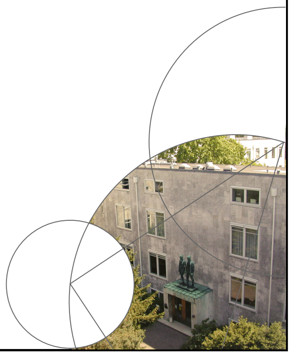
Faculty of Science

Compilers (Oversættere): Lexical Analysis

Jost Berthold
berthold@diku.dk
Department of Computer Science



Lexical Analysis (Scanning): First phase of Frontend

Program text
↓
Lexical analysis
↓
Token sequence
↓
Syntax analysis
↓
Syntax tree
↓
Typecheck
↓
Syntax tree
↓
Intermediate code generation ⟶ Intermediate code
↑
Machine code generation
↑
Symbolic machine code
↑
Register allocation
↑
Ditto with named registers
↑
Assembly and linking
↑
Binary machine code

Lexical

of or relating to words or the vocabulary of a language as distinguished from its grammar and construction.
Merriam-Webster Dictionary, `m-w.com`

- Lexical analyser, also called lexer, scanner or tokeniser.
- splits input (stream of characters) into tokens.
- Token: smallest meaningful unit for a programming language
  Keyword, number, comment, parenthesis, semicolon, . . .
- Tokens are classes of concrete input (called lexeme).

Contents and Goals of this Part

Goals:

- Understand regular expressions and the concept of formal languages, and apply them for lexical analysis
- Use scanner generators for lexical analysis
- Construct, convert, and minimise finite automata
- Know automata limitations and use them as arguments

Lexical analysis – Example

An SML program

```
(* My first
 *  SML program.
 *)
val result
   = let val x = 10 :: 020 :: 0x30 :: []
     in List.map (fn x => x div 2) x
     end
```

Converting textual input into a token sequence

- Input file read as a string
- Contains comments (and preprocessor directives)
- Easy to read with layout/colours.

. . . read by the lexer

```
(*␣My␣first\n␣*␣␣SML␣program.␣\n␣*)\nval␣result\n=let␣val␣x␣=␣10␣::␣020
␣::␣0x30␣::␣[]\nin␣List.map␣(fn␣x␣=>␣x␣div␣2)␣x\nend\n
```

- Machine-read. Formatting only helps human readers.
- Lexical analysis: character stream to token sequence

Lexical analysis – Example

An SML program

```
(* My first
 *  SML program.
 *)
val result
   = let val x = 10 :: 020 :: 0x30 :: []
     in List.map (fn x => x div 2) x
     end
```

Converting textual input into a token sequence

- Input file read as a string
- Contains comments (and preprocessor directives)
- Split into token sequence for machine processing

resulting token sequence

```
[Keyword "val", Id "result", Equal, Keyword "let", Keyword "val", Id "x",
Equal, Int 10, Doublecolon, Int 20, Doublecolon, Int 48, Doublecolon,
LBracket, RBracket, Keyword "in", Id "List", Dot, Id "map", LParen,...
```

- Tokens: classes of input lexemes.
- No comments or formatting (only position for error messages)
- Sometimes: special processing instructions (including files).

Describing Tokens

What are the typical tokens of a programming language?

- Separators and (various) parentheses: `; , { [ ( ) ] } ...`
- String and number literals: `"Hello", 10, 0x30, 3.14, 1.2e-12, ...`
- Operators: $*, +, -, !, ', ||, \&\&, <=, =>, \ldots$
- Identifiers: `x, result, main, map, __a323_int, ...`
  Includes: variable names, user-defined type names, function names, library function and type names
- Keywords: `let, val, fun, fn, end, while, for ...`
  Keywords are reserved identifiers, constructs for statements and declarations.
  Built-in type names are often considered as keywords.

Formal Languages

Building a machine to process words and compile programmes requires formal definitions.

Definition (Formal Language)

Let Σ be an alphabet: a finite set of allowed characters.

- A word over Σ is a string $w = a_1 a_2 \ldots a_n$ of n characters $a_i \in \Sigma$.
  $n = 0$ is allowed, and results in the empty string ε.
  We write Σ^* for the set of all words over Σ.
- A language L over Σ is a set of words over Σ: $L \subset \Sigma^*$

Examples (alphabet: small latin letters)

- Σ^* and $\emptyset$.
- $\{a^n b^n c^n \mid n \in \mathbb{N}\}$
- All C++ keywords: {`if, else, return, do, while, int, char...`}
- All palindromes (words that are the same backward and forward): { `kayak, racecar, mellem, retter, ...`}.

