Phases of Syntax Analysis

1. Identify the words: **Lexical Analysis**.
   Converts a stream of characters (input program) into a stream of tokens.
   Also called *Scanning* or *Tokenizing*.

2. Identify the sentences: **Parsing**.
   Derive the structure of sentences: construct *parse trees* from a stream of tokens.

**Lexical Analysis**

Convert a stream of characters into a stream of *tokens*.

- **Simplicity**: Conventions about “words” are often different from conventions about “sentences”.
- **Efficiency**: Word identification problem has a much more efficient solution than sentence identification problem.
- **Portability**: Character set, special characters, device features.

**Terminology**

- **Token**: Name given to a family of words.
  e.g., `integer_constant`
- **Lexeme**: Actual sequence of characters representing a word.
  e.g., `32894`
- **Pattern**: Notation used to identify the set of lexemes represented by a token.
  e.g., `[0−9]`

**Patterns**

How do we *compactly* represent the set of all lexemes corresponding to a token?

For instance:

The token `integer_constant` represents the set of all integers: that is, all sequences of digits (0–9), preceded by an optional sign (+ or −).

Obviously, we cannot simply enumerate all lexemes.

Use **Regular Expressions**.

**Regular Expressions**

Notation to represent (potentially) infinite sets of strings over alphabet Σ.
\begin{itemize}
\item \(a\): stands for the set \(\{a\}\) that contains a single string \(a\).
\item \(a | b\): stands for the set \(\{a, b\}\) that contains two strings \(a\) and \(b\).
  \(\triangleright\) Analogous to *Union*.
\item \(ab\): stands for the set \(\{ab\}\) that contains a single string \(ab\).
  \(\triangleright\) Analogous to *Product*.
\item \((a | b)(a | b)\): stands for the set \(\{aa, ab, ba, bb\}\).
\item \(a^*\): stands for the set \(\{\epsilon, a, aa, aaa, \ldots\}\) that contains all strings of zero or more \(a\)'s.
  \(\triangleright\) Analogous to *closure* of the product operation.
\end{itemize}

**Regular Expressions**

Examples of Regular Expressions over \(\{a, b\}\):

- \((a | b)^*\): Set of strings with zero or more \(a\)'s and zero or more \(b\)'s:
  \(\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}\)
- \((a^*b^*)\): Set of strings with zero or more \(a\)'s and zero or more \(b\)'s such that all \(a\)'s occur before any \(b\):
  \(\{\epsilon, a, b, aa, ab, bb, aaa, aab, abb, \ldots\}\)
- \((a^*b^*)^*\): Set of strings with zero or more \(a\)'s and zero or more \(b\)'s:
  \(\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, abb, \ldots\}\)

**Language of Regular Expressions**

Let \(R\) be the set of all regular expressions over \(\Sigma\). Then,

- Empty String: \(\epsilon \in R\)
- Unit Strings: \(\alpha \in \Sigma \Rightarrow \alpha \in R\)
- Concatenation: \(r_1, r_2 \in R \Rightarrow r_1r_2 \in R\)
- Alternative: \(r_1, r_2 \in R \Rightarrow (r_1 | r_2) \in R\)
- Kleene Closure: \(r \in R \Rightarrow r^* \in R\)

**Regular Expressions**

Example: \((a | b)^*\)

\[
\begin{align*}
L_0 &= \{\epsilon\} \\
L_1 &= L_0 \cdot \{a, b\} \\
&= \{\epsilon\} \cdot \{a, b\} \\
&= \{a, b\} \\
L_2 &= L_1 \cdot \{a, b\} \\
&= \{a, b\} \cdot \{a, b\} \\
&= \{aa, ab, ba, bb\} \\
L_3 &= L_2 \cdot \{a, b\} \\
L &= \bigcup_{i=0}^{\infty} L_i = \{\epsilon, a, b, aa, ab, ba, bb, \ldots\}
\end{align*}
\]
Semantics of Regular Expressions

