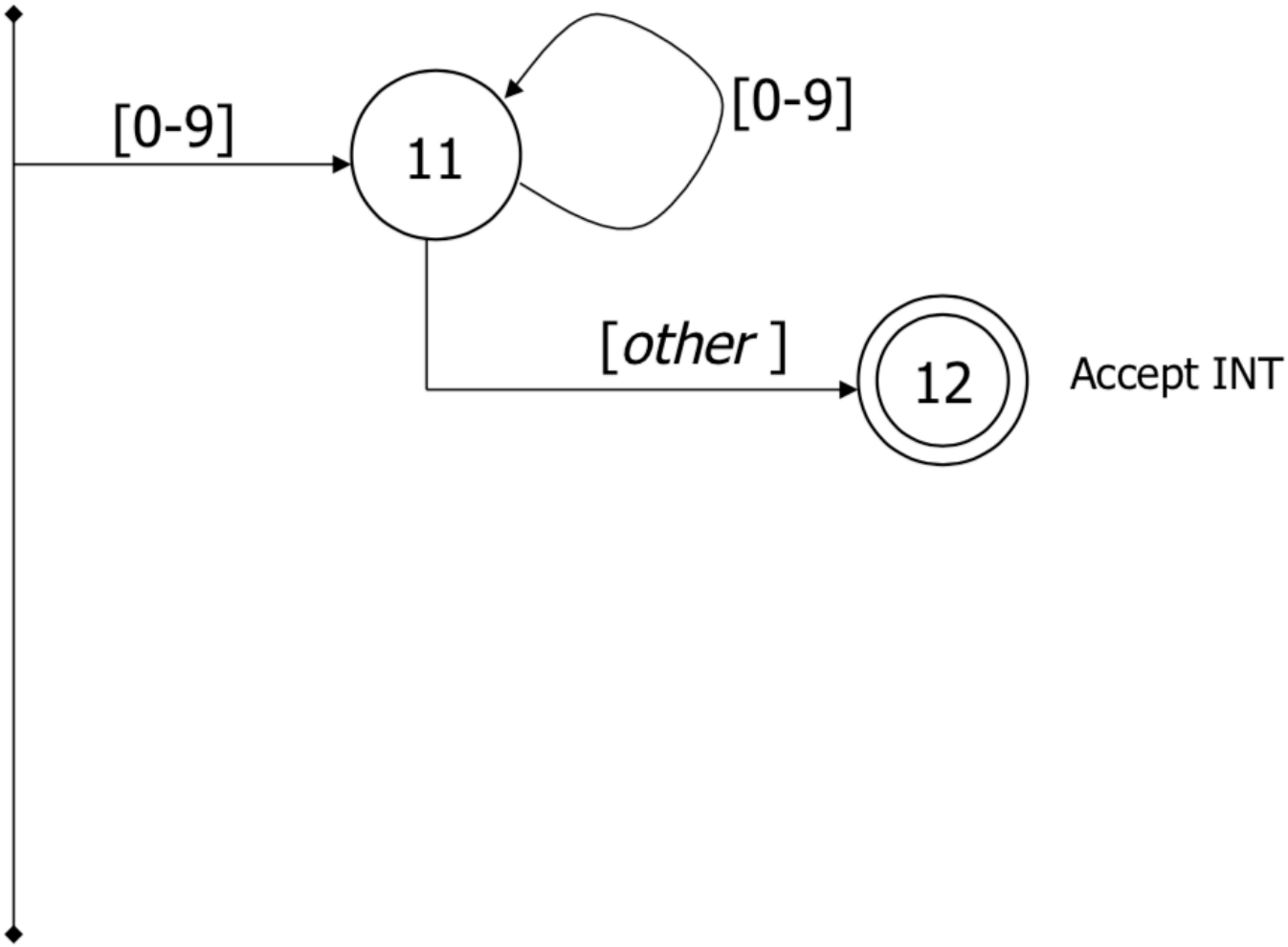
Scanner DFA Example (1)



