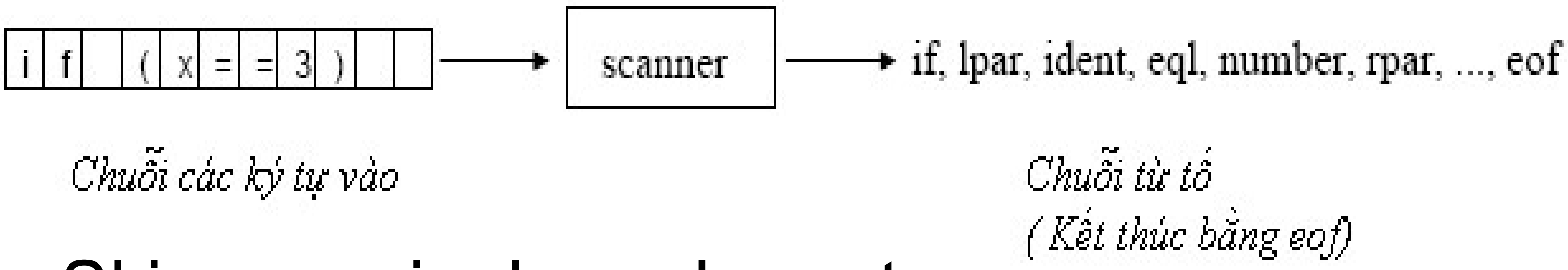
Unit 5
Scanner

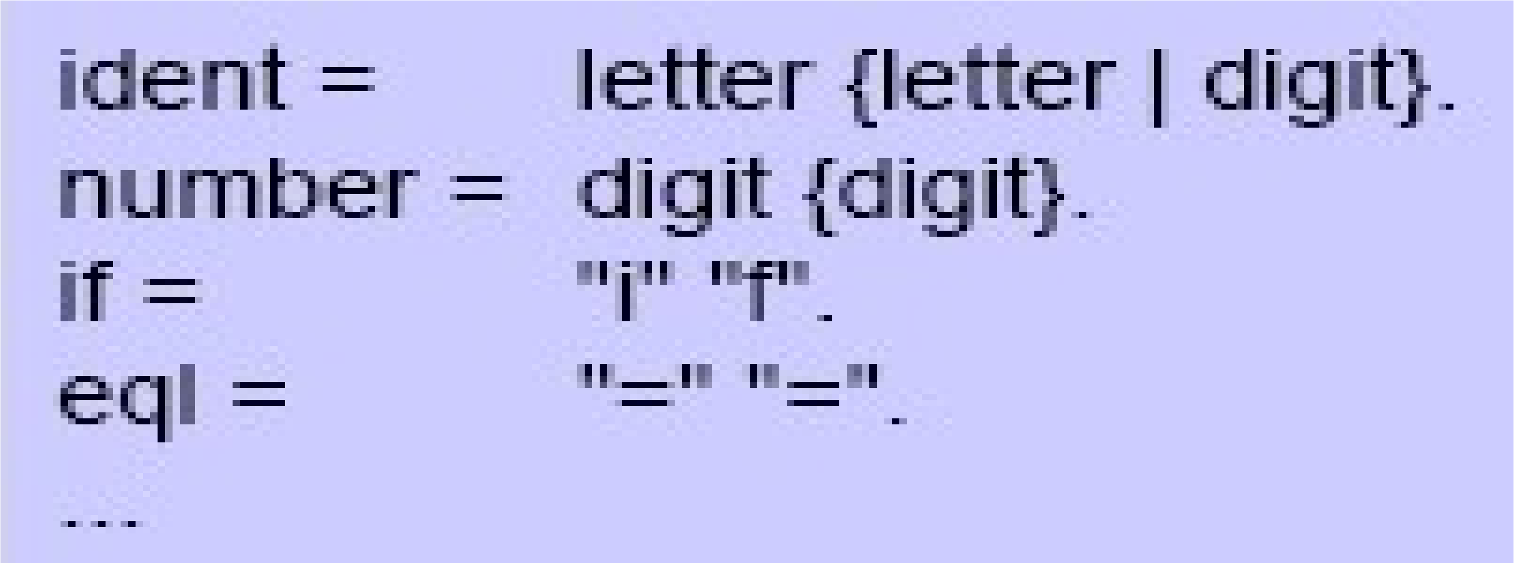
Task of a scanner

- Delivers tokens



- Skip meaningless characters
  - □ blanks
  - □ Tabulator characters
  - □ End-of-line characters (CR,LF)
  - □ Comments

Tokens have a syntactic structure



```
ident =       letter {letter | digit}.
number =   digit {digit}.
if =          "i" "f".
eql =        "=" "=".
...
```

- Why is scanning not a part of parsing?

