Theory of Compilation 236360

Erez Petrank

Lecture 1:
Introduction, Lexical Analysis
Theory of Compilation

• Lecturer: Erez Petrank
  – erez@cs.technion.ac.il
  – Reception hour: Thursday 14:30—15:30, Taub 528.

• Teaching assistants:
  – Yuri Meshman (responsible TA)
  – Adi Sosnovich
  – Idan Schwartz
Administration

- Site: http://webcourse.cs.technion.ac.il/236360
- Grade:
  - 25% homework (important!)
    - 5% dry (MAGEN)
    - 20% wet: compulsory
  - 75% Test
- Failure in test means failure in course, independent of homework grade.
- Prerequisite: Automata and formal languages 236353
- MOED GIMMEL for Miluim only
- ... העתקות
Books

- **Main book**

- **Additional book:**
Goals

• Understand what a compiler is,
• How a compiler works,
• Tools and techniques that can be used in other settings.
Complexity

- Complexity
- Methodology
- Algorithmics
- Engineering
- Computer Architecture
- Computational Complexity

• Computer Models are more meaningful for
• Masculine Thinking and Inward
• Masculine Thinking and Practical

7
What is a Compiler?

• “A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language). The most common reason for wanting to transform source code is to create an executable program.”

--Wikipedia
“The Education of a Computer” 1952
Grace Hopper 1906 -- 1992

Image source:
“The Education of a Computer”  
(Hopper 1952)
1957

John Backus and team at IBM

The first complete compiler
What is a Compiler?

source language

C
C++
Pascal
Java
Postscript
Perl
JavaScript
Python
Ruby
Prolog
Lisp
Scheme
ML
OCaml

target language

IA32
IA64
SPARC
C
C++
Pascal
Java code
Java Bytecode
...

Compiler

"I THINK YOU SHOULD BE MORE EXPlicit HERE IN STEP TWO."
What is a Compiler?

- The source and target program are semantically equivalent.
- Since the translation is difficult, it is partitioned into standard modular steps.

```
int a, b;
   a = 2;
 b = a*2 + 1;
```

```
MOV R1,2
SAL R1
INC R1
MOV R2,R1
```
Anatomy of a Compiler
(Coarse Grained)

Compiler

Source text

Frontend (analysis)
Intermediate Representation
Backend (synthesis)

Executable code

int a, b;
a = 2;
b = a*2 + 1;

MOV R1,2
SAL R1
INC R1
MOV R2,R1
Modularity

Build m+n modules instead of m•n compilers…

int a, b;
a = 2;
b = a*2 + 1;

SET R1,2
STORE #0,R1
SHIFT R1,1
STORE #1,R1
ADD R1,1
STORE #2,R1

MOV R1,2
SAL R1
INC R1
MOV R2,R1
Anatomy of a Compiler

Compiler

Source text

Executable code

Frontend (analysis)

Intermediate Representation

Backend (synthesis)

int a, b;
a = 2;
b = a*2 + 1;

Lexical Analysis

Syntax Analysis

Parsing

Semantic Analysis

Intermediate Representation (IR)

Code Generation

MOV R1,2
SAL R1
INC R1
MOV R2,R1
int a, b;
a = 2;
b = a*2 + 1;
Compiler vs. Interpreter

1. **Source text** (txt) → **Frontend (analysis)** → **Intermediate Representation** → **Backend (synthesis)** → **Executable code** (exe)

2. **Source text** (txt) → **Frontend (analysis)** → **Intermediate Representation** → **Execution Engine** → **Output**
Compiler vs. Interpreter

b = a*2 + 1;

Frontend (analysis) ➔ Intermediate Representation ➔ Backend (synthesis)

MOV R1,8(ebp)
SAL R1
INC R1
MOV R2,R1

b = a*2 + 1;

Frontend (analysis) ➔ Intermediate Representation ➔ Execution Engine

3 ➔ 7

7 ➔ 3
Just-in-time Compiler (e.g., Java)

Just-in-time compilation: bytecode interpreter (in the JVM) compiles program fragments during interpretation to avoid expensive re-interpretation.
Importance

