2. Lexical Analysis

2.1 Tasks of a Scanner

2.2 Regular Grammars and Finite Automata

2.3 Scanner Implementation

Tasks of a Scanner

1. Delivers terminal symbols (tokens)

| i | f | | (| x | = | = | 3 |) | | | → scanner → IF, LPAR, IDENT, EQ, NUMBER, RPAR, ..., EOF

character stream

token stream
(must end with EOF)

2. Skips meaningless characters

- blanks
- tabulator characters
- end-of-line characters (CR, LF)
- comments

Tokens have a syntactical structure, e.g.

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

Why is scanning not part of parsing?

Why is Scanning not Part of Parsing?

It would make parsing more complicated
(e.g. difficult distinction between keywords and names)

```
Statement =  ident "=" Expr ";"
          |  "if" "(" Expr ")" ... .
```

One would have to write this as follows:

```
Statement = "i" (  "f" "(" Expr ")" ...
                |  notF {letter | digit} "=" Expr ";"
                )
          |  notI {letter | digit} "=" Expr ";".
```

The scanner must eliminate blanks, tabs, end-of-line characters and comments
(these characters can occur anywhere => would lead to very complex grammars)

```
Statement =  "if" {Blank} "(" {Blank} Expr {Blank} ")" {Blank} ... .
Blank = " " | "\r" | "\n" | "\t" | Comment.
```

Tokens can be described with regular grammars
(simpler and more efficient than context-free grammars)

2. Lexical Analysis

2.1 Tasks of a Scanner

2.2 Regular Grammars and Finite Automata

2.3 Scanner Implementation

Regular Grammars

Definition

A grammar is called regular if it can be described by productions of the form:

```
A = a.
A = b B.
```

a, b ∈ TS
A, B ∈ NTS

Example Grammar for names

```
Ident  =  letter
       |  letter Rest.
Rest   =  letter
       |  digit
       |  letter Rest
       |  digit Rest.
```

e.g., derivation of the name xy3

Ident ⇒ letter Rest ⇒ letter letter Rest ⇒ letter letter digit

Alternative definition

A grammar is called regular if it can be described by a single non-recursive EBNF production.

Example Grammar for names

```
Ident  =  letter { letter | digit }.
```

Examples

Can we transform the following grammar into a regular grammar?

```
E = T { "+" T }.
T = F { "*" F }.
F = id.
```

Can we transform the following grammar into a regular grammar?

```
E = F { "*" F }.
F = id | "(" E ")".
```

Limitations of Regular Grammars

Regular grammars cannot deal with *nested structures*
because they cannot handle *central recursion*!

But central recursion is important in most programming languages.

- nested expressions — Expr ⇒ ... "(" Expr ")" ...
- nested statements — Statement ⇒ "do" Statement "while" "(" Expr ")"
- nested classes — Class ⇒ "class" "{" ... Class ... "}"

For productions like these we need context-free grammars.

But most lexical structures are regular

names	letter { letter | digit }
numbers	digit { digit }
strings	"\"" { noQuote } "\""
keywords	letter { letter }
operators	">" "="

Exception: nested comments

/* ..... /* ... */ ..... */

The scanner must treat them in a special way

Regular Expressions

Alternative notation for regular grammars

Definition

1. ε (the empty string) is a regular expression
2. A terminal symbol is a regular expression
3. If α and β are regular expressions the following expressions are also regular:

α β	
(α | β)	
(α)?	ε | α
(α)*	ε | α | αα | ααα | ...
(α)+	α | αα | ααα | ...

