Syntactic Structure & Lexical Analysis

Subodh Sharma

S. Arun Kumar

IIT Delhi, Computer Science Department

Language Description

- **Need:** *Clear* and *complete* descriptions are needed by designers, implementers and programmers.

Language Description

- **Need:** *Clear* and *complete* descriptions are needed by designers, implementers and programmers.
  - $DD/DD/DDDD$: 01/02/2021. How do you interpret it?

Language Description

- **Need:** *Clear* and *complete* descriptions are needed by designers, implementers and programmers.
  - $DD/DD/DDDD$: 01/02/2021. How do you interpret it?
- Every language (natural or computer) is a set of strings of *symbols* from some *alphabet*

$$\begin{array}{lcl}
\texttt{nonzero_digit} & \longrightarrow & 1|2|3|4\ldots|9 \\
\texttt{digit} & \longrightarrow & 0|\texttt{nonzero_digit} \\
\texttt{natural_number} & \longrightarrow & \texttt{nonzero_digit}^{\star}
\end{array}$$

Language Description

- **Need:** *Clear* and *complete* descriptions are needed by designers, implementers and programmers.
  - $DD/DD/DDDD$: 01/02/2021. How do you interpret it?
- Every language (natural or computer) is a set of strings of *symbols* from some *alphabet*

  $$\begin{array}{lcl} \texttt{nonzero\_digit} & \longrightarrow & 1|2|3|4\ldots|9 \\ \texttt{digit} & \longrightarrow & 0|\texttt{nonzero\_digit} \\ \texttt{natural\_number} & \longrightarrow & \texttt{nonzero\_digit}^{\star} \end{array}$$

- The string of symbols (or *words*) in a programming language could be: delimiter, expressions, commands, functions, procedures and programs.

Language Description

- **Need:** *Clear* and *complete* descriptions are needed by designers, implementers and programmers.
  - $DD/DD/DDDD$: 01/02/2021. How do you interpret it?
- Every language (natural or computer) is a set of strings of *symbols* from some *alphabet*

$$\begin{array}{lcl} \texttt{nonzero\_digit} & \longrightarrow & 1|2|3|4\ldots|9 \\ \texttt{digit} & \longrightarrow & 0|\texttt{nonzero\_digit} \\ \texttt{natural\_number} & \longrightarrow & \texttt{nonzero\_digit}^{\star} \end{array}$$

- The string of symbols (or *words*) in a programming language could be: delimiter, expressions, commands, functions, procedures and programs.
- ASCII set is one such alphabet for programming languages.

Syntax

- Syntax: Form of program's expressions, statements, etc.

Syntax

- Syntax: Form of program's expressions, statements, etc.
- The smallest units of syntax are called *lexemes*. Eg: numeric literals, operators, special words etc.

Syntax

- Syntax: Form of program's expressions, statements, etc.
- The smallest units of syntax are called *lexemes*. Eg: numeric literals, operators, special words etc.
- Group of lexemes are represented by a name or *token*. Eg: An identifier is a token that can have lexemes such as `sum` or `total`.

Syntax

- Syntax: Form of program's expressions, statements, etc.
- The smallest units of syntax are called *lexemes*. Eg: numeric literals, operators, special words etc.
- Group of lexemes are represented by a name or *token*. Eg: An identifier is a token that can have lexemes such as `sum` or `total`.
- Some tokens have single lexemes. For eg: token for $+$.

Syntax

- Syntax: Form of program's expressions, statements, etc.
- The smallest units of syntax are called *lexemes*. Eg: numeric literals, operators, special words etc.
- Group of lexemes are represented by a name or *token*. Eg: An identifier is a token that can have lexemes such as `sum` or `total`.
- Some tokens have single lexemes. For eg: token for $+$.
- Tokens: Constants, Identifiers, Keywords, Operators, Punctuation, Brackets

Syntax

- Syntax: Form of program's expressions, statements, etc.
- The smallest units of syntax are called *lexemes*. Eg: numeric literals, operators, special words etc.
- Group of lexemes are represented by a name or *token*. Eg: An identifier is a token that can have lexemes such as `sum` or `total`.
- Some tokens have single lexemes. For eg: token for $+$.
- Tokens: Constants, Identifiers, Keywords, Operators, Punctuation, Brackets
- Non-tokens: Whitespaces, comments, preprocessor directives, macros etc.

