

Programming Language Implementation II

In this lecture we will look at **lexical analysis**, **scanning** and **screening**.

- **Regular languages and regular expressions**
- **Finite state automata (FSA)**
 - deterministic and non-deterministic FSA
 - Equivalence and minimization of FSA
 - Pumping lemma
- **Scanner generators: MLex, Flex.**

The material is (loosely) based on Appelt, Chapter 2 and Wilhelm and Maurer Chapter 7.

Lexical Analysis

Recall:

The **scanner** (lexical analyser) converts the source program (string of characters) into a sequence of tokens (i.e. lexical units).

Tokens are, for example

- identifiers (“this”, “x”, ...)
- numbers and other constants (“3.14”, “99”, “A String t”, ...)
- operators (“+”, “-”, “(”, ...)

At the same time meaningless whitespace (blanks, tabs, line breaks, etc.) are stripped.

In some cases the scanner already strips comments, too.

Lexical analysis is based on the theory of **regular expressions**.

Screening

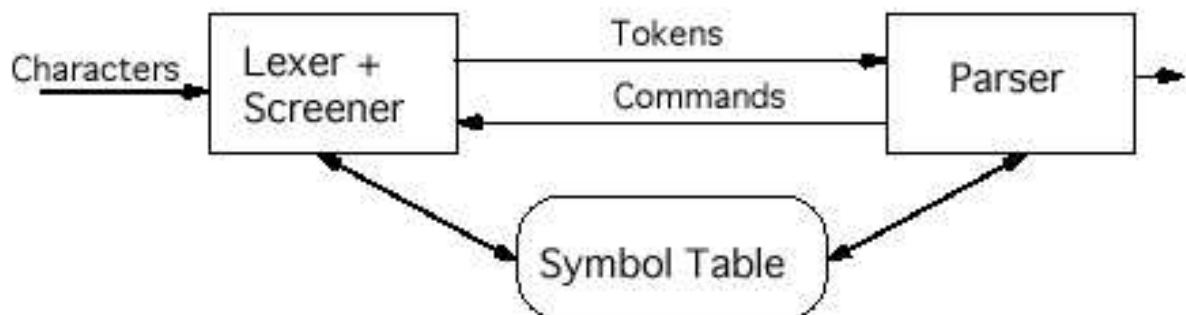
Recall:

The Screener performs further analysis and classification of the tokens

This includes

- recognition of reserved words (“while”, “return” ...)
- stripping comments
- recognizing compiler directives (so-called pragmas)

Finally it creates the *symbol table* and links the tokens to the entries in the symbol table.



Words and Languages

A **word** x over an **alphabet** Σ is a finite sequence of characters from (the set of characters) Σ .

We let ϵ be the **empty word**, ie the null string.

A (**formal**) **language** L over Σ is a set of words over Σ .

We will use the following operations on formal languages:

$L_1 \cup L_2$		Union of languages
$L_1 L_2$	$\{xy \mid x \in L_1, y \in L_2\}$	Concatenation of languages
L^n	$\{x_1 \cdots x_n \mid x_i \in L, 1 \leq i \leq n\}$	
L^*	$\bigcup_{n \geq 0} L^n$	Closure of a language
L^+	$\bigcup_{n > 0} L^n$	
\bar{L}	$\Sigma^* - L$	Complement of a language

Let $D = \{0, 1, \dots, 9\}$ and $L = \{a, \dots, z, A, \dots, Z\}$. Then:

- $L \cup D$ is the set of letters and digits
- $LD = \{a0, \dots, a9, \dots, Z0, \dots, Z9\}$
- D^+ are the strings of digits.

Exercise: What are $L(L \cup D)^*$ and $(D - \{0\})D^*$?

Regular Languages and Regular Expressions

The **regular expressions** over alphabet Σ are:

- ϵ , which describes the language $\{\epsilon\}$
- a for any $a \in \Sigma$, which describes the language $\{a\}$
- If regular expressions r and s describe the languages R and S then,
 - $(r|s)$ is a regular expression describing $R \cup S$
 - (rs) is a regular expression describing RS
 - (r^*) is a regular expression describing R^* .

