

Languages and Compilers **(SProg og Oversættere)**

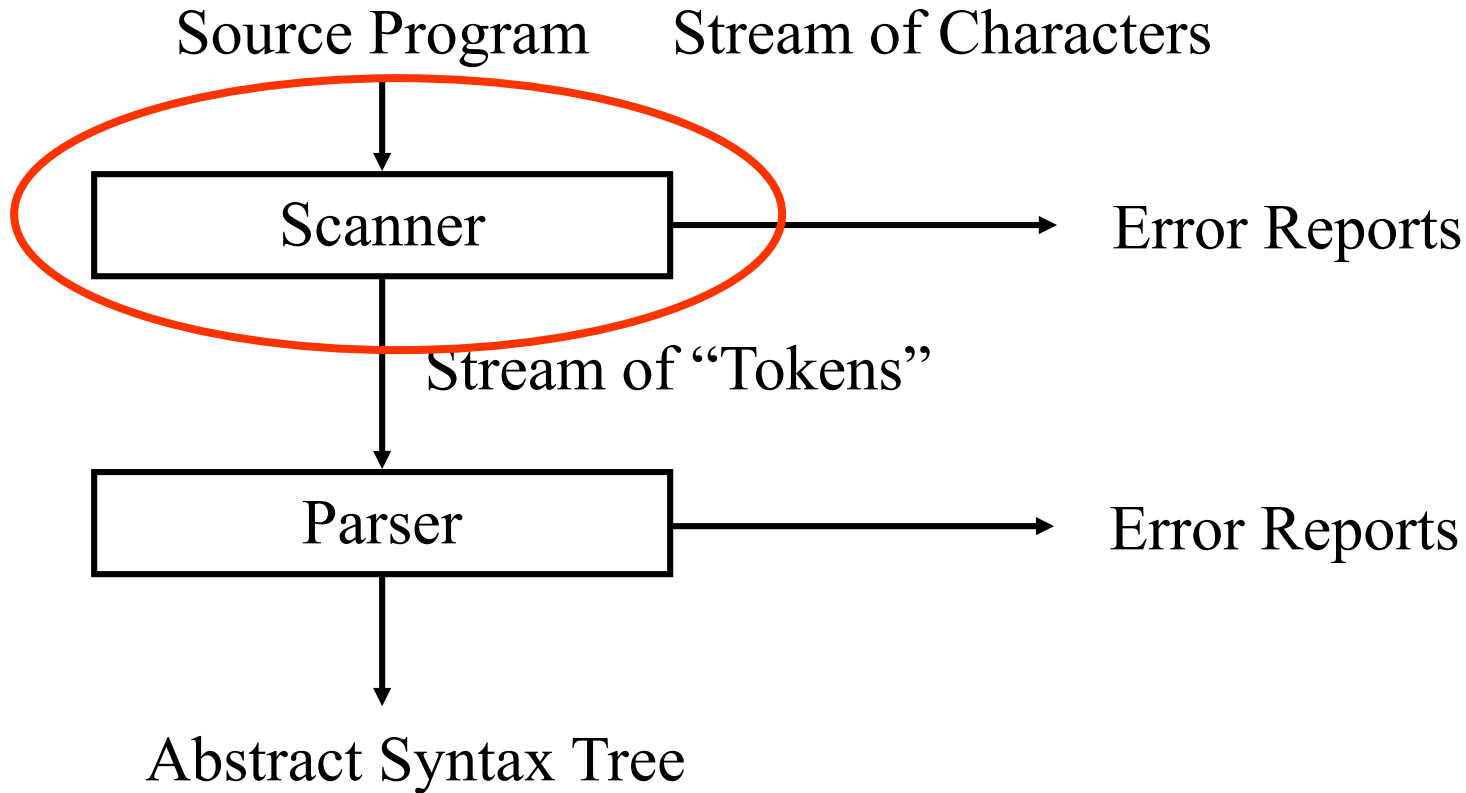
Lexical analysis

Lexical analysis

- Describe the role of the lexical analysis phase
- Describe how a scanner can be implemented by hand or auto-generated
- Describe regular expressions and finite automata

Syntax Analysis: Scanner

Dataflow chart



1) Scan: Divide Input into Tokens

An example ac source program:

```
f b
i a
a = 5
b = a + 3.2
p b
```

Lexems are “words” in the input, for example keywords, operators, identifiers, literals, etc.

