50006 - Compilers - Lecture 3

Oliver Killane

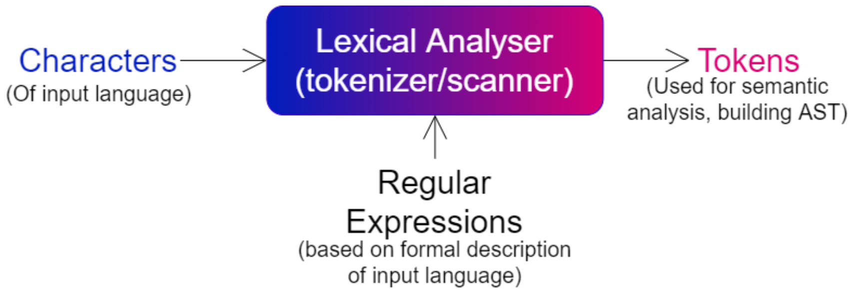
09/01/22

Lecture Recording

Lecture recording is available here

Lexical Analysis

Transforming a stream of characters into a stream of tokens, based on the formal description of the tokens of the input language.



Identifier Tokens

Lexical analyser needs to identify keywords quickly, so often fast string lookup is achieved using a perfect hash function (hash with no possible collisions).

- **Keyword Identifiers**
  Have special meaning in a programming language and are normnally represented by their own token.
  e.g 'class' → CLASS, 'package' → PACKAGE, 'while' → WHILE, etc.
- **Non-Keyword Identifiers**
  Programmer defined identifiers, such as variable names. Typically a generic token is used that uses the provided string name.
  e.g 'var1' → IDENT("var1")

Literal Tokens

Literals (constant values embedded into the input program)

- **Unsigned Integers**
  Represented as a literal token for integers, containing the value used. Tokenizer needs to account for negative integers, as well as varying integer sizes (e.g larger than typical default of 4 bytes) to prevent overflow and correctly assign the value from the input program.
  e.g '123' → INTEGER(123), '1e400' → BIGINTEGER(1e400), '0x11' → INTEGER(17), etc.
- **Unsigned Reals**
  Represented by a literal token for floating-point values, which containg the value. Tokenizer must take into account large and negative floats.
  e.g '17.003' → FLOAT(17.003)
- **Strings**
  Represented by string tokens containing the string (much like non-keyword identifiers). Tokeinzer

Needs to account for input language characters are encoding (e.g unicode, ascii etc), as well as escape characters (backslash)
e.g '"hello, world!"' $\rightarrow$ STRING("hello, world!")

Other Tokens

- **Operators**
  Usually one or two characters, with their own token. e.g $+, -, *, /, ::, <=$
- **Whitespace**
  Normally removed unless inside a string literal. Some information may be included as metadata for later stages of the compiler (e.g for nice error handling). Normally needed to separate identifiers ('pub mod' $\rightarrow$ [PUBLIC, MODULE] but 'pubmod' $\rightarrow$ IDENT("pubmod"))
- **Comments**
  Normally Removed, have no effect on program logic.
- **Pre-processing directives & Macros**
  Most languages remove/process before lexical analysis either by an external tool or an earlier stage of the compiler (e.g pre-processor). One exception to this is **Rust's** macros system, which allows macros to process tokens and is incredibly powerful as a result.

Regular Expressions

Expressions to match strings, cannot be recursive.

a	Match symbol	x matches 'x' only.
$\backslash symbol$	Escape regex char	\(matches '(' only.
ϵ	Match empty string.	ϵ matches '' only.
R_1R_2	Match adjacent.	ab89 matches 'ab89' only.
$R_1 \vert R_2$	Alternation (or), match either regex	$abc \vert 1$ matches 'abc' and '1'.
(R)	Group regexes together	$(a \vert b)c$ matches 'ac' and 'bc'.
$R+$	One or more repetitions of expression R	$(a \vert b \vert j)+$ matches all non-empty strings of a,b & j (e.g 'abbjab'
$R*$	Zero or more repetitions	$R*$ is equivalent to $(R+) \vert \epsilon$

Precedence (highest $\rightarrow$ lowest) grouping, repetition, concatenation, alternation.

The following can be derived from the previous rules.

