Compiler Construction

Lecture 5: Lexical Analysis IV (Practical Aspects)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH Aachen University

noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/cc10/

Winter semester 2010/11

Lehrpreis der Fachgruppe Informatik

Wir suchen:

Beste selbstständige Lehre

Professor/-innen, Juniorprofessor/-innen und (habilitierte) Mitarbeiter/-innen, die selbstständig Vorlesungen, Seminare, Praktika oder ähnliches anbieten.

Beste Unterstützung in der Lehre

Wissenschaftliche Mitarbeiter/-innen und studentische Hilfskräfte, die maßgeblich an der Betreuung von Vorlesungen, Übungen, Seminaren und Praktika oder ähnlichem mitwirken.

Vorschläge (mit Begründung) bitte bis zum 10. November an:

`lehrpreis@informatik.rwth-aachen.de`

Die Preisverleihung erfolgt im Rahmen des diesjährigen Tages der Informatik.

Weitere Informationen:
`www.informatik.rwth-aachen.de`

RWTH AACHEN UNIVERSITY

FACHGRUPPE INFORMATIK

Announcement

Due to the Fachschaftsvollversammlung (whatever this means in English...), the Tuesday lecture on November 2 will start at 14:15.

Outline

The Extended Matching Problem

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.

To ensure uniqueness:

1. Principle of the longest match ("maximal munch tokenization")
   - for uniqueness of decomposition
   - make lexemes as long as possible
   - motivated by applications: usually every (nonempty) prefix of an identifier is also an identifier
2. Principle of the first match
   - for uniqueness of analysis
   - choose first matching regular expression (in the order given)

Implementation of FLM Analysis

Algorithm (FLM analysis)

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)*
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, and let it run on* w

Output: *FLM analysis of* w *(if it exists)*

Outline

A Backtracking NFA

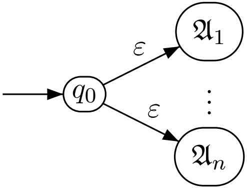
A similar construction is possible for the NFA method:

1. $\mathfrak{A}_i = \langle Q_i, \Omega, \delta_i, q_0^{(i)}, F_i \rangle \in NFA_\Omega$ $(i \in [n])$ by NFA method

A Backtracking NFA

A similar construction is possible for the NFA method:

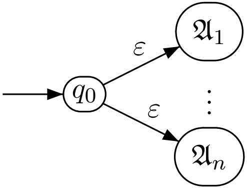
1. $\mathfrak{A}_i = \langle Q_i, \Omega, \delta_i, q_0^{(i)}, F_i \rangle \in NFA_\Omega$ $(i \in [n])$ by NFA method
2. "Product" automaton: $Q := \{q_0\} \uplus \biguplus_{i=1}^{n} Q_i$



A Backtracking NFA

A similar construction is possible for the NFA method:

1. $\mathfrak{A}_i = \langle Q_i, \Omega, \delta_i, q_0^{(i)}, F_i \rangle \in NFA_\Omega$ $(i \in [n])$ by NFA method
2. "Product" automaton: $Q := \{q_0\} \uplus \biguplus_{i=1}^n Q_i$

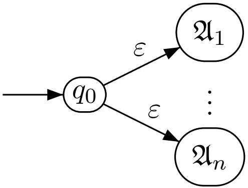


3. Partitioning of final states:
   - $M \subseteq Q$ is called a T_i-matching if
   $$M \cap F_i \neq \emptyset \text{ and for all } j \in [i-1] : M \cap F_j = \emptyset$$
   - yields set of T_i-matchings $F^{(i)} \subseteq 2^Q$
   - $M \subseteq Q$ is called productive if there exists a productive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

A Backtracking NFA

A similar construction is possible for the NFA method:

1. $\mathfrak{A}_i = \langle Q_i, \Omega, \delta_i, q_0^{(i)}, F_i \rangle \in NFA_\Omega$ ($i \in [n]$) 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

Longest Match in Practice

- In general: lookahead of arbitrary length required
  - that is, $|v|$ unbounded in configurations (T, vqw, W)
  - see Example 4.15: $\alpha_1 = a$, $\alpha_2 = a^*b$, $w = a \ldots a$

Longest Match in Practice

- In general: lookahead of arbitrary length required
  - that is, $|v|$ unbounded in configurations (T, vqw, W)
  - see Example 4.15: $\alpha_1 = a$, $\alpha_2 = a^*b$, $w = a \dots a$
- "Modern" programming languages (Pascal, ...): 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 Example 4.15: $\alpha_1 = a$, $\alpha_2 = a^*b$, $w = a \dots a$
- "Modern" programming languages (Pascal, ...): 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 5.1 (Longest Match in FORTRAN)

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

Inadequacy of Longest Match I

Example 5.1 (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: $(\mathsf{int}, 12)(\mathsf{relop}, \mathsf{eq})(\mathsf{int}, 12)$

Inadequacy of Longest Match I

Example 5.1 (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: $(\mathsf{int}, 12)(\mathsf{relop}, \mathsf{eq}), (\mathsf{int}, 12)$
   - LM yields: $(\mathsf{real}, 12.0)(\mathsf{id}, \mathtt{EQ})(\mathsf{real}, 0.12)$

   $\Rightarrow$ wrong interpretation

Example 5.1 (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: $(\mathsf{int}, 12)(\mathsf{relop}, \mathsf{eq}), (\mathsf{int}, 12)$
   - LM yields: $(\mathsf{real}, 12.0)(\mathsf{id}, \mathtt{EQ})(\mathsf{real}, 0.12)$

