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]+`

---

A few more examples:

<table>
<thead>
<tr>
<th>Token</th>
<th>Sample Lexemes</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>while</code></td>
<td><code>while</code></td>
<td><code>while</code></td>
</tr>
<tr>
<td><code>integer_constant</code></td>
<td><code>32894, -1093, 0</code></td>
<td><code>[0-9]+</code></td>
</tr>
<tr>
<td><code>identifier</code></td>
<td><code>buffer_size</code></td>
<td><code>[a-zA-Z]+</code></td>
</tr>
</tbody>
</table>

---

**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.

- **a**: stands for the set `{a}` that contains a single string `a`. 
Analogous to *Union*.

- *ab*: stands for the set \{ab\} that contains a single string `ab`.
  - Analogous to *Product*.
  - \((a|b)(a|b)\): stands for the set \{aa, ab, ba, bb\}.

- \(a^*\): stands for the set \{\(\epsilon, a, aa, aaa, \ldots\)\} that contains all strings of zero or more `a`'s.
  - Analogous to *closure* of the product operation.

### 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:
  \[\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}\]

### 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: \(r_1, r_2 \in R \Rightarrow (r_1 \mid r_2) \in R\)
- Kleene Closure: \(r \in R \Rightarrow r^* \in R\)

### Regular Expressions

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

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

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

\[
\begin{align*}
\mathcal{L}(\varepsilon) &= \{\varepsilon\} \\
\mathcal{L}(\alpha) &= \{\alpha\} \quad (\alpha \in \Sigma) \\
\mathcal{L}(r_1 | r_2) &= \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1 \cdot r_2) &= \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \\
\mathcal{L}(r^*) &= \{\varepsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*))
\end{align*}
\]

Computing the Semantics

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

Computing the Semantics of Closure

Example: $\mathcal{L}((a | b)^*)$
\[
= \{\varepsilon\} \cup (\mathcal{L}(a | b) \cdot \mathcal{L}((a | b)^*))
\]

\[
\begin{align*}
L_0 &= \{\varepsilon\} & \text{Base case} \\
L_1 &= \{\varepsilon\} \cup (\{a, b\} \cdot L_0) \\
&= \{\varepsilon\} \cup (\{a, b\} \cdot \{\varepsilon\}) \\
&= \{\varepsilon, a, b\} \\
L_2 &= \{\varepsilon\} \cup (\{a, b\} \cdot L_1) \\
&= \{\varepsilon\} \cup (\{a, b\} \cdot \{\varepsilon, a, b\}) \\
&= \{\varepsilon, a, b, aa, ab, ba, bb\} \\
&\vdots
\end{align*}
\]

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

Another Example

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

\[
\begin{align*}
\mathcal{L}(a^*) &= \{\varepsilon, a, aa, \ldots\} \\
\mathcal{L}(b^*) &= \{\varepsilon, b, bb, \ldots\} \\
\mathcal{L}(a^*b^*) &= \{\varepsilon, a, b, aa, ab, bb, \\
&\qquad\quad\quad\quad aab, abb, bbb, \ldots\} \\
\mathcal{L}((a^*b^*)^*) &= \{\varepsilon\} \\
&\cup \{\varepsilon, a, b, aa, ab, bb, \\
&\qquad\quad\quad\quad aab, abb, bbb, \ldots\} \\
&\cup \{\varepsilon, a, b, aa, ab, ba, bb, \\
&\qquad\quad\quad\quad aab, aba, abb, baa, bab, bba, bbb, \ldots\} \\
&\vdots
\end{align*}
\]
Regular Definitions

Assign “names” to regular expressions.
For example,

\[
\begin{align*}
\text{digit} & \rightarrow 0 \mid 1 \cdots 9 \\
\text{natural} & \rightarrow \text{digit digit}^* \\
\end{align*}
\]

SHORTHANDS:

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

Example:

\[
\text{integer} \rightarrow (+|−)?\text{digit}^+ \\
\]

Regular Definitions: Examples

\[
\begin{align*}
\text{float} & \rightarrow \text{integer} \cdot \text{fraction} \\
\text{integer} & \rightarrow (+|−)?\text{no_leading_zero} \\
\text{no_leading_zero} & \rightarrow (\text{nonzero_digit} \text{digit}^*) | 0 \\
\text{fraction} & \rightarrow \text{no_trailing_zero} \text{exponent}^* \\
\text{no_trailing_zero} & \rightarrow (\text{digit}^* \text{nonzero_digit}) | 0 \\
\text{exponent} & \rightarrow (E | e) \text{integer} \\
\text{digit} & \rightarrow 0 | 1 \cdots 9 \\
\text{nonzero_digit} & \rightarrow 1 | 2 \cdots 9 \\
\end{align*}
\]

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 \(\text{ababbaabb}\) in \(\mathcal{L}((a^*b^*)^*)\)?
A: Hm. Well. Let’s see.

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

Recognizers

Construct automata that recognize strings belonging to a language.

