THEORY OF COMPILATION

LECTURE 02

Lexical Analysis
You are here

Compiler

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Optimization
Code Generation

Source text

Executable code
The process of generating executable code from source text involves several steps:

1. **Source text input**: The source text is the input to the process.
2. **Lexical Analysis**: Converts the source text into a sequence of tokens.
3. **Syntax Analysis**: Parses the tokens into a syntax tree (AST).
4. **Semantic Analysis**: Checks the AST for semantic errors.
5. **Intermediate code generation**: Generates an intermediate representation (IR) from the AST.
6. **Intermediate code optimization**: Optimizes the IR.
7. **Code generation**: Translates the IR into machine code.
8. **Target code optimization**: Further optimizes the machine code.
9. **Write executable output**: Outputs the final executable code.

This diagram illustrates the flow from source text to executable code, including the stages of lexical, syntactic, and semantic analysis, as well as intermediate and target code generation and optimization.
From Characters to Tokens

• What is a token?
  ‣ Roughly – a “word” in the source language
  ‣ Really – anything that should appear in the input to syntax analysis as a single unit

• Technically
  ‣ Usually a pair of \langle kind, value \rangle
# Example Tokens

<table>
<thead>
<tr>
<th>Kind</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>x, y, z0, foo, bar</td>
</tr>
<tr>
<td>NUM</td>
<td>42</td>
</tr>
<tr>
<td>FLOATNUM</td>
<td>3.141592654</td>
</tr>
<tr>
<td>STRING</td>
<td>&quot;so long, and thanks for all the fish&quot;</td>
</tr>
<tr>
<td>LPAREN</td>
<td>(</td>
</tr>
<tr>
<td>RPAREN</td>
<td>)</td>
</tr>
<tr>
<td>IF</td>
<td>if</td>
</tr>
<tr>
<td>MINUS</td>
<td>−</td>
</tr>
</tbody>
</table>
## Strings with Special Handling

<table>
<thead>
<tr>
<th>Kind</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments</td>
<td>/* Ceci n'est pas un commentaire */</td>
</tr>
<tr>
<td>Preprocessor directives</td>
<td>#include &lt;foo.h&gt;</td>
</tr>
<tr>
<td>Macros</td>
<td>#define THE_ANSWER 42</td>
</tr>
<tr>
<td>White spaces</td>
<td>\t \r\n</td>
</tr>
</tbody>
</table>
Strings with Special Handling

<table>
<thead>
<tr>
<th>Kind</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comments</td>
<td>/* Ceci n'est pas un commentaire */</td>
</tr>
<tr>
<td>Preprocessor directives</td>
<td><code>#include &lt;foo.h&gt;</code></td>
</tr>
<tr>
<td>Macros</td>
<td><code>#define THE_ANSWER 42</code></td>
</tr>
<tr>
<td>White spaces</td>
<td>\t \r \n</td>
</tr>
</tbody>
</table>

Symbols:
- \t: tab
- \r: carriage return
- \n: line feed
Some basic terminology

- **Lexeme** (aka symbol) – a series of characters separated from the rest of the program according to a convention (space, semicolon, word boundary, ...)

- **Pattern** – a rule specifying a set of strings. Example: “an identifier is a string that starts with a letter, followed by letters and digits”

- **Token** – a pair of ⟨pattern, attributes⟩
From Characters to Tokens

\[ x = b^2 - 4ac \]

Token Stream

\[ \langle \text{ID,"x"} \rangle \langle \text{EQ} \rangle \langle \text{ID,"b"} \rangle \langle \text{MULT} \rangle \langle \text{ID,"b"} \rangle \langle \text{MINUS} \rangle \langle \text{INT,4} \rangle \langle \text{MULT} \rangle \langle \text{ID,"a"} \rangle \langle \text{MULT} \rangle \langle \text{ID,"c"} \rangle \]
Errors in Lexical Analysis

\[
\text{pi} = 3.141.562
\]

Illegal token

\[
\text{pi} = 3\text{oranges}
\]

Illegal token

\[
\text{pi} = \text{oranges3}
\]

\[
\langle \text{ID,"pi"}, \langle \text{EQ} \rangle, \langle \text{ID,"oranges3"} \rangle \rangle
\]
Error Handling

• Many errors cannot be identified at this stage
• Example: should “fi” be “if”? Or is it a routine name?
  - We will discover this later in the analysis
  - At this point, we just create an identifier token

• Sometimes the lexeme does not match any pattern
  - Easiest fix: skip characters until the beginning of a legitimate lexeme
  - Alternatives: eliminate/add/replace one letter, replace order of two adjacent letters, etc.

• Pro: allow the compilation to continue
• Con: errors that spread all over
How can we define tokens?

• Keywords — easy!
  ‣ *if, then, else, for, while, ...*

• Identifiers?
• Numerical Values?
• Strings?

• Characterize **unbounded** sets of values using a **bounded** description?
How can we define tokens?