$R?$	Optional, zero or one occurence	$a?$ matches '' and 'a'.
.	wildcard, match any character	$.*$ matches every possible string.
$[abcd]$	Character Set, match any character in the set	$[abc]$ matches 'a', 'b' and 'c'.
$[0-9]$	Match characters in range	$[a-zA-Z]$ matches all single alphabet character
$[\wedge abc]$	Match all characters except those in the character set.	$[^0-9]*$ matches all strings with no numbers.

These expressions can be used in production rules:

$$\begin{array}{lcl} Digit & \rightarrow & [0-9] \\ Int & \rightarrow & Digit+ \\ Signedint & \rightarrow & (+|-)?Int \\ Keyword & \rightarrow & 'if' \mid 'while' \mid 'do' \\ Letter & \rightarrow & [a-zA-Z] \\ Identifier & \rightarrow & Letter(Letter|Digit)* \end{array}$$

Disambiguation Rules

When more than one expression matches, choose the longest matching character sequence. Otherwise assume regular expression rules are ordered (textual precedence, earlier rule takes precedence).

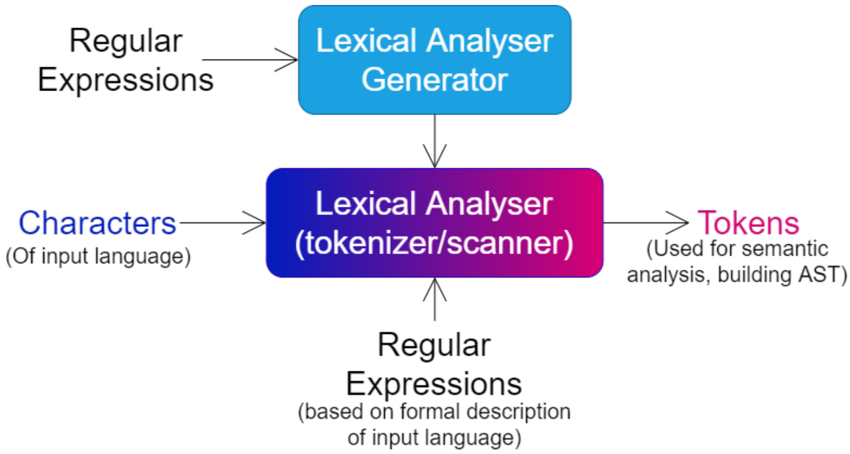
$$Keyword \rightarrow while \mid if \mid do \quad \text{and} \quad Identifier \rightarrow Letter(Letter|Digit)*$$

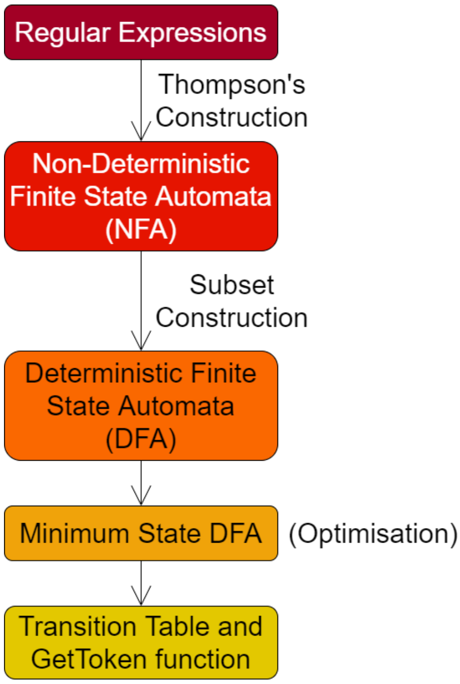
'whileactive' matches $Identifier$ and $Keyword$, however $Identifier$ matches all of the string (longer) so is chosen.

Lexical Analyser Implementation

- **Manually Implement**
  Relatively easy to write from scratch, however can become very difficult to change to change rules for tokens, and many rules implemented in an ad-hoc fashion.
- **Lexical Analyser Generators**
  For example **Logos** which take programmer defined token structures & functions to consume matches specified by regular expressions. They generate a tokenizer convert inputs into the defined tokens.

  An example is included in the code directory of this lecture.