A language which can be exactly described by a regular expression is called **regular**.

Many useful languages are not regular, eg the set of palindromes.

* has highest precedence, followed by concatenation, then disjunction.

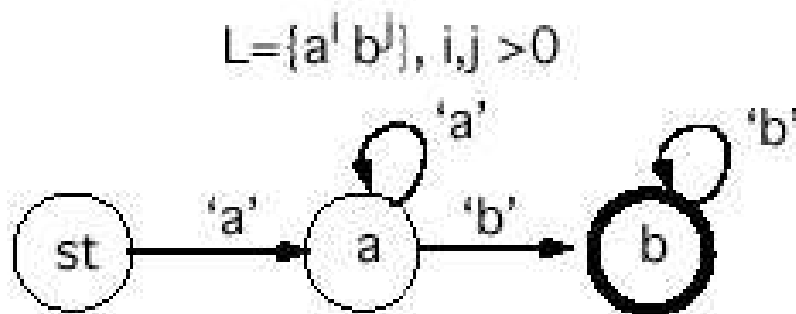
Finite-State Automata

A **finite state automata** can be used to recognize if a string is in a regular language or not.

A (**non-deterministic**) **finite state automata (NFA)** consists of an:

- **input alphabet** Σ ,
- a finite set of **states** Q ,
- an **initial state** $q_0 \in Q$,
- a set of **final states** $F \subseteq Q$, and
- a **transition relation** $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$.

A NFA **accepts** (or **recognizes**) words for which it has a path from the initial state to a final state.



Transition Table

An example NFA has

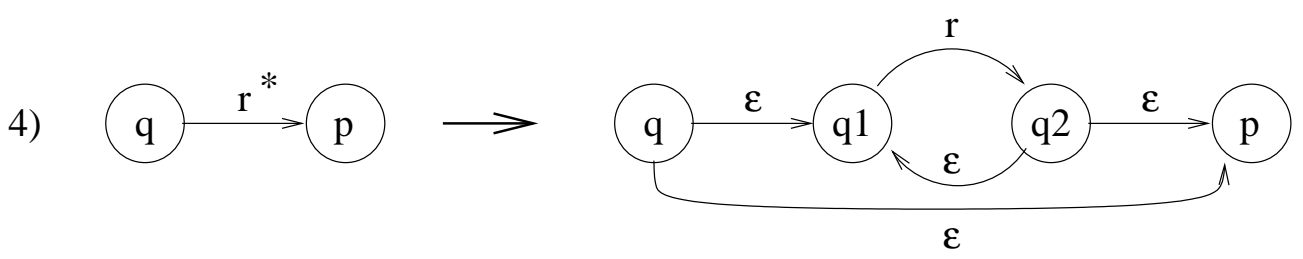
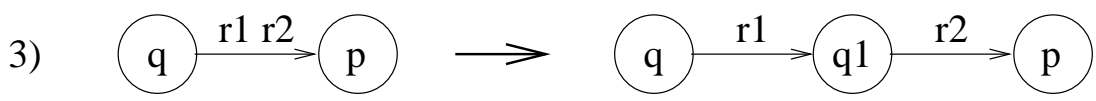
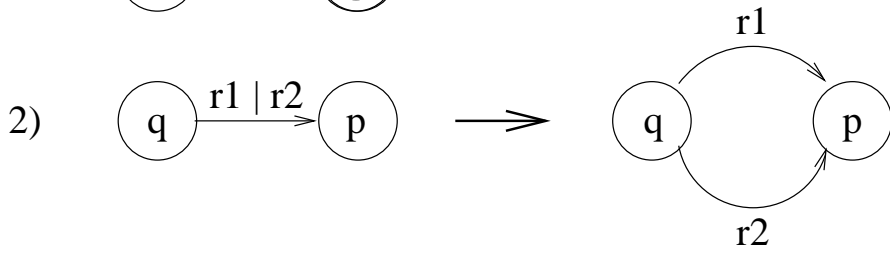
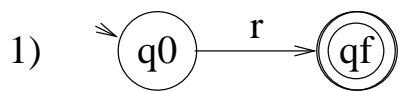
- input alphabet $\{0, 1, \dots, 9, \epsilon, E\}$,
- states $\{0, \dots, 7\}$,
- initial state 0,
- final states $\{1, 7\}$, and
- transition table,