• Many in this class will build a parser some day
  – Or wish they knew how to build one…
• Useful techniques and algorithms
  – Lexical analysis / parsing
  – Intermediate representation
  – …
  – Register allocation
• Understand programming languages better
• Understand internals of compilers
• Understand compilation versus runtime,
• Understand how the compiler treats the program (how to improve the efficiency, how to use error messages),
• Understand (some) details of target architectures
## Turing Award

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1966</td>
<td>Alan Perlis</td>
<td>Compiler construction</td>
</tr>
<tr>
<td>1967</td>
<td>Maurice Wilkes</td>
<td>EDSAC Computer</td>
</tr>
<tr>
<td>1968</td>
<td>Richard Hamming</td>
<td>Information Theory</td>
</tr>
<tr>
<td>1969</td>
<td>Marvin Minsky</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>1970</td>
<td>James Wilkinson</td>
<td>Numerical Analysis</td>
</tr>
<tr>
<td>1971</td>
<td>John McCarthy</td>
<td>Lisp</td>
</tr>
<tr>
<td>1972</td>
<td>Edsger Dijkstra</td>
<td>Algol, Science of programming</td>
</tr>
<tr>
<td>1973</td>
<td>Charles Bachman</td>
<td>Database Technology</td>
</tr>
<tr>
<td>1974</td>
<td>Donald Knuth</td>
<td>The Art of Computer Programming</td>
</tr>
<tr>
<td>1975</td>
<td>Newell &amp; Simon</td>
<td>AI &amp; Cognition</td>
</tr>
<tr>
<td>1976</td>
<td>Rabin &amp; Scott</td>
<td>Automata Theory</td>
</tr>
<tr>
<td>1977</td>
<td>John Backus</td>
<td>Fortran, Functional Programming</td>
</tr>
<tr>
<td>1978</td>
<td>Bob Floyd</td>
<td>Parsing, Semantics, Verification</td>
</tr>
<tr>
<td>1979</td>
<td>Ken Iverson</td>
<td>APL</td>
</tr>
<tr>
<td>1980</td>
<td>Tony Hoare</td>
<td>Definition &amp; design of languages</td>
</tr>
<tr>
<td>1981</td>
<td>Edgar Codd</td>
<td>Relational Databases</td>
</tr>
<tr>
<td>1982</td>
<td>Stephen Cook</td>
<td>Complexity of Computation</td>
</tr>
<tr>
<td>1983</td>
<td>Thompson &amp; Ritchie</td>
<td>Unix (also C)</td>
</tr>
<tr>
<td>1984</td>
<td>Niklaus Wirth</td>
<td>Algol-W, Pascal, Modula</td>
</tr>
<tr>
<td>1985</td>
<td>Dick Karp</td>
<td>Theory of NP-Completeness</td>
</tr>
<tr>
<td>1986</td>
<td>Hopcroft &amp; Tarjan</td>
<td>Algorithms &amp; Data Structures</td>
</tr>
<tr>
<td>1988</td>
<td>John Cocke</td>
<td>Compilers &amp; RISC Architecture</td>
</tr>
<tr>
<td>1989</td>
<td>Ivan Sutherland</td>
<td>Computer Graphics</td>
</tr>
<tr>
<td>1990</td>
<td>Velvel Kahan</td>
<td>Numerical Analysis, IEEE FP</td>
</tr>
<tr>
<td>1991</td>
<td>Fernando Corbato</td>
<td>Operating Systems, Timesharing (CTSS &amp; Multics)</td>
</tr>
<tr>
<td>1992</td>
<td>Butler Lampson</td>
<td>Workstations</td>
</tr>
<tr>
<td>1993</td>
<td>Hartmanis &amp; Steams</td>
<td>Computational Complexity</td>
</tr>
<tr>
<td>1994</td>
<td>Feigenbaum &amp; Reddy</td>
<td>Large-scale AI</td>
</tr>
<tr>
<td>1995</td>
<td>Manuel Blum</td>
<td>Complexity &amp; Cryptography</td>
</tr>
<tr>
<td>1996</td>
<td>Amir Pnueli</td>
<td>Temporal Logic</td>
</tr>
<tr>
<td>1997</td>
<td>Doug Engelbart</td>
<td>Interactive Computing</td>
</tr>
<tr>
<td>1998</td>
<td>Jim Gray</td>
<td>Transaction Processing</td>
</tr>
<tr>
<td>1999</td>
<td>Fred Brooks</td>
<td>Software Engineering</td>
</tr>
<tr>
<td>2000</td>
<td>Andrew Yao</td>
<td>Complexity-based Theory</td>
</tr>
<tr>
<td>2001</td>
<td>Dahl &amp; Nygaard</td>
<td>Object-oriented Programming</td>
</tr>
<tr>
<td>2002</td>
<td>R-S-A</td>
<td>Public-Key Cryptography</td>
</tr>
<tr>
<td>2003</td>
<td>Alan Kay</td>
<td>Smalltalk</td>
</tr>
<tr>
<td>2004</td>
<td>Cerf &amp; Kahn</td>
<td>Internetworking</td>
</tr>
<tr>
<td>2005</td>
<td>Peter Naur</td>
<td>Languages, Compilers, ALGOL-60</td>
</tr>
<tr>
<td>2006</td>
<td>Fran Allen</td>
<td>Optimizing Compilers</td>
</tr>
<tr>
<td>2007</td>
<td>C-E-S</td>
<td>Model Checking</td>
</tr>
</tbody>
</table>