Semantic Function $\mathcal{L}$: Maps regular expressions to sets of strings.

$$
\begin{align*}
\mathcal{L}(\epsilon) &= \{\epsilon\} \\
\mathcal{L}(\alpha) &= \{\alpha\} \quad (\alpha \in \Sigma) \\
\mathcal{L}(r_1 \mid r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1 \cdot r_2) &= \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \\
\mathcal{L}(r^*) &= \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*))
\end{align*}
$$

Computing the Semantics

$$
\begin{align*}
\mathcal{L}(a) &= \{a\} \\
\mathcal{L}(a \mid b) &= \mathcal{L}(a) \cup \mathcal{L}(b) \\
&= \{a\} \cup \{b\} \\
&= \{a, b\} \\
\mathcal{L}(ab) &= \mathcal{L}(a) \cdot \mathcal{L}(b) \\
&= \{a\} \cdot \{b\} \\
&= \{ab\} \\
\mathcal{L}((a \mid b)(a \mid b)) &= \mathcal{L}(a \mid b) \cdot \mathcal{L}(a \mid b) \\
&= \{a, b\} \cdot \{a, b\} \\
&= \{aa, ab, ba, bb\}
\end{align*}
$$

Computing the Semantics of Closure

Example: $\mathcal{L}((a \mid b)^*)$

$$
\begin{align*}
\mathcal{L}((a \mid b)^*) &= \{\epsilon\} \cup (\mathcal{L}(a \mid b) \cdot \mathcal{L}((a \mid b)^*)) \\
L_0 &= \{\epsilon\} \quad \text{Base case} \\
L_1 &= \{\epsilon\} \cup (\{a, b\} \cdot L_0) \\
&= \{\epsilon\} \cup (\{a, b\} \cdot \{\epsilon\}) \\
&= \{\epsilon, a, b\} \\
L_2 &= \{\epsilon\} \cup (\{a, b\} \cdot L_1) \\
&= \{\epsilon\} \cup (\{a, b\} \cdot \{\epsilon, a, b\}) \\
&= \{\epsilon, a, b, aa, ab, ba, bb\}
\end{align*}
$$

$$
\mathcal{L}((a \mid b)^*) = L_\infty = \{\epsilon, a, b, aa, ab, ba, bb, \ldots\}
$$

Another Example

$\mathcal{L}((a^*b^*)^*) :$
\[ L(a^*) = \{ \epsilon, a, aa, \ldots \} \]
\[ L(b^*) = \{ \epsilon, b, bb, \ldots \} \]
\[ L(a^*b^*) = \{ \epsilon, a, b, aa, ab, bb, \ldots \} \]
\[ L((a^*b^*)^*) = \{ \epsilon \} \]
\[ \cup \{ \epsilon, a, b, aa, ab, bb, \ldots \} \]
\[ \cup \{ \epsilon, a, b, aa, ab, ba, bb, \ldots \} \]
\[ \vdots \]
\[ = \{ \epsilon, a, b, aa, ab, ba, bb, \ldots \} \]

**Regular Definitions**

Assign “names” to regular expressions.
For example,

\[
\text{digit} \rightarrow 0 | 1 | \cdots | 9 \\
\text{natural} \rightarrow \text{digit digit}^* \\
\]

**Shorthands:**
- \( a^+ \): Set of strings with one or more occurrences of \( a \).
- \( a? \): Set of strings with zero or one occurrences of \( a \).