Example Languages: Number Literals in C++

- Integers in decimal format: `123 4 0 8` but not ~~`08`~~ ~~`abc`~~
- Integers in octal format: `0123 07 007` but not ~~`08`~~ ~~`abc`~~
- Integers in hexadecimal format: `0X123 0xCafe` but not ~~`0X`~~ ~~`0XG`~~
- Floating point decimals: `0.   .123   0123.456`
- Scientific notation: `0123E-456   0.E123   .123e+456`

A decimal integer is a sequence of digits 0-9 which does not start by 0 or is only a single 0. An octal integer is a sequence of digits starting with 0, followed by any number of digits 0-7.

Floating-point constants have a “mantissa,” [..][and] an “exponent,” [..] The mantissa is specified as a sequence of digits followed by a period, followed by an optional sequence of digits[..] The exponent, if present, specifies the magnitude[..] using `e` or `E`[..] followed by an optional sign (+ or -) and a sequence of digits. If an exponent is present, the trailing decimal point is unnecessary in whole numbers. `http://msdn.microsoft.com/en-us/library/tfh6f0w2.aspx.`

We need a more formal description for automatic processing.

Regular Expressions

Definition (Regular Expression)

Let Σ be an alphabet of allowed characters.
The set $RE(\Sigma)$ of regular expressions over Σ is defined recursively.

- $\varepsilon \in RE(\Sigma)$: describes the empty word.
- $\mathsf{a} \in RE(\Sigma)$ for $\mathsf{a} \in \Sigma$: describes word `a`.

Furthermore, for every $\alpha, \beta \in RE(\Sigma)$:

- $\alpha \cdot \beta \in RE(\Sigma)$: Sequence, one word described by α, followed by one described by β.
- $\alpha \mid \beta \in RE(\Sigma)$: Alternative, a word described by α or by β.
- $\alpha^* \in RE(\Sigma)$: Repetition, zero or more words described by α.

- Round parentheses $(\ldots)$ for grouping regular expressions.
- Sequence binds tighter than alternative, $\mathsf{a(b|c)} = \mathsf{ab|ac}$

Example Languages: Number Literals in C

- Integers in decimal format: `123 4 0 8` but not ~~`08`~~ ~~`abc`~~
- Integers in octal format: `0123 07 007` but not ~~`08`~~ ~~`abc`~~
- Integers in hexadecimal format: `0X123 0xCafe` but not ~~`0X`~~ ~~`0XG`~~
- Floating point decimals: `0.   .123   0123.456`
- Scientific notation: `0123E-456   0.E123   .123e+456`

- Decimal Numbers: `(1|2|...|9)(0|1|2|...|9)*| 0`
  Shorthand – character range: `[1-9][0-9]*| 0`
- Octal format: `0 [0-7]*`
- Hexadecimal format: `0 ([xX] [0-9a-fA-F][0-9a-fA-F]*`
  Shorthand – at least once: `0 [xX] [0-9a-fA-F]+`
- Floating point numbers: ... (later)

Common Abbreviations for Regular Expressions

- Character Sets
  $[a_1 a_2 \ldots a_n] := (a_1 \mid a_2 \mid \ldots \mid a_n)$
  One of $a_1, \ldots, a_n \in \Sigma$.
- Character Ranges
  $[a_1 - a_n] := (a_1 \mid a_2 \mid \ldots \mid a_n)$ when $\{a_i\}$ is ordered.
  One character in the range between a_1 and a_n.
- Optional Parts
  $\alpha? := (\alpha \mid \varepsilon)$ for $\alpha \in \mathrm{RA}(\Sigma)$.
  Optionally a string described by α.
- Repeated Parts
  $\alpha^+ := \alpha\alpha^*$ for $\alpha \in \mathrm{RA}(\Sigma)$.
  At least one string described by α (maybe more).