2008: Barbara Liskov, programming languages and system design.

2009: Charles P Thacker, architecture.

2010: Leslie Valiant, theory of computation.

2011: Judea Pearl, artificial intelligence.

2012: Goldwasser & Micali, Cryptography

2013: Leslie Lamport, verification of distributed systems.
Complexity:
Various areas, theory and practice.

- Useful formalisms
  - Regular expressions
  - Context-free grammars
  - Attribute grammars

- Data structures

- Algorithms

Source

Programing Languages
Software Engineering

Compiler

Target

Operating systems
Runtime environment
Garbage collection
Architecture
Course Overview

Compiler

Lexical Analysis

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

Token Stream:

\[
<\text{ID,"x"}> \ <\text{EQ}> \ <\text{ID,"b"}> \ <\text{MULT}> \ <\text{INT,4}> \ <\text{MULT}> \ <\text{ID,"a"}> \ <\text{MULT}> \ <\text{ID,"c"}> 
\]
Syntax Analysis (Parsing)

\[
<\text{ID,"}b\text{"}> \ <\text{MULT}> \ <\text{ID,"}b\text{"}> \ <\text{MINUS}> \ <\text{INT,}4\text{}> \ <\text{MULT}> \ <\text{ID,"}a\text{"}> \ <\text{MULT}> \ <\text{ID,"}c\text{"}>
\]

Syntax Tree

```
expression
  /\  
expression    MINUS    term
  /\         /\        /\   
term        term    term
  /\      /\      /\     /\   
term  MULT  factor  term MULT factor
     /\    /\     /\     /\   
ID  'b' ID     ID     ID     'c'
```

Semantic Analysis

Annotated Abstract Syntax Tree

```
MULT
   \b\n   MULT
     |  \b  |
     | type: int |
     | loc: R1   |
     | type: int |
     | loc: sp+16|
     | 'b'      |
     | type: int |
     | loc: const|
     | '4'      |
     | type: int |
     | loc: R1   |
     | type: int |
     | loc: R1   |
     | type: int |
     | loc: R2   |
     | type: int |
     | loc: R2   |
     | 'c'      |
     | type: int |
     | loc: sp+24|
     | 'a'      |
     | type: int |
     | loc: sp+8|
```
Intermediate Representation

R2 = 4*a
R1 = b*b
R2 = R2*c
R1 = R1 - R2
**Generating Code**

Intermediate Representation

\[
\begin{align*}
R2 &= 4 \cdot a \\
R1 &= b \cdot b \\
R2 &= R2 \cdot c \\
R1 &= R1 - R2
\end{align*}
\]

Assembly Code

\[
\begin{align*}
\text{MOV R2, (sp+8)} \\
\text{SAL R2, 2} \\
\text{MOV R1, (sp+16)} \\
\text{MUL R1, (sp+16)} \\
\text{MUL R2, (sp+24)} \\
\text{SUB R1, R2}
\end{align*}
\]
The Symbol Table

• A data structure that holds attributes for each identifier, and provides fast access to them.
• Example: location in memory, type, scope.
• The table is built during the compilation process.
  – E.g., the lexical analysis cannot tell what the type is, the location in memory is discovered only during code generation, etc.
Error Checking

• Done at each stage.