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

Inadequacy of Longest Match I

Example 5.1 (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: $(\mathsf{int}, 12)(\mathsf{relop}, \mathsf{eq}), (\mathsf{int}, 12)$
   - LM yields: $(\mathsf{real}, 12.0)(\mathsf{id}, \mathtt{EQ})(\mathsf{real}, 0.12)$

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

Inadequacy of Longest Match I

Example 5.1 (Longest Match in FORTRAN)

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

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

Inadequacy of Longest Match I

Example 5.1 (Longest Match in FORTRAN)

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

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

Inadequacy of Longest Match II

Example 5.2 (Longest Match in C)

- valid lexemes:
  - `x` (identifier)
  - `=-` (decrement operator; ANSI-C: `-=`)
  - `1`, `-1` (integers)
- input string: `x=-1`

Inadequacy of Longest Match II

Example 5.2 (Longest Match in C)

- valid lexemes:
  - `x` (identifier)
  - `=-` (decrement operator; ANSI-C: `-=`)
  - `1`, `-1` (integers)
- input string: `x=-1`
- intended analysis: (id, x)(gets,)(int, -1)

Inadequacy of Longest Match II

Example 5.2 (Longest Match in C)

- valid lexemes:
  - `x` (identifier)
  - `=-` (decrement operator; ANSI-C: `-=`)
  - `1`, `-1` (integers)
- input string: `x=-1`
- intended analysis: $(\mathsf{id}, \mathtt{x})(\mathsf{gets},)$, $(\mathsf{int}, -1)$
- LM yields: $(\mathsf{id}, \mathtt{x})(\mathsf{dec},)(\mathsf{int}, 1)$

$\Rightarrow$ wrong interpretation

Inadequacy of Longest Match II

Example 5.2 (Longest Match in C)

- valid lexemes:
  - `x` (identifier)
  - `=-` (decrement operator; ANSI-C: `-=`)
  - `1`, `-1` (integers)
- input string: `x=-1`
- intended analysis: $(\mathsf{id}, \mathtt{x})(\mathsf{gets},), (\mathsf{int}, -1)$
- LM yields: $(\mathsf{id}, \mathtt{x})(\mathsf{dec},)(\mathsf{int}, 1)$

$\Rightarrow$ wrong interpretation

Possible solutions:

- Hand-written (non-FLM) scanners
- Lookahead (later)

Outline

Regular Definitions I

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

Regular Definitions I

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

Definition 5.3 (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: modularizing the representation of regular sets by introducing additional identifiers

Definition 5.3 (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 no recursion is involved, every R_i can (iteratively) be substituted by a regular expression $\alpha \in RE_\Omega$
(otherwise $\Longrightarrow$ context-free languages)

Regular Definitions II

Example 5.4 (Symbol classes in Pascal)

Identifiers:

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

Numerals: (unsigned)

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

Rel. operators:

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

Keywords:

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

Outline

The [f]lex Tool

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

$$\underset{\texttt{[f]lex specification}}{\texttt{spec.l}} \xrightarrow{\texttt{[f]lex}} \underset{\text{Scanner (in C)}}{\texttt{lex.yy.c}} \xrightarrow{\texttt{cc}} \underset{\text{Executable}}{\texttt{a.out}}$$

$$\text{Program} \xrightarrow{\texttt{a.out}} \text{Symbol sequence}$$

The [f]lex Tool

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

$$\underset{\texttt{[f]lex} \text{ specification}}{\texttt{spec.l}} \xrightarrow{\texttt{[f]lex}} \underset{\text{Scanner (in C)}}{\texttt{lex.yy.c}} \xrightarrow{\texttt{cc}} \underset{\text{Executable}}{\texttt{a.out}}$$

$$\text{Program} \xrightarrow{\texttt{a.out}} \text{Symbol sequence}$$

A `[f]lex` specification is of the form

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

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 *Names*
- *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 in 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 5.5 (Lookahead in FORTRAN)

1. `DO` loops (cf. Example 5.1)
   - input string: `DO 5 I = 1, 20`
   - LM yields: (id,)(gets,)(int, 1)(comma,)(int, 20)
   - observation: decision for do only possible after reading "`,`"
   - specification of `DO` keyword in `[f]lex`, using lookahead:
     `DO / ({LETTER}|{DIGIT})* = ({LETTER}|{DIGIT})* ,`

Using the Lookahead Operator

Example 5.5 (Lookahead in FORTRAN)

1. `DO` loops (cf. Example 5.1)
   - input string: `DO 5 I = 1, 20`
   - LM yields: (id,)(gets,)(int, 1)(comma,)(int, 20)
   - observation: decision for do 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 ...` (`IF` keyword)
   - specification of `IF` keyword in `[f]lex`, using lookahead:
     `IF / \( .* \) THEN`

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'`

$\Longrightarrow$ lookahead counts for length of match

Outline

Further Problems

Identifiers that influence subsequent parsing

- **Example:** (type definitions in C)
  The program fragment `(T *)` is
  - valid in the scope of declaration `typedef int T;`
    (casting to type "pointer to `T`")
  - invalid in the scope of declaration `int T;`
    (incorrect expression with missing second multiplication factor)
- **Solution:** exploit symbol table information

Further Problems

Identifiers that influence subsequent parsing

- **Example:** (type definitions in C)
  The program fragment `(T *)` is
  - valid in the scope of declaration `typedef int T;`
    (casting to type "pointer to `T`")
  - invalid in the scope of declaration `int T;`
    (incorrect expression with missing second multiplication factor)
- **Solution:** exploit symbol table information

Macro processing

- macro substitution
- parameter substitution
- file inclusion
- conditional compilation
- **Solution:** separate preprocessing phase between reading and lexical analysis