  $\alpha\{n,m\} := \overbrace{(\alpha)\ldots(\alpha)}^{n}\overbrace{(\alpha)?\ldots(\alpha)?}^{m-n}$ for $\alpha \in \mathrm{RA}(\Sigma)$, $n < m$.
  A string described by α between n and m times.

Mosml-lex: Generating Lexical Analysis Programs

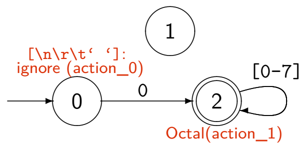
```
{  (* initial part containing SML code *)
   (* helper functions and data types   *)

   data type MyTokens = Decimal of int | Octal of int | ...
   fun decodeInBase (base:int) (s:string) :int = ...
}
let oct = ['0'-'7'] (* a helper definition for reg. expr.s *)
(* Rules section:  regular expr., action (returning a token) *)
rule Token = parse
     '0' oct*      { Octal (decodeInBase 8 (getLexeme lexbuf)) }
   | ..regexp2..   { ...action.2.. (* (SML code) *) }
     ...
;
```

Demo

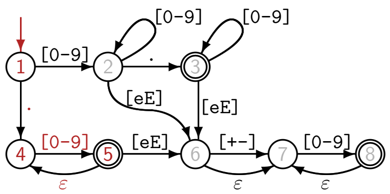
Under the Hood: States and Actions

Octal numbers, from `mosml-lex` code:



- Start in `state_0`:
  Input whitespace: restart (`action_0`)
  Input `0`: go to `state_2`
- When in `state_2`:
  Input `0-7`: continue
  otherwise: do `action_1`
- `state_1`: never reached

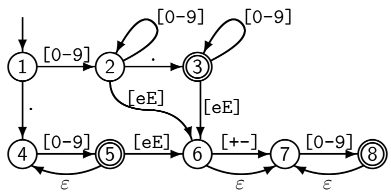
An Automaton to Analyse Input. . .



.31 ⟵ .3ε1. ⟵ .31.4159

- Starting at the pointed state
- Transitions to new states, possibly reading input
- At end of input: check if state "accepting"
- If no transition: stuck, input refused.

An Automaton to Analyse Input. . .



.31 ⟵ . ⟵ 4159

- Starting at the pointed state
- Transitions to new states, possibly reading input
- At end of input: check if state "accepting"
- If no transition: stuck, input refused.

Non-deterministic Finite Automaton – Definition

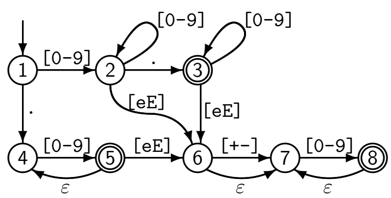
Definition (NFA)

Let Σ be an alphabet of (input) characters.
A Non-deterministic Finite Automaton (NFA) consists of

- A finite set S of states,
- an alphabet Σ of input characters,
- a start state $s_0 \in S$,
- a set of final states $F \subset S$
- and a relation $T \subset S \times (\Sigma \cup \{\varepsilon\}) \times S$ describing state transitions (notation: $s_i \,{}^{c} s_j \in T$)

- Meaning of $s_1 \,{}^{a} s_2 \in T$: in s_1 with input `a`, go to s_2
- Meaning of $s_1 \,{}^{\varepsilon} s_2 \in T$: in s_1, go to s_2.
- Several options may exist, T is a relation.

Example NFA – Formalised



$S = \{1,\ldots,8\}$, $\Sigma = \{0,1\ldots,9,\ .,\ e,\ E\}$, $s_0 = 1$, $F = \{3,5,8\}$

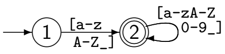
$T = \{1\,{}^{d}2,\ 1\,{}^{.}4,\ 2\,{}^{d}2,\ 2\,{}^{.}3,\ 2\,{}^{e}6,\ 2\,{}^{E}6,\ 3\,{}^{d}3,\ 3\,{}^{e}6,\ 3\,{}^{E}6,$
$4\,{}^{d}5,\ 5\,{}^{\varepsilon}4,\ 5\,{}^{e}6,\ 5\,{}^{E}6,\ 6\,{}^{+}7,\ 6\,{}^{-}7,\ 6\,{}^{\varepsilon}7,\ 7\,{}^{d}8,\ 8\,{}^{\varepsilon}7\}$

$(d \in \{0,1\ldots,9\})$

Picture sufficient as definition – Formalisation: machine-processing.