• Lexical analysis: illegal tokens
• Syntax analysis: illegal syntax
• Semantic analysis: incompatible types, undefined variables, ...

• Each phase tries to recover and proceed with compilation (why?)
Errors in lexical analysis

- `pi = 3.141.562` → Illegal token
- `pi = 3oranges` → Illegal token
- `pi = oranges3` → `<ID,"pi">, <EQ>, <ID,"oranges3">`
Error detection: type checking

\[
x = 4*a*"oranges"
\]
The Real Anatomy of a Compiler

- **Source text**
- **Process text input**
- **Lexical Analysis**
  - characters → tokens
- **Syntax Analysis**
  - AST
- **Sem. Analysis**
- **Intermediate code generation**
- **Intermediate code optimization**
- **Code generation**
- **Target code optimization**
- **Machine code generation**
- **Write executable output**
- **Executable code**
- **exe**
Optimizations

• “Optimal code” is out of reach
  – many problems are undecidable or too expensive (NP-complete)
  – Use approximation and/or heuristics
  – Must preserve correctness, should (mostly) improve code, should run fast.

• Improvements in time, space, energy, etc.

• This part takes most of the compilation time.

• A major question: how much time should be invested in optimization to make the code run faster.
  – Answer changes for the JIT setting.
Optimization Examples

- Loop optimizations: invariants, unrolling, ...
- Peephole optimizations
- Constant propagation
  - Leverage compile-time information to save work at runtime (pre-computation)
- Dead code elimination
  - space
- ...

40
Modern Optimization Challenges

• Main challenge is to exploit modern platforms
  – Multicores
  – Vector instructions
  – Memory hierarchy
  – Is the code sent on the net (Java byte-code)?
Machine code generation

- A major goal: determine location of variables.
- Register allocation
  - Optimal register assignment is NP-Complete
  - In practice, known heuristics perform well
- assign variables to memory locations
- Instruction selection
  - Convert IR to actual machine instructions

- Modern architectures
  - Multicores
  - Challenging memory hierarchies
Compiler Construction Toolset

• Lexical analysis generators
  – lex
• Parser generators
  – yacc
• Syntax-directed translators
• Dataflow analysis engines
Summary

• A compiler is a **program that translates** code from **source language** to **target language**

• **Compilers play a critical role**
  – Bridge from programming languages to the machine
  – Many useful techniques and algorithms
  – Many useful tools (e.g., lexer/parser generators)

• **Compiler are constructed from modular phases**
  – Reusable
  – Debug-able, understandable.
  – Different front/back ends
Theory of Compilation

Lexical Analysis
You are here

Compiler

Source text

Lexical Analysis

Syntax Analysis

Semantic Analysis

Inter. Rep. (IR)

Code Gen.

Executable code

txt

exe
From characters to tokens

• What is a token?
  – Roughly - a “word” in the source language
  – Identifiers
  – Values
  – Language keywords
  – (Really - anything that should appear in the input to syntax analysis)

• Technically
  – A token is a pair of (kind,value)
## Example: kinds of tokens

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>\textit{x, y, z, foo, bar}</td>
</tr>
<tr>
<td>NUM</td>
<td>42</td>
</tr>
<tr>
<td>FLOATNUM</td>
<td>\textit{3.141592654}</td>
</tr>
<tr>
<td>STRING</td>
<td>\textit{“so long, and thanks for all the fish”}</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>...</td>
<td></td>
</tr>
</tbody>
</table>
## Strings with special handling

<table>
<thead>
<tr>
<th>Type</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 \n</td>
</tr>
</tbody>
</table>
The Terminology

• **Lexeme** (aka symbol): a series of letters separated from the rest of the program according to a convention (space, semi-column, comma, etc.)