Tokens is a datastructure for lexems and additional information



<i>floatdl</i>	<i>id</i>	<i>intdcl</i>	<i>id</i>	<i>id</i>	<i>assign</i>	<i>inum</i>	...
f	b	i	a	a	=	5	

...	<i>assign</i>	<i>id</i>	<i>plus</i>	<i>fnum</i>	<i>print</i>	<i>id</i>	<i>eot</i>
	=	a	+	3.2	p	b	

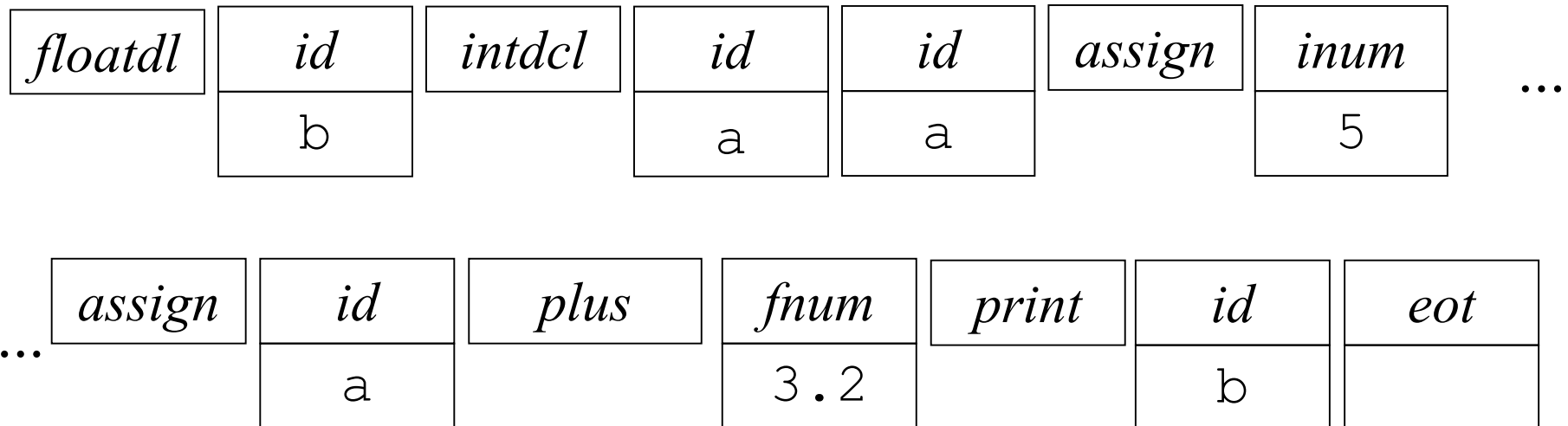
1) Scan: Divide Input into Tokens

An example ac source program:

```
f b
i a
a = 5
b = a + 3.2
p b
```

Lexems are “words” in the input, for example keywords, operators, identifiers, literals, etc.

Tokens is a datastructure for lexems and additional information



Implement Scanner based on RE by hand

- 1) Express the “lexical” grammar as RE
(sometimes it is easier to start with a BNF or an EBNF and do necessary transformations)
 - For each variant make a switch on the first character by peeking the input stream
 - For each repetition (..)* make a while loop with the condition to keep going as long as peeking the input still yields an expected character
 - Sometimes the “lexical” grammar is not reduced to one single RE but a small set of REs – in this case a switch or if-then-else case analysis is used to determine which rule is being recognized, before following the first two steps

