Compiler Construction

Lecture 4: Lexical Analysis III (Practical Aspects)

Winter Semester 2018/19

Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ws-1819/cc/

Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

Outline of Lecture 4

Recap: First-Longest-Match Analysis

The Extended Matching Problem

Definition

Let $n \geq 1$ and $\alpha_1, \ldots, \alpha_n \in RE_\Omega$ with $\varepsilon \notin [\![\alpha_i]\!] \neq \emptyset$ for every $i \in [n]$ (where $[n] := \{1, \ldots, n\}$). Let $\Sigma := \{T_1, \ldots, T_n\}$ be an alphabet of corresponding tokens and $w \in \Omega^+$. If $w_1, \ldots, w_k \in \Omega^+$ such that

- $w = w_1 \ldots w_k$ and
- for every $j \in [k]$ there exists $i_j \in [n]$ such that $w_j \in [\![\alpha_{i_j}]\!]$,

then

- $(w_1, \ldots, w_k)$ is called a decomposition and
- $(T_{i_1}, \ldots, T_{i_k})$ is called an analysis

of w w.r.t. $\alpha_1, \ldots, \alpha_n$.

Problem (Extended matching problem)

Given $\alpha_1, \ldots, \alpha_n \in RE_\Omega$ and $w \in \Omega^+$, decide whether there exists a decomposition of w w.r.t. $\alpha_1, \ldots, \alpha_n$ and determine a corresponding analysis.

Recap: First-Longest-Match Analysis

Ensuring Uniqueness

Two principles

1. Principle of the longest match (“maximal munch tokenisation”)
   - for uniqueness of decomposition
   - make lexemes as long as possible
   - motivated by practical considerations:
     - every proper prefix of an identifier is also an identifier
     - real-valued constants contain integer constants as prefixes
2. Principle of the first match
   - for uniqueness of analysis
   - choose first matching regular expression (in the given order)
   - therefore: arrange keywords before identifiers (if keywords protected)

Implementation of FLM Analysis

Algorithm (FLM analysis – overview)

Input: expressions $\alpha_1, \ldots, \alpha_n \in RE_\Omega$, tokens $\{T_1, \ldots, T_n\}$, input word $w \in \Omega^+$

Procedure:

1. for every $i \in [n]$, construct $\mathfrak{A}_i \in DFA_\Omega$ such that $L(\mathfrak{A}_i) = [\![\alpha_i]\!]$
   (see DFA method; Algorithm 2.9)
2. construct the product automaton $\mathfrak{A} \in DFA_\Omega$ such that $L(\mathfrak{A}) = \bigcup_{i=1}^{n} [\![\alpha_i]\!]$
3. partition the set of final states of $\mathfrak{A}$ to follow the first-match principle
4. extend the resulting DFA to a backtracking DFA which implements the longest-match principle
5. let the backtracking DFA run on w

Output: FLM analysis of w (if existing)

Recap: First-Longest-Match Analysis

(4) The Backtracking DFA

Definition (Backtracking DFA)

- The set of configurations of $\mathfrak{B}$ is given by
$$(\{N\} \uplus \Sigma) \times \Omega^* \cdot Q \cdot \Omega^* \times \Sigma^* \cdot \{\varepsilon, \text{lexerr}\}$$
- The initial configuration for an input word $w \in \Omega^+$ is $(N, q_0 w, \varepsilon)$.
- The transitions of $\mathfrak{B}$ are defined as follows (where $q' := \delta(q, a)$):
  - normal mode: look for initial match
$$(N, qaw, W) \vdash \begin{cases} (T_i, q'w, W) & \text{if } q' \in F^{(i)} \quad (1) \\ (N, q'w, W) & \text{if } q' \in P \setminus F \quad (2) \\ \textbf{output: } W \cdot \text{lexerr} & \text{if } q' \notin P \quad (3) \end{cases}$$
  - backtrack mode: look for longest match
$$(T, vqaw, W) \vdash \begin{cases} (T_i, q'w, W) & \text{if } q' \in F^{(i)} \quad (4) \\ (T, vaq'w, W) & \text{if } q' \in P \setminus F \quad (5) \\ (N, q_0 vaw, WT) & \text{if } q' \notin P \quad (6) \end{cases}$$
  - end of input:
$$\begin{aligned} (T, q, W) &\vdash \textbf{output: } WT && \text{if } q \in F && (7) \\ (N, q, W) &\vdash \textbf{output: } W \cdot \text{lexerr} && \text{if } q \in P \setminus F && (8) \\ (T, vaq, W) &\vdash (N, q_0 va, WT) && \text{if } q \in P \setminus F && (9) \end{aligned}$$

Outline of Lecture 4

Time Complexity of FLM Analysis

Lemma 4.1

The worst-case time complexity of FLM analysis using the backtracking DFA on input $w \in \Omega^+$ is $\mathcal{O}(|w|^2)$.