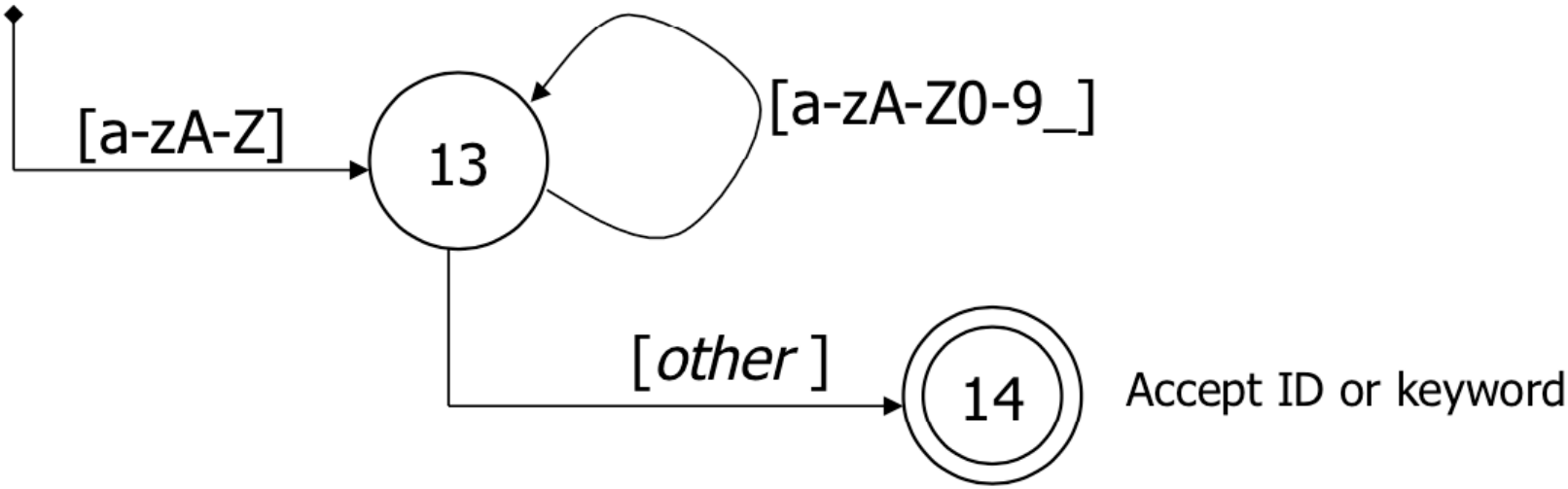
Scanner DFA Example (2)



Scanner DFA Example (3)



Scanner DFA Example (4)



- Strategies for handling identifiers vs keywords
  - Hand-written scanner: look up ID-like strings in a table of keywords to classify (e.g., hash table)
  - Machine-generated scanner: generate DFA with appropriate transitions to recognize keywords
    - Lots of states, but very efficient (no extra lookup step)

Implementing a Scanner by Hand – Token Representation

- A token is a simple, tagged structure

```
class Token {
   int kind;                  // token's lexical class
   int intVal;                // integer value if class = INT
   String id;                 // actual identifier if class = ID
   // lexical classes
   const int EOF = 0;         // "end of file" token
   const int ID   = 1;        // identifier, not keyword
   const int INT = 2;         // integer
   const int LPAREN = 4;
   const int SCOLN   = 5;
   const int WHILE   = 6;
   // etc. etc. etc. …
```

better: use enums if you have them

Simple Scanner Example

```
// global state and methods

static char nextch;      // next unprocessed input character

// advance to next input char
void getch() { … }

// skip whitespace and comments
void skipWhitespace() { … }
```

Scanner getToken() method

```
// return next input token
Token getToken() {
 Token result;

 skipWhiteSpace();

 if (no more input) {
    result = new Token(Token.EOF); return result;
 }

 switch(nextch) {
    case '(': result = new Token(Token.LPAREN); getch(); return result;
    case ')': result = new Token(Token.RPAREN); getch(); return result;
    case ';': result = new Token(Token.SCOLON); getch(); return result;

    // etc. …
```

getToken() (2)

```
case '!': // ! or !=
          getch();
          if (nextch == '=') {
            result = new Token(Token.NEQ); getch(); return result;
          } else {
            result = new Token(Token.NOT); return result;
          }

case '<': // < or <=
          getch();
          if (nextch == '=') {
            result = new Token(Token.LEQ); getch(); return result;
          } else {
            result = new Token(Token.LESS); return result;
          }
// etc. ...
```

getToken() (3)

```
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
        // integer constant
        String num = nextch;
        getch();
        while (nextch is a digit) {
           num = num + nextch; getch();
        }
        result = new Token(Token.INT, Integer(num).intValue());
        return result;
...
```

getToken (4)

```
case 'a': … case 'z':
case 'A': … case 'Z':  // id or keyword
      string s = nextch; getch();
      while (nextch is a letter, digit, or underscore) {
        s = s + nextch; getch();
      }
      if (s is a keyword) {
        result = new Token(keywordTable.getKind(s));
      } else {
        result = new Token(Token.ID, s);
      }
      return result;
```