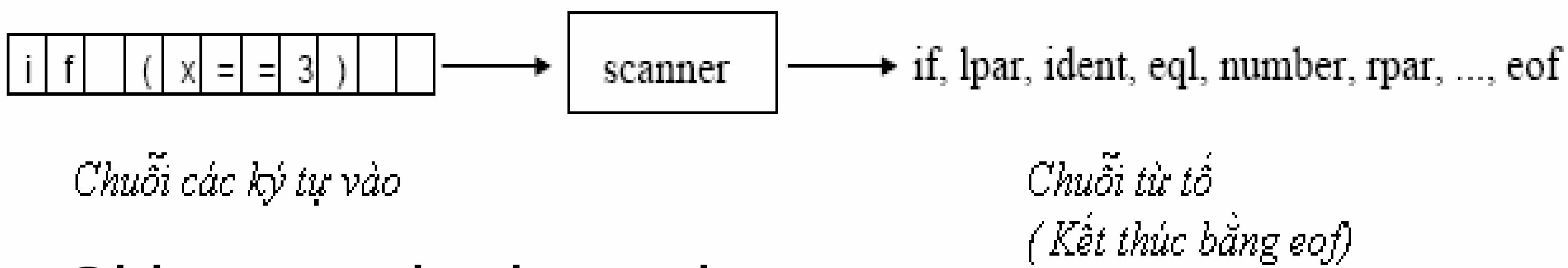
Unit 5
Scanner

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

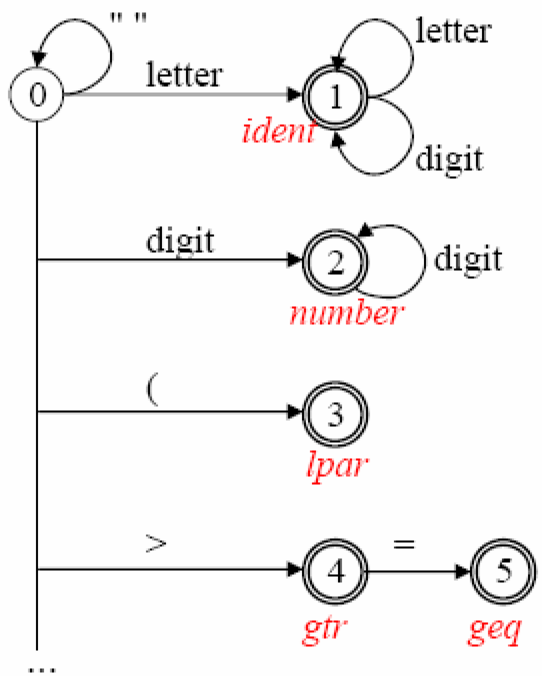
- Unsigned integer
- Identifier
- Key word: begin,end, if,then, while, do, call, const, var, procedure, program,type, function,of,integer,char,else,for, to,array
- Character constant
- Operators:
  - Arithmetic

    + - */
  - Relational

    =   !=   <   >   <=   >=
- Separators

  ( ) . : ; (. .)
- Assign :=

The scanner as Finite Automaton



Example input: MAX >= 30

$s0 \xrightarrow{M} s1 \xrightarrow{A} s1 \xrightarrow{X} s1$

- no transition with " " in s1
- *ident* recognized

$s0 \xrightarrow{\text{" "}} s0 \xrightarrow{>} s4 \xrightarrow{=} s5$

- skips blanks at the beginning
- does not stop in s4
- no transition with " " in s5
- *geq* recognized

$s0 \xrightarrow{\text{" "}} s0 \xrightarrow{3} s2 \xrightarrow{0} s2$

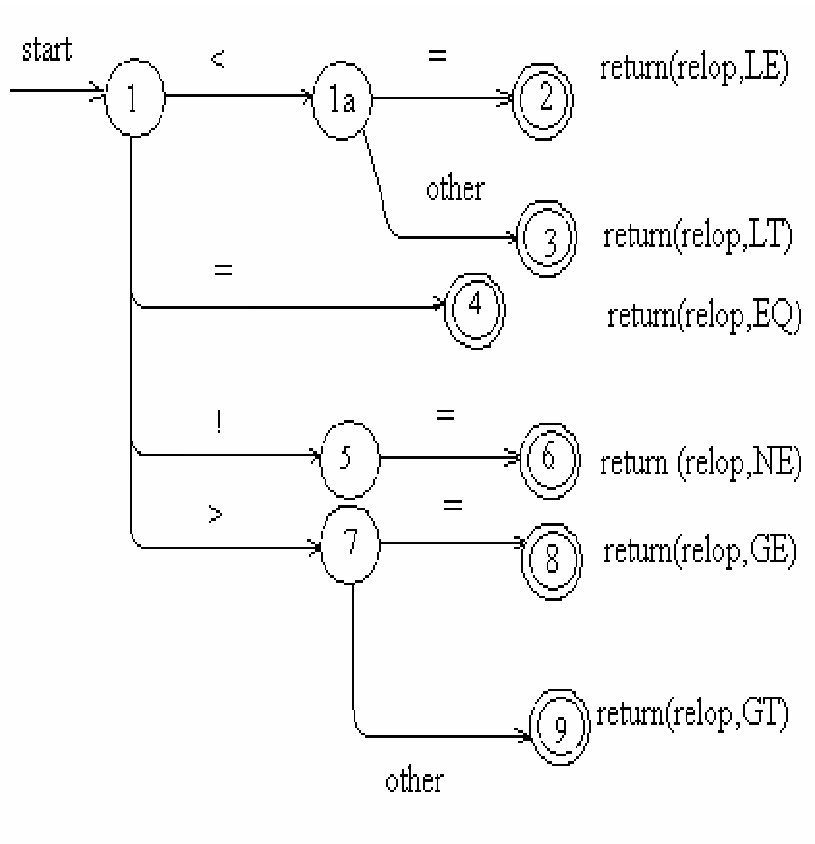
- skips blanks at the beginning
- no transition with " " in s2
- *number* recognized