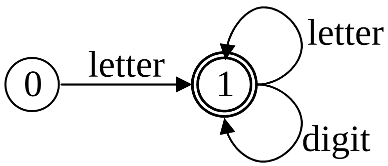
Examples

"w" "h" "i" "l" "e"	while
letter (letter | digit)*	names
digit+	numbers

Deterministic Finite Automaton (DFA)

Can be used to analyze regular languages

Example



final state

start state is always
state 0 by convention

State transition function as a table

δ	letter	digit
s0	s1	error
s1	s1	s1

"finite", because δ can be written down explicitly

Definition

A deterministic finite automaton is a 5 tuple (S, I, δ, s0, F)

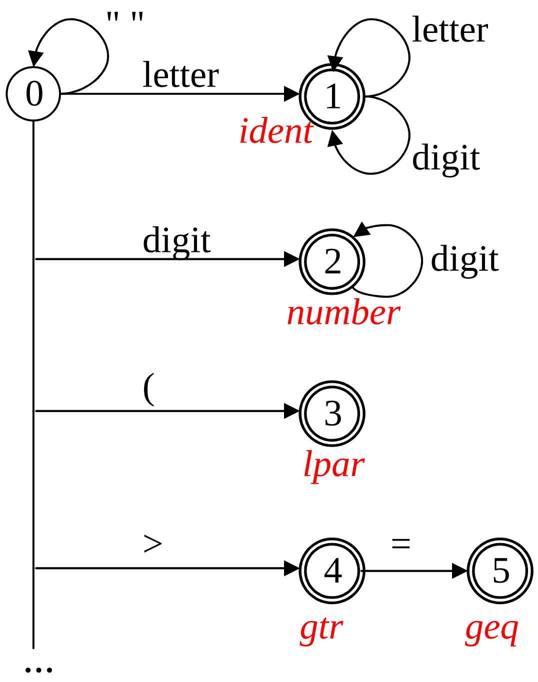
- S — set of states
- I — set of input symbols
- δ: S x I → S — state transition function
- s0 — start state
- F — set of final states

The **language** recognized by a DFA is the set of all symbol sequences that lead from the start state into one of the final states

A DFA has recognized a sentence
- if it is in a final state
- <u>and</u> if the input is totally consumed <u>or</u> there is no possible transition with the next input symbol

The Scanner as a DFA

The scanner can be viewed as a big DFA



Example

input: max >= 30

s0 —m a x→ s1
- no transition with " " in s1
- *ident* recognized

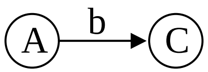
s0 —> =→ s5
- skips blanks at the beginning
- does not stop in s4
- no transition with " " in s5
- *geq* recognized

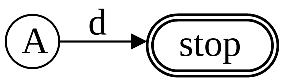
s0 —3 0→ s2
- skips blanks at the beginning
- no transition with " " in s2
- *number* recognized

After every recognized token the scanner starts in s0 again

Transformation: reg. grammar ↔ DFA

A reg. grammar can be transformed into a DFA according to the following scheme

A = b C. ⇔ 

A = d. ⇔ 

Example

grammar

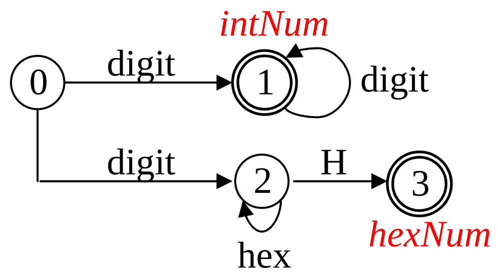
A = a B | b C | c.
B = b B | c.
C = a C | c.

automaton

Nondeterministic Finite Automaton (NDFA)

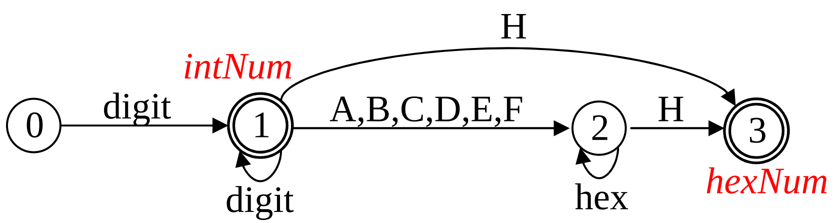
Example

```
intNum =   digit { digit }.
hexNum = digit { hex } "H".
digit =       "0" | "1" | ... | "9".
hex =         digit | "A" | ... | "F".
```



nondeterministic because there are 2 possible transitions with *digit* in s0

Every NDFA can be transformed into an equivalent DFA
(algorithm see for example: Aho, Sethi, Ullman: Compilers)