Example:

\[
\text{integer} \rightarrow (\pm)\text{digit}^+ \\
\]

**Regular Definitions: Examples**

- \( \text{float} \rightarrow \text{integer} . \text{fraction} \)
- \( \text{integer} \rightarrow (\pm)^? \text{no_leading_zero} \)
- \( \text{no_leading_zero} \rightarrow (\text{nonzero_digit digit}^*) | 0 \)
- \( \text{fraction} \rightarrow \text{no_trailing_zero exponent}^* \)
- \( \text{no_trailing_zero} \rightarrow (\text{digit}^* \text{nonzero_digit}) | 0 \)
- \( \text{exponent} \rightarrow (E | e) \text{integer} \)
- \( \text{digit} \rightarrow 0 | 1 | \cdots | 9 \)
- \( \text{nonzero_digit} \rightarrow 1 | 2 | \cdots | 9 \)

**Regular Definitions and Lexical Analysis**

Regular Expressions and Definitions specify sets of strings over an input alphabet.

- They can hence be used to specify the set of lexemes associated with a token.
  - Used as the pattern language

How do we decide whether an input string belongs to the set of strings specified by a regular expression?

**Using Regular Definitions for Lexical Analysis**
Q: Is $ababbaabbb$ in $L((a^*b^*)^*)$?
A: Hm. Well. Let’s see.

\[
L((a^*b^*)^*) = \{ \epsilon \} \\
\cup \{ \epsilon, a, b, aa, ab, bb, \\
    aaa, aab, abb, bbb, \ldots \} \\
\cup \{ \epsilon, a, b, aa, ab, ba, bb, \\
    aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots \} \\
\vdots \\
= ???
\]

**Lexical Analysis**

- Regular Expressions and Definitions are used to specify the set of strings (lexemes) corresponding to a *token*.
- An automaton (DFA/NFA) is built from the above specifications.
- Each final state is associated with an *action*: emit the corresponding token.

**Specifying Lexical Analysis**

Consider a recognizer for integers (sequence of digits) and floats (sequence of digits separated by a decimal point).

\[
[0-9]+ \quad \{ \text{emit(INTEGER_CONSTANT); } \} \\
[0-9]+\".\"[0-9]+ \quad \{ \text{emit(FLOAT_CONSTANT); } \}
\]

**Lex**

Tool for building lexical analyzers.
Input: lexical specifications (.l file)
Output: C function (yylex) that returns a token on each invocation.

```
%%
[0-9]+ \{ \text{return(INTEGER_CONSTANT); } \}
[0-9]+\".\"[0-9]+ \{ \text{return(FLOAT_CONSTANT); } \}
```

Tokens are simply integers (#define’s).
C header statements for inclusion

Regular Definitions e.g.:
  digit [0-9]

Token Specifications e.g.:
  \{digit\}+ { return(INTEGER_CONSTANT); }

Support functions in C

---

**Regular Expressions in Lex**

Adds “syntactic sugar” to regular expressions:

- **Range**: [0-7]: Integers from 0 through 7 (inclusive)
  [a-\nx-zA-Q]: Letters a thru n, x thru z and A thru Q.
- **Exception**: [^/]: Any character other than /.
- **Definition**: \{digit\}: Use the previously specified regular definition digit.
- **Special characters**: Connectives of regular expression, convenience features.
  e.g.: | * ^

### Special Characters in Lex

- \* + ? ( ) Same as in regular expressions
- [ ] Enclose ranges and exceptions
- \{ \} Enclose “names” of regular definitions
- ^ Used to negate a specified range (in Exception)
- . Match any single character except newline
- \ Escape the next character
- \n, \t Newline and Tab

For literal matching, enclose special characters in double quotes (") e.g.: "*"

Or use \ to escape. e.g.: \\

### Examples

<table>
<thead>
<tr>
<th>for</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>C-style OR operator (two vert. bars)</td>
</tr>
<tr>
<td>*</td>
<td>Sequence of non-newline characters</td>
</tr>
<tr>
<td>[^*/]+</td>
<td>Sequence of characters except * and /</td>
</tr>
<tr>
<td>&quot;[^&quot;]&quot;</td>
<td>Sequence of non-quote characters beginning and ending with a quote</td>
</tr>
<tr>
<td>{{letter}</td>
<td>__}{{letter}</td>
</tr>
</tbody>
</table>

---

**A Complete Example**
%{
#include <stdio.h>
#include "tokens.h"
%

digit [0-9]
hexdigit [0-9a-f]
%

"+" { return(PLUS); }
"-" { return(MINUS); }
{digit}+ { return(INTEGER_CONSTANT); }
{digit}+"."{digit}+ { return(FLOAT_CONSTANT); }
. { return(SYNTAX_ERROR); }
%

Actions

Actions are attached to final states.

