Lexical Analysis

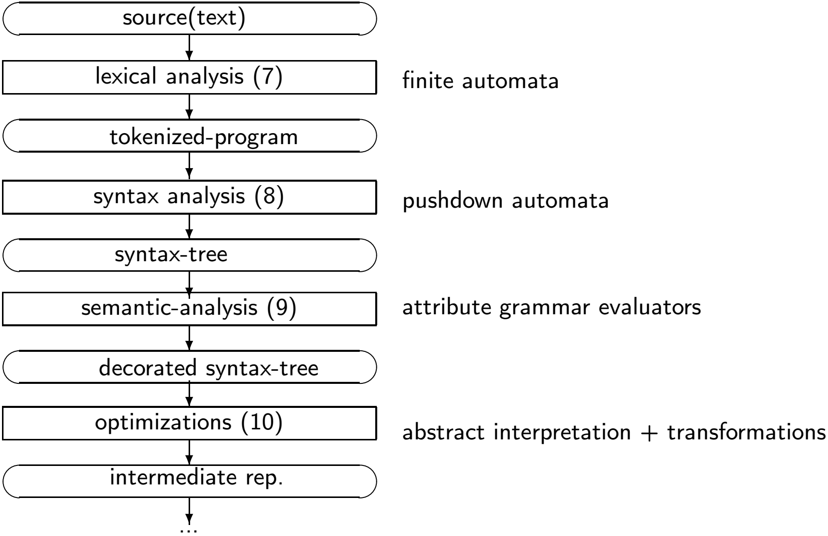
Reinhard Wilhelm
Universität des Saarlandes
wilhelm@cs.uni-sb.de
and
Mooly Sagiv
Tel Aviv University
sagiv@math.tau.ac.il

20. Oktober 2009

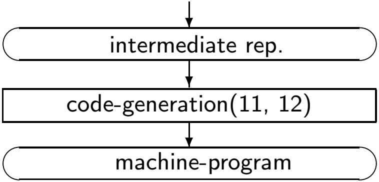
Subjects

- Role of lexical analysis
- Regular languages, regular expressions
- Finite automata
- From regular expressions to finite automata
- A language for specifying lexical analysis
- The generation of a scanner
- Flex

"Standard" Structure



"Standard" Structure cont'd



tree automata + dynamic programming + $\cdots$

Lexical Analysis (Scanning)

- Functionality

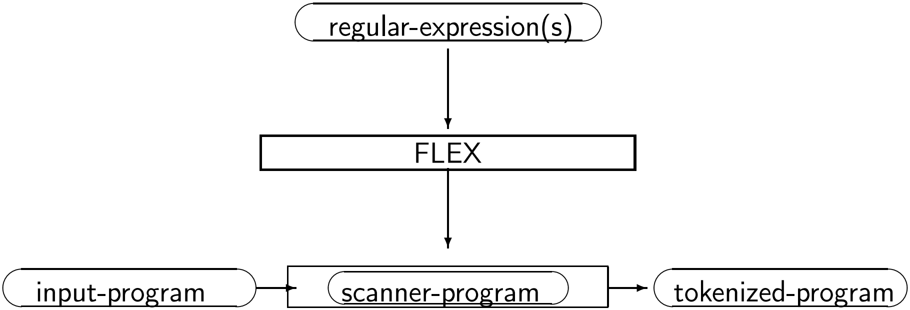
  Input: program as sequence of characters
  Output: program as sequence of symbols (tokens)

- Produce listing
- Report errors, symbols illegal in the programming language
- Screening subtask:
  - Identify language keywords and standard identifiers
  - Eliminate “white-space”, e.g., consecutive blanks and newlines
  - Count line numbers
  - Construct table of all symbols occurring

Automatic Generation of Lexical Analyzers

- The symbols of programming languages can be specified by regular expressions.
- Examples:
  - `program` as a sequence of characters.
  - `(alpha (alpha | digit)*)` for Pascal identifiers
  - ‘‘(*‘‘ until ‘‘*)‘‘ for Pascal comments
- The recognition of input strings can be performed by a finite automaton.
- A table representation or a program for the automaton is automatically generated from a regular expression.