Time Complexity of FLM Analysis

Lemma 4.1

The worst-case time complexity of FLM analysis using the backtracking DFA on input $w \in \Omega^+$ is $\mathcal{O}(|w|^2)$.

Proof.

- lower bound: $\alpha_1 = a$, $\alpha_2 = a^*b$, $w = a^m$ requires $\mathcal{O}(m^2)$

Time Complexity of FLM Analysis

Lemma 4.1

The worst-case time complexity of FLM analysis using the backtracking DFA on input $w \in \Omega^+$ is $\mathcal{O}(|w|^2)$.

Proof.

- lower bound: $\alpha_1 = a$, $\alpha_2 = a^*b$, $w = a^m$ requires $\mathcal{O}(m^2)$
- upper bound:
  - each run from mode N to $T \in \Sigma$ consumes at least one input symbol (and possibly inspects all input symbols), involving at most $\sum_{i=1}^{|w|} i = \frac{n(n+1)}{2}$ transitions
  - if no Σ-mode is reached, lexerr is reported after $\leq |w|$ transitions

□

Time Complexity of FLM Analysis

Lemma 4.1

The worst-case time complexity of FLM analysis using the backtracking DFA on input $w \in \Omega^+$ *is* $\mathcal{O}(|w|^2)$.

Proof.

- lower bound: $\alpha_1 = a$, $\alpha_2 = a^*b$, $w = a^m$ requires $\mathcal{O}(m^2)$
- upper bound:
  - each run from mode N to $T \in \Sigma$ consumes at least one input symbol (and possibly inspects all input symbols), involving at most $\sum_{i=1}^{|w|} i = \frac{n(n+1)}{2}$ transitions
  - if no Σ-mode is reached, lexerr is reported after $\leq |w|$ transitions

□

Remark: possible improvement by tabular method (similar to Knuth-Morris-Pratt Algorithm for pattern matching in strings)

Literature: Th. Reps: *"Maximal-Munch" Tokenization in Linear Time*, ACM TOPLAS 20(2), 1998, 259–273

Outline of Lecture 4

A Backtracking NFA

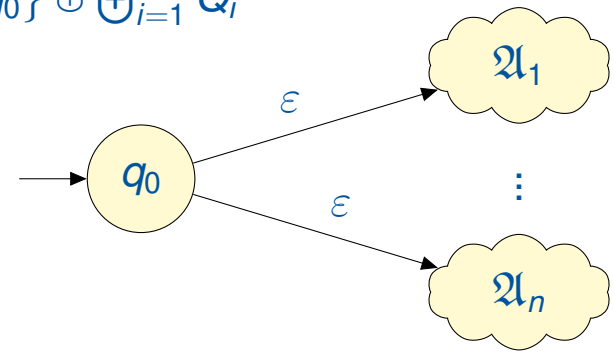
A similar construction is possible for the NFA method:

1. For every $i \in [n]$: $\mathfrak{A}_i = \langle Q_i, \Omega, \delta_i, q_0^{(i)}, F_i \rangle \in \mathit{NFA}_\Omega$ with $\delta : Q \times \Omega_\varepsilon \to 2^Q$ and $L(\mathfrak{A}_i) = [\![\alpha_i]\!]$ by NFA method

A Backtracking NFA

A similar construction is possible for the NFA method:

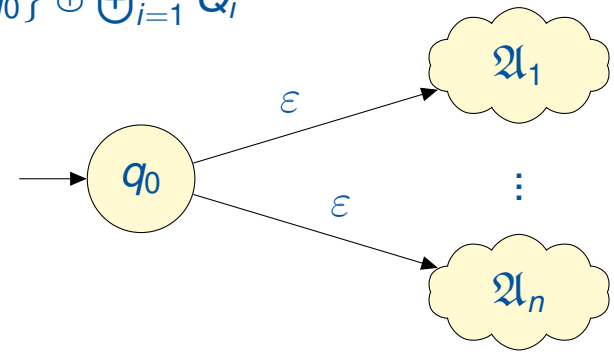
1. For every $i \in [n]$: $\mathfrak{A}_i = \langle Q_i, \Omega, \delta_i, q_0^{(i)}, F_i \rangle \in NFA_\Omega$ with $\delta : Q \times \Omega_\varepsilon \to 2^Q$ and $L(\mathfrak{A}_i) = [\![\alpha_i]\!]$ by NFA method
2. "Product" automaton: $Q := \{q_0\} \uplus \biguplus_{i=1}^{n} Q_i$



A Backtracking NFA

A similar construction is possible for the NFA method:

1. For every $i \in [n]$: $\mathfrak{A}_i = \langle Q_i, \Omega, \delta_i, q_0^{(i)}, F_i \rangle \in NFA_\Omega$ with $\delta : Q \times \Omega_\varepsilon \to 2^Q$ and $L(\mathfrak{A}_i) = [\![\alpha_i]\!]$ by NFA method
2. "Product" automaton: $Q := \{q_0\} \uplus \biguplus_{i=1}^{n} Q_i$

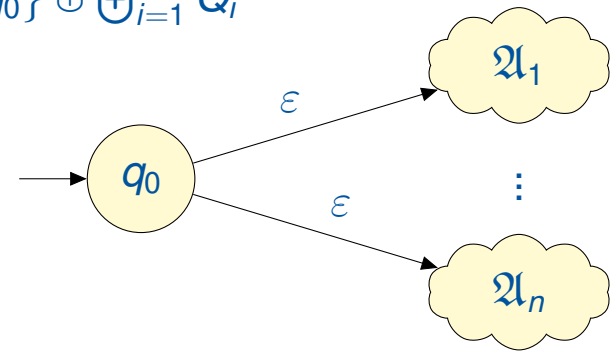


3. Partitioning of final states:
   - $M \subseteq Q$ is called a T_i-matching if
   $$M \cap F_i \neq \emptyset \text{ and for all } j \in [i-1] : M \cap F_j = \emptyset$$
   - yields set of T_i-matchings $F^{(i)} \subseteq 2^Q$
   - $M \subseteq Q$ is called productive if there exists a productive $q \in M$
   - yields productive state sets $P \subseteq 2^Q$

A Backtracking NFA

A similar construction is possible for the NFA method:

1. For every $i \in [n]$: $\mathfrak{A}_i = \langle Q_i, \Omega, \delta_i, q_0^{(i)}, F_i \rangle \in NFA_\Omega$ with $\delta : Q \times \Omega_\varepsilon \to 2^Q$ and $L(\mathfrak{A}_i) = [\![\alpha_i]\!]$ by NFA method
2. "Product" automaton: $Q := \{q_0\} \uplus \biguplus_{i=1}^{n} Q_i$



3. Partitioning of final states:
   - $M \subseteq Q$ is called a T_i-matching if
   $$M \cap F_i \neq \emptyset \text{ and for all } j \in [i-1] : M \cap F_j = \emptyset$$
   - yields set of T_i-matchings $F^{(i)} \subseteq 2^Q$
   - $M \subseteq Q$ is called productive if there exists a productive $q \in M$
   - yields productive state sets $P \subseteq 2^Q$
4. Backtracking automaton: similar to DFA case

Outline of Lecture 4

Longest Match in Practice

- In general: lookahead of arbitrary length required
  - that is, $|v|$ unbounded in configurations (T, vqw, W)
  - see Lemma 4.1: $\alpha_1 = a$, $\alpha_2 = a^* b$, $w \in [\![a^+]\!]$

Longest Match in Practice

- In general: lookahead of arbitrary length required
  - that is, $|v|$ unbounded in configurations (T, vqw, W)
  - see Lemma 4.1: $\alpha_1 = a$, $\alpha_2 = a^*b$, $w \in [\![a^+]\!]$
- "Modern" programming languages (Pascal, Java, ...): lookahead of one or two characters sufficient
  - separation of keywords, identifiers, etc. by spaces
  - Pascal: two-character lookahead required to distinguish `1.5` (real number) from `1..5` (integer range)

Longest Match in Practice

- In general: lookahead of arbitrary length required
  - that is, $|v|$ unbounded in configurations (T, vqw, W)
  - see Lemma 4.1: $\alpha_1 = a$, $\alpha_2 = a^*b$, $w \in [\![a^+]\!]$
- "Modern" programming languages (Pascal, Java, ...):
  lookahead of one or two characters sufficient
  - separation of keywords, identifiers, etc. by spaces
  - Pascal: two-character lookahead required to distinguish `1.5` (real number) from `1..5` (integer range)

However: principle of longest match not always applicable!