- Distinguish the different final states.
- Used to return tokens.
- Can be used to set attribute values.
- Fragment of C code (blocks enclosed by ‘{‘ and ‘}’).

Attributes

Additional information about a token’s lexeme.

- Stored in variable yylval
- Type of attributes (usually a union) specified by YYSTYPE
- Additional variables:
  - yytext: Lexeme (Actual text string)
  - yyleng: length of string in yytext
  - yylineno: Current line number (number of ‘\n’ seen thus far)
    * enabled by %option yylineno

Priority of matching

What if an input string matches more than one pattern?

- A pattern that matches the longest string is chosen.
  Example: if1 is matched with an identifier, not the keyword if.
- Of patterns that match strings of same length, the first (from the top of file) is chosen.
  Example: while is matched as an identifier, not the keyword while.

Constructing Scanners using (f)lex
• Scanner specifications: *specifications.l*
  
  (f)lex
  
  *specifications.l*  →  *lex.yy.c*

• Generated scanner in *lex.yy.c*
  
  (g)cc
  
  *lex.yy.c*  →  executable

  – yywrap(): hook for signalling end of file.
  – Use `-lfl` (flex) or `-ll` (lex) flags at link time to include default function `yywrap()` that always returns 1.

**Recognizers**

Construct *automata* that recognize strings belonging to a language.

• Finite State Automata ⇒ Regular Languages
  
  ▶ Finite State → cannot maintain arbitrary counts.

• Push Down Automata ⇒ Context-free Languages
  
  ▶ Stack is used to maintain counter, but only one counter can go arbitrarily high.

**Recognizing Finite Sets of Strings**

Identifying words from a small, finite, fixed vocabulary is straightforward.

For instance, consider a stack machine with `push`, `pop`, and `add` operations with two constants: 0 and 1.

We can use the *automaton*:

![Finite State Automata Diagram](image)

**Finite State Automata**

Represented by a labeled directed graph.

• A finite set of *states* (vertices).
• *Transitions* between states (edges).
• *Labels* on transitions are drawn from Σ ∪ {ε}.
• One distinguished *start* state.
• One or more distinguished *final* states.

**Finite State Automata: An Example**
Consider the Regular Expression \((a \mid b)^*a(a \mid b)\).
\(\mathcal{L}((a \mid b)^*a(a \mid b)) = \{aa, ab, aaa, aab, baa, bab, aaaa, aaab, abaa, abab, baaa, \ldots\}\).

The following automaton determines whether an input string belongs to \(\mathcal{L}((a \mid b)^*a(a \mid b))\):

![Automaton Diagram]

**Using States in Lex**

- Some regular languages are more easily expressed as FSA
  - Set of all strings representing binary numbers divisible by 3
- Lex allows you to use FSA concepts using *start states*

```c
%\x MOD1 MOD2
"0" { }
"1" {BEGIN MOD1}
<MOD1> "0" {BEGIN MOD2}
<MOD1> "1" {BEGIN 0}
```

**Other Special Directives**

- **ECHO** causes Lex to echo current lexeme
- **REJECT** causes abandonment of current match in favor of the next.

- Example

```c
a| ab| abc| abcd {ECHO; REJECT;}
.\n{ /* eat up the character */}
```

**Deterministic Vs Nondeterministic FSA**

\((a \mid b)^*a(a \mid b)\):

- **Nondeterministic (NFA)**
- **Deterministic (DFA)**
Acceptance Criterion

A finite state automaton (NFA or DFA) accepts an input string $x$ if beginning from the start state, we can trace some path through the automaton such that the sequence of edge labels spells $x$ and end in a final state.

