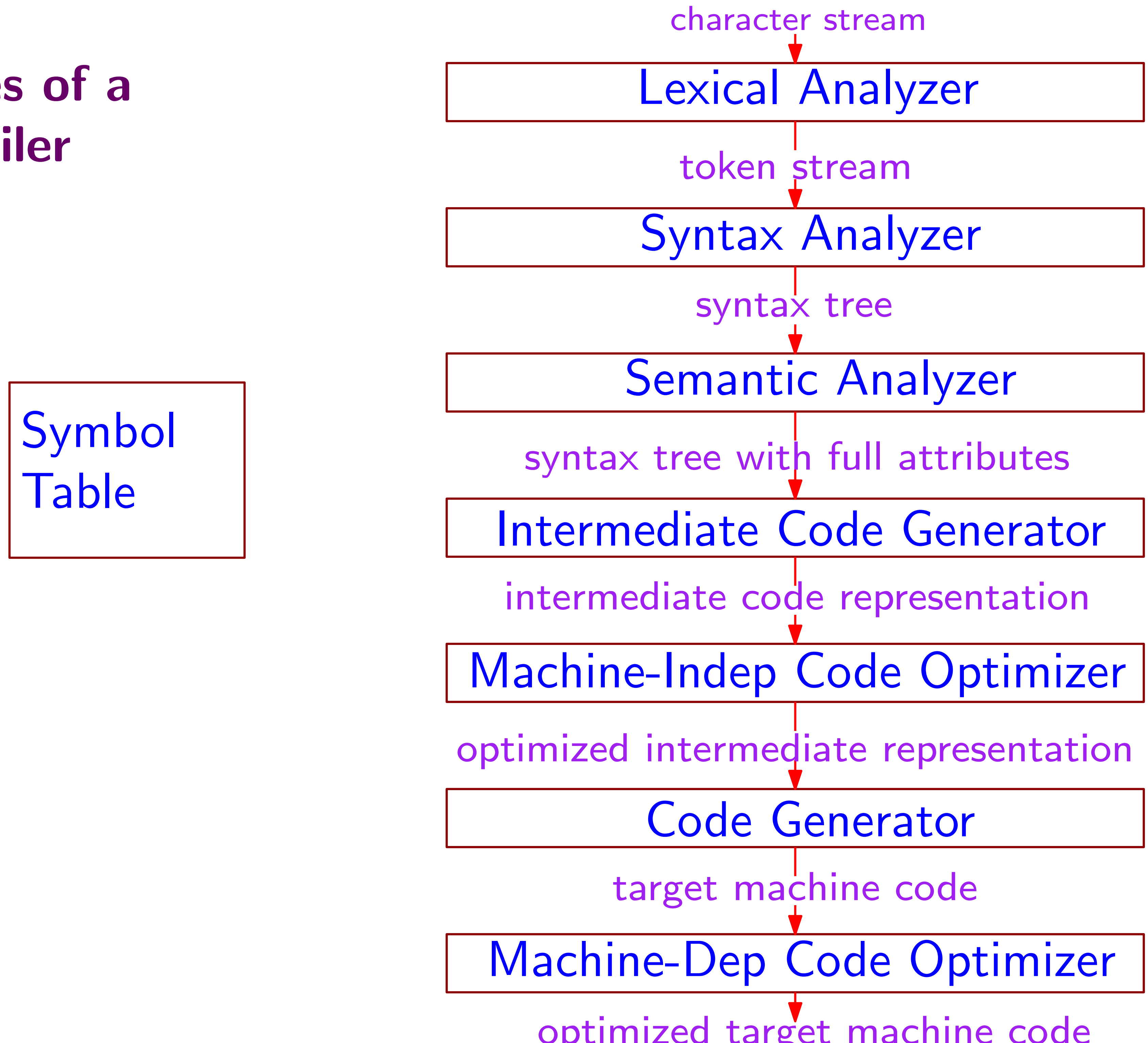
Lexical Analysis

Phases of a Compiler



Example

Symbol Table

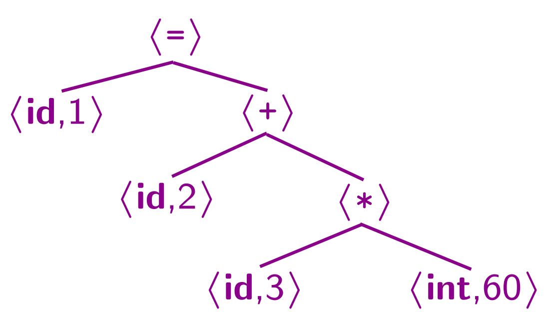
1	position	...
2	initial	...
3	rate	...

```
position = initial + rate*60
```

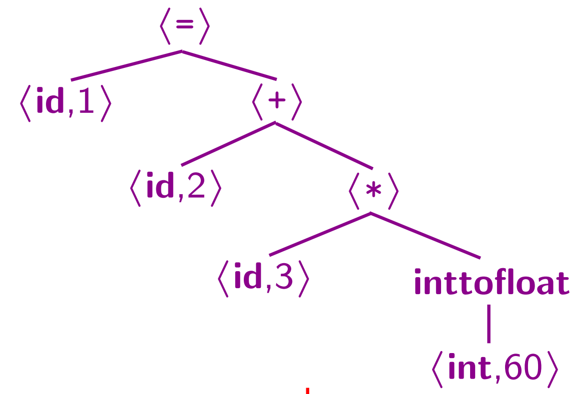
Lexical Analyzer

$\langle \textbf{id},1\rangle\ \langle=\rangle\ \langle \textbf{id},2\rangle\ \langle+\rangle\ \langle \textbf{id},3\rangle\ \langle*\rangle\ \langle \textbf{int},60\rangle$

Syntax Analyzer



Semantic Analyzer



Intermediate Code Generator

```
t1 = inttofloat(60)
t2 = id3*t1
t3 = id2+t2
id1 = t3
```

Machine-Indep Code Optimizer

```
t1 = id3*60.0
id1 = id2+t1
```

Code Generator

```
flds LC0
fmuls id3
fadds id2
fstps id1
...
LC0: .float 60.0
```

Lexical Analysis Terminology

- **Token:** pair $\langle$*token type,optional attribute*$\rangle$
  - *token type:* abstract symbol representing a kind of lexical unit
    - keyword
    - identifier
    - operator
    - constant