Implementation of a DFA (Variant 1)

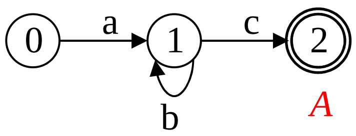
Implementation of δ as a matrix

```
int[,] delta = new int[maxStates, maxSymbols];
int lastState, state = 0;   // DFA starts in state 0
do {
    int sym = next symbol;
    lastState = state;
    state = delta[state, sym];
} while (state != undefined);
assert(lastState ∈ F);  // F is set of final states
return recognizedToken[lastState];
```

This is an example of a universal *table-driven algorithm*

Example for δ

A = a { b } c.

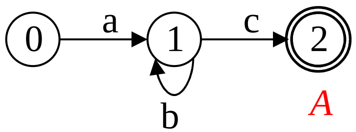


δ	a	b	c	
0	1	-	-	
1	-	1	2	
2	-	-	-	F

```
int[,] delta = { {1, -1, -1}, {-1, 1, 2}, {-1, -1, -1} };
```

This implementation would be too inefficient for a real scanner.

Implementation of a DFA (Variant 2)



Hard-coding the states in source code

```
char ch = read();
s0:   if (ch == 'a') { ch = read(); goto s1; }
      else goto err;
s1:   if (ch == 'b') { ch = read(); goto s1; }
      else if (ch == 'c') { ch = read(); goto s2; }
      else goto err;
s2:   return A;
err:  return errorToken;
```

In Java this is more tedious:

```
int state = 0;
loop:
  for (;;) {
    char ch = read();
    switch (state) {
      case 0:  if (ch == 'a') { state = 1; break; }
               else break loop;
      case 1:  if (ch == 'b') { state = 1; break; }
               else if (ch == 'c') { state = 2; break; }
               else break loop;
      case 2: return A;
    }
  }
return errorToken;
```

2. Lexical Analysis

2.1 Tasks of a Scanner

2.2 Regular Grammars and Finite Automata

2.3 Scanner Implementation

Scanner Interface

```
class Scanner {
    static void     Init (TextReader r) {...}
    static Token  Next () {...}
}
```

For efficiency reasons methods are static (there is just one scanner per compiler)

Initializing the scanner

```
Scanner.Init(new StreamReader("myProg.zs"));
```

Reading the token stream

```
Token t;
for (;;) {
    t = Scanner.Next();
    ...
}
```

Tokens

```
class Token {
    int kind;       // token code
    int line;       // token line (for error messages)
    int col;        // token column (for error messages)
    int val;        // token value (for number and charCon)
    string str;     // token string (for numbers and identifiers)
}
```

Token codes for Z#

const int

error token	token classes	operators and special characters		keywords	end of file
NONE = 0,	IDENT = 1,	PLUS = 4, /* + */	ASSIGN = 17, /* = */	BREAK = 29,	EOF = 40;
	NUMBER = 2,	MINUS = 5, /* - */	PPLUS = 18, /* ++ */	CLASS = 30,	
	CHARCONST = 3,	TIMES = 6, /* * */	MMINUS = 19, /* -- */	CONST = 31,	
		SLASH = 7, /* / */	SEMICOLON = 20, /* ; */	ELSE = 32,	
		REM = 8, /* % */	COMMA = 21, /* , */	IF = 33,	
		EQ = 9, /* == */	PERIOD = 22, /* . */	NEW = 34,	
		GE = 10, /* >= */	LPAR = 23, /* (*/	READ = 35,	
		GT = 11, /* > */	RPAR = 24, /*) */	RETURN = 36,	
		LE = 12, /* <= */	LBRACK = 25, /* [*/	VOID = 37,	
		LT = 13, /* < */	RBRACK = 26, /*] */	WHILE = 38,	
		NE = 14, /* != */	LBRACE = 27, /* { */	WRITE = 39,	
		AND = 15, /* && */	RBRACE = 28, /* } */		
		OR = 16, /* || */			