Inadequacy of Longest Match I

Example 4.2 (Longest match in FORTRAN)

1. Relational expressions
   - valid lexemes: `.EQ.` (relational operator), `EQ` (identifier), `12` (integer), `12.`, `.12` (reals)
   - input string: `12␣.EQ.␣12` $\rightsquigarrow$ `12.EQ.12` (ignoring blanks!)

Inadequacy of Longest Match I

Example 4.2 (Longest match in FORTRAN)

1. Relational expressions
   - valid lexemes: `.EQ.` (relational operator), `EQ` (identifier), `12` (integer), `12.`, `.12` (reals)
   - input string: `12␣.EQ.␣12` $\rightsquigarrow$ `12.EQ.12` (ignoring blanks!)
   - intended analysis: (int, 12)(relop, eq)(int, 12)

Inadequacy of Longest Match I

Example 4.2 (Longest match in FORTRAN)

1. Relational expressions
   - valid lexemes: `.EQ.` (relational operator), `EQ` (identifier), `12` (integer), `12.`, `.12` (reals)
   - input string: `12␣.EQ.␣12` $\rightsquigarrow$ `12.EQ.12` (ignoring blanks!)
   - intended analysis: (int, 12)(relop, eq), (int, 12)
   - LM yields: (real, 12.0)(id, EQ)(real, 0.12)

   $\Rightarrow$ wrong interpretation

Inadequacy of Longest Match I

Example 4.2 (Longest match in FORTRAN)

1. Relational expressions
   - valid lexemes: `.EQ.` (relational operator), `EQ` (identifier), `12` (integer), `12.`, `.12` (reals)
   - input string: `12␣.EQ.␣12` $\rightsquigarrow$ `12.EQ.12` (ignoring blanks!)
   - intended analysis: (int, 12)(relop, eq), (int, 12)
   - LM yields: (real, 12.0)(id, EQ)(real, 0.12)

   $\Rightarrow$ wrong interpretation
2. `DO` loops
   - (correct) input string: `DO␣5␣I␣=␣1,␣20` $\rightsquigarrow$ `DO5I=1,20`

Inadequacy of Longest Match I

Example 4.2 (Longest match in FORTRAN)

1. Relational expressions
   - valid lexemes: `.EQ.` (relational operator), `EQ` (identifier), `12` (integer), `12.`, `.12` (reals)
   - input string: `12␣.EQ.␣12` $\rightsquigarrow$ `12.EQ.12` (ignoring blanks!)
   - intended analysis: (int, 12)(relop, eq), (int, 12)
   - LM yields: (real, 12.0)(id, EQ)(real, 0.12)

   $\Rightarrow$ wrong interpretation
2. `DO` loops
   - (correct) input string: `DO␣5␣I␣=␣1,␣20` $\rightsquigarrow$ `DO5I=1,20`
     - intended analysis: (do,)(label, 5)(id, I)(gets,)(int, 1)(comma,)(int, 20)

Inadequacy of Longest Match I

Example 4.2 (Longest match in FORTRAN)

1. Relational expressions
   - valid lexemes: `.EQ.` (relational operator), `EQ` (identifier), `12` (integer), `12.`, `.12` (reals)
   - input string: `12␣.EQ.␣12` $\rightsquigarrow$ `12.EQ.12` (ignoring blanks!)
   - intended analysis: (int, 12)(relop, eq), (int, 12)
   - LM yields: (real, 12.0)(id, EQ)(real, 0.12)

   $\Rightarrow$ wrong interpretation
2. `DO` loops
   - (correct) input string: `DO␣5␣I␣=␣1,␣20` $\rightsquigarrow$ `DO5I=1,20`
     - intended analysis: (do,)(label, 5)(id, I)(gets,)(int, 1)(comma,)(int, 20)
     - LM analysis (wrong): (id, DO5I)(gets,)(int, 1)(comma,)(int, 20)