  - *attribute:* specific *value* of the token

- **Pattern:** description of the form that the lexemes of a token may take.

- **Lexeme:** sequence of characters in the source program that matches a pattern.

Lexical Tokens

Type	**Examples**
ID	`foo nl4 last`
NUM	`73 0 00 515 082`
REAL	`66.1 .5 10.  1e67 5.5e-10`
IF	`if`
COMMA	`,`
NOTEQ	`!=`
LPAREN	{
RPAREN	}

Token Stream

```
float match0(char *s) /* find a zero */
{if (!strncmp(s, "0.0", 3))
  return 0.;
}
```

Token stream returned by lexer:

```
FLOAT   ID(match0)   LPAREN   CHAR   STAR   ID(s)   RPAREN
LBRACE   IF   LPAREN   BANG   ID(strncmp)   LPAREN   ID(s)
COMMA   STRING(0.0)   COMMA   NUM(3)   RPAREN   RPAREN
RETURN   REAL(0.0)   SEMI   RBRACE   EOF
```

Languages

- **Alphabet:** set of symbols

- **String:** sequence of symbols from an alphabet
  - given two strings s_1 and s_2, we denote by $s_1 s_2$ their *concatenation*
  - the *empty string*, denoted by ϵ is the string with no symbols; for all strings s, $\epsilon s = s\epsilon = s$