Automatic Generation of Lexical Analyzers cont'd



Notations

A language, L, is a set of words, x, over an alphabet, Σ.

$a_1 a_2 \ldots a_n$,	a word over Σ, $a_i \in \Sigma$
ε	The empty word
Σ^n	The words of length n over Σ
Σ^*	The set of finite words over Σ
Σ^+	The set of non-empty finite words over Σ
$x.y$	The concatenation of x and y

Language Operations

$L_1 \cup L_2$		Union
$L_1 L_2$	$= \{x.y \| x \in L_1, y \in L_2\}$	Concatenation
$\overline{L}$	$= \Sigma^* - L$	Complement
L^n	$= \{x_1 \ldots x_n \| x_i \in L, 1 \leq i \leq n\}$	
L^*	$= \bigcup_{n \geq 0} L^n$	Closure
L^+	$= \bigcup_{n \geq 1} L^n$	

Regular Languages

Defined inductively

- $\emptyset$ is a regular language over Σ
- $\{\varepsilon\}$ is a regular language over Σ
- For all $a \in \Sigma$, $\{a\}$ is a regular language over Σ
- If R_1 and R_2 are regular languages over Σ, then so are:
  - $R_1 \cup R_2$,
  - $R_1 R_2$, and
  - R_1^*

Regular Expressions and the Denoted Regular Languages

Defined inductively

- $\underline{\emptyset}$ is a regular expression over Σ denoting $\emptyset$,
- $\underline{\varepsilon}$ is a regular expression over Σ denoting $\{\varepsilon\}$,
- For all $a \in \Sigma$, a is a regular expression over Σ denoting $\{a\}$,
- If r_1 and r_2 are regular expressions over Σ denoting R_1 and R_2, resp., then so are:
  - $\underline{(}r_1\underline{|}r_2\underline{)}$, which denotes $R_1 \cup R_2$,
  - $\underline{(}r_1 r_2\underline{)}$, which denotes $R_1 R_2$, and
  - $\underline{(}r_1\underline{)}\underline{^*}$, which denotes R_1^*.
- Metacharacters, $\underline{\emptyset}, \underline{\varepsilon}, \underline{(}, \underline{)}, \underline{|}, \underline{*}$ don't really exist, are replaced by their non-underlined versions.
  Attention: Clash between characters in Σ and metacharacters $\{\underline{(}, \underline{)}, \underline{|}, \underline{^*}\}$

Example

Expression	Language	Example Words
$a \| b$	$\{a, b\}$	a, b
ab^*a	$\{a\}\{b\}^*\{a\}$	$aa, aba, abba, abbba, \ldots$
$(ab)^*$	$\{ab\}^*$	$\varepsilon, ab, abab, \ldots$
$abba$	$\{abba\}$	$abba$

Regular Expressions for Symbols (Tokens)

Alphabet for the symbol classses listed below:

$\Sigma =$

integer-constant

real-constant

identifier

string

comments

matching-parentheses?

Automata

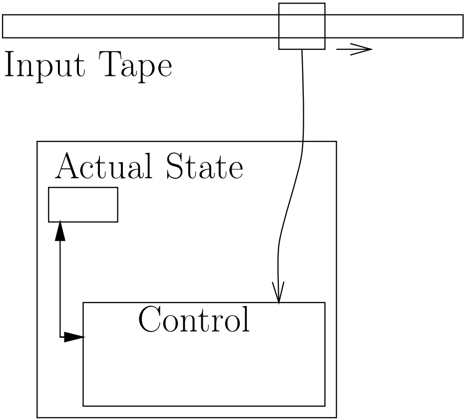
In the following, we will meet different types of automata.

