Announcements

- We have moved!
  (Surprised me too...)

- Essay is due.

- Look for HW2 tomorrow.
Compilers

The process by which programming languages are turned into assembly or machine code is important in programming languages. We'll spend some time on these compilers, although it isn't a focus of this U class.
Compilers

Compilers are essentially translators, so must semantically understand the code.

Output: either assembly, machine code, some other output.

Java → bytecode

C++ → assembler
Compilers begin by preprocessing:
- remove white space and comments
- include macros or libraries
- group characters into tokens
  ex: `for (int i = 0; i < n; i++) {}`
- identify high-level syntactical structures
  ex: `...`
Overview of Compilation

Character stream
Token stream
Parse tree
Abstract syntax tree or other intermediate form
Modified intermediate form
Target language (e.g., assembler)
Modified target language

Scanner (lexical analysis)
Parser (syntax analysis)
Semantic analysis and intermediate code generation
Machine-independent code improvement (optional)
Target code generation
Machine-specific code improvement (optional)

Symbol table

Front end
Back end
The steps:

Front end:

A) Scanner
B) Parser
C) Semantic Analysis

Let's dive into these first...
Scanning (lexical analysis)

- Divide program into tokens, or smallest meaningful units
  Ex: keywords, &, |, +, variables, etc.

- Scanning & tokenizing makes parsing much simpler.

- While parsers can work character by character, it is slow.

- Note: Scanning is recognizing a regular language, e.g. via NFA
Parsing

- Recognizing a context-free language, e.g. via PDA
- Finds the structure of the program (or the syntax)

Example:

```
iteration-statement →
  while (expression) statement

statement → compound-statement
```

Outputs a parse tree.
Semantic Analysis (after parsing)
This discovers the meaning of the commands.
Actually only does static semantic analysis, consisting of all that is known at compile time.

(Some things - e.g., array out of bounds - are unknown until run time.)

→ error generation
Ex: (semantic analysis)
- Variables can't be used before being declared. (C-like)
- Type checking.
- Identifiers are used in proper context.
- Functions have correct inputs & returns.
  etc... (very language dependent)
Intermediate Form

This is the output of the "front end"

- Often, this is an abstract syntax tree - a simplified version of a parse tree
- May also be a type of assembly-like code

<table>
<thead>
<tr>
<th>index</th>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>void</td>
<td>type</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>type</td>
</tr>
<tr>
<td>3</td>
<td>getint</td>
<td>func : (1) → (2)</td>
</tr>
<tr>
<td>4</td>
<td>putint</td>
<td>func : (2) → (1)</td>
</tr>
<tr>
<td>5</td>
<td>i</td>
<td>(2)</td>
</tr>
<tr>
<td>6</td>
<td>j</td>
<td>(2)</td>
</tr>
</tbody>
</table>
Back end: (Actual code generation)

Creating correct code is generally not difficult. Optimization of that code is.
Back to front end:

1. How is this actually done?
   Input is actually a string of ASCII.

   Need to find a way to scan letter by letter and decide what is a token.

   Then pass the tokens on to the parsers.
Regular Expressions: some theory

Defined as: Language generated as follows:

- A character: 0 or 1
- The empty string, ε
- 2 regular expressions concatenated
- 2 regular expressions separated by an or (written |)
- A regular expression followed by * (Kleene star - 0 or more occurrences)
Regular Languages

The class of languages described by a regular expression.

Ex: \( \epsilon \cdot 0^* 1 0^* = L \)

Any number of 0's, followed by a single 1, followed by any # of 0's.

\( 1 \in L \)  \( 0 \notin L \)

\( 001 \in L \)  \( 3 \notin L \)
Ex: Give the regular expression for \( \{ w \mid w \text{ begins with a 1 and ends with a 0} \} \)

\[
1 \ (1 \ 1 \ 0)^* \ 0 \\
(1 \ 1 \ 0)(1 \ 1 \ 0) \ (1 \ 1 \ 0) \ \text{for any # of times}
\]

Ex: \( \{ w \mid w \text{ starts with 0 and has an odd length} \} \)

\[
= 0 \ (1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0)^* \\
= 0 \ ((1 \ 1 \ 0)(1 \ 1 \ 0))^*
\]
Example: Numbers in Pascal

digit → 0 1 2 3 4 5 6 7 8 9

unsigned_int → digit digit*

unsigned_number → unsigned_int

unsigned_number → unsigned_int (\times 10^{\pm unsigned_int})

unsigned_number → (\times 10^{\pm unsigned_int})

digit → [0-9]

unsigned_number → unsigned_int \times digit digit*

unsigned_number → 1
Deterministic Finite Automate (DFA)

Regular languages are precisely the 0 things recognized by DFAs.
- A set of states
- Input alphabet
- A start state
- A set of accept states
- A transition function: given a state and an input, output a new state
Example:

word: 0010 ∈ L, recognized by DFA

0

S1

Start

1

S2

= 0*10*

0, 1

double circle indicates accept state

00 ∈ L
Ex: unsigned int → digit digit*
digit → [0-9]
DFA vs NFA
Non-deterministic Finite Automata

Note: No ambiguity is allowed in DFA's. So given a state or input, can't be multiple options.

Also—no $\varepsilon$-transitions. (In DFA)

If we allow several choices to exist, this is called an NFA.

Ex:
Ex: \( L = 1 (0|1)^* 0 \)
Ex: Some things are easier with NFA!

```
unsigned number -> unsigned int (3 | unsigned int)
unsigned int -> [0-9]
```
Essentially, we can think of an NFA as modeling a parallel set of possibilities (or a tree of them).

Thm: Every NFA has an equivalent DFA. (Size of DFA is much bigger ~ $2^n$)

So: Both recognize regular languages!
Limitations of Regular Expressions

Certain languages are not regular.

Ex: \( \exists w \mid w \text{ has an equal number of } 0\text{'s and } 1\text{'s} \)?

Somehow, this needs a type of memory, which regular expressions do not have.

Ex: \( 0^n 1^n \)

Reg? 0 0^n 1^n

(Pumping lemma)
Why do we need this?

Need to "nest" expressions.

Ex: $expr \rightarrow id \mid number \mid -expr \mid (expr) \mid expr \ op \ expr$

$op \rightarrow + \mid - \mid \mid \mid \ast$

Regular expressions can't quite do this.
Context Free Languages

Described in terms of productions (called Backus-Naur Form, or BNF)

- A set of terminals $T : \{d, b, +, \ldots\}$
- A set of non-terminals $N \uparrow$
- A start symbol $S$ (a non-terminal)
- A set of productions
Ex: \( \exists n \in \mathbb{N} \mid n > 0 \)

\[
S \rightarrow 0T1 \\
T \rightarrow 0T1 \mid \varepsilon
\]

\[
S \rightarrow 0S1 \mid 01 \\
n > 1
\]
Ex: 3 word has an equal number of 0's and 1's.
Expression grammars: Simple calculator

expr → term | expr add-op term

term → factor | term mult-op factor

factor → id | number | - factor | (expr)

add-op → + | -

mult-op → * | /
Example: Show how rules can generate $3 + 4 = 5$
Parse Tree

\[ 6 : 3 + 4 \times 5 \]