Developing a Scanner

- Express the “lexical” grammar in EBNF

```
Token ::= Identifier | Integer-Literal | Operator |  
        ; | : | := | ~ | ( | ) | eot  
Identifier ::= Letter (Letter | Digit)*  
Integer-Literal ::= Digit Digit*  
Operator ::= + | - | * | / | < | > | =  
Separator ::= Comment | space | eol  
Comment ::= ! Graphic* eol
```

Now perform substitution and left factorization...

```
Token ::= Letter (Letter | Digit)*  
        | Digit Digit*  
        | + | - | * | / | < | > | =  
        | ; | : ( = | ε ) | ~ | ( | ) | eot  
Separator ::= ! Graphic* eol | space | eol
```

```

Token ::= Letter ( Letter | Di gi t ) *
        | Di gi t Di gi t *
        | + | - | * | / | < | > | =
        | ; | : ( = | ε ) | ~ | ( | ) | eot

```

```

private byte scanToken() {
    switch (current Char) {
    case 'a': case 'b': ... case 'z':
    case 'A': case 'B': ... case 'Z':
        scan Letter ( Letter | Di gi t ) *
        return Token. IDENTIFIER;
    case '0': ... case '9':
        scan Di gi t Di gi t *
        return Token. INTLITERAL ;
    case '+' : case '-' : ... : case '=' :
        takeIt();
        return Token. OPERATOR;
        ... etc ...
    }
}

```


Developing a Scanner

Let's look at the identifier case in more detail

```
...  
return ...  
case 'a': case 'b': ... case 'z':  
case 'A': case 'B': ... case 'Z':  
    acceptIt();  
    while (isLetter(currentChar)  
           || isDigit(currentChar) )  
        acceptIt();  
    return Token.IDENTIFIER;  
case '0': ... case '9':  
...
```

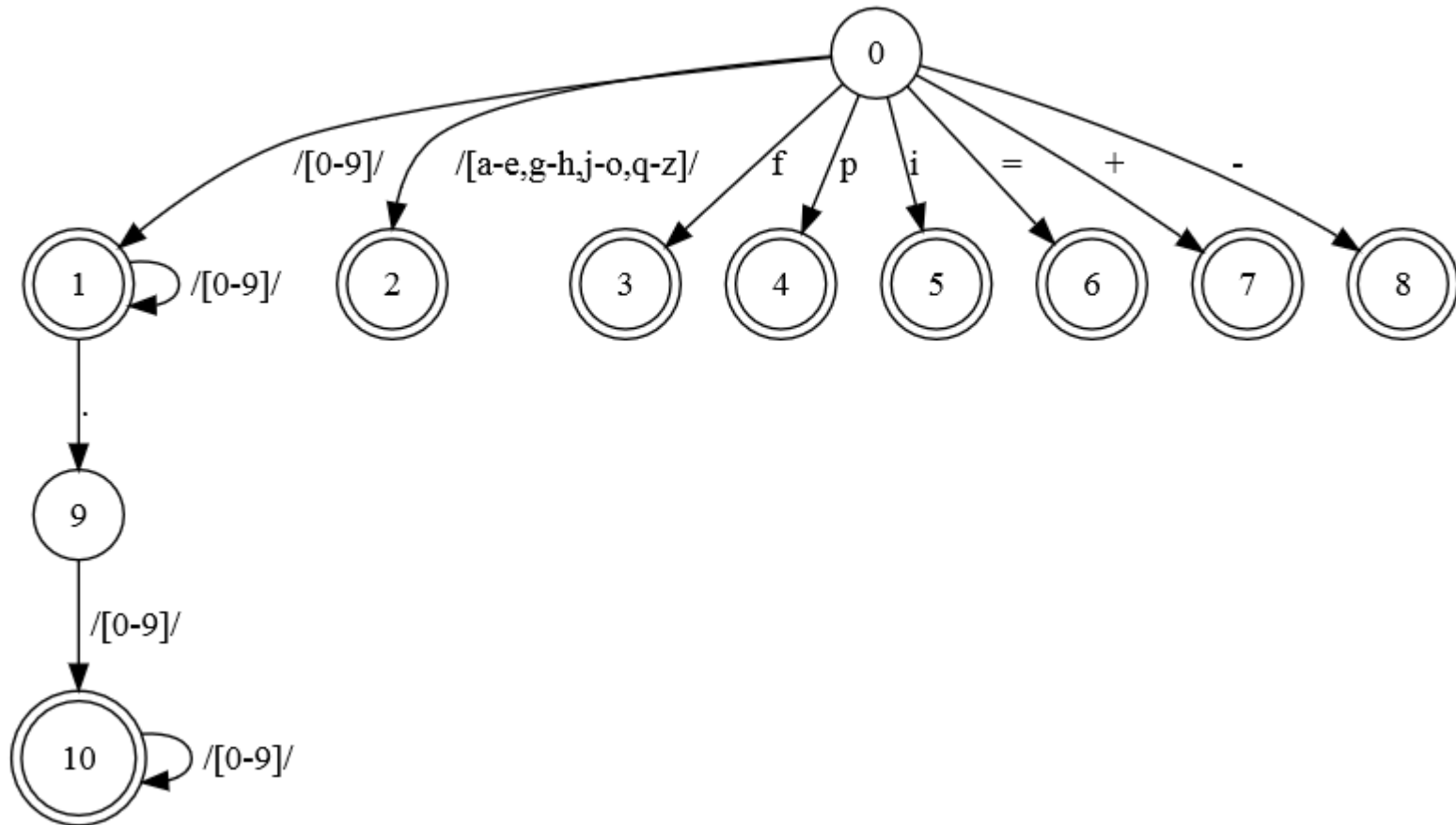
Thus developing a scanner is a mechanical task.