Scanner Implementation

Static variables in the scanner

```
static TextReader input;        // input stream
static char ch;                 // next input character (still unprocessed)
static int line, col;           // line and column number of the character ch

const int EOF = '\u0080';       // character that is returned at the end of the file
```

Init()

```
public static void Init (TextReader r) {
    input = r;
    line = 1; col = 0;
    NextCh();  // reads the first character into ch and increments col to 1
}
```

NextCh()

```
static void NextCh() {
    try {
        ch = (char) input.Read(); col++;
        if (ch == '\n') { line++; col = 0; }
        else if (ch == '\uffff') ch = EOF;
    } catch (IOException e) { ch = EOF; }
}
```

- *ch* = next input character
- returns *EOF* at the end of the file
- increments *line* and *col*

Method Next()

```
public static Token Next () {
   while (ch <= ' ') NextCh();  // skip blanks, tabs, eols
   Token t = new Token(); t.line = line, t.col = col;
   switch (ch) {
      case 'a': ... case 'z': case 'A': ... case 'Z': ReadName(t); break;
      case '0': case '1': ... case '9': ReadNumber(t); break;
      case ';': NextCh(); t.kind = Token.SEMICOLON; break;
      case '.': NextCh(); t.kind = Token.PERIOD; break;
      case EOF: t.kind = Token.EOF; break;  // no NextCh() any more
      ...
      case '=':    NextCh();
                   if (ch == '=') { NextCh(); t.kind = Token.EQ; }
                   else t.kind = Token.ASSIGN;
                   break;
      case '&':    NextCh();
                   if (ch == '&') { NextCh(); t.kind = Token.AND; }
                   else t.kind = NONE;
                   break;
      ...
      case '/':    NextCh();
                   if (ch == '/') {
                      do NextCh(); while (ch != '\n' && ch != EOF);
                      t = Next();  // call scanner recursively
                   } else t.kind = Token.SLASH;
                   break;
      default:     NextCh(); t.kind = Token.NONE; break;
   }
   return t;
} // ch holds the next character that is still unprocessed
```

- names, keywords
- numbers
- simple tokens
- composite tokens
- comments
- invalid character

Further Methods

static void ReadName (Token t)

- At the beginning *ch* holds the first letter of the name
- Reads further letters, digits and '_' and stores them in *t.str*
- Looks up the name in a keyword table (using hashing or binary search)
  if found: t.kind = *token number of the keyword*;
  otherwise: t.kind = Token.IDENT;
- At the end *ch* holds the first character after the name

static void ReadNumber (Token t)

- At the beginning *ch* holds the first digit of the number
- Reads further digits, storing them in *t.str*; then converts the digit string into a number and stores the value in *t.val*.
  if overflow: report an error
- t.kind = Token.NUMBER;
- At the end *ch* holds the first character after the number

Efficiency Considerations

Typical program size

about 1000 statements
⇒ about 6000 tokens
⇒ about 60000 characters

Scanning is one of the most time-consuming phases of a compiler
(takes about 20-30% of the compilation time)

Touch every character as seldom as possible

therefore *ch* is global and not a parameter of N*extCh()*

For large input files it is a good idea to use buffered reading

```
Stream file = new FileStream("MyProg.zs");
Stream buf = new BufferedStream(file);
TextReader r = new StreamReader(buf);
Scanner.Init(r);
```

Does not pay off for small input files