- **Language:** set of strings over an alphabet

- **Language Operations:**
  - **Union** : same as set union
  - **Concatenation** : $L_1 L_2 = \{s_1 s_2 \mid s_1 \in L_1,\, s_2 \in L_2\}$
  - **Power** : $L^0 = \{\epsilon\},\ L^{k+1} = L(L^k)$
  - **Kleene closure** : $L^* = \bigcup_{i \geq 0} L^i$ (shortcut: $L^+ = L(L^*)$

Regular Expressions

Regular expressions over an alphabet Σ are language specifications defined by the following rules:

- ◇ Every symbol $a \in \Sigma$ is an RE
- ◇ If r_1 and r_2 are regular expressions, then so are $r_1|r_2$, r_1r_2, and r_1^*
- ◇ We use r^+ as a shortcut for rr^*

Given a regular expression r, the *language $L(r)$ specified by r* is defined by the following rules:

- ◇ $L(a) = \{a\}$, for all $a \in \Sigma$
- ◇ $L(r_1r_2) = L(r_1)L(r_2)$, for all regular expressions r_1 and r_2
- ◇ $L(r_1|r_2) = L(r_1) \cup L(r_2)$for all regular expressions r_1 and r_2
- ◇ $L(r^*) = (L(r))^*$for all regular expressions r

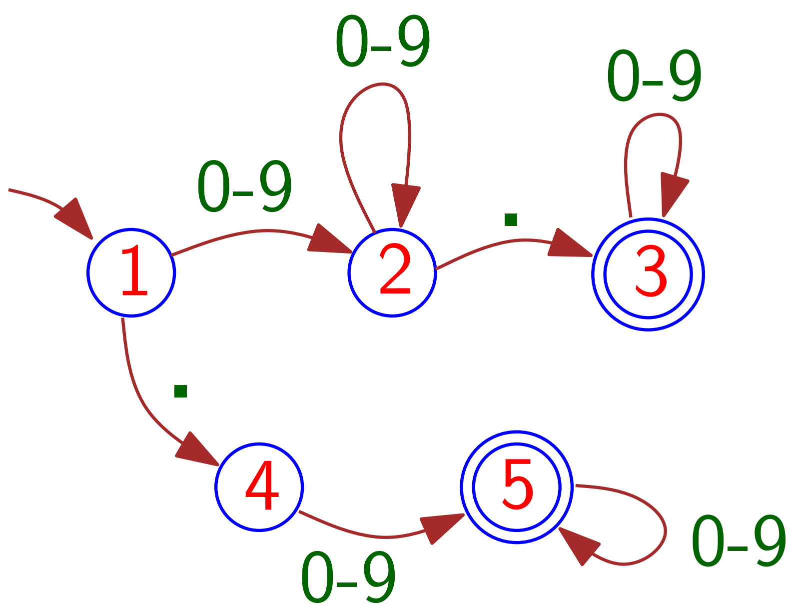
Examples

Using this language, we can specify the lexical tokens of a programming language

`if`	`IF`
`[a-z][a-z0-9]*`	`ID`
`[0-9]+`	`NUM`
`([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)`	`REAL`
`("--"[a-z]*"\n")|(" "|"\n"|"\t")+`	`no token, just white space`
`.`	`error`

Regular expressions are a tool to *generate* a language. We also need a tool to *recognize* a language, that is, detect whether a string belongs to the language of interest or not.

Finite Automata



Scanned lexeme is *accepted* if current state is final, and we cannot transition out of it (current symbol not part of accepted lexeme).