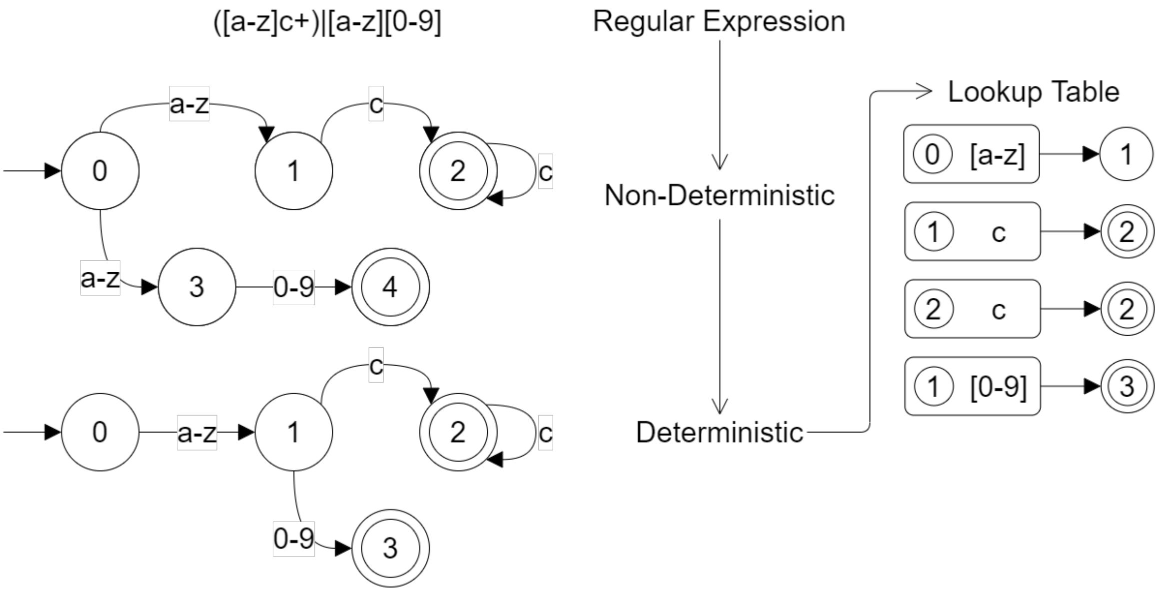




Finite Automata

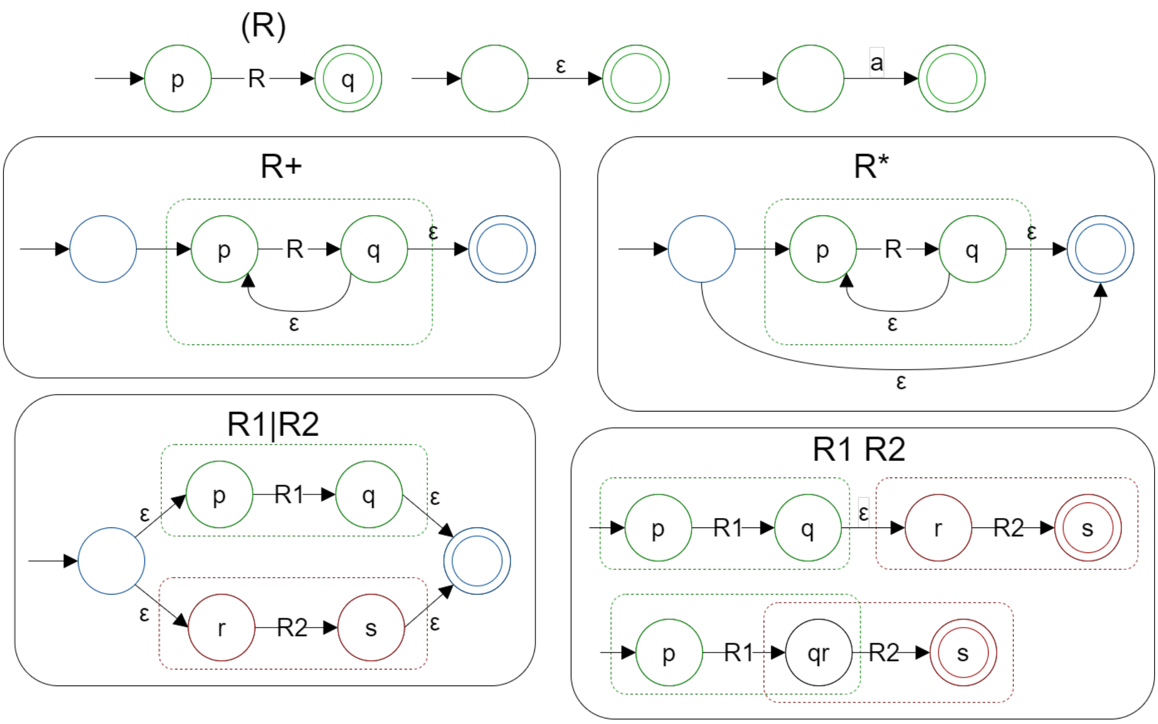
Also called finite state machines.