	0, 1, ...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

We usually represent an NFA using a **state transition diagram**.
That for the previous NFA is:

Finite-State Automata (Cont)

It is straightforward to give a NFA for recognizing words in the language of an arbitrary regular expression. The rules are



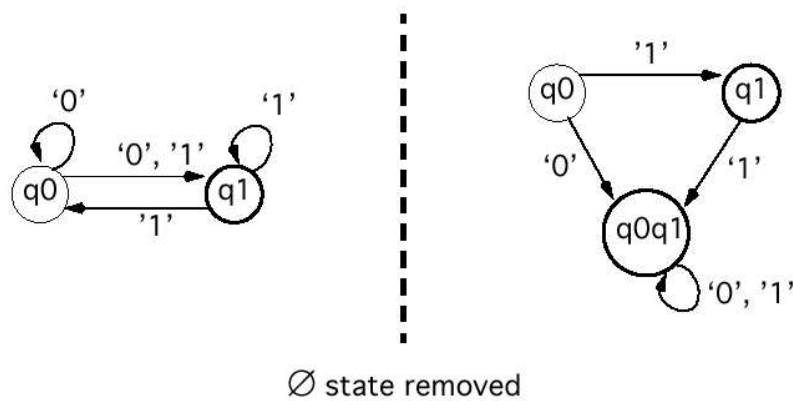
Exercise: What is a NFA for recognizing the language of $a(a|b)^*$?

NFA-DFA Equivalence

A finite state automata is **deterministic** if it has no transitions via ϵ and at most one successor state for each pair (q, a) where $q \in Q$ and $a \in \Sigma$. We call such an automata a **DFA**.

Every non-deterministic state automaton (NFA) can be transformed into an equivalent deterministic state automaton (DFA) such that both automata accept the same language.

The “trick” is to model all possible state combinations (in the NFA) explicitly as separate states (in the DFA). This is called the *subset construction*.



The obvious problem is that this leads to a combinatorial explosion in the number of states.

Subset Construction

$$\text{NFA } N = (S, V, T, S_0, F)$$

$$\text{DFA } D = (S', V, T', S_0', F')$$

$$S' = 2^S$$

$$S_0' = \{S_0\}$$

$$F' = \{s \in S' \mid \exists x \in s : (x \in F)\}$$

$$((q, x) \rightarrow p) \in T' \text{ with } q = \{q_1, q_2, \dots, q_n\}$$

if and only if

$$((q_i, x) \rightarrow s) \in T \text{ and } p = \bigcup_i s$$

Construction of D for the previous example:

$$N = (\{q_0, q_1\}, \{0, 1\}, \{(q_0, 0) \rightarrow \{q_0, q_1\}, (q_0, 1) \rightarrow \{q_1\}, (q_1, 1) \rightarrow \{q_0, q_1\}\}, q_0, q_1)$$

yields

$$D = (\{q_0, q_1, \{q_0, q_1\}\}, \{0, 1\}, \{(q_0, 0) \rightarrow \{q_0, q_1\}, (q_0, 1) \rightarrow \{q_1\}, (q_1, 1) \rightarrow \{q_0, q_1\}, (\{q_0, q_1\}, 0) \rightarrow \{\{q_0, q_1\}\}, (\{q_0, q_1\}, 1) \rightarrow \{\{q_0, q_1\}\}\}, \{q_0\}, \{q_1, \{q_0, q_1\}\})$$

Minimizing DFAs

The DFA for recognizing the regular language generated by the above two steps is usually not the smallest DFA recognizing the language.

The DFA **minimization** algorithm takes a DFA and returns a DFA with the smallest number of states which recognizes the same language.

The idea is to partition the original states into states which have **equivalent** accepting behaviour.

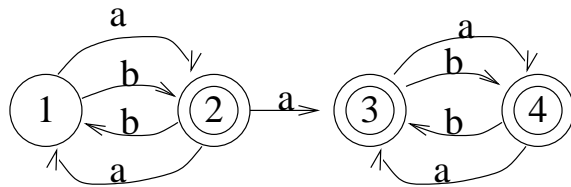
Initially the states are partitioned into the final states and the non-final states.