Other Examples

- Identifiers
  (Σ: letters, digits, underscore)
  Starting with a letter or underscore, then any number of letters, underscores, and digits.




```
[a-zA-Z_][a-zA-Z0-9_]*
```

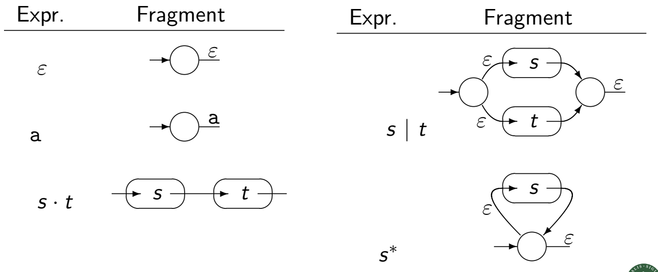
- Binary numbers
  without leading zeros.

$$\Sigma = \{0,1\},\ S = \{0,1,2\}$$
$$s_0 = 0,\ F = \{1,2\}$$
$$T = \{0\,{}^0 1,\ 0\,{}^1 2,\ 2\,{}^0 2,\ 2\,{}^1 2\}$$

```
0 | 1 [01]*
```

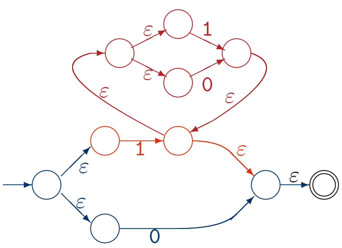

NFA Construction from Regular Expression

- Define an NFA fragment for every regular expression
  Fragments have exactly one entry (arrow) and exit (line)
- Fragment composition follows expression composition
  A single final state is added at the end of the construction.



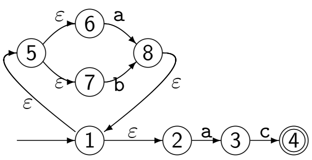
Construction Examples

```
0 | 1 (0 | 1)*
```



Construction Examples

(a|b)*ac



Undesired Non-determinism:

- Many ε transitions (branch and exit for alternatives)
- Undefined transitions when automaton gets stuck
- Several alternatives for same input

Deterministic Finite Automaton – Definition

Definition (DFA)

Let Σ be an alphabet of (input) characters.
A Deterministic Finite Automaton (DFA) consists of

- A finite set S of states,
- an alphabet Σ of input characters,
- a start state $s_0 \in S$,
- a set of final states $F \subset S$
- and a function $\delta : S \times \Sigma \longrightarrow S$ describing state transitions.

- Meaning of $\delta(s_1, \mathrm{a}) = s_2$: in s_1 with input a, go to s_2
- No empty input: always read an input character.
- Only one transition possible, δ is a (partial) function.

Extended Transition Function (whole words)

We define an extended version:
A walk through the DFA reading a whole word $w \in \Sigma^*$ at once.

Definition (Word Transition Function)

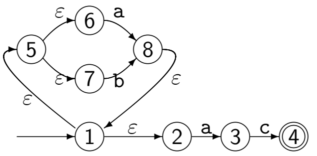
Let $D = (S, \Sigma, s_0, F, \delta)$ a DFA.
The word transition function $\overline{\delta} : S \times \Sigma^* \to S$ of D is defined recursively over words:

1. For the empty word: $\overline{\delta}(s, \varepsilon) = s$
2. For $\mathrm{a} \in \Sigma,\ w \in \Sigma^*$: $\overline{\delta}(s, \mathrm{a}w) = \overline{\delta}(\delta(s, \mathrm{a}), w)$

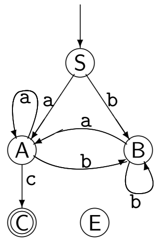
if $\delta(s, \mathrm{a})$ is defined.

Language accepted by the DFA: $\{w \in \Sigma^* \mid \overline{\delta}(s_0, w) \in F\} \subset \Sigma^*$.

Converting an NFA to a DFA: Idea



- States 1,2,5,6,7 reachable from the start state 1.
- With input a, the NFA can go to state 3 and 8.
  On input b, only state 8 possible.
- States 1,2,5,6,7,8 reachable from 8.
- If in state 3, the NFA can go to state 4 on input c (otherwise nowhere).