Longest Match in Practice

Inadequacy of Longest Match I

Example 4.2 (Longest match in FORTRAN)

1. Relational expressions
   - valid lexemes: `.EQ.` (relational operator), `EQ` (identifier), `12` (integer), `12.`, `.12` (reals)
   - input string: `12␣.EQ.␣12` $\rightsquigarrow$ `12.EQ.12` (ignoring blanks!)
   - intended analysis: (int, 12)(relop, eq), (int, 12)
   - LM yields: (real, 12.0)(id, EQ)(real, 0.12)

   $\Rightarrow$ wrong interpretation
2. `DO` loops
   - (correct) input string: `DO␣5␣I␣=␣1,␣20` $\rightsquigarrow$ `DO5I=1,20`
     - intended analysis: (do,)(label, 5)(id, I)(gets,)(int, 1)(comma,)(int, 20)
     - LM analysis (wrong): (id, DO5I)(gets,)(int, 1)(comma,)(int, 20)
   - (erroneous) input string: `DO␣5␣I␣=␣1.␣20` $\rightsquigarrow$ `DO5I=1.20`
     - LM analysis (correct): (id, DO5I)(gets,)(real, 1.2)

Inadequacy of Longest Match II

Example 4.3 (Longest match in C)

```
int i, j, k;
int *ref;
int **refref;
...
i = **refref+1;
j = 2***refref;
k = 3**ref;
```

Inadequacy of Longest Match II

Example 4.3 (Longest match in C)

```
int i, j, k;
int *ref;
int **refref;
...
i = **refref+1;
j = 2***refref;
k = 3**ref;
```

Possible solutions:

- Hand-written (non-FLM) scanners
- FLM with lookahead (later)

Outline of Lecture 4

Regular Definitions I

Goal: modularising the representation of regular sets by introducing additional identifiers

Regular Definitions I

Goal: modularising the representation of regular sets by introducing additional identifiers

Definition 4.4 (Regular definition)

Let $\{R_1, \ldots, R_n\}$ be a set of symbols disjoint from Ω. A regular definition (over Ω) is a sequence of equations

$$\begin{aligned} R_1 &= \alpha_1 \\ &\vdots \\ R_n &= \alpha_n \end{aligned}$$

such that, for every $i \in [n]$, $\alpha_i \in RE_{\Omega \uplus \{R_1, \ldots, R_{i-1}\}}$.

Regular Definitions I

Goal: modularising the representation of regular sets by introducing additional identifiers

Definition 4.4 (Regular definition)

Let $\{R_1, \ldots, R_n\}$ be a set of symbols disjoint from Ω. A regular definition (over Ω) is a sequence of equations

$$\begin{aligned} R_1 &= \alpha_1 \\ &\vdots \\ R_n &= \alpha_n \end{aligned}$$

such that, for every $i \in [n]$, $\alpha_i \in RE_{\Omega \uplus \{R_1, \ldots, R_{i-1}\}}$.

Remark: since recursion is excluded, every R_i can (iteratively) be substituted by a regular expression $\alpha \in RE_\Omega$ (otherwise $\Longrightarrow$ context-free languages)

Regular Definitions II

Example 4.5 (Symbol classes in Pascal)

Identifiers:

$$\begin{aligned} \mathit{Letter} &= \texttt{A} \mid \ldots \mid \texttt{Z} \mid \texttt{a} \mid \ldots \mid \texttt{z} \\ \mathit{Digit} &= \texttt{0} \mid \ldots \mid \texttt{9} \\ \mathit{Id} &= \mathit{Letter}\,(\mathit{Letter} \mid \mathit{Digit})^* \end{aligned}$$

Numerals (unsigned):

$$\begin{aligned} \mathit{Digits} &= \mathit{Digit}^+ \\ \mathit{Empty} &= \emptyset^* \\ \mathit{OptFrac} &= \texttt{.}\,\mathit{Digits} \mid \mathit{Empty} \\ \mathit{OptExp} &= \texttt{E}\,(\texttt{+} \mid \texttt{-} \mid \mathit{Empty})\,\mathit{Digits} \mid \mathit{Empty} \\ \mathit{Num} &= \mathit{Digits}\ \mathit{OptFrac}\ \mathit{OptExp} \end{aligned}$$