Repeatedly **split** a partition if its states lead to different accepting behavior for some character in the alphabet.

Dead (no path to final state) and **unreachable** (no path from start state) states are removed.

Minimizing DFA Example

Find the minimal DFA for



Exercise: What is the minimal DFA for $a(a|b)^*$?

Pumping Lemma

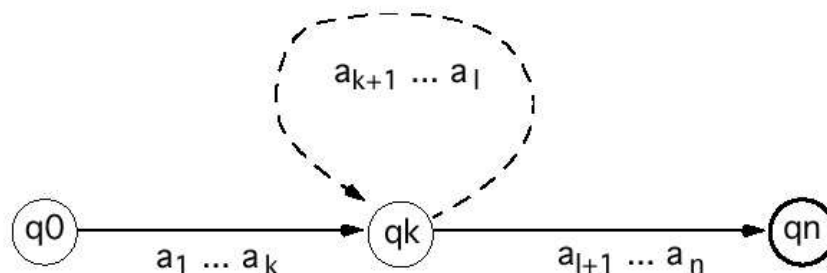
a fundamental result about regular languages is the *pumping lemma*.

It can be used to **prove that a language L is not regular**, i.e. that we cannot construct a DFA that accepts exactly this language.

Theorem: (Pumping Lemma)

Let L be some regular language. There is a finite constant n such that any word x in L can be split into three words u, v, w with $x = uvw$ with $\text{length}(uv) \leq n$ and $\text{length}(v) > 0$ and $\forall i > 0 : uv^i w \in L$.

Consider a word in L with more symbols than D has state. If it is accepted some state must have been repeated. Since the DFA cannot keep a memory of its execution history, it is possible to traverse the loop over and over again to accept other (longer) words.



Example: $a^i b^i$ is not regular.

Let $x = a^i b^i$ with $i = n$ (number of states). Then $u = a^j, v = a^k, w = a^l b^i$. Thus $\forall m > 0 : a^j a^{mk} a^l b^i \in L$.

So D also accepts $a^* b^i$. **Contradiction.**

Regular Grammars

An alternative characterization of regular languages is via *regular grammars*. These are also called *type 3 grammars*.

A grammar is a quadruple $G = (N, T, P, S)$ consisting of:

- A set of **terminal** symbols T ;
- A set of **non-terminal** symbols N ;
- A **start** symbol, $S \in N$;
- A set of **productions** each of form $X \rightarrow \alpha$ where $X \in N$ and α is a sequence from $(N \cup T)^*$.

Such a grammar is a mechanism to construct words. Let x be a word over $N \cup T$. If it is possible to split x into uAv where u, v are words over $N \cup T$ and $A \in N$ such that $(A \rightarrow q) \in P$, we write

$$x \Rightarrow_G uqv$$

and say that uqv can be derived from x according to G (in a single step).

We write $x_0 \Rightarrow_G^* x_n$ for $x_0 \Rightarrow x_1 \Rightarrow \dots \Rightarrow x_n$.

A grammar $G = (N, T, P, S)$ generates (or accepts) the language $L = \{x \mid S \Rightarrow_G^* x \wedge x \in T^*\}$.

Regular Grammars (cont.)

If all productions in P have the form $A \rightarrow wB$ or $A \rightarrow w$ where $A, B \in N$ and w is a word over T the grammar is regular and right-linear. Alternatively we can demand that all productions have the form $A \rightarrow Bw$ or $A \rightarrow w$. In this case the grammar is regular and left-linear.

A regular grammar generates a regular language.

Example:

$G = (\{X, Y\}, \{a, b\}, \{X \rightarrow aX, X \rightarrow aY, Y \rightarrow bY, Y \rightarrow b\}, X)$
generates $L = a^i b^j, i > 0, j > 0$.