Automata

- process some *input*, e.g. strings or trees,
- make *transitions* from configurations to configurations;
- *configurations* consist of (the rest of) the input and some *memory*;
- the *memory* may be small, just one variable with finitely many values,
- but the memory may also be able to grow without bound, adding and removing values at one of its ends;
- the type of memory an automaton has determines its ability to *recognize* a class of languages,
- in fact, the more powerful an automaton type is, the better it is in *rejecting* input.

Finite State Machine

The simplest type of automaton, its memory consists of only one variable, which can store one out of finitely many values, its *states*,



A Non-Deterministic Finite Automaton (NFA)

$M = \langle \Sigma, Q, \Delta, q_0, F \rangle$ where:

- Σ — finite alphabet
- Q — finite set of states
- $q_0 \in Q$ — initial state
- $F \subseteq Q$ — final states
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ — transition relation

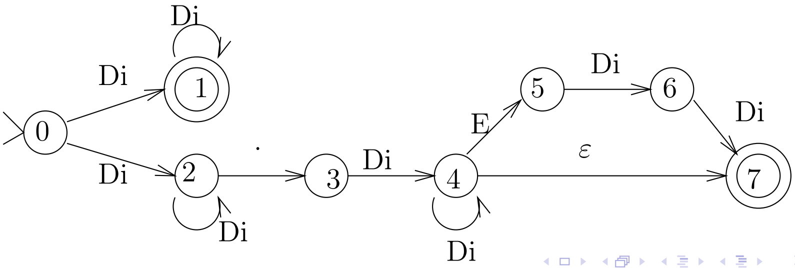
May be represented as a transition diagram

- Nodes — States
- q_0 has a special "entry" mark
- final states doubly encircled
- An edge from p into q labeled by a if $(p, a, q) \in \Delta$

Example: Integer and Real Constants

	Di $\in \{0, 1, \ldots, 9\}$	.	E	ε
0	{1,2}	$\emptyset$	$\emptyset$	$\emptyset$
1	{1}	$\emptyset$	$\emptyset$	$\emptyset$
2	{2}	{3}	$\emptyset$	$\emptyset$
3	{4}	$\emptyset$	$\emptyset$	$\emptyset$
4	{4}	$\emptyset$	{5}	{7}
5	{6}	$\emptyset$	$\emptyset$	$\emptyset$
6	{7}	$\emptyset$	$\emptyset$	$\emptyset$
7	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

$$q_0 = 0$$
$$F = \{1, 7\}$$



Finte Automata — Scanners

Finite automata

- get an input word,
- start in their initial state,
- make a series of transitions under the characters constituting the input word,
- accept (or reject).

Scanners

- get an input string (a sequence of words),
- start in their initial state,
- attempt to find the end of the next word,
- when found, restart in their initial state with the rest of the input,
- terminate when the end of the input is reached or an error is encountered.

Maximal Munch strategy

Find longest prefix of remaining input that is a legal symbol.

- first input character of the scanner — first "non-consumed" character,
- in final state, and exists transition under the next character: make transition and remember position,
- in final state, and exists no transition under the next character: Symbol found,
- actual state not final and no transition under the next character: backtrack to last passed final state
  - There is none: Illegal string
  - Otherwise: Actual symbol ended there.

Warning: Certain overlapping symbol definitions will result in quadratic runtime: Example: $(a|a^*;)$

Other Example Automata

- integer-constant
- real-constant
- identifier
- string
- comments

The Language Accepted by an Automaton

- $M = \langle \Sigma, Q, \Delta, q_0, F \rangle$
- For $q \in Q$, $w \in \Sigma^*$, (q, w) is a configuration
- The binary relation step on configurations is defined by:

$$(q, aw) \vdash_M (p, w)$$

  if $(q, a, p) \in \Delta$
- The reflexive transitive closure of $\vdash_M$ is denoted by $\vdash_M^*$
- The language accepted by M

$$L(M) = \{w \mid w \in \Sigma^* \mid \exists q_f \in F : (q_0, w) \vdash_M^* (q_f, \varepsilon)\}$$

