Compiler Construction

Lecture 4: Lexical Analysis III (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/cc08/`

Summer semester 2008

Outline

Repetition: 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)

Repetition: 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

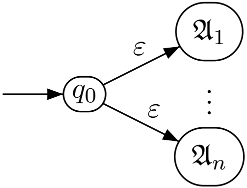
Output: *FLM analysis of* w *(if it exists)*

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
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

Outline

Regular Definitions I

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

Definition 4.1 (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 4.2 (Symbol classes in Pascal)

Identifiers:

$$Letter = \mathtt{A} + \ldots + \mathtt{Z} + \mathtt{a} + \ldots + \mathtt{z}$$
$$Digit = \mathtt{0} + \ldots + \mathtt{9}$$
$$Id = Letter\,(Letter + Digit)^*$$

Numerals: (unsigned)

$$Digits = Digit^+$$
$$Empty = \Lambda^*$$
$$OptFrac = \mathtt{.}\,Digits + Empty$$
$$OptExp = \mathtt{E}\,(\mathtt{+} + \mathtt{-} + Empty)\,Digits + Empty$$
$$Num = Digits\ OptFrac\ OptExp$$

Rel. operators:

$$RelOp = \mathtt{<} + \mathtt{<=} + \mathtt{=} + \mathtt{<>} + \mathtt{>} + \mathtt{>=}$$

Keywords:

$$If = \mathtt{if}$$
$$Then = \mathtt{then}$$
$$Else = \mathtt{else}$$

Outline

The [f]lex Tool

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

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

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

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!)

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 `.` or `.|\n` (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
α`$`	α at end of line
`{`*Name*`}`	*RegExp* for *Name*
α`?`	zero or one α
α`*`	zero or more α
α`+`	one or more α
α`{`n,m`}`	between n and m times α ("$,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)

Outline

Longest Match in Practice

- In general: lookahead of arbitrary length (backtracking phase) required
  - see example on Slide 3.18: $\alpha_1 = a$, $\alpha_2 = a^*b$
- "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)

Example 4.3 (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)
2. `DO` loops
   - (erroneous) input string: `DO␣5␣I␣=␣1.␣20` ⇝ `DO5I=1.20`
     - LM analysis (correct): (id, DO5I)(gets,)(real, 1.2)
   - (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 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})*,`

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