2. Lexical Analysis

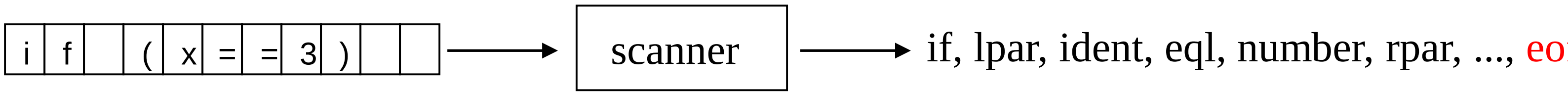
2.1 Tasks of a Scanner

2.2 Regular Grammars and Finite Automata

2.3 Scanner Implementation

Tasks of a Scanner

1. Recognizes tokens



character stream

token stream
(must end with eof)

2. Skips meaningless characters

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

Why is Scanning not Part of Parsing?

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?

E.g., why is *ident* considered to be a terminal symbol and not a nonterminal symbol?

Why is Scanning not Part of Parsing?

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

```
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:

```
X = a.
X = b Y.
```

a, b Î TS
X, Y Î NTS

Example Regular grammar for identifiers

```
Ident = letter.
Ident = letter Rest.

Rest  = letter.
Rest  = digit.
Rest  = '_'.
Rest  = letter Rest.
Rest  = digit Rest.
Rest  = '_' Rest.
```

e.g., derivation of the name xy3

Ident Þ letter Rest Þ letter letter Rest Þ letter letter digit

Alternative definition

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

Example Regular grammar for identifiers

```
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 Þ * ... "(" Expr ")" ...
• nested statements	Statement Þ "do" Statement "while" "(" Expr ")"
• nested classes	Class Þ "class" "{" ... Class ... "}"

For productions like these we need context-free grammars

But most lexical structures are regular

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

Exception: nested comments

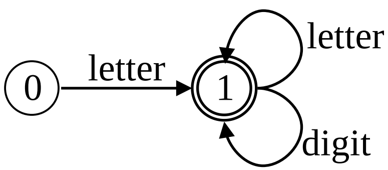
/* ..... /* ... */ ..... */

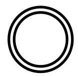
The scanner must treat them in a special way

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

d	letter	digit
s0	s1	error
s1	s1	s1

"finite", because d can be written down explicitly

Definition

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

- S set of states
- I set of input symbols
- d: S x I ® S state transition function
- s0 start state
- F set of final states

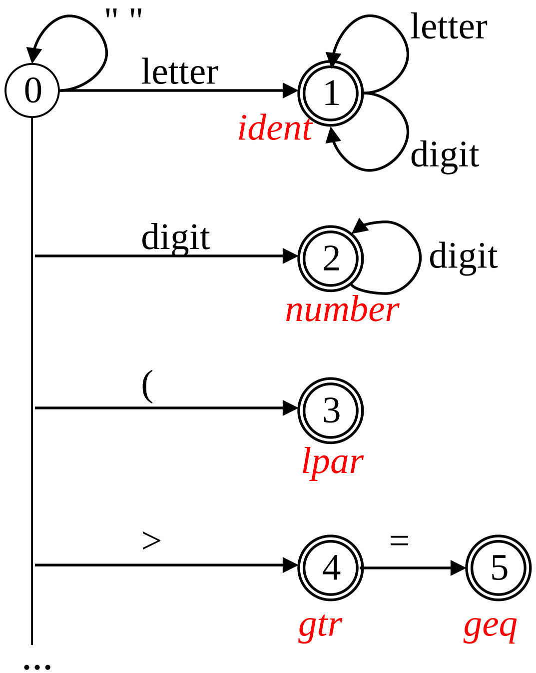
The **language** recognized by a DFA is the set of all symbol sequences that lead from the start state into one of the final states

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`

$s0 \xrightarrow{m} s1 \xrightarrow{a} s1 \xrightarrow{x} s1$

- 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 s0 again

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 (Reader r) {...}
    static Token    next () {...}
}
```

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

Example: Initializing the scanner