Output in Automata

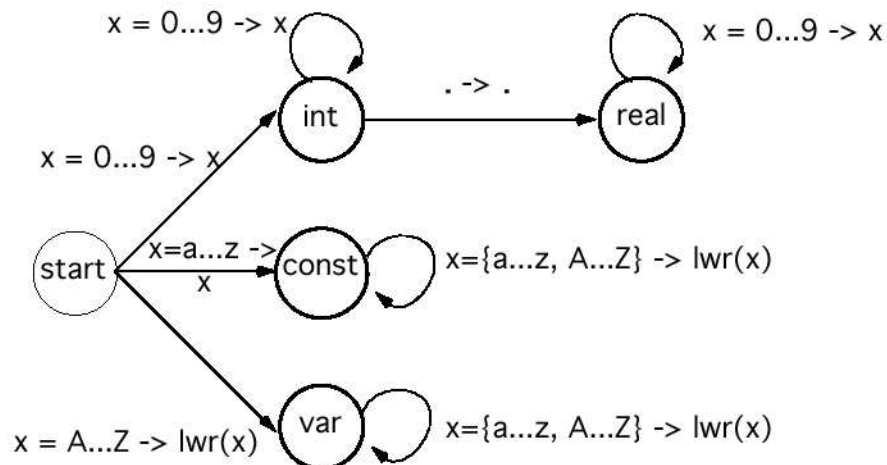
In practice our FSAs also need to generate output, i.e. they synthesize the tokens and should return a token type. There are two different ways of generating output in state automata (or state machines).

Mealy machines are FSAs in which each transition generates an output.

Moore machines are FSA in which each state generates an output.

Note: We use the state names of the different end states to derive the classification of the recognized token.

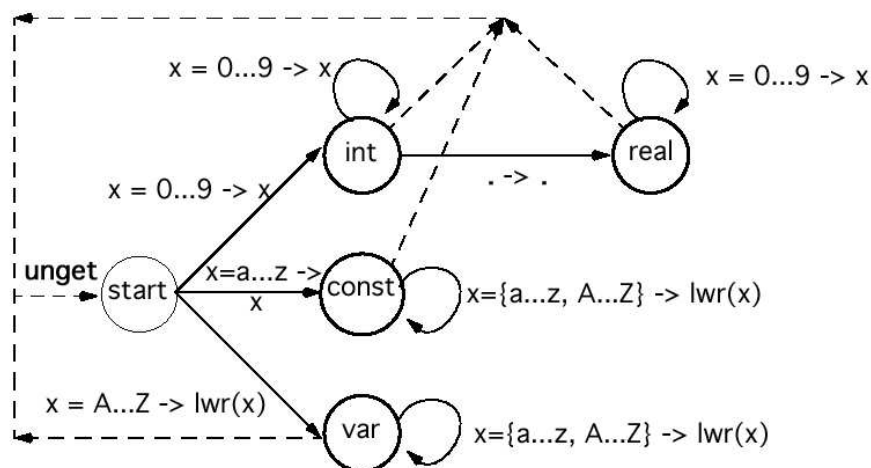
Example: A Mealy machine to recognize, classify and normalize numbers and identifiers.



Lookahead

If we implement a DFA for the lexical level, at least one character of lookahead is needed, because we operate on a sequence of word (i.e. the character which terminates a token can be the first character of the next token and must be available to the next recognition step).

Example: input is **word1234rest**. If we recognize numbers and word consisting only of letters and blanks are not significant, this sequence consists of 3 tokens.