• **Pattern**: a rule specifying a set of strings. Example: “an identifier is a string that starts with a letter and continues with letters and digits”.

• **Token**: a pair of (pattern, attributes)
From characters to tokens

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

Token Stream

\[
\text{<ID,"x"> <EQ> <ID,"b"> <MULT> <ID,"b"> <MINUS> \\
<INT,4> <MULT> <ID,"a"> <MULT> <ID,"c">}
\]
Errors in lexical analysis

- pi = 3.141.562
  - Illegal token
- pi = 3oranges
  - Illegal token
- pi = oranges3
  - \(<\text{ID,"pi"}, \text{EQ}, \text{ID,"oranges3"}>\)
Error Handling

• Many errors cannot be identified at this stage.
• For example: “fi (a==f(x))”. 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.
• But sometimes the lexeme does not satisfy any pattern.
  – Easiest: eliminate letters until the beginning of a legitimate lexeme.
  – Alternatives: eliminate one letter, add one letter, replace one letter, replace order of two adjacent letters, etc.

• Goal: allow the compilation to continue.
• Problem: errors that spread all over.
How can we define tokens?

• Keywords - easy!
  – if, then, else, for, while, ...

• We need a precise, formal description (that a machine can understand) of things like:
  – Identifiers
  – Numerical Values
  – Strings

• Solution: regular expressions.
# Regular Expressions over $\Sigma$

<table>
<thead>
<tr>
<th>Basic Patterns</th>
<th>Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>No string</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>The empty string</td>
</tr>
<tr>
<td>$a$</td>
<td>A single letter 'a' in $\Sigma$</td>
</tr>
</tbody>
</table>

**Repetition Operators**

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

**Grouping**

<table>
<thead>
<tr>
<th>(R)</th>
<th>R itself</th>
</tr>
</thead>
</table>
Examples

- \( ab^* | cd? = \)
- \( (a|b)^* = \)
- \( (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^* = \)
Simplifications

- **Precedence**: * is of highest priority, then concatenation, and then unification.
  
  \[ a \mid (b\ (c)^\ast) = a \mid bc^\ast \]

- ‘R?’ stands for R or ε.
- ‘.’ stands for any character in \( \Sigma \).
- \([xyz]\) for letters x, y, z in \( \Sigma \) means \( x\mid y\mid z \)
- **Use hyphen** to denote a range
  - letter = a-z | A-Z
  - digit = 0-9
More Simplifications

• Assign names for expressions:
  - letter = a | b | ... | z | A | B | ... | Z
  - letter_ = letter | _
  - digit = 0 | 1 | 2 | ... | 9
  - id = letter_ (letter_ | digit)*
**An Example**

- A number is
  \[(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+\]
  \[(\varepsilon \mid .)(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+\]
  \[(\varepsilon \mid E(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^+\)
  \]

- Using simplifications it is:
  - digit = 0-9
  - digits = digit+
  - number = digits \((\varepsilon \mid .)\) digits \((\varepsilon \mid e (\varepsilon|+|-) \) digits \))
Additional (Practical) Examples

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

• What is the expression for one or more '+' symbols coming after the letter 'C'?
  – C++ won’t work
  – C(\+)+ will

• backslash \ before an operator turns it to standard character

• \*, \?, \+, ...
Ambiguity

• Consider the two definitions:
  – if = if
  – id = letter_ (letter_ | digit)*

• The string “if” is valid for the pattern if and also for the pattern id... so what should it be?

• How about the string “iffy”?
  – Is it an identifier? Or an “if” followed by the identifier “fy”?