- Keywords — easy!
  - \textit{if}, \textit{then}, \textit{else}, \textit{for}, \textit{while}, ... \\

- Identifiers?
  - $x_0$, $b_2b$, \textit{heIsNoOne}, ...

- Numerical Values?

- Strings?

- Characterize \textit{unbounded} sets of values using a \textit{bounded} description?
How can we define tokens?

• Keywords — easy!
  ‣ if, then, else, for, while, ...

• Identifiers?
• Numerical Values?
• Strings?

• Characterize **unbounded** sets of values using a **bounded** description?
How can we define tokens?

- Keywords — easy!
  - `if, then, else, for, while, ...`

- Identifiers?
  - `x0, b2b, heIsNoOne, ...`

- Numerical Values?
  - `0, 1, 2, 3.4, 5.67, ...`

- Strings?
  - "there are infinitely many", ...

- Characterize **unbounded** sets of values using a **bounded** description?
## Regular Expressions over $\Sigma$

<table>
<thead>
<tr>
<th>Basic Patterns</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>A single letter ‘x’ from the alphabet $\Sigma$</td>
</tr>
<tr>
<td>.</td>
<td>Any character (usually excluding a new-line)</td>
</tr>
<tr>
<td>[xyz]</td>
<td>Any of the characters x,y,z</td>
</tr>
</tbody>
</table>

### Repetition Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R?$</td>
<td>An $R$ or nothing (= optionally an $R$)</td>
</tr>
<tr>
<td>$R^*$</td>
<td>Zero or more occurrences of $R$</td>
</tr>
<tr>
<td>$R+$</td>
<td>One or more occurrences of $R$</td>
</tr>
</tbody>
</table>

### Composition Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_1R_2$</td>
<td>An $R_1$ followed by $R_2$</td>
</tr>
<tr>
<td>$R_1</td>
<td>R_2$</td>
</tr>
</tbody>
</table>

### Grouping

<table>
<thead>
<tr>
<th>Grouping</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R)</td>
<td>$R$ itself (used to override default precedence)</td>
</tr>
</tbody>
</table>
Examples

• \((a|b)^* =\)

• \((0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+ =\)

• \(ab^* | cd? =\)
Escape characters

• What is the expression for one or more “+” symbols?
  – (+) + won’t work
  – (\+) + will

• backslash \ before an operator turns it into a standard character

• \*, \?, \+, ...
Shorthands

• Use names for expressions
  ‣ letter = a | b | ... | z | A | B | ... | Z
  ‣ letter_ = letter | _
  ‣ digit = 0 | 1 | 2 | ... | 9
  ‣ id = letter_ (letter_ | digit)*

• Use hyphen in groups to denote a range
  ‣ letter = [a-z] | [A-Z]
  ‣ digit = [0-9]
Examples

- if = if
- then = then
- relop = < | > | <= | >= | = | <>

- digit = [0-9]
- digits = digit+

- ?? = digits ( \. digits ( e (\+|-)? digits )? )?
Examples

• if = if
• then = then
• relop = < | > | <= | >= | = | <>

• digit = [0-9]
• digits = digit+

• number = digits ( \. digits ( e (\+|\-)? digits )? )?
Ambiguity

• \texttt{if} = \texttt{if}
• \texttt{id} = \texttt{letter\_} (\texttt{letter\_} \mid \texttt{digit})*

• “\texttt{if}” matches both the pattern for reserved words and the pattern for identifiers... so what should it be?
• How about the identifier “\texttt{iffy}”?  

• Solution
  – Always find \textit{longest matching token}
  – Break ties using \textit{order of definitions}; first definition wins
    \textit{(⇒ list rules for keywords before identifiers)}
Creating a Lexical Analyzer

• Input
  ‣ List of token definitions (pattern name, regex)
  ‣ String to be analyzed

• Output
  ‣ List of tokens

• How do we build an analyzer?
Character Classification

#define is_end_of_input(ch) ((ch) == '\0')
#define is_uc_letter(ch) ('A' <= (ch) && (ch) <= 'Z')
#define is_lc_letter(ch) ('a' <= (ch) && (ch) <= 'z')
#define is_letter(ch) (is_uc_letter(ch) || is_lc_letter(ch))
#define is_digit(ch) ('0' <= (ch) && (ch) <= '9')
...