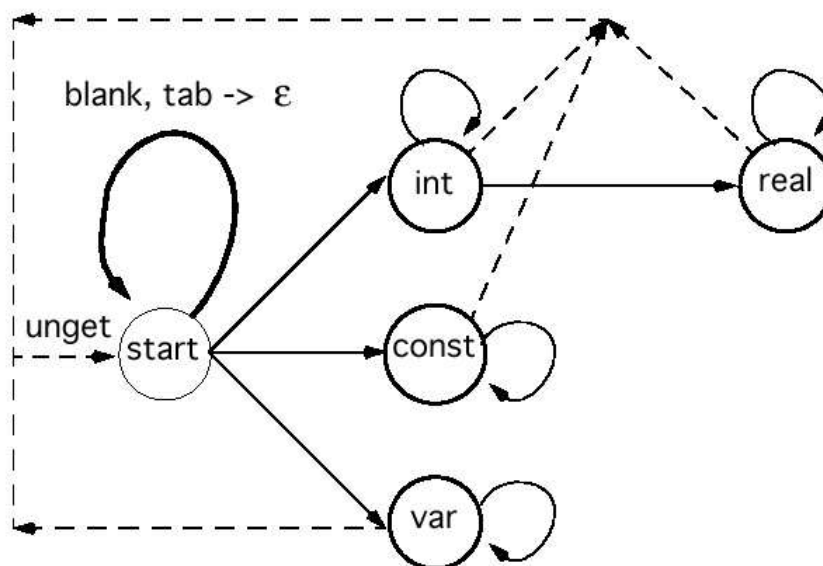
Whitespace Removal

The lexer can remove redundant characters.

Fully redundant characters (e.g. blanks in Fortran) can even be stripped before lexing.

Partially redundant characters (e.g. token-delimiting whitespaces like blanks, tabs, etc. in most languages) have to be removed by the lexer.

As these can only occur between tokens this is trivial:



Scanning in ML

We are now ready to implement a small scanner in ML. Our scanner will recognize integer numbers, real numbers, some operators, variables (only alpha characters, all uppercase) and constant identifiers (only alpha characters, all lowercase).

We declare a type for the different token types as well as some other additional types:

```
exception LEXER_ERROR of string;

datatype statename = ERROR | START | ID | VAR
                  | INT | REAL | OP;

type inputtype = char;

val whitespaces = [" ", "\t", "\n"];

type state = statename * inputtype * statename;

type statetable = state list;
```

Scanning in ML (cont.)

The FSA itself is specified by its final states and transition table

```
val endstates = [VAR, ID, INT, REAL, OP];
```

```
val automaton =  
  [ ( START, #" ", START),  
  
    ( START, #"+", OP),  
    ( START, #"-", OP),  
    ( START, #"*", OP),  
    ( START, #"/", OP),  
  
    ( START, #"A", VAR ),  
    ( START, #"B", VAR ),  
    ( START, #"C", VAR ), ...  
  
    ( VAR, #"A", VAR ),  
    ( VAR, #"B", VAR ),  
    ( VAR, #"C", VAR ), ...  
  
    ( START, #"a", ID ),  
    ( START, #"b", ID ),  
    ( START, #"c", ID ), ...  
  
    ( ID, #"a", ID ),  
    ( ID, #"b", ID ),  
    ( ID, #"c", ID ), ...  
  
    ( START, #"1", INT ),  
    ( START, #"2", INT ), ...  
  
    ( INT, #"1", INT ),  
    ( INT, #"2", INT ), ...  
  
    ( INT, #".", REAL ),  
  
    ( REAL, #"1", REAL ),  
    ( REAL, #"2", REAL ), ...  
  
  ];
```

Scanning in ML (cont.)

We need a function that finds the applicable transition (if any) and another one that checks whether we are in an endstate (if there is no transition for the lookahead character):

```
fun findTransition (state, symbol) [] =
    (false, ERROR)
  | findTransition (x as (thisState, thisSymbol))
    ((state, symbol, follow)::Rest) =
    if thisState=state andalso thisSymbol=symbol
    then (true, follow)
    else findTransition x Rest;

fun checkEndstate stateName chars c =
  let val token = implode(reverse chars) in
    if (member stateName endstates)
      then (stateName, token)
      else raise LEXER_ERROR (str(c)^token)
  end;
```

Scanning in ML (cont.)

With these the complete scanner can be implemented as:

```
fun skip_ws [] = []
  | skip_ws (all as c::cs) = if (member c whitespaces)
                             then (skip_ws cs)
                             else all;

fun lex1 Cs automaton =
  lex2 START nil (skip_ws Cs) automaton
and lex2 currentState soFar (C::Cs) automaton =
  (let val {1=found, 2=followState}
       = findTransition (currentState, C) automaton
   in if found then lex2 followState (C::soFar) Cs
      automaton
      else (checkEndstate currentState soFar C)
          :: (lex1 (C::Cs) automaton)
   end
  handle LEXER_ERROR t =>
    (print("Illegal token: "^t^"\n");
     lex1 Cs automaton)
)
| lex2 currentState soFar [] automaton =
  [(checkEndstate currentState soFar (chr 0))]
  handle LEXER_ERROR t =>
    (print("Unexpected end of input\n"); []);

fun lex S = lex1 (explode S) automaton;
```

Note how redundant whitespace is skipped in the start state.