$S:$ $\{1,2,5,6,7\}$
$A:$ $\{1,2,3,5,6,7,8\}$
$B:$ $\{1,2,5,6,7,8\}$
$C:$ $\{4\}$
$E:$ $\emptyset$

E is an error state.

The Subset Construction: Preparation

To formalise the idea, we first define this reachability.

Definition (ε-Closure $\hat{\varepsilon}(.)$)

Let $N = (S, \Sigma, s_0, F, T)$ a given NFA, and $M \subset S$ a set of states. The ε-Closure of M, written $\hat{\varepsilon}(M)$ contains all states reachable from states in M. It is recursively defined:

1. $M \subset \hat{\varepsilon}(M)$
2. If $s \in \hat{\varepsilon}(M)$, then $\{s' \mid s\,^{\varepsilon}s' \in T\} \subset \hat{\varepsilon}(M)$.

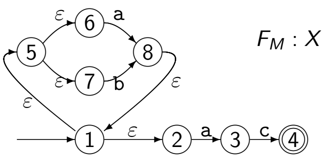
$\hat{\varepsilon}(M)$ is the smallest subset of S that fulfills these conditions.
As a set equation: $X = M \cup \{s' \mid \exists_{s\in X} : s\,^{\varepsilon}s' \in T\}$

Solve this equation by computing a fixed point of F_M:

$$F_M : X \longmapsto \underline{M} \cup \{s' \mid \exists_{s\in X} : s\,^{\varepsilon}s' \in T\}$$

Starting by $X_0 = \emptyset$, compute $X_i = F(X_{i-1})$ until $X_n = F(X_n)$ (works because F is monotonic: $X \subseteq Y \Rightarrow F(X) \subseteq F(Y)$).

ε-Closure $\hat{\varepsilon}(.)$: Example

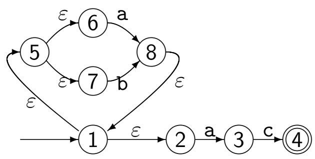


$$F_M : X \longmapsto M \cup \{s' \mid \exists_{s\in X} : s\,^{\varepsilon}s' \in T\}$$

Starting with $X_0 = \emptyset$, compute:
$X_i = F_M(X_{i-1}) = F_M^i(\emptyset)$
... until $X_n = F(X_n)$.

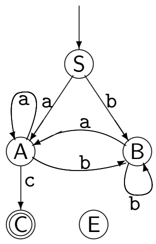
Obviously: $\hat{\varepsilon}(\emptyset) = \emptyset$
$\hat{\varepsilon}(\{1\}) = \underline{\{1,2,5,6,7\}}$
$X_1 = \{1\}, F(X_1) = X_2 = \{1,2,5\},$
$F(X_2) = X_3 = \{1,2,5,6,7\} = F(X_3) = \hat{\varepsilon}(\{1\})$
$\hat{\varepsilon}(\{8\}) = \underline{\{1,2,5,6,7,8\}}$
$\hat{\varepsilon}(\{3,8\}) = \underline{\{1,2,3,5,6,7,8\}}$

ε-Closure $\hat{\varepsilon}(.)$: Example



$\hat{\varepsilon}(1) = S:$ $\{1,2,5,6,7\}$
$\hat{\varepsilon}(3,8) = A:$ $\{1,2,3,5,6,7,8\}$
$\hat{\varepsilon}(8) = B:$ $\{1,2,5,6,7,8\}$
$\hat{\varepsilon}(\emptyset) = E:$ $\emptyset$
More transitions: No new states.
$\hat{\varepsilon}(4) = C:$ $\{4\}$

E is an error state.



The Subset Construction: Definition

Theorem (Subset Construction)

Let $N = (S, \Sigma, s_0, F, T)$ a given NFA.
Define a DFA $D = (S^d, \Sigma, s_0^d, F_d, \delta)$ as follows:

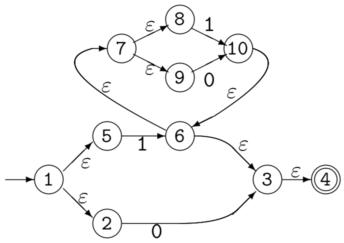
- $S_d = \mathbb{P}(S)$ *(all subsets of S).*
- $s_0^d = \hat{\varepsilon}(\{s_0\})$
- $F_d = \{M \subset S \mid M \cap F \neq \emptyset\}$ *(subsets with a final NFA state).*
- $\delta(s^d, a) = \hat{\varepsilon}(\{t \mid s \in s^d, s\,^{a}t) \in T\})$
  (t reachable from an $s \in s^d$ on input a, and their ε-Closure).