Syntax: Exercise

```
fun gcd(a:int, b:int): int =
        if b = 0 then a
        else gcd(b, a mod b)
```

- Keywords:
- Identifiers:
- Constants:
- Operators:

Scanning

- Reads as input a stream of symbols and identifies *tokens* from the *lexemes*.
- In the process it removes comments and whitespaces.
- May keep meta information for error reporting (such as line number in the file etc.)
- Eg: b*b - 4*a*c is represented by a token sequence $\mathtt{name}_\mathtt{b} * \mathtt{name}_\mathtt{b} - \mathtt{number}_4 * \mathtt{name}_\mathtt{a} * \mathtt{name}_\mathtt{c}$
- Tokens are constructed from individual characters using just three kinds of formal rules

Scanning:Regular expressions

- Rules for token generation: *concatenation*, *alternation*, *repetition* (or Kleene closure)
  - **Concatenation**: Given two strings x and y, then $x.y$ or simply xy is a concatenation of the two strings. One can apply similar ideas to sets of strings.
  - **Alternation**: Given two x and y, $x|y$ is the set of strings specified by x and y.
  - **Repetition**: For any string x, we may use concatenation to create a string y with as many repetitions of y as we want. Eg: $x^0 = "", x^3 = x.x.x, x^* = \{x^n | n \geq 0\}$.

Scanning:Regular expressions

- Rules for token generation: *concatenation*, *alternation*, *repetition* (or Kleene closure)
  - **Concatenation**: Given two strings x and y, then $x.y$ or simply xy is a concatenation of the two strings. One can apply similar ideas to sets of strings.
  - **Alternation**: Given two x and y, $x|y$ is the set of strings specified by x and y.
  - **Repetition**: For any string x, we may use concatenation to create a string y with as many repetitions of y as we want. Eg: $x^0 = "", x^3 = x.x.x, x^* = \{x^n | n \geq 0\}$.
- Set of strings defined in terms of these rules is called a *regular language* or a regular set.

Scanning:Regular expressions

- Rules for token generation: *concatenation*, *alternation*, *repetition* (or Kleene closure)
  - **Concatenation**: Given two strings x and y, then $x.y$ or simply xy is a concatenation of the two strings. One can apply similar ideas to sets of strings.
  - **Alternation**: Given two x and y, $x|y$ is the set of strings specified by x and y.
  - **Repetition**: For any string x, we may use concatenation to create a string y with as many repetitions of y as we want. Eg: $x^0 = "", x^3 = x.x.x, x^* = \{x^n | n \geq 0\}$.
- Set of strings defined in terms of these rules is called a *regular language* or a regular set.
- Regular sets are generated by *regular expressions* and **recognized** by *scanners*.

Scanning:Regular expressions

- Rules for token generation: *concatenation*, *alternation*, *repetition* (or Kleene closure)
  - **Concatenation**: Given two strings x and y, then $x.y$ or simply xy is a concatenation of the two strings. One can apply similar ideas to sets of strings.
  - **Alternation**: Given two x and y, $x|y$ is the set of strings specified by x and y.
  - **Repetition**: For any string x, we may use concatenation to create a string y with as many repetitions of y as we want. Eg: $x^0 = "", x^3 = x.x.x, x^* = \{x^n | n \geq 0\}$.
- Set of strings defined in terms of these rules is called a *regular language* or a regular set.
- Regular sets are generated by *regular expressions* and **recognized** by *scanners*.
- Regular expressions are *finite descriptions* or specifications of rules to generate regular sets which could be potentially infinite.

Scanning:Regular expressions

- Rules for token generation: *concatenation*, *alternation*, *repetition* (or Kleene closure)
  - **Concatenation**: Given two strings x and y, then $x.y$ or simply xy is a concatenation of the two strings. One can apply similar ideas to sets of strings.
  - **Alternation**: Given two x and y, $x|y$ is the set of strings specified by x and y.
  - **Repetition**: For any string x, we may use concatenation to create a string y with as many repetitions of y as we want. Eg: $x^0 = "", x^3 = x.x.x, x^* = \{x^n | n \geq 0\}$.
- Set of strings defined in terms of these rules is called a *regular language* or a regular set.
- Regular sets are generated by *regular expressions* and **recognized** by *scanners*.
- Regular expressions are *finite descriptions* or specifications of rules to generate regular sets which could be potentially infinite.
- Example: $[A - Za - z][A - Za - z0 - 9]^*$

Simple Language of Regular Expressions

- Assume a (finite) alphabet A of symbols.

Simple Language of Regular Expressions

- Assume a (finite) alphabet A of symbols.
- Each regular expression r denotes a set of strings $\mathcal{L}(r)$. $\mathcal{L}(r)$ is also called the language specified by the regular expression r.
  - Epsilon ϵ denotes the language with a single element the empty string $("")$ i.e. $\mathcal{L}(\epsilon) = \{""\}$.
  - ϵ as a string has zero symbols in it.
  - ϵ acts as *identity* for concatenation.