• Convention:
  – Always find longest matching token
  – Break ties using order of definitions... first definition wins (=> list rules for keywords before identifiers)
Creating a lexical analyzer

• **Input**
  - List of token definitions (pattern name, regular-expression)
  - 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();
        switch(c) {
            case is_letter(c): return recognize_identifier(c);
            case is_digit(c): return recognize_number(c);
            ...
        }
    } while (c != EOF);
}
```
But we have a much better way!

• *Generate a lexical analyzer automatically* from token definitions

• **Main idea**
  – Use finite-state automata to match regular expressions
Overview

• Produce an finite-state automaton from a regular expression (automatically).
• Simulate a final-state automaton on a computer (automatically).
• Immediately obtain a lexical analyzer.

• Let’s recall material from the Automata course...
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
  - \( \delta : Q \times \Sigma \to Q \) - transition function
Reminder: Finite-State Automaton

• 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

• Possible \( \varepsilon \)-transitions
• For a word \( w \), \( M \) can reach a number of states or get stuck. If some reached state is final, \( M \) accepts \( w \).
Identifying Patterns = Lexical Analysis

• Step 1: remove shortcuts and obtain pure regular expressions $R_1 \ldots R_m$ for the $m$ patterns.

• Step 2: construct an NFA $M_i$ for each regular expression $R_i$

• Step 3: combine all $M_i$ into a single NFA

• Step 4: convert the NFA into a DFA

• DFA is ready to identify the patterns.

• Ambiguity resolution: prefer longest accepting word
A Comment

• In the Automata course you study of automata as identifiers only: is input in the language or not?
• But when you run an automaton on an input there is no reason to not gather information along the way.
  – E.g., letters read from input so far, line number in the code, etc.
Building NFA: Basic constructs

- $R = \varepsilon$
- $R = a$
- $R = \phi$
The starting and final states in the original automata become regular states after the composition.
Building NFA: Repetition

$R = R1^*$
Use of Automata

- Naïve approach: try each automaton separately
- Given a word \( w \):
  - Try \( M_1(w) \)
  - Try \( M_2(w) \)
  - ...
  - Try \( M_n(w) \)

- Requires resetting after every attempt.
- A more efficient method: combine all automata into one.
Combine automata: an example.

Combine $a$, $abb$, $a^*b^+$, $abab$. 

Diagram:

- Start state 0
- Transition from 0 to 1 on $\epsilon$
- Transition from 1 to 2 on $a$
- Transition from 2 to 0 on $a$
- Transition from 0 to 3 on $\epsilon$
- Transition from 3 to 4 on $a$
- Transition from 4 to 5 on $b$
- Transition from 5 to 6 on $b$
- Transition from 6 to abb
- Transition from 6 to 13 on $\epsilon$
- Transition from 0 to 7 on $\epsilon$
- Transition from 7 to 8 on $b$
- Transition from 8 to 0 on $\epsilon$
- Transition from 8 to 9 on $a^*b^+$
- Transition from 9 to 10 on $a$
- Transition from 10 to 11 on $b$
- Transition from 11 to 12 on $a$
- Transition from 12 to 13 on $b$
- Transition from 12 to 13 on $\epsilon$

States:
- 0: Start state
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
Ambiguity resolution

• Longest word
• Tie-breaker based on order of rules when words have same length.
  – Namely, if an accepting state has two labels then we can select one of them according to the rule priorities.

• Recipe
  – Turn NFA to DFA
  – Run until stuck, remember last accepting state, this is the token to be returned.
Example: \((a|b)^*|(cbb)\)

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

```
(a|b)^*
```

```
cb
```

```
cb
```

```
cb
```
Example (continued)

(a|b)*|(cbb)
Now let’s return to the previous example…
Corresponding DFA
abaa: 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), token is abb because it comes first in spec
Summary of Construction

• Developer describes the tokens as regular expressions (and decides which attributes are saved for each token).

• The regular expressions are turned into a deterministic automata (a transition table) that describes the expressions and specifies which attributes to keep.

• The lexical analyzer simulates the run of an automata with the given transition table on any input string.
Good News

- All of this construction is done automatically for you by common tools.
- **Lex** automatically generates a lexical analyzer from declaration file.
- Advantages: a short declaration, easily checked, easily modified and maintained.

Intuitively:
- Lex builds a DFA table,
- The analyzer simulates the DFA on a given input.
Summary

- Lexical analyzer
  - Turns character stream into token stream
  - Tokens defined using regular expressions
  - Regular expressions $\rightarrow$ NFA $\rightarrow$ DFA construction for identifying tokens
  - Automated constructions of lexical analyzer using lex

Lex will be presented in the exercise.