Main reading routine

```c
void get_next_token() {
    do {
        char c = getchar();
        if is_letter(c)
            return recognize_identifier(c);
        else if is_digit(c)
            return recognize_number(c);
        ...
    } while (!is_end_of_input(c));
}
```
• Generate a lexical analyzer automatically from token definitions

• Main idea
  ‣ Use finite-state automata to match regular expressions

But we have a much better way!
Overview

• Construct a **nondeterministic finite-state automaton** (NFA) from regular expression (automatically)

• Determinize the NFA into a **deterministic finite-state automaton** (DFA)

• DFA can be directly used to identify tokens
Reminder: Finite-State Automaton

Deterministic automaton

- $M = (\Sigma, Q, \delta, q_0, F)$
  - $\Sigma$ – alphabet
  - $Q$ – finite set of state
  - $q_0 \in Q$ – initial state
  - $F \subseteq Q$ – final states
  - $\delta : Q \times \Sigma \rightarrow Q$ – transition function
Non-Deterministic automaton

- \( M = (\Sigma, Q, \delta, q_0, F) \)
  - \( \Sigma \) - alphabet
  - \( Q \) – finite set of states
  - \( q_0 \in Q \) – initial state
  - \( F \subseteq Q \) – final states
  - \( \delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q \) – transition function

- Allows \( \varepsilon \)-transitions
- For a word \( w \), \( M \) can reach a number of states or get stuck. If some state reached is final, \( M \) accepts \( w \).
Non-Deterministic automaton

- $M = (\Sigma, Q, \delta, q_0, F)$
  - $\Sigma$ - alphabet
  - $Q$ – finite set of state
  - $q_0 \in Q$ – initial state
  - $F \subseteq Q$ – final states
  - $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ – transition function

- Allows $\varepsilon$-transitions
- For a word $w$, $M$ can reach a number of states or get stuck. If some state reached is final, $M$ accepts $w$. 
From Regular Expressions to NFA

- Step 1: assign expression names and obtain pure regular expressions $R_1...R_m$
- Step 2: construct an NFA $M_i$ for each regular expression $R_i$
- Step 3: combine all $M_i$ into a single NFA

- Ambiguity resolution: prefer longest accepting word
Basic constructs

\[ R = \epsilon \]

\[ R = a \]

\[ R = \phi \]
Composition

\[ R = R_1 \mid R_2 \]

\[ R = R_1R_2 \]
Repetition

\[ R = R_1^* \]
What now?

• We have one automaton per pattern. Naïve approach: try each automaton separately

• Given a word w:
  ‣ Try $M_1(w)$
  ‣ Try $M_2(w)$
  ‣ ...
  ‣ Try $M_n(w)$

• Requires “rewinding” after every attempt
Combine Automata

`combines`

- `a`
- `abb`
- `a*b+`
- `abab`

```
ε
```

```
a
```

```
abb
```

```
a*b+
```

```
abab
```

```
a
```

```
a
```

```
abb
```

```
a
```

```
abab
```
Ambiguity resolution

• Recall...
  ✴ Longest word
  ✴ Tie-breaker based on order of rules when words have same length

• Recipe
  ‣ Turn NFA to DFA
  ‣ Run until stuck, remember last accepting state, this is the token to be returned
Corresponding DFA

```
0 1 3 7 9
2 4 7 10
5 8 11
12
13
```

```
0 → 7 (a)
7 → 0 (b)
7 → 8 (b)
8 → 7 (b)
8 → 6 (a)
6 → 8 (b)
6 → 12 (a*bb+)
```

```
1 → 2 (a)
2 → 3 (a)
3 → 4 (b)
4 → 5 (b)
5 → 6 (abb)
5 → 9 (a*bb+)
9 → 10 (b)
10 → 11 (a)
11 → 12 (b)
12 → 13 (abab)
```

```
ε → 0 (ε)
ε → 4 (ε)
ε → 8 (ε)
ε → 12 (ε)
```
Example Inputs

**aba**: gets stuck after **aba** in state 12, backs up to state (5 8 11) pattern is a*b+, token is **ab**

**abba**: stops after second **b** in (6 8), pattern is abb because it comes first in spec
Summary of Construction

✓ Describe tokens as regular expressions
✓ Regular expressions turned into a DFA
✓ Lexical analyzer simulates the run of an automata with the given transition table on any input string
Good News

• Construction is done automatically by common tools
• **Flex** is your friend
  ‣ Automatically generates a lexical analyzer from declaration file
• Advantages: short declaration file, easily checked, easily modified and maintained

Intuitively:
• **Flex** builds DFA table
• Analyzer simulates (runs) the DFA on a given input
Flex declarations file

{%
#include <math.h>
int line_number = 1;
%

WS     [ \t]
LETTER [a-zA-Z]
DIGIT  [0-9]
ID     {LETTER}({LETTER}|{DIGIT})*

%%

{DIGIT}+     { printf("number: %d\n", atoi(yytext)); return 1; }
{ID}         { return 2;}
{WS}         { /* ignore */ }
\n     { line_number++; /* and ignore */ }
.            { return -1; /* ERROR */ }

%%

int main() { return yylex(); }
Summary

- **Lexical analyzer**
  - ✓ Turns character stream into token stream
  - ✓ Tokens defined using regular expressions
  - ✓ Regex $\rightarrow$ NFA $\rightarrow$ DFA construction for identifying tokens
  - ✓ Automated constructions of lexical analyzer using Flex
Coming Up

Syntax
Analysis