Why is scanning not a part of parsing?

- It would make parsing more complicated,e.g.
  - Difficult distinction between identifiers and keywords
  - The scanner must have complicated rules for eliminating blanks, tabs, comments,etc.
  - => would lead to very complicated grammars

Token classes of KPL

- Unsigned integer
- Identifier
- Key word: begin,end, if,then, while, do, call, const, var, procedure, program,type, function,of,integer,char,else,for, to,array
- Character constant
- Operators:
  - □ Arithmetic

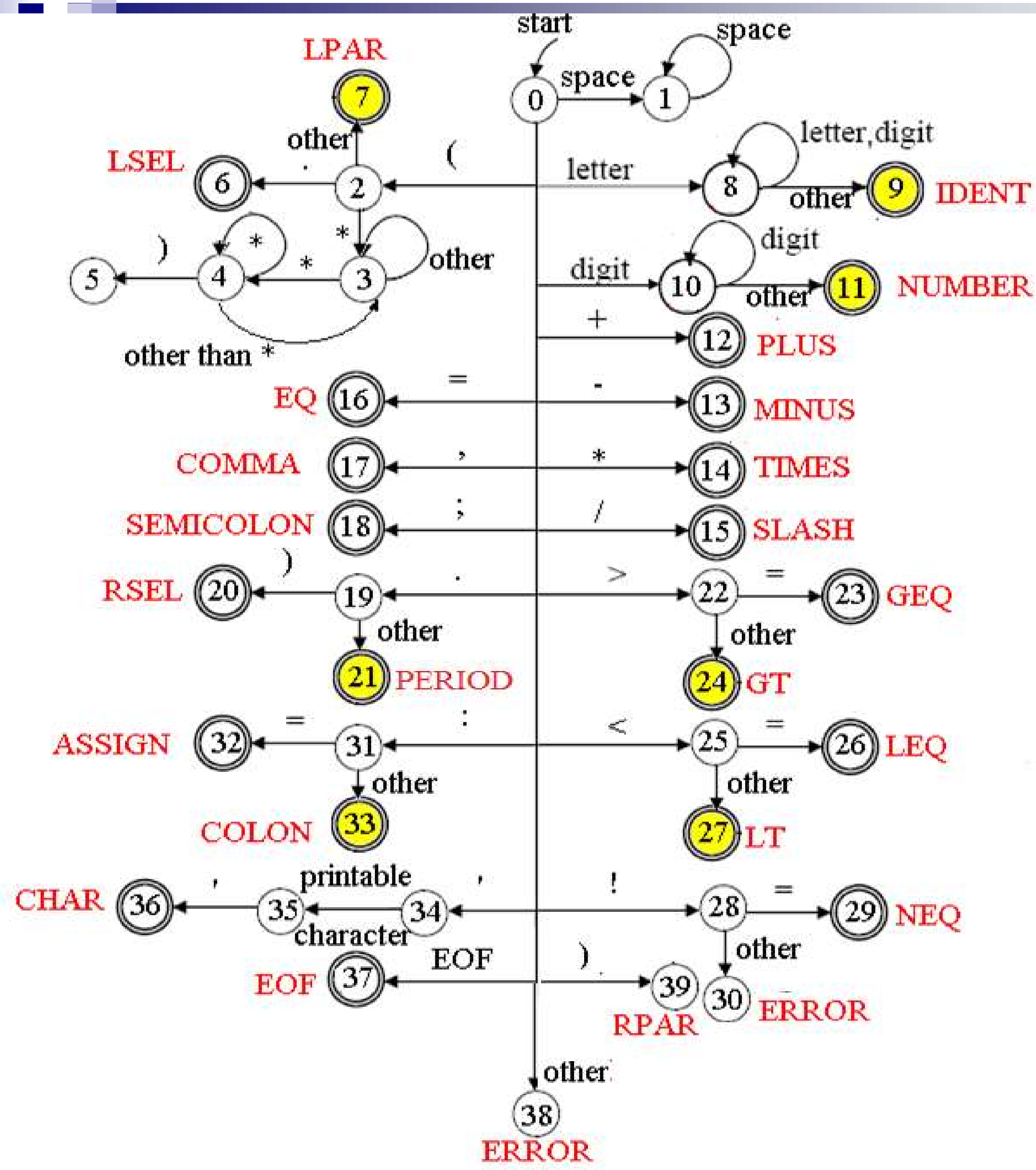
    + - */
  - □ Relational

    = != < > <= >=

  Assign :=
- Separators

  ( ) . : ; (. .)