Token Specification

Terminal	Regular Expression
floatdcl	"f"
intdcl	"i"
print	"p"
id	[a - e] [g - h] [j - o] [q - z]
assign	"="
plus	"+"
minus	"_"
inum	[0 - 9] ⁺
fnum	[0 - 9] ⁺ .[0 - 9] ⁺
blank	(" ") ⁺

Figure 2.3: Formal definition of ac tokens.

$[0-9]^+ \mid [0-9]^+.[0-9]^+ \mid [a-e,g-h,j-o,q-z] \mid f \mid p \mid i \mid = \mid \backslash \mid + \mid -$



```

function SCANNER( ) returns Token
  while s.PEEK( ) = blank do call s.ADVANCE( )
  if s.EOF( )
  then ans.type  $\leftarrow$  $
  else
    if s.PEEK( )  $\in$  { 0, 1, ..., 9 }
    then ans  $\leftarrow$  SCANDIGITS( )
    else
      ch  $\leftarrow$  s.ADVANCE( )
      switch (ch)
        case { a, b, ..., z } - { i, f, p }
          ans.type  $\leftarrow$  id
          ans.val  $\leftarrow$  ch
        case f
          ans.type  $\leftarrow$  floatdcl
        case i
          ans.type  $\leftarrow$  intdcl
        case p
          ans.type  $\leftarrow$  print
        case =
          ans.type  $\leftarrow$  assign
        case +
          ans.type  $\leftarrow$  plus
        case -
          ans.type  $\leftarrow$  minus
        case default
          call LEXICALERROR( )
      return (ans)
    end
  end

```

Figure 2.5: Scanner for the ac language. The variable *s* is an input stream of characters.

```

function SCANDIGITS( ) returns token
    tok.val  $\leftarrow$  " "
    while s.PEEK( )  $\in \{0, 1, \dots, 9\}$  do
        tok.val  $\leftarrow$  tok.val + s.ADVANCE( )
    if s.PEEK( )  $\neq$  "."
    then tok.type  $\leftarrow$  inum
    else
        tok.type  $\leftarrow$  fnum
        tok.val  $\leftarrow$  tok.val + s.ADVANCE( )
        while s.PEEK( )  $\in \{0, 1, \dots, 9\}$  do
            tok.val  $\leftarrow$  tok.val + s.ADVANCE( )
    return (tok)
end