Recognition with an NFA

Is $abab \in \mathcal{L}((a \ | \ b)^*a(a \ | \ b))$?

Input: $a \ b \ a \ b$

Path 1: $a \ b \ a \ b$
Path 2: $a \ b \ a \ b$
Path 3: $a \ b \ a \ b$

Accept

Recognition with an NFA

Is $abab \in \mathcal{L}((a \ | \ b)^*a(a \ | \ b))$?

Input: $a \ b \ a \ b$

Path 1: $1 \ 1 \ 1$
Path 2: $1 \ 1 \ 1$
Path 3: $1 \ 1 \ 1$

Accept
Is $abab \in L((a | b)^*a(a | b))$?

Recognition with an NFA

Input: a b a b
Path 1: 1 1 1 1 1
Path 2: 1 1 1 2 3 Accept
Path 3: 1 2 3 Accept

Recognition with an NFA

Input: a b a b
Path 1: 1 1 1 1 1
Path 2: 1 1 1 2 3 Accept
Path 3: 1 2 3 Accept

Recognition with an NFA

Input: a b a b
Path 1: 1 1 1 1 1
Path 2: 1 1 1 2 3 Accept
Path 3: 1 2 3 Accept

Recognition with a DFA

Is $abab \in L((a | b)^*a(a | b))$?
For every NFA, there is a DFA that accepts the same set of strings.

- NFA may have transitions labeled by $\epsilon$.
  (Spontaneous transitions)
- All transition labels in a DFA belong to $\Sigma$.
- For some string $x$, there may be many accepting paths in an NFA.
- For all strings $x$, there is one unique accepting path in a DFA.
- Usually, an input string can be recognized faster with a DFA.
- NFAs are typically smaller than the corresponding DFAs.

Regular Expressions to NFA

Thompson’s Construction: For every regular expression $r$, derive an NFA $N(r)$ with unique start and final states.

$\epsilon$

$\alpha \in \Sigma$

$(r_1 | r_2)$

$\epsilon$

$r_1 r_2$

$r^*$
Example

\[(a \mid b)^*a(a \mid b):\]

Recognition with an NFA

Is \(abab \in \mathcal{L}((a \mid b)^*a(a \mid b))\)?

Input:
Path 1: 1 1 1 1 1
Path 2: 1 1 1 2 3  \textbf{Accept}
Path 3: 1 2 3 \perp \perp

All Paths \{1\} \{1,2\} \{1,3\} \{1,2\} \{1,3\}  \textbf{Accept}

Recognition with an NFA (contd.)

Is \(aaab \in \mathcal{L}((a \mid b)^*a(a \mid b))\)?

Input:
Path 1: 1 1 1 1 1 1
Path 2: 1 1 1 1 2
Path 3: 1 1 1 2 3  \textbf{Accept}
Path 4: 1 1 2 3 \perp
Path 5: 1 2 3 \perp \perp

All Paths \{1\} \{1,2\} \{1,2,3\} \{1,2,3\} \{1,2,3\}  \textbf{Accept}

Recognition with an NFA (contd.)

Is \(aabb \in \mathcal{L}((a \mid b)^*a(a \mid b))\)?

Input:
Path 1: 1 1 1 1 1 1
Path 2: 1 1 2 3 \perp
Path 3: 1 2 3 \perp \perp

All Paths \{1\} \{1,2\} \{1,2,3\} \{1,3\} \{1\} \textbf{REJECT}

Converting NFA to DFA

Subset construction
Given a set \(S\) of NFA states,

- compute \(S_\epsilon = \epsilon\text{-closure}(S)\): \(S_\epsilon\) is the set of all NFA states reachable by zero or more \(\epsilon\)-transitions from \(S\).
• compute $S_{\alpha} = \text{goto}(S, \alpha)$:
  
  – $S'$ is the set of all NFA states reachable from $S$ by taking a transition labeled $\alpha$.
  