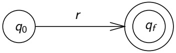
From Regular Expressions to Finite Automata

Theorem

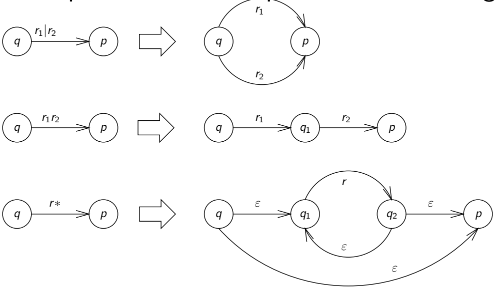
(i) For every regular language R, there exists an NFA M, such that $L(M) = R$.
(ii) For every regular expression r, there exists an NFA that accepts the regular language defined by r.

A Constructive Proof for (ii) (Algorithm)

- A regular language is defined by a regular expression r
- Construct an "NFA" with one final state, q_f, and the transition



- Decompose r and develop the NFA according to the following rules



until only transitions under single characters and ε remain.

Examples

- $a(a|0)^*$ over $\Sigma = \{a, 0\}$
- Identifier
- String

Nondeterminism

- Several transitions may be possible under the same character in a given state
- ε-moves (next character is not read) may "compete" with non-ε-moves.
- Deterministic simulation requires "backtracking"

Deterministic Finite Automaton (DFA)

- No ε-transitions
- At most one transition from every state under a given character, i.e. for every $q \in Q$, $a \in \Sigma$,

$$|\{q' \mid (q, a, q') \in \Delta\}| \leq 1$$

From Non-Deterministic to Deterministic Automata

Theorem

For every NFA, $M = \langle \Sigma, Q, \Delta, q_0, F \rangle$ there exists a DFA, $M' = \langle \Sigma, Q', \delta, q_0', F' \rangle$ such that $L(M) = L(M')$.

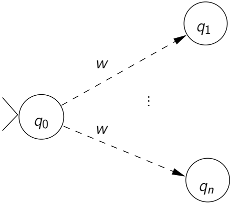
A Scheme of a Constructive Proof (Powerset Construction)

Construct a DFA whose states are sets of states of the NFA.

The DFA simulates all possible transition paths under an input word in parallel.

Set of new states

$\{\{q_1, \ldots, q_n\} \mid n \geq 1 \wedge \exists w \in \Sigma^* : (q_0, w) \vdash_M^* (q_i, \varepsilon)\}$



The Construction Algorithm

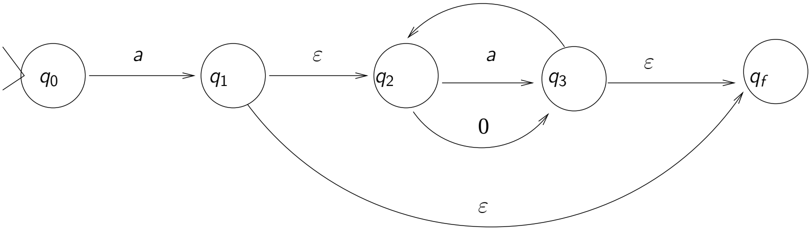
Used in the construction: the set of ε-*Successors*,
$\varepsilon\text{-}SS(q) = \{p \mid (q, \varepsilon) \vdash_M^* (p, \varepsilon)\}$

- Starts with $q_0' = \varepsilon\text{-}SS(q_0)$ as the initial DFA state.
- Iteratively creates more states and more transitions.
- For each DFA state $S \subseteq Q$ already constructed and character $a \in \Sigma$,

$$\delta(S, a) = \bigcup_{q \in S} \bigcup_{(q,a,p) \in \Delta} \varepsilon\text{-}SS(p)$$

  if non-empty
  add new state $\delta(S, a)$ if not previously constructed;
  add transition from S to $\delta(S, a)$.
- A DFA state S is accepting (in F') if there exists $q \in S$ such that $q \in F$

Example: $a(a|0)^*$

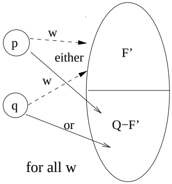


DFA minimization

DFA need not have minimal size, i.e. minimal number of states and transitions.

q and p are undistinguishable iff

for all words w $(q, w) \vdash_M^*$ and $(p, w) \vdash_M^*$ lead into either F' or $Q' - F'$.