1. *This indeed defines a DFA.*
2. *This DFA D accepts the same language as the NFA N.*

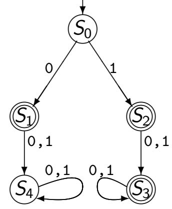
Proof idea: Consider a word $w = a_1a_2\ldots a_n$ accepted by the NFA. There is a state sequence $s_1 \ldots s_n$ such that $(s_{i-1}\,^{a_i}s_i') \in T$ and $s_i \in \hat{\varepsilon}(\{s_i'\})$ and $\hat{\varepsilon}(\{s_n\}) \cap F \neq \emptyset$.
Therefore: $s_1 \in s_1^d = \delta(\hat{\varepsilon}(\{s_0\}), a_1)$, $s_2 \in s_2^d = \delta(s_1^d, a_2)$, $\ldots s_n \in s_n^d = \delta(s_{n-1}^d, a_n)$.
$\Rightarrow$There exists an accepting state sequence $s_1^d s_2^d \ldots s_n^d$ in the DFA (since $s_n^d \in F_d$).

Subset Construction: A Second Example

0 | 1 (0 | 1) *



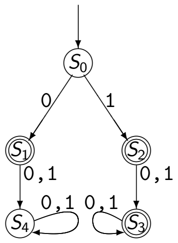
$S_0:$ $\{1,2,5\}$
$S_1:$ $\{3,4\}$
$S_2:$ $\{6,3,7,4,8,9\}$
$S_3:$ $\{10,6,3,7,4,8,9\}$
$S_4:$ $\emptyset$ (error)



Minimising a DFA

The DFAs we obtain from the subset construction are big!
Very often, they contain superfluous states.
Want to optimise the DFA (size and runs).
But, what exactly does "superfluous" mean?

- States that cannot lead to a final state (dead states).
- States that have identical transitions as others.
  More generally: States that lead to the same outcome (acceptance, rejection) for any input.

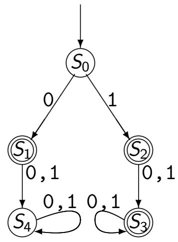


Minimising a DFA: Preparation

Definition (DFA State Equivalence)

Let $D = (S, \Sigma, s_0, F, \delta)$ a DFA.

- A state s is called dead if and only if no final state can be reached from s with any input. Formally:
  s dead $:\Leftrightarrow \bar{\delta}(s, w) \cap F = \emptyset$ for all $w \in \Sigma^*$.
- States s and $s' \in S$ are called equivalent, $s \sim s'$, if and only if both lead to either acceptance or rejection with any input. Formally:
  $s \sim s' :\Leftrightarrow \bar{\delta}(s, w) \in F \Leftrightarrow \bar{\delta}(s', w) \in F$ for all $w \in \Sigma^*$.



$S_2 \sim S_3$
S_4 is a dead state.

Minimising a DFA: Algorithm

We compute the equivalent states backwards from final states.

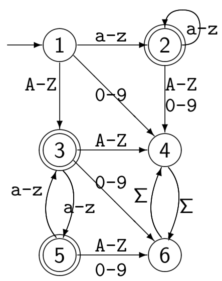
Algorithm (DFA minimisation)

Let $D = (S, \Sigma, s_0, F, \delta)$ a DFA. We assume δ is total.
Determine state equivalence for a minimised DFA as follows:

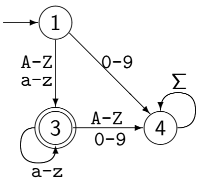
1. Start with two unmarked groups, F and $S \setminus F$.
2. While there are unmarked groups:
   - Pick an unmarked group G.
   - For all $a \in \Sigma$, check for all states $s \in G$ to which group a transition $\delta(s, a)$ leads.
   - If for any respective input a, all transitions lead to the same group: mark the group.
   - Otherwise: Split the group into maximal groups that lead to the same group on transitions and unmark all groups.
3. Repeat from 2 until all groups are marked.