  – $S_{\alpha} = \epsilon$-closure($S'$).

Converting NFA to DFA (contd).

Each state in DFA corresponds to a set of states in NFA.
Start state of DFA = $\epsilon$-closure(start state of NFA).
From a state $s$ in DFA that corresponds to a set of states $S$ in NFA:
add a transition labeled $\alpha$ to state $s'$ that corresponds to a non-empty $S'$ in NFA,
such that $S' = \text{goto}(S, \alpha)$.

$s$ is a state in DFA such that the corresponding set of states $S$ in NFA contains a final state of NFA,
$\Rightarrow s$ is a final state of DFA

\[
\text{NFA} \rightarrow \text{DFA: An Example}
\]

\[
\begin{align*}
\epsilon\text{-closure}\{1\} & = \{1\} \\
go\{1\}, a & = \{1, 2\} \\
go\{1\}, b & = \{1\} \\
go\{1, 2\}, a & = \{1, 2, 3\} \\
go\{1, 2\}, b & = \{1, 3\} \\
go\{1, 2, 3\}, a & = \{1, 2, 3\} \\
\end{align*}
\]

\[
\text{NFA} \rightarrow \text{DFA: An Example (contd.)}
\]

\[
\begin{align*}
\epsilon\text{-closure}\{1\} & = \{1\} \\
go\{1\}, a & = \{1, 2\} \\
go\{1\}, b & = \{1\} \\
go\{1, 2\}, a & = \{1, 2, 3\} \\
go\{1, 2\}, b & = \{1, 3\} \\
go\{1, 2, 3\}, a & = \{1, 2, 3\} \\
\end{align*}
\]

\[
\text{NFA} \rightarrow \text{DFA: An Example (contd.)}
\]

14
\[
goto(\{1\}, a) = \{1, 2\} \\
goto(\{1\}, b) = \{1\} \\
goto(\{1, 2\}, a) = \{1, 2, 3\} \\
goto(\{1, 2\}, b) = \{1, 3\} \\
goto(\{1, 2, 3\}, a) = \{1, 2, 3\} \\
\]

\[
\begin{array}{|c|c|c|}
\hline
 & \text{NFA} & \text{DFA} \\
\hline
\text{Size of Automaton} & O(R) & O(2^R) \\
\text{Recognition time per input string} & O(N \times R) & O(N) \\
\hline
\end{array}
\]

NFA vs. DFA

\( R = \text{Size of Regular Expression} \)
\( N = \text{Length of Input String} \)

Implementing a Scanner

\[
\text{transition} : \text{state} \times \Sigma \rightarrow \text{state}
\]

\[
\text{algorithm scanner}() \{ \\
\quad \text{current\_state} = \text{start state}; \\
\quad \text{while (1)} \{ \\
\quad\quad c = \text{getc}(); /* \text{on end of file, ... */} \\
\quad\quad \text{if defined(transition(current\_state, c))} \\
\quad\quad\quad \text{current\_state} = \text{transition(current\_state, c)}; \\
\quad\quad\quad \text{else} \\
\quad\quad\quad\quad \text{return s;}; \\
\quad\} \\
\}
\]

Implementing a Scanner (contd.)

Implementing the \text{transition} function:

- Simplest: 2-D array.
  - Space inefficient.
- Traditionally compressed using row/column equivalence. (default on \texttt{flex})
  - Good space-time tradeoff.
- Further table compression using various techniques:
– Example: RDM (Row Displacement Method):
  Store rows in overlapping manner using 2 1-D arrays.
Smaller tables, but longer access times.

**Lexical Analysis: A Summary**

Convert a stream of characters into a stream of tokens.

- Make rest of compiler independent of character set
- Strip off comments
- Recognize line numbers
- Ignore white space characters
- Process macros (definitions and uses)
- Interface with *symbol (name)* table.