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

Scanner implementation

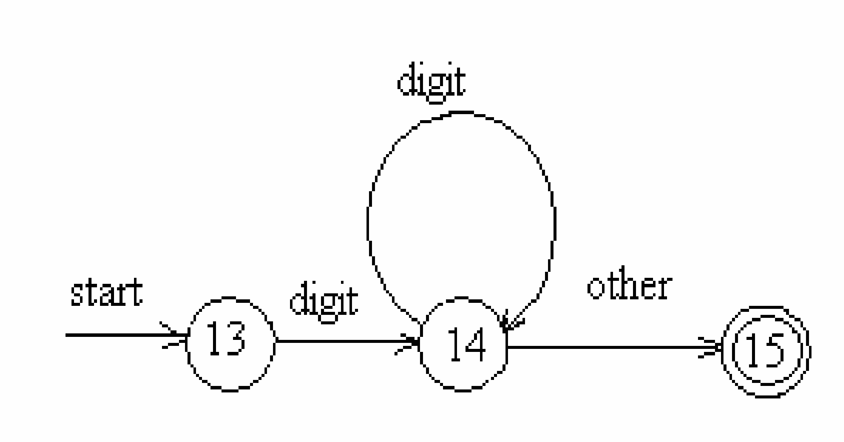
```
switch(state)
  {
   case 0 // Skip spaces
   case 1// Recognize Relational Operators
   case 10
   // Recognize identifiers
   case 13
   // Recognize numbers
   . . .
   }
```

Relational operators



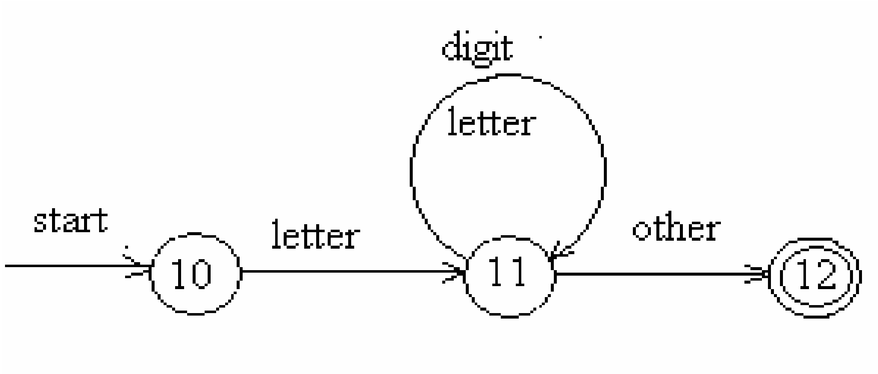

```
case 0 : c= nextchar();
     // c is lookahead character
              if(c==blank || c= =tab || c= =newline ){
                             state = 0;
              lexeme_beginning++;
    //advance beginning of lexeme
              }
case 1:
     if(c= = '<') state = 1a;
     else if (c= = '=') state = 4;
     else if (c= = '!') state = 5;
     else if (c= = '>') state = 7;
     else state =fail();break;
case 1a: c:=nextchar();
if (c= = '=') state = 2;
Else state=3;
case 2: return (leq)
```

Unsigned Integers




```
case 13: c = nextchar();
if(isdigit(c)) state = 14;
case 14: c=nextchar();
if(isdigit(c)) state = 14;
else state = 15; break;
case 15 : retract(1);
   install_num();
return(num);
```

Identifier




```
case 10:
c = nextchar();
if(isletter(c)) state =11;
else state = fail; break;
case 11:
c = nextchar();
if(isletter(c)) state =11;
else if (isdigit(c)) state = 11;
else state = 12; break;
case 12: retract(1) ;
   install_id();
return (gettoken());
```

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 representing tokens

```
enum symbol
{
    nul, ident, number,
    plus,minus, times, slash,
    eql,neq, lss,leq,grt,geq,
    lparen, rparen, comma, semicolon, period, becomes,
    quote, colon,       lsquare,rsquare,
    beginsym, endsym, ifsym,thensym, whilesym,dosym,
    callsym,constsym,varsym,procsym,programsym,typesym,
    funcsym,ofsym,integersym,charsym,elsesym,forsym,
    tosym, arraysym
};
```