Simple Language of Regular Expressions

- Assume a (finite) alphabet A of symbols.
- Each regular expression r denotes a set of strings $\mathcal{L}(r)$. $\mathcal{L}(r)$ is also called the language specified by the regular expression r.
  - Epsilon ϵ denotes the language with a single element the empty string ("") i.e. $\mathcal{L}(\epsilon) = \{""\}$.
  - ϵ as a string has zero symbols in it.
  - ϵ acts as *identity* for concatenation.
- **Symbol**: The regular expression a where $a \in \Sigma$ denotes the set $\{a\}$ as $\mathcal{L}(a)$.

Simple Language of Regular Expressions

- Assume a (finite) alphabet A of symbols.
- Each regular expression r denotes a set of strings $\mathcal{L}(r)$. $\mathcal{L}(r)$ is also called the language specified by the regular expression r.
  - Epsilon ϵ denotes the language with a single element the empty string ("") i.e. $\mathcal{L}(\epsilon) = \{""\}$.
  - ϵ as a string has zero symbols in it.
  - ϵ acts as *identity* for concatenation.
- **Symbol**: The regular expression a where $a \in \Sigma$ denotes the set $\{a\}$ as $\mathcal{L}(a)$.
- **Concatenation**: For any two regular expressions r and s, $r.s$ or simply rs denotes the concatenation of the languages specified by r and s. That is,

$$\mathcal{L}(rs) = \mathcal{L}(r)\mathcal{L}(s)$$

Simple Language of Regular Expressions

- Assume a (finite) alphabet A of symbols.
- Each regular expression r denotes a set of strings $\mathcal{L}(r)$. $\mathcal{L}(r)$ is also called the language specified by the regular expression r.
  - Epsilon ϵ denotes the language with a single element the empty string ("") i.e. $\mathcal{L}(\epsilon) = \{""\}$.
  - ϵ as a string has zero symbols in it.
  - ϵ acts as *identity* for concatenation.
- **Symbol**: The regular expression a where $a \in \Sigma$ denotes the set $\{a\}$ as $\mathcal{L}(a)$.
- **Concatenation**: For any two regular expressions r and s, $r.s$ or simply rs denotes the concatenation of the languages specified by r and s. That is,
$$\mathcal{L}(rs) = \mathcal{L}(r)\mathcal{L}(s)$$
- **Alternation**: Given any two regular expressions r and s, $r|s$ is the set union of the languages specified by the individual expressions r and s respectively, i.e. $\mathcal{L}(r|s) = \mathcal{L}(r) \cup \mathcal{L}(s)$.

Simple Language of Regular Expressions

- **Repetition**: For any language X, we may use concatenation to create a another language Y with as many repetitions of the strings in X as we want, by defining repetitions by induction i.e. $X^* = \bigcup_{n \geq 0} X^n$.

Simple Language of Regular Expressions

- **Repetition**: For any language X, we may use concatenation to create a another language Y with as many repetitions of the strings in X as we want, by defining repetitions by induction i.e. $X^* = \bigcup_{n \geq 0} X^n$.
- Some identities:

Simple Language of Regular Expressions

- **Repetition**: For any language X, we may use concatenation to create a another language Y with as many repetitions of the strings in X as we want, by defining repetitions by induction i.e. $X^* = \bigcup_{n \geq 0} X^n$.
- Some identities:
  - : $r^* = \epsilon | r.r^*$, $r^* = (r^*)^*, r^+ = r.r^*$

Scanning: Recognising Tokens

- Identification of tokens is usually done by a *Deterministic Finite-state automaton* (DFA).
- A DFA is specified through a transition graph as shown below and represented as a tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$.

Scanning: Recognising Tokens

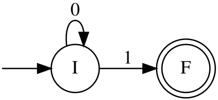
- Identification of tokens is usually done by a *Deterministic Finite-state automaton* (DFA).
- A DFA is specified through a transition graph as shown below and represented as a tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$.
- A DFA D take input strings and accepts a string w if $w \in \mathcal{L}(D)$.
  - Strings are read in the process of a single left-to-right scan.
  - At each step, *at most* one symbol is allowed to be read.
  - The entire stings has to be read before the machine can accept/reject it.