Rel. operators:

$$\mathit{RelOp} = \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{<>} \mid \texttt{>} \mid \texttt{>=}$$

Keywords:

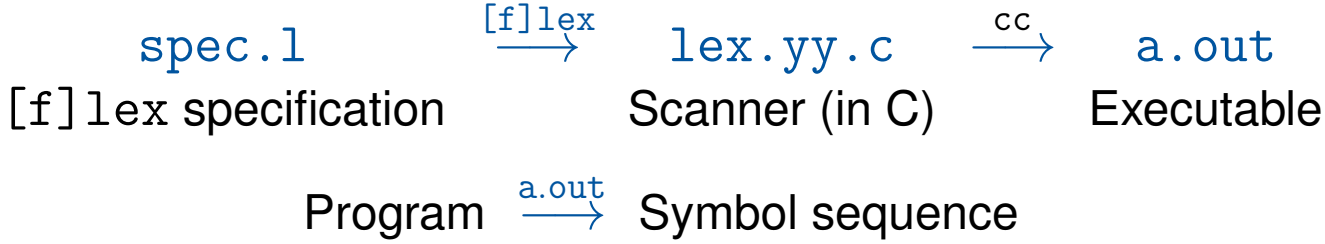
$$\begin{aligned} \mathit{If} &= \texttt{if} \\ \mathit{Then} &= \texttt{then} \\ \mathit{Else} &= \texttt{else} \end{aligned}$$

Outline of Lecture 4

Generating Scanners Using `[f]lex`

The `[f]lex` Tool

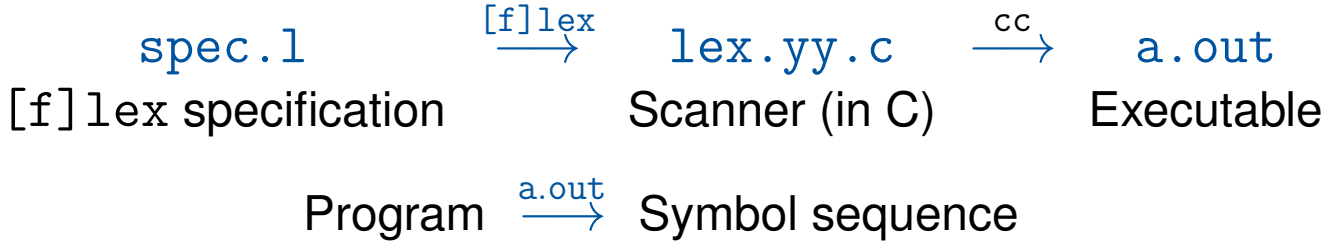
Usage of `[f]lex` (“[fast] lexical analyzer generator”):



Generating Scanners Using [f]lex

The [f]lex Tool

Usage of [f]lex ("[fast] lexical analyzer generator"):



A [f]lex specification is of the form

Definitions (optional)
%%
Rules
%%
Auxiliary procedures (optional)

[f]lex Specifications

Definitions:
- C code for declarations etc.: %{ *Code* %}
- Regular definitions: *Name RegExp* ... (non-recursive!)

[f]lex Specifications

Definitions:
- C code for declarations etc.: %{ *Code* %}
- Regular definitions: *Name RegExp* ... (non-recursive!)

Rules: of the form *Pattern* { *Action* }
- *Pattern*: regular expression, possibly using *Name*s
- *Action*: C code for computing symbol = (token, attribute)
  - token: integer `return` value, 0 = `EOF`
  - attribute: passed in global variable `yylval`
  - lexeme: accessible by `yytext`
- matching rule found by FLM strategy
- lexical errors catched by `.` (any character)

Example [f]lex Specification

```
%{#include <stdio.h>
  typedef enum {EOF, IF, ID, RELOP, LT, ...} token_t;
  unsigned int yylval;   /* attribute values */
%}
LETTER     [A-Za-z]
DIGIT      [0-9]
ALPHANUM   {LETTER}|{DIGIT}
SPACE      [ \t\n]
%%
"if"                 { return IF; }
"<"                  { yylval = LT; return RELOP; }
{LETTER}{ALPHANUM}*  { yylval = install_id(); return ID; }
{SPACE}+             /* eat up whitespace */
.                    { fprintf (stderr, "Invalid character '%c'\n", yytext[0]); }
%%
int main(void) {
  token_t token;
  while ((token = yylex()) != EOF)
    printf ("(Token %d, Attribute %d)\n", token, yylval);
  exit (0);
}
unsigned int install_id () {...}   /* identifier name passed through yytext */
```