- Arrows denote treansitions between states.
- Start state has an unlabelled transition to it.
- Accepting states are double circles.
- Technically each non-accepting state should have a transition for every symbol (potentially to an error state), however these are ommitted from the diagram for conciseness.
- Will stop when no transition can be made (as a result matches the longest string possible to a state).



Regular Expressions to Nondeterministic Finite Automata

Thompson's construction is used to translate, it uses ϵ transitions to stick NFAs together.



Subset Construction

- NFA traversal requires backtracking (for a state, input must try every branch for that input).
- Backtracking is slow.
- DFA traversal is faster (no backtracking) so compilers convert to DFA by removing all ϵ transitions instances of multiple paths for an input.
- DFAs often require more memory than NFAs (up to 2^n states for an n state NFA) so some application such as regex searches in some editors use them.

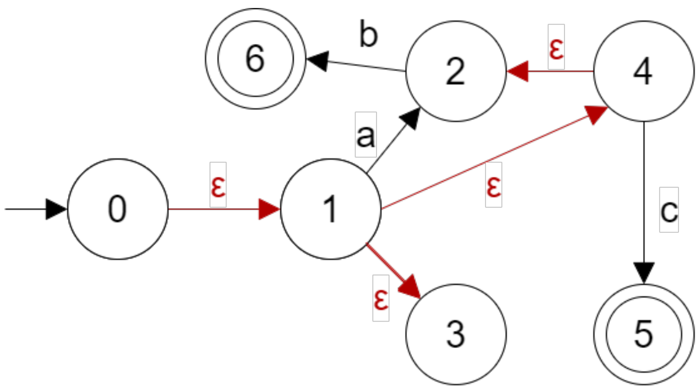
Subset Construction eliminates ϵ transitions, converts to a DFA. States of DFA are subsets of states in the NFA, hence the name.

	Space	**Time**
NFA	$O(len\ R)$	$O(len\ R \times len\ X)*$
DFA	$O(2^{len\ R})$	$O(len\ X)$

It is very rare for the space of the DFA to be an issue with compilers, so they are always used for the increased speed.

ϵ Closures

ϵ-**Closure**(s)	Set of all states that can be reached through only ϵ transitions (including itself).
ϵ-**Closure**$(s_1, \ldots, s_n)$	union of the closures for $s_1, \ldots, s_n$.



$$
\begin{aligned}
\epsilon\text{-Closure}(0) &= \{0,1,2,3,4\} \\
\epsilon\text{-Closure}(1) &= \{1,2,3,4\} \\
\epsilon\text{-Closure}(2) &= \{2\} \\
\epsilon\text{-Closure}(3) &= \{3\} \\
\epsilon\text{-Closure}(4) &= \{2,4\} \\
\epsilon\text{-Closure}(5) &= \{5\}
\end{aligned}
$$

Generating Subset Construction

1. Start at NFA start state.
2. Get the ϵ-Closure (all states the machine could possibly be in)
3. For each possible transition (not including erroneous) create a transition to the possible states (e.g if in the current ϵ-Closure 'a' goes to states 7 and 10, then 'a' should now transition to state $\{7, 10\}$).

4. COntinue step 2 for the states transitions have been created to.

A good example walkthrough can be found here.