Scanning: Recognising Tokens

- Identification of tokens is usually done by a *Deterministic Finite-state automaton* (DFA).
- A DFA is specified through a transition graph as shown below and represented as a tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$.
- A DFA D take input strings and accepts a string w if $w \in \mathcal{L}(D)$.
  - Strings are read in the process of a single left-to-right scan.
  - At each step, *at most* one symbol is allowed to be read.
  - The entire stings has to be read before the machine can accept/reject it.
- What kind of strings are accepted by the DFA shown below? Can you specify a RE for the same?

Scanning: Recognising Tokens

- Identification of tokens is usually done by a *Deterministic Finite-state automaton* (DFA).
- A DFA is specified through a transition graph as shown below and represented as a tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$.
- A DFA D take input strings and accepts a string w if $w \in \mathcal{L}(D)$.
  - Strings are read in the process of a single left-to-right scan.
  - At each step, *at most* one symbol is allowed to be read.
  - The entire stings has to be read before the machine can accept/reject it.
- What kind of strings are accepted by the DFA shown below? Can you specify a RE for the same?



Language to represent Graphs: Dot

```
digraph G{
/* defaults */
   fontsize=12;
   ratio=compress;
   rankdir=LR;
/* Bounding box */
   size="4,4";

/* Node Definitions */
   I [shape=circle, peripheries=1];
   F [shape=circle, peripheries=2];
   "" [shape=plaintext];

/* Graph defn */
   "" -> I
   I -> I [label="0"];
   I -> F [label="1"];
}

$ dot -Tps dfa.dot > dfa.ps
```

Strengths/Power and Limitations of DFAs

- L is a regular language iff there is a regular expression r such that $L(r) = L$ iff there is a DFA M such that $L(M) = L$.
  - DFAs as scanners: DFAs recognize patterns hidden within strings representing keywords, numbers, etc.

Strengths/Power and Limitations of DFAs

- L is a regular language iff there is a regular expression r such that $L(r) = L$ iff there is a DFA M such that $L(M) = L$.
  - DFAs as scanners: DFAs recognize patterns hidden within strings representing keywords, numbers, etc.
- DFAs can be used to compactly represent Boolean functions or even help determine the validity of formulas in certain branches of mathematic logic.

Strengths/Power and Limitations of DFAs

- L is a regular language iff there is a regular expression r such that $L(r) = L$ iff there is a DFA M such that $L(M) = L$.
  - DFAs as scanners: DFAs recognize patterns hidden within strings representing keywords, numbers, etc.
- DFAs can be used to compactly represent Boolean functions or even help determine the validity of formulas in certain branches of mathematic logic.
- Despite being adequately expressive, they may require too many states, thus adversely affecting space/time requirements of DFA-based algorithms.

Strengths/Power and Limitations of DFAs

- L is a regular language iff there is a regular expression r such that $L(r) = L$ iff there is a DFA M such that $L(M) = L$.
  - DFAs as scanners: DFAs recognize patterns hidden within strings representing keywords, numbers, etc.
- DFAs can be used to compactly represent Boolean functions or even help determine the validity of formulas in certain branches of mathematic logic.
- Despite being adequately expressive, they may require too many states, thus adversely affecting space/time requirements of DFA-based algorithms.
- DFAs are simply inadequate to capture some common occurring patterns, eg: a language of all and only those strings that contain equal number of 0's and 1's.

Strengths/Power and Limitations of DFAs

- L is a regular language iff there is a regular expression r such that $L(r) = L$ iff there is a DFA M such that $L(M) = L$.
  - DFAs as scanners: DFAs recognize patterns hidden within strings representing keywords, numbers, etc.
- DFAs can be used to compactly represent Boolean functions or even help determine the validity of formulas in certain branches of mathematic logic.
- Despite being adequately expressive, they may require too many states, thus adversely affecting space/time requirements of DFA-based algorithms.
- DFAs are simply inadequate to capture some common occurring patterns, eg: a language of all and only those strings that contain equal number of 0's and 1's.
- Question: Is the language $L = \{(^n)^n | n \geq 0\}$ regular?

Recognition Using DFA

Algorithm 1: Recognizer using DFA

Require: A string $w \in A^*$.
Ensure: Boolean

```
S := q_0
a := nextChar(w)
while a ≠ endOfString do
    S := S ∪ δ_S(a)
    a := nextChar(w)
end while
return S ∩ F ≠ ∅
```

Where $\delta_S(a) = \{q' | \exists q \in S : q \xrightarrow{a} q'\}$

An example Lexer

See exlexer.l.

- `lex exlexer.l`.This will produce the file `lex.yy.c`
- `gcc lex.yy.c`. This will produce the executable `a.out`
- Run the executable against a list of strings.