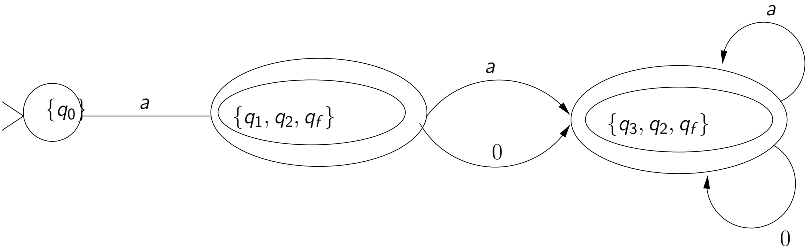


After termination merge undistinguishable states.

DFA minimization algorithm

- Input a DFA $M = \langle \Sigma, Q, \delta, q_0, F \rangle$
- Iteratively refine a partition of the set of states, where each set in the partition consists of states so far undistinguishable.
- Start with the partition $\Pi = \{F, Q - F\}$
- Refine the current Π by splitting sets $S \in \Pi$ if there exist $q_1, q_2 \in S$ and $a \in \Sigma$ such that
  - $\delta(q_1, a) \in S_1$ and $\delta(q_2, a) \in S_2$ and $S_1 \neq S_2$
- Merge sets of undistinguishable states into a single state.

Example: $a(a|0)^*$



A Language for specifying lexical analyzers

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$
$$(\varepsilon|.(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$$
$$(\varepsilon|E(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)))$$

Descriptional Comfort

Character Classes:
: Identical meaning for the DFA (exceptions!), e.g.,
le = a - z A - Z
di = 0 - 9
Efficient implementation: Addressing the transitions indirectly through an array indexed by the character codes.

Symbol Classes:
: Identical meaning for the parser, e.g.,
Identifiers
Comparison operators
Strings

Descriptional Comfort cont'd

Sequences of regular definitions:

$$
\begin{array}{rcl}
A_1 & = & R_1 \\
A_2 & = & R_2 \\
 & \cdots & \\
A_n & = & R_n
\end{array}
$$

Sequences of Regular Definitions

Goal: Separate final states for each definition

1. Substitute right sides for left sides
2. Create an NFA for every regular expression separately;
3. Merge all the NFAs using ε transitions from the start state;
4. Construct a DFA;
5. Minimize starting with partition

$$\{F_1, F_2, \ldots, F_n, Q - \bigcup_{i=1}^{n} F_i\}$$

Flex Specification

Definitions
%%
Rules
%%
C-Routines

Flex Example

```
%{
extern int line_number;
extern float atof(char *);
%}
DIG      [0-9]
LET     [a-zA-Z]
%%
[=#<>+-*]            { return(*yytext); }
({DIG}+)  { yylval.intc  = atoi(yytext); return(301); }
({DIG}*\.{DIG}+(E(\+|\-)?{DIG}+)?)
           {yylval.realc = atof(yytext); return(302); }
\"(\\.|[^\"\\])*\" { strcpy(yylval.strc, yytext);
                     return(303); }
"<="                 { return(304); }
:=                   { return(305); }
\.\.                 { return(306); }
```

Flex Example cont'd

```
ARRAY                  { return(307); }
BOOLEAN                { return(308); }
DECLARE                { return(309); }
{LET}({LET}|{DIG})* { yylval.symb = look_up(yytext);
                       return(310); }
[ \t]+                  { /* White space */ }
\n                      { line_number++; }
.                       { fprintf(stderr,
   "WARNING: Symbol '%c' is illegal, ignored!\n", *yytext);}
%%
```