



Compiler Construction

Lecture 4: Lexical Analysis III (Practical Aspects)

Summer Semester 2017

Thomas Noll

Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ss-17/cc/>

Recap: First-Longest-Match Analysis

The Extended Matching Problem I

Definition

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

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

then

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

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

Problem (Extended matching problem)

Given $\alpha_1, \dots, \alpha_n \in RE_\Omega$ and $w \in \Omega^+$, decide whether there exists a decomposition of w w.r.t. $\alpha_1, \dots, \alpha_n$ 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)

Recap: First-Longest-Match Analysis

Implementation of FLM Analysis

Algorithm (FLM analysis – overview)

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

Procedure: 1. for every $i \in [n]$, construct $\mathfrak{A}_i \in DFA_\Omega$ such that $L(\mathfrak{A}_i) = \llbracket \alpha_i \rrbracket$

(see *DFA method*; Algorithm 2.9)

2. construct the *product automaton* $\mathfrak{A} \in DFA_\Omega$ such that $L(\mathfrak{A}) = \bigcup_{i=1}^n \llbracket \alpha_i \rrbracket$

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_0w, ε) .
- The **transitions** of \mathfrak{B} are defined as follows (where $q' := \delta(q, a)$):

- normal mode: look for initial match

$$(N, qaw, W) \vdash \begin{cases} (T_i, q'w, W) & \text{if } q' \in F^{(i)} & (1) \\ (N, q'w, W) & \text{if } q' \in P \setminus F & (2) \\ \mathbf{output: } W \cdot \text{lexerr} & \text{if } q' \notin P & (3) \end{cases}$$

- backtrack mode: look for longest match

$$(T, vqaw, W) \vdash \begin{cases} (T_i, q'w, W) & \text{if } q' \in F^{(i)} & (4) \\ (T, vaq'w, W) & \text{if } q' \in P \setminus F & (5) \\ (N, q_0vaw, WT) & \text{if } q' \notin P & (6) \end{cases}$$

- end of input:

$$\begin{aligned} (T, q, W) &\vdash \mathbf{output: } WT && \text{if } q \in F && (7) \\ (N, q, W) &\vdash \mathbf{output: } W \cdot \text{lexerr} && \text{if } q \in P \setminus F && (8) \\ (T, vaq, W) &\vdash (N, q_0va, WT) && \text{if } q \in P \setminus F && (9) \end{aligned}$$

Time Complexity of First-Longest-Match Analysis

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)

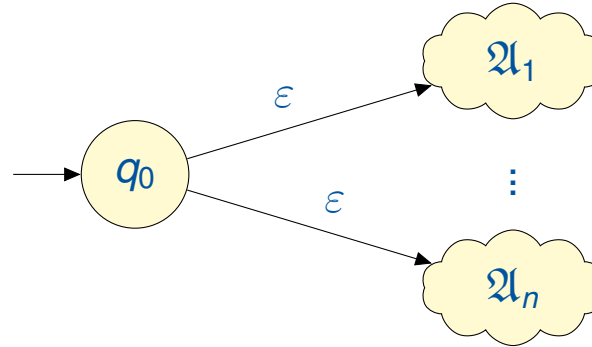
Literature: Th. Reps: “Maximal-Munch” Tokenization in Linear Time, ACM TOPLAS 20(2), 1998, 259–273

First-Longest-Match Analysis with NFA

A Backtracking NFA

A similar construction is possible for the **NFA method**:

1. For every $i \in [n]$: $\mathcal{A}_i = \langle Q_i, \Omega, \delta_i, q_0^{(i)}, F_i \rangle \in NFA_\Omega$ with $L(\mathcal{A}_i) = \llbracket \alpha_i \rrbracket$ 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

Longest Match in Practice

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!

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: `DO5I=1,20` \rightsquigarrow `D05I=1,20`
 - intended analysis: `(do,)(label, 5)(id, I)(gets,)(int, 1)(comma,)(int, 20)`
 - LM analysis (wrong): `(id, D05I)(gets,)(int, 1)(comma,)(int, 20)`
- (erroneous) input string: `DO5I=1.20` \rightsquigarrow `D05I=1.20`
 - LM analysis (correct): `(id, D05I)(gets,)(real, 1.2)`

Longest Match in Practice

Inadequacy of Longest Match II

Example 4.3 (Longest match in C)

- valid lexemes:
 - `x` (identifier)
 - `=` (assignment)
 - `--` (subtractive assignment; K&R/ANSI-C: `--`)
 - `1`, `-1` (integers)
 - input string: `x=-1`
 - intended analysis: `(id, x)(gets,)(int, -1)`
 - LM yields: `(id, x)(dec,)(int, 1)`
- ⇒ wrong interpretation

Possible solutions:

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

Regular Definitions

Regular Definitions I

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

Definition 4.4 (Regular definition)

Let $\{R_1, \dots, R_n\}$ 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 $i \in [n]$, $\alpha_i \in RE_{\Omega \uplus \{R_1, \dots, R_{i-1}\}}$.