Scanner Generator

Luckily we don't have to go to this effort everytime that we need a scanner in ML. Instead we can use a scanner generator.

A **scanner generator** automates the process of writing an efficient scanner from a regular expression description of the language tokens.

Basically a scanner generator does the following:

Input: Regular expressions R_1, \dots, R_n describing the tokens of interest T_1, \dots, T_n .

Combine R_1, \dots, R_n into a single regular expression R .

Compute the corresponding NFA for R .

Transform this into a DFA.

Minimize the DFA.

Output: An efficient scanning function based on the minimal DFA.

When the generated scanner is used, it begins with the first character that has not yet been **consumed**.

The generated DFA is then used to process the characters.

The scanner reports that it has found a symbol when it is in a final state and there is no transition using the next character. If there is no transition from the actual state and the actual state is not a final state, it **backtracks** to the last final state it went through. If there is no such state, an error has occurred.

ML Lex

ML Lex is a complete scanner generator in ML in which the required screening functions can be embedded in ML. It is part of the standard ML distribution.

Documentation can also be found at:

<http://www.cs.princeton.edu/~appel/modern/ml/>

ML Lex reads a lexer definition file `*.lex` and produces a standard ML file `*.sml` as output which implements this scanner.

An ML Lex specification has the format

```
user declarations
%% ML-Lex definitions
%% rules
```

The rules are the core part of the specification. Each rule has the form

```
regular expression => ( code );
```

and defines a token type (with a regular expression) and some code to be executed when this token type is recognized.

ML Lex (cont.)

The code part can use some special values: `yypos` and `yylineno` give the current character position and line number. The function `REJECT()` can be called to let the current rule application be rejected (as if it had not matched).

`lex()` is another useful function. It invokes the lexer recursively and can be used to skip whitespace with a rule like:

```
[\\ \\t\\n]+      => ( lex() );
```

Regular expressions in ML Lex

any character except for the special characters

? * + | () ^ \$ / ; . = < > [{ " \.

. stands for any newline character

\b backspace

\n newline

\t tab

\ddd where ddd is a 3 digit decimal escape

[abcde...] denotes exactly one of a, b, c, d, e, \dots

sequences of characters may be enclosed in double quotes

{name} refers to a defined named regular expression (see below)

(r), r*, r+, r1|r2 are the usual regular expression derived from other regular expressions $r, r1, r2$.

r? denotes zero or one occurrences

r{n} denotes exactly n occurrences of r

Declarations in ML Lex

The *user declarations* contain user-defined code and must specify two values: a type `lexresult` (the token types) and a function to be called when the “end of file” is reached.

The *ML-Lex definitions section* can contain various declarations for the lexer generator (see documentation). The most important declarations are:

Regular expressions are defined below. The following commands are also available:

`%reject` to create the `REJECT()` function

and

`%count` to create the function to count newlines using `yylineno`

using either of these slows down the lexer significantly!

In this section named regular expressions can also be defined via

`identifier = regular expression`

An ML Lex example

The following is an ML-Lex definition of a scanner for arithmetic expressions (adapted from the ML-Lex documentation):

```
datatype lexresult= DIV | EOF | EOS | ID of string |
                  LPAREN | NUM of int | PLUS | PRINT |
                  RPAREN | SUB | TIMES

val linenum = ref 1
val error = fn x => print(x ^ "\n")
val eof = fn () => EOF
%%
%structure CalcLex
alpha=[A-Za-z];
digit=[0-9];
ws = [\ \t];
%%
\n      => (linenum := 1 + !linenum; lex());
{ws}+   => (lex());
"/"     => (DIV);
";"     => (EOS);
"("     => (LPAREN);
{digit}+ => (NUM (foldl
                (fn(a,r)=>ord(a)-ord("#0")+10*r) 0
                (explode ytext)));
")"     => (RPAREN);
"+"     => (PLUS);
{alpha}+ => (if ytext="print" then PRINT else ID ytext);
"-"     => (SUB);
"*"     => (TIMES);
```

ML Lex example

To use ML-Lex you must first generate the scanner from its definition:

```
bruce_27%ml-lex calcLex.lex
```

```
Number of states = 18
```

```
Number of distinct rows = 5
```

```
Approx. memory size of trans. table = 645 bytes
```

This generates a file `calcLex.lex.sml` which contains the executable scanner.