The resulting groups contain equivalent states (all dead states will be equivalent).

Minimising a DFA: Example



Blackboard
$\Longrightarrow$



in the book: no dead states!

Back to our original question. . .

Result so far: Is an input w described by the regular expression α?
Decision problem: for $w \in \Sigma^*$: is w in the language described by $\alpha \in RE(\Sigma)$?

How can we recognise a whole sequence of tokens?

...read by the lexer
```
(*␣My␣first\n␣*␣␣SML␣program.␣\n␣*)\nval␣result\n=let␣val␣x␣=␣10␣::␣020
␣::␣0x30␣::␣[]\nin␣List.map␣(fn␣x␣=>␣x␣div␣2)␣x\nend\n
```

resulting token sequence
```
[Keyword "val", Id "result", Equal, Keyword "let", Keyword "val", Id "x",
Equal, Int 10, Doublecolon, Int 20, Doublecolon, Int 48, Doublecolon,
LBracket, RBracket, Keyword "in", Id "List", Dot, Id "map", LParen,...
```

- Recognise prefixes of input as tokens.
- Restart on remaining input after recognising something.
- Often, several decompositions of the input possible.

Principles of Longest and First Match

Principle of Longest Match

A lexical analyser usually outputs the token that consumes the longest part of the input.

This is important when reading in identifiers and numbers (prefixes could otherwise be recognised instead).

Principle of First Match

Tokens are usually prioritised, so the lexical analyser can decide which token to recognise if two tokens are possible for the same input.

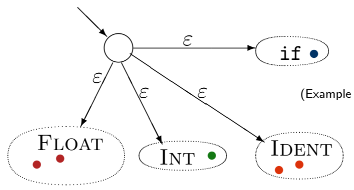
This is especially important when recognising keywords (they could otherwise be recognised as identifiers).

- Define combined NFA with prioritised final states, backtrack.

Lexical Analysis: Putting it All Together

Construction of the automaton:

1. Define an NFA for each token class.
2. Mark final states in each NFA with the respective token name.
3. Combine the NFAs using new start state and ε transitions.
4. Construct a small combined DFA, using subset construction and minimisation. Prioritise token classes in final DFA states to decide what to recognise.

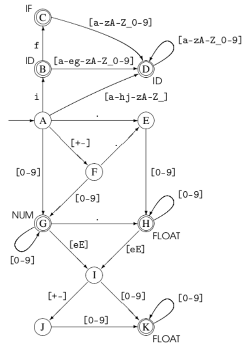


(Example taken from the book)

Lexical Analysis: Putting it All Together (2)

Processing input with the automaton:

1. Start the DFA in normal mode.
2. When reaching a final state: save it, enter read-ahead mode.
3. In read-ahead mode:
   - Buffer all input when reading.
   - When reaching a new final state: clear buffer.
   - End of input or DFA stuck: output last final state, restore input from buffer.
4. Restart in normal mode until input ends.



Example taken from the book

Example: Input 12E.3, recognised as NUM, ID, FLOAT

More About Regular Languages. . .

Regular languages are (by definition) described by regular expressions, but likewise by NFAs, and DFAs. Therefore, we can argue:

- For two regular languages $L_1, L_2 \subset \Sigma^*$, their union $L_1 \cup L_2$ and intersection $L_1 \cap L_2$ are regular.
- For a regular language $L \subset \Sigma^*$, the complement $\Sigma^* \setminus L$ is regular.
- Regular languages are also closed under common string operations: Prefix, Suffix, Subsequence, Reversal.
- The minimised DFA is uniquely determined. Two regular expressions are thus equivalent if their minimised DFAs are the same (apart from renaming states).

Regular languages are limited. Typically, what requires unbounded memory cannot be expressed as a regular language.
Palindromes (`{kayak,racecar,mellem,retter, ...}`) not regular.

Summary

In this part, you have seen

- Formal languages: Sets of words over a finite alphabet.
- Regular expressions, describing regular languages (a subset of all formal languages)
- A compiler tool for lexical analysis (mosmllex)

  . . . and how the tool works internally:
- Deterministic and non-deterministic finite automata

  . . . and how to convert, minimise, and combine them.