The scanner as Finite Automaton



After every recognized token, the scanner starts in state 0 again

If an illegal character is met, the scanner would change to the states 30 or 38 which tell the scanner to stop scanning and return error messages.

Notice the yellow states

Scanner implementation based on DFA

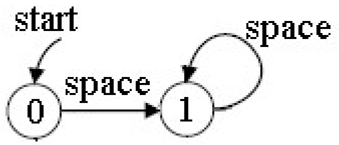
```
state = 0;
currentChar = getCurrentChar;
token = getToken();
while ( token!=EOF)
   {
        state =0;
        token = getToken();
    }
```

Token recognizer

```
switch (state)
{
case 0 : currentChar =
    getCurrentChar();
    switch (currentChar)
    {
      case space
           state = 1;
      case  lpar
           state = 2;
      case  letter
           state = 8;
      case digit
           state =10;
      case plus
           state = 12;
      ……
    }
```

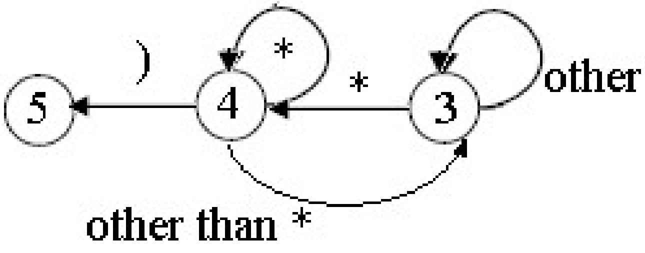
Token recognizer (cont’d)

```
case 1:
   while (current Char== space) // skip blanks
        currentChar = getCurrentChar();
   state =0;
 case 2:
   currentChar = getCurrentChar();
     switch  (currentChar)
        {
        case period
                state = 6;//  token lsel
        case times
                state =3; //skip comment
       else
                state =7; // token lpar
        }
```



Token recognizer (cont’d)

```
case 3: // skip comment
    currentChar = getCurrentChar();
    while (currentChar != times)
    {
          state = 3;
          currentChar = getCurrentChar(
    }
    state = 4;
case 4:
    currentChar = getCurrentChar();
    while (currentChar == times)
    {
          state = 4;
          currentChar = getCurrentChar();
    }
If  (currentChar == lpar)  state = 5; else state =3;
```



Token recognizer (cont’d)

```
case 9:
  if (checkKeyword (token) == TK_NONE)
  install_ident();// save to symbol table
  else
  return checkKeyword(token);
…………
```

Initialize a symbol table

- The following information about identifiers is saved
  - Name:string
  - Attribute : type name, variable name, constant name. . .
  - Data type
  - Scope
  - Address and size of the memory where the lexeme is located
  - . . .

Distinction between identifiers and keywords

- Variable ch is assigned with the first character of the lexeme.
- Read all digits and letters into string t
- Use binary search algorithm to find if there is an entry for that string in table of keyword
- If found t.kind = order of the keyword
- Otherwise, t.kind =ident
- At last, variable ch contains the first character of the next lexeme

Data structure for tokens

```
enum {
  TK_NONE, TK_IDENT, TK_NUMBER, TK_CHAR, TK_EOF,

  KW_PROGRAM, KW_CONST, KW_TYPE, KW_VAR,
  KW_INTEGER, KW_CHAR, KW_ARRAY, KW_OF,
  KW_FUNCTION, KW_PROCEDURE,
  KW_BEGIN, KW_END, KW_CALL,
  KW_IF, KW_THEN, KW_ELSE,
  KW_WHILE, KW_DO, KW_FOR, KW_TO,

  SB_SEMICOLON, SB_COLON, SB_PERIOD, SB_COMMA,
  SB_ASSIGN, SB_EQ, SB_NEQ, SB_LT, SB_LE, SB_GT, SB_GE,
  SB_PLUS, SB_MINUS, SB_TIMES, SB_SLASH,
  SB_LPAR, SB_RPAR, SB_LSEL, SB_RSEL
};
```