- Alphabet: $\{0, \ldots, 9, \cdot\}$
- Set of states: $\{0, 1, 2, 3, 4, 5\}$
- Start state: 1
- End states: $\{3, 5\}$
- Transition function:

Next state as a function of the current state and current symbol

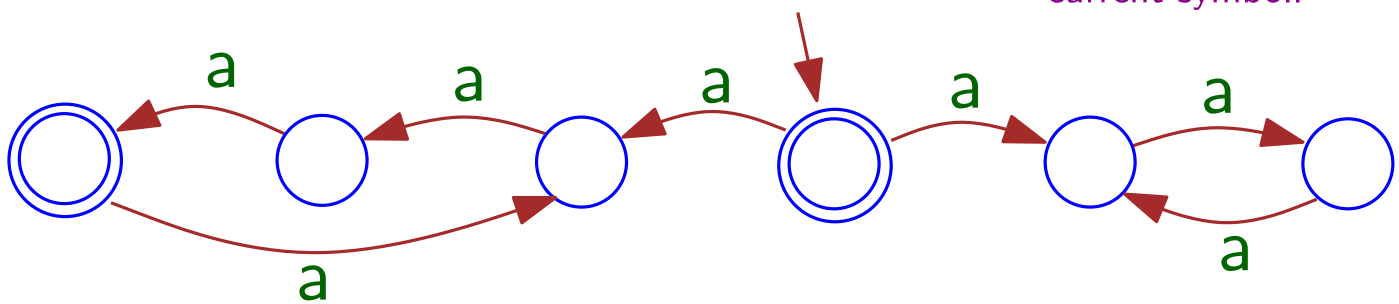
	0-9	•
1	2	4
2	2	3
3	3	
4	5	
5	5	

Types of Finite Automata

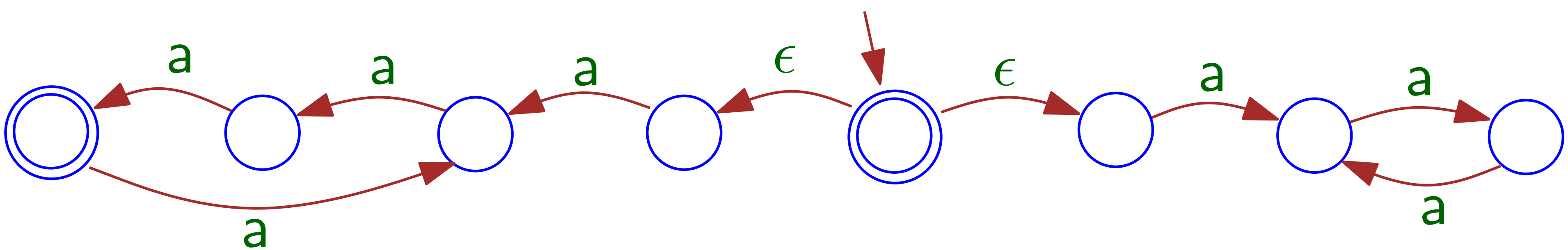
- *Deterministic:* only single out-transitions on the same symbol for any given state.
- *Non-Deterministic:* multiple out-transitions allowed.
- The two types are equivalent.
- Deterministic automata are easy to implement – transition table is a constant array;
  moving to next state can be done with
  `state = transition(state,input[current_pos++]);`
- Non-deterministic automata are easy to construct from regular expressions.
- We need conversion procedure to bridge the gap

NFA Example

Acceptance of scanned lexeme: if there is a chance that, through the non-deterministic transitions, the machine would end up in a final state with no out-transition on the current symbol.



Accepts strings of as whose length is either a multiple of 2 or 3.