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

Regular Definitions

Regular Definitions II

Example 4.5 (Symbol classes in Pascal)

Identifiers: $Letter = A \mid \dots \mid Z \mid a \mid \dots \mid z$
 $Digit = 0 \mid \dots \mid 9$
 $Id = Letter (Letter \mid Digit)^*$

Numerals:
(unsigned) $Digits = Digit^+$
 $Empty = \emptyset^*$
 $OptFrac = . Digits \mid Empty$
 $OptExp = E (+ \mid - \mid Empty) Digits \mid Empty$
 $Num = Digits OptFrac OptExp$

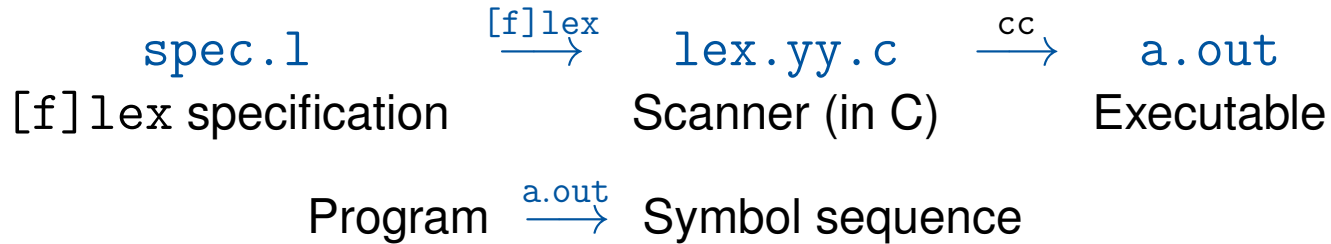
Rel. operators: $RelOp = < \mid <= \mid = \mid <> \mid > \mid >=$

Keywords: $If = if$
 $Then = then$
 $Else = else$

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)

Generating Scanners Using `[f]lex`

`[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 *Names*
- **Action:** C code for computing `symbol = (token, attribute)`
 - **token:** integer `return` value, `0 = EOF`
 - **attribute:** passed in global variable `yylval`
 - **lexeme:** accessible by `ytext`
- matching rule found by **FLM strategy**
- **lexical errors** caught by `.` (any character)

Generating Scanners Using [f]lex

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 */
```

Generating Scanners Using `[f]lex`

Regular Expressions in `[f]lex`

Syntax	Meaning
printable character	this character
<code>\n, \t, \123</code> , etc.	newline, tab, octal representation, etc.
<code>.</code>	any character except <code>\n</code>
<code>[Chars]</code>	one of <i>Chars</i> ; ranges possible (“0–9”)
<code>[^Chars]</code>	none of <i>Chars</i>
<code>\\, \., \[,</code> etc.	<code>\, ., [,</code> etc.
<code>"Text"</code>	<i>Text</i> without interpretation of <code>.</code> , <code>[</code> , <code>\</code> , etc.
<code>^α</code>	<i>α</i> at beginning of line
<code>α\$</code>	<i>α</i> at end of line
<code>{Name}</code>	<i>RegExp</i> for <i>Name</i>
<code>α?</code>	zero or one <i>α</i>
<code>α*</code>	zero or more <i>α</i>
<code>α+</code>	one or more <i>α</i>
<code>α{n, m}</code>	between <i>n</i> and <i>m</i> times <i>α</i> (“ <i>m</i> ” optional)
<code>(α)</code>	<i>α</i>
<code>α₁α₂</code>	concatenation
<code>α₁ α₂</code>	alternative
<code>α₁/α₂</code>	<i>α₁</i> but only if followed by <i>α₂</i> (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: `(id,)(gets,)(int, 1)(comma,)(int, 20)`
- observation: decision for `do` token only possible after reading “,”
- specification of `DO` keyword in [f]lex, using lookahead:

$$\text{DO} / (\{\text{LETTER}\} | \{\text{DIGIT}\})^* = (\{\text{LETTER}\} | \{\text{DIGIT}\})^* ,$$

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:

$$\text{IF} / \backslash (. * \backslash) \text{ THEN}$$

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);
}
```

returns on input

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

⇒ **lookahead counts for length of match**

Preprocessing

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)