Regular Expressions in [f]lex

Syntax	Meaning
printable character	this character
\n, \t, \123, etc.	newline, tab, octal representation, etc.
.	any character except \n
[*Chars*]	one of *Chars*; ranges possible ("0-9")
[^*Chars*]	none of *Chars*
\\, \., \[, etc.	\, ., [, etc.
"*Text*"	*Text* without interpretation of ., [, \, etc.
^α	α at beginning of line
$\alpha$$	α at end of line
{*Name*}	*RegExp* for *Name*
α?	zero or one α
α*	zero or more α
α+	one or more α
α{*n*, *m*}	between *n* and *m* times α (", *m*" optional)
(α)	α
$\alpha_1\alpha_2$	concatenation
α_1 | α_2	alternative
α_1/α_2	α_1 but only if followed by α_2 (lookahead)

Using the Lookahead Operator

Example 4.6 (Lookahead in FORTRAN)

1. DO loops (cf. Example 4.2)
   - input string: `DO 5 I = 1, 20`
   - LM yields: $(\text{id}, \texttt{DO5I})(\text{gets},)(\text{int}, 1)(\text{comma},)(\text{int}, 20)$
   - observation: decision for do token only possible after reading "`,`"
   - specification of `DO` keyword in `[f]lex`, using lookahead:

```
DO / ({LETTER}|{DIGIT})+ = ({LETTER}|{DIGIT})+ ,
```

Using the Lookahead Operator

Example 4.6 (Lookahead in FORTRAN)

1. DO loops (cf. Example 4.2)
   - input string: `DO 5 I = 1, 20`
   - LM yields: $(\text{id}, \texttt{DO5I})(\text{gets},)(\text{int}, 1)(\text{comma},)(\text{int}, 20)$
   - observation: decision for do token only possible after reading "`,`"
   - specification of `DO` keyword in `[f]lex`, using lookahead:

```
DO / ({LETTER}|{DIGIT})+ = ({LETTER}|{DIGIT})+ ,
```

2. IF statement
   - problem: FORTRAN keywords not reserved
   - example: `IF(I,J) = 3` (assignment to an element of matrix `IF`)
   - conditional: `IF (condition) THEN ...` (with keyword `IF`)
   - specification of `IF` keyword in `[f]lex`, using lookahead:

```
IF / \( .+ \) THEN
```

Generating Scanners Using [f]lex

Longest Match and Lookahead in [f]lex

```
%{#include <stdio.h>
  typedef enum {EOF, AB, A} token_t;
%}
%%
ab      { return AB; }
a/bc    { return A; }
.       { fprintf (stderr, "Invalid character '%c'\n", yytext[0]); }
%%
int main(void) {
  token_t token;
  while ((token = yylex()) != EOF) printf ("Token %d\n", token);
  exit (0);
}
```

Longest Match and Lookahead in [f]lex

```
%{#include <stdio.h>
  typedef enum {EOF, AB, A} token_t;
%}
%%
ab      { return AB; }
a/bc    { return A; }
.       { fprintf (stderr, "Invalid character '%c'\n", yytext[0]); }
%%
int main(void) {
  token_t token;
  while ((token = yylex()) != EOF) printf ("Token %d\n", token);
  exit (0);
}
```

returns on input

- a: `Invalid character 'a'`
- ab: `Token 1`
- abc: `Token 2  Invalid character 'b'  Invalid character 'c'`

$\Rightarrow$ lookahead counts for length of match

Outline of Lecture 4

Preprocessing

Preprocessing = preparation of source code before (lexical) analysis

Preprocessing steps

- macro substitution

```
#define is_capital(ch) ((ch) >= 'A' && (ch) <= 'Z')
```

- file inclusion

```
#include "header.h"
```

- conditional compilation

```
#ifdef UNIX
char* separator = '/'
#endif
#ifdef WINDOWS
char* separator = '\\'
#endif
```

- elimination of comments (or by scanner)