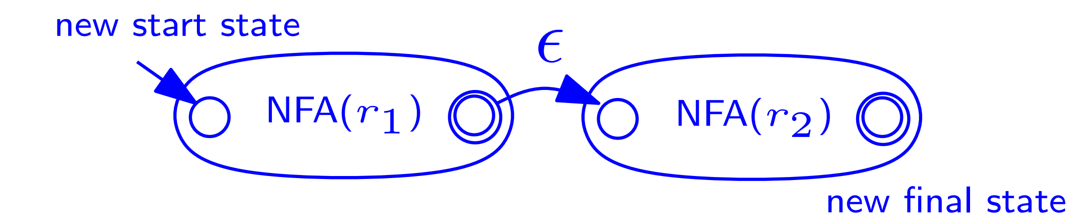


Equivalent NFA

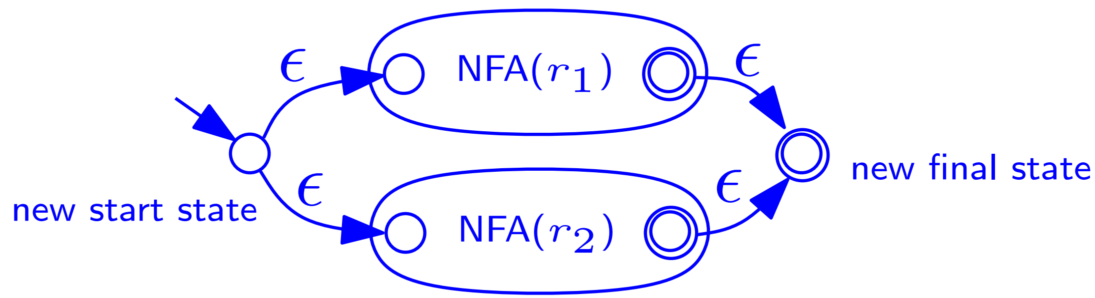
Conversion of RE to NFA

Though NFAs can have multiple final states, this conversion will produce NFAs with single final state.

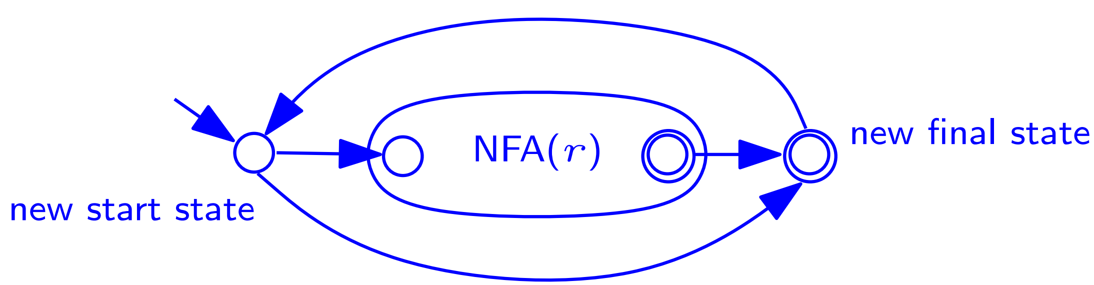
NFA$(r_1 r_2)$



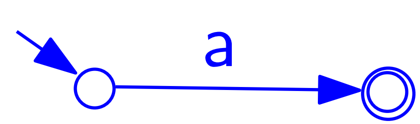
NFA$(r_1 | r_2)$



NFA(r^*)



NFA(a)



Conversion of NFA to DFA

- ◇ Starting from the start state, construct sets of states that are reachable with transitions on a given symbol (epsilon-transitions must be included too)

- ◇ Process must be applied exhaustively

- ◇ The new states of the resulting DFA are the NFA sets of states computed in the process — Each DFA state is *a set of NFA states*.

Example in separate set of slides.

Lex

```
%{
        /* definitions of manifest constants
        LT, LE, EQ, NE, GT, GE,
        IF, THEN, ELSE, ID, NUMBER, RELOP */
%}
/* regular definitions */
delim     [ \t\n]
ws        (delim)+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter} | {digit}) *
number    {digit}+ (\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF) ; }
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID() ; return(ID) ; }
{number}  {yylval = (int) valueOfNumber();  return(NUMBER) ; }
"<"       {yylval = LT ; return(RELOP) ; }
"<="      {yylval = LE ; return(RELOP) ; }
...

%%

int installID() (/* function to install the lexeme, whose
        first character is pointed to by yytext,
        and whose length is yyleng, into the
        symbol table and return a pointer
        thereto */
}
```