```

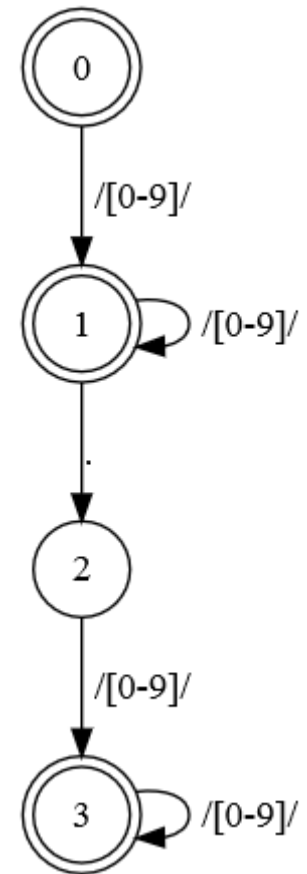


Figure 2.6: Finding inum or fnum tokens for the ac language.

Generating Scanners

- Generation of scanners is based on
 - Regular Expressions: to describe the tokens to be recognized
 - Finite State Machines: an execution model to which RE's are “compiled”

A possible algorithm:

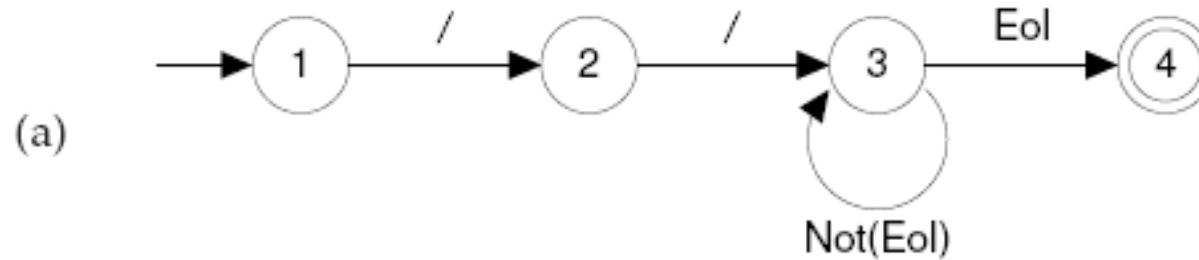
- Convert RE into NDFA- ϵ
- Convert NDFA- ϵ into NDFA
- Convert NDFA into DFA
- generate Java/C/... code

Implementing a DFA

```
/* Assume CurrentChar contains the first character to be scanned */  
State ← StartState  
while true do  
    NextState ← T[State, CurrentChar]  
    if NextState = error  
    then break  
    State ← NextState  
    CurrentChar ← READ( )  
if State ∈ AcceptingStates  
then /* Return or process the valid token */  
else /* Signal a lexical error */
```

Figure 3.3: Scanner driver interpreting a transition table.

Comment \rightarrow $//(\text{Not}(\text{Eol}))^*\text{Eol}$



(b)

State	Character				
	/	Eol	a	b	...
1	2				
2	3				
3	3	4	3	3	3
4					

Figure 3.2: DFA for recognizing a single-line comment. (a) transition diagram; (b) corresponding transition table.

Implementing a Scanner as a DFA

Slightly different from previously shown implementation (but similar in spirit):

- Not the goal to match entire input
=> when to stop matching?
 - Token(if), Token(Ident i) vs. Token(Ident ifi)

Match longest possible token

Report error (and continue) when reaching error state.

- How to identify matched token class (not just true|false)
Final state determines matched token class

Performance considerations

- Performance of scanners is important for production compilers, for example:
 - 30,000 lines per minute (500 lines per second)
 - 10,000 characters per second (for an average line of 20 characters)
 - For a processor that executes 10,000,000 instructions per second, 1,000 instructions per input character
 - Considering other tasks in compilers, 250 instructions per character is more realistic
- Size of scanner sometimes matters
 - Including keyword in scanner increases table size
 - E.g. Pascal has 35 keywords, including them increases states from 37 to 165
 - Uncompressed this increases table entries from 4699 to 20955