```
InputStream s = new FileInputStream("myfile.mj");
Reader r = new InputStreamReader(s);
Scanner.init(r);
```

Example: Reading the token stream

```
for (;;) {
    Token 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 string;   // token string
}
```

Token codes for MicroJava

static final int

error token	token classes	operators and special characters		keywords	end of file
none = 0,	ident = 1,	plus = 4, /* + */	assign = 15, /* = */	class_ = 25,	eof = 36;
	number = 2,	minus = 5, /* - */	semicolon = 16, /* ; */	else_ = 26,	
	charCon = 3,	times = 6, /* * */	comma = 17, /* , */	final_ = 27,	
		slash = 7, /* / */	period = 18, /* . */	if_ = 28,	
		rem = 8, /* % */	lpar = 19, /* (*/	new_ = 29,	
		eql = 9, /* == */	rpar = 20, /*) */	print_ = 30,	
		neq = 10, /* != */	lbrack = 21, /* [*/	program_ = 31,	
		lss = 11, /* < */	rbrack = 22, /*] */	read_ = 32,	
		leq = 12, /* <= */	lbrace = 23, /* { */	return_ = 33,	
		gtr = 13, /* > */	rbrace = 24, /* } */	void_ = 34,	
		geq = 14, /* >= */		while_ = 35,	

Scanner Implementation

Static fields in class *Scanner*

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

static final int eofCh = '\u0080';     // character that is returned at the end of the file
```

init()

```
public static void init (Reader r) {
    in = r;
    line = 1; col = 0;
    nextCh();  // reads the first character into ch and increments col to 1
}
```

nextCh()

```
private static void nextCh() {
    try {
        ch = (char) in.read(); col++;
        if (ch == '\n') { line++; col = 0; }
        else if (ch == '\uffff') ch = eofCh;
    } catch (IOException e) { ch = eofCh; }
}
```

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

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 'b': ... case 'z': case 'A': case 'B': ... case 'Z':
                readName(t); break;
        case '0': case '1': ... case '9':
                readNumber(t); break;
        case ';':   nextCh(); t.kind = semicolon; break;
        case '.':   nextCh(); t.kind = period; break;
        case eofCh: t.kind = eof; break;  // no nextCh() any more
        ...
        case '=':   nextCh();
                if (ch == '=') { nextCh(); t.kind = eql; } else t.kind = assign;
                break;
        ...
        case '/':   nextCh();
                if (ch == '/') {
                    do nextCh(); while (ch != '\n' && ch != eofCh);
                    t = next();  // call scanner recursively
                } else t.kind = slash;
                break;
        default:    nextCh(); t.kind = none; break;
    }
    return t;
} // ch holds the next character that is still unprocessed
```

names, keywords

numbers

simple tokens

compound tokens

comments

invalid character

Further Methods

private static void readName(Token t)

- At the beginning *ch* holds the first letter of the name
- Reads further letters and digits and stores them in *t.string*
- 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 = ident;
- At the end *ch* holds the first character after the name

private static void readNumber(Token t)

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

Further Methods

private static void readCharCon(Token t)

- At the beginning *ch* holds a single quote
- Reads further characters and stores them in *t.string*
- At the end *ch* holds the first character after the closing quote
- Sets the following token fields:
  t.kind = charCon;
  t.val = *numeric char value*;

valid char constants	*invalid char constants*
'x'	'xy'
'\r'	''
'\n'	'x
'\t'	

↑ Scanner reports an error, but returns a *charCon*

What you should do in the lab

1. Study the specification of MicroJava carefully (Appendix A of the handouts).
2. Create a package *MJ*;
   Download *Scanner.java* and *Token.java* from http://ssw.jku.at/Misc/CC/ into this package.
   Try to understand what they do.
3. Complete *Scanner.java* according to the slides of the course;
   Compile *Token.java* and *Scanner.java*.
4. Download *TestScanner.java* into the package *MJ* and compile it.
5. Download the MicroJava source program *sample.mj* and run *TestScanner* on it.
6. Download the MicroJava source program *BuggyScannerInput.mj* and run *TestScanner* on it