- Finite State Automata \(\Rightarrow\) Regular Languages
• 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

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 \((a \mid b)^*a(a \mid b)\).
\(L((a \mid b)^*a(a \mid b)) = \{aa, ab, aaa, aab, baa, bab, aaaa, aaab, abaa, abab, baaa, \ldots\}\).
The following automaton determines whether an input string belongs to \(L((a \mid b)^*a(a \mid b))\):

Determinism

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

\[
\begin{array}{c}
\text{Input:} & a & b & a & b \\
\text{Path 1:} & 1 & 1 & 1 & 1 & 1 \\
\text{Path 2:} & 1 & 1 & 1 & 2 & 3 & \text{Accept} \\
\text{Path 3:} & 1 & 2 & 3 & \bot & \bot \\
\end{array}
\]

Accept

Recognition with an NFA

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

\[
\begin{array}{c}
\text{Input:} & a & b & a & b \\
\text{Path 1:} & 1 & 1 & 1 & 1 & 1 \\
\text{Path 2:} & 1 & 1 & 1 & 2 & 3 & \text{Accept} \\
\text{Path 3:} & 1 & 2 & 3 & \bot & \bot \\
\end{array}
\]

Accept

Recognition with a DFA

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

\[
\begin{array}{c}
\text{Input:} & a & b & a & b \\
\text{Path 1:} & 1 & 1 & 1 & 1 & 1 \\
\text{Path 2:} & 1 & 1 & 1 & 2 & 3 & \text{Accept} \\
\text{Path 3:} & 1 & 2 & 3 & \bot & \bot \\
\end{array}
\]

Accept
### NFA vs. DFA

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.

- $\epsilon$
- $\alpha \in \Sigma$
- $r_1 \mid r_2$
- $r^*$

**Example**

$(a \mid b)^*a(a \mid b)$:
Recognition with an NFA

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

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    ⊥    ⊥

All Paths: {1} {1,2} {1,3} {1,2} {1,3}    Accept

Recognition with an NFA (contd.)

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

Input: $a \ a \ a \ b$
Path 1: 1 1 1 1 1
Path 2: 1 1 1 1 2
Path 3: 1 1 1 2 3
Path 4: 1 1 2 3    ⊥
Path 5: 1 2 3    ⊥    ⊥

All Paths: {1} {1,2} {1,2,3} {1,2,3} {1,2,3}    Accept

Recognition with an NFA (contd.)

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

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

All Paths: {1} {1,2} {1,2,3} {1,2,3} {1,2,3}    REJECT

Converting NFA to DFA

Subset construction
Given a set $S$ of NFA states,

- compute $S_\epsilon = \epsilon$-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 $S' = \text{goto}(S, \alpha)$.


$\varepsilon$-closure($\{1\}$) = $\{1\}$

goto($\{1\}, a$) = $\{1, 2\}$

goto($\{1\}, b$) = $\{1\}$

goto($\{1, 2\}, a$) = $\{1, 2, 3\}$

goto($\{1, 2\}, b$) = $\{1, 3\}$

goto($\{1, 2, 3\}, a$) = $\{1, 2, 3\}$

\[\vdots\]

goto($\{1, 2, 3\}$, $b$) = $\{1\}$

goto($\{1, 3\}, a$) = $\{1, 2\}$

goto($\{1, 3\}, b$) = $\{1\}$

\[\vdots\]

$R = \text{Size of Regular Expression}$

$N = \text{Length of Input String}$

<table>
<thead>
<tr>
<th></th>
<th>NFA</th>
<th>DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Automaton</td>
<td>$O(R)$</td>
<td>$O(2^R)$</td>
</tr>
</tbody>
</table>

\[\text{NFA vs. DFA}\]
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]+ { emit(INTEGER_CONSTANT); }
[0-9]+.""[0-9]+ { 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]+ { return(INTEGER_CONSTANT); }
[0-9]+.""[0-9]+ { return(FLOAT_CONSTANT); }
```

Tokens are simply integers (#define’s).

Lex Specifications

```
%
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
• 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: \(\{\text{digit}\}\): Use the previously specified regular definition \(\text{digit}\).
• Special characters: Connectives of regular expression, convenience features.
  e.g.:  
  
  \(|\ast\|^\sim\)

Special Characters in Lex

\(|\ast\ast\ast\|^\sim\)

Same as in regular expressions

[ ]

Enclose ranges and exceptions

\(\{\}\)

Enclose “names” of regular definitions

\(^\sim\)

Used to negate a specified range (in Exception)

\(.\)

Match any single character except newline

\(\n, \t\)

Newline and Tab

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

Or use \(\backslash\) to escape. e.g.: "\n"

Examples

<table>
<thead>
<tr>
<th>for</th>
<th>Sequence of (f, o, r)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>(\ast)</td>
<td>Sequence of non-newline characters</td>
</tr>
<tr>
<td>[^//]+</td>
<td>Sequence of characters except (\ast) 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>({\text{letter}}</td>
<td>&quot;_&quot;{\text{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.
• 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 `yy1val`
• 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?

```
"if" { return(TOKEN_IF); }
{letter}+ { return(TOKEN_ID); }
"while" { return(TOKEN_WHILE); }
```

• 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 `-lf1` (flex) or `-ll` (lex) flags at link time to include default function `yywrap()` that always returns 1.

## Implementing a Scanner

```
transition : state × Σ → state

algorithm scanner() {
  current_state = start state;
  while (1) {
    c = getc(); /* on end of file, ... */
    if defined(transition(current_state, c))
      current_state = transition(current_state, c);
    else
      return s;
  }
}
```
Implementing a Scanner (contd.)

Implementing the transition function:

- Simplest: 2-D array.
  
  Space inefficient.

- Traditionally compressed using row/column equivalence. (default on (f)lex)
  
  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.