Using the Generated Scanner

You can now simply load the generated lexer (with extension `*.lex.sml`)

```
Standard ML of New Jersey ...
val it = () : unit
- use "calcLex.lex.sml";
[opening calcLex.lex.sml]
GC #0.0.0.0.1.18:    (0 ms)
GC #0.0.0.1.2.28:  (10 ms)
structure CalcLex :
  sig
    structure Internal : <sig>
    structure UserDeclarations : <sig>
    exception LexError
    val makeLexer : (int -> string) ->
                    unit -> Internal.result
  end
val it = () : unit
```

To perform the scanning, open a file and generate a lexer. The constructor function is passed a function argument

```
- val inputfile = TextIO.openIn("testfile");
val it = - : TextIO.instream
- val lexer = CalcLex.makeLexer(
    fn n => TextIO.inputLine inputfile);
val lexer = fn : unit -> CalcLex.Internal.result
```

Using the Generated Scanner

We assume the input testfile has contents

```
1 + ( 3 * 4 ) / abc + 2
```

Each successive call of `lexer()` will yield one token

```
- lexer();
val it = NUM 1 : CalcLex.UserDeclarations.lexresult
- lexer();
val it = PLUS : CalcLex.UserDeclarations.lexresult
- lexer();
val it = LPAREN : CalcLex.UserDeclarations.lexresult
- lexer();
val it = NUM 3 : CalcLex.UserDeclarations.lexresult
- lexer();
val it = TIMES : CalcLex.UserDeclarations.lexresult
- lexer();
val it = NUM 4 : CalcLex.UserDeclarations.lexresult
- lexer();
val it = RPAREN : CalcLex.UserDeclarations.lexresult
- lexer();
val it = DIV : CalcLex.UserDeclarations.lexresult
- lexer();
val it = ID "abc" : CalcLex.UserDeclarations.lexresult
- lexer();
val it = PLUS : CalcLex.UserDeclarations.lexresult
- lexer();
val it = NUM 2 : CalcLex.UserDeclarations.lexresult
- lexer();
val it = EOF : CalcLex.UserDeclarations.lexresult
```

Flex

flex is a similar public domain scanner generator for UNIX systems. It works with C instead of ML.

flex provides an interface to C so that the programmer can directly program the desired screening functionality into the scanner itself.

It also has a (large) number of extensions to standard regular expressions to make the task of specification easier.

flex takes a lexical specification and produces a C file `lex.yy.c` which contains the function `yylex` which can be called by other programs such as a parser.

A flex specification has almost the same form as an ML Lex specification (ML Lex was modelled on Flex), but Flex is (still) a bit more powerful and has more flexible expressions.

Summary

We have looked at lexical analysis:

- Regular languages and expressions.
- Nondeterministic and deterministic finite state automata.
- Computing a minimal DFA to recognize a given regular language.
- Implementing a lexer in ML.
- Scanner generators: MLLex and Flex.

Homework

- Read Chapter 7 of Wilhelm and Maurer.
- Give a corresponding NFA to $(a|b)^*abb$. Convert this to a DFA. Now minimize the DFA.
- Read the MLLex documentation at <http://www.cs.princeton.edu/~appel/modern/ml/>.
- Implement a lexer with MLLex that can scan the rules of an MLLex definition file.
- extend the scanner that was given in plain ML in the lectures to call arbitrary ML functions when a token has been recognized. These functions should take the identifier type (`statename`) as a single argument and return a boolean (successful execution). The functions to be called should be specified in a value of type $(state_name * (statename \rightarrow bool)) list$.