The Compiler’s Front End
(viewed from a lab 1 perspective)

Comp 412
Lab 1

In Lab 1, you will build a **local register allocator**

- Handouts and software will be online Friday
  - Handouts at the course web site under “programming exercises”
  - Software is on CLEAR under ~comp412/students/lab1
- Friday’s lecture will cover the “register allocation” portion of the lab
- Q&A session early next week
- I have posted a short tutorial (PowerPoint) on the ILOC Virtual Machine and the ILOC Simulator’s trace facility (under “Additional Materials”)
- Today’s lecture should help you get started on the lab’s front end

See remarks in lecture 1 on programming language choice
In Lab 1, the input is written in a subset of ILOC

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>$r_1 \rightarrow r_2$</td>
<td>$r_2 \leftarrow \text{MEM}(r_1)$</td>
</tr>
<tr>
<td>store</td>
<td>$r_1 \rightarrow r_2$</td>
<td>$\text{MEM}(r_2) \leftarrow r_1$</td>
</tr>
<tr>
<td>loadI</td>
<td>$c \rightarrow r_2$</td>
<td>$r_2 \leftarrow c$</td>
</tr>
<tr>
<td>add</td>
<td>$r_1, r_2 \rightarrow r_3$</td>
<td>$r_3 \leftarrow r_1 + r_2$</td>
</tr>
<tr>
<td>sub</td>
<td>$r_1, r_2 \rightarrow r_3$</td>
<td>$r_3 \leftarrow r_1 - r_2$</td>
</tr>
<tr>
<td>mult</td>
<td>$r_1, r_2 \rightarrow r_3$</td>
<td>$r_3 \leftarrow r_1 \times r_2$</td>
</tr>
<tr>
<td>lshift</td>
<td>$r_1, r_2 \rightarrow r_3$</td>
<td>$r_3 \leftarrow r_1 \ll r_2$</td>
</tr>
<tr>
<td>rshift</td>
<td>$r_1, r_2 \rightarrow r_3$</td>
<td>$r_3 \leftarrow r_1 \gg r_2$</td>
</tr>
<tr>
<td>output</td>
<td>$c$</td>
<td>prints $\text{MEM}(c)$ to stdout</td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td>idles for one cycle</td>
</tr>
</tbody>
</table>

**ILOC** is an abstract assembly language for a simple RISC processor. We will use it at times as an IR and at other times as a target language. For details on Lab 1 **ILOC**, see the lab handout, the **ILOC** simulator document and Appendix A in EaC2e. (**ILOC** appears throughout the book.) Note that **ILOC** is case sensitive. The ‘I’ in “loadI” must be an uppercase letter and the others must be lowercase letters. ‘c’ represents a constant.

The same **ILOC** subset (syntax & meaning) will be used in Lab 3, with different per-operation latencies. You should plan to reuse your **ILOC** front end in Lab 3.
Lab 1

In Lab 1, you will build a register allocator for straight-line code.

The allocator will:

• take as input some sequence of operations in a subset of ILOC
• produce as output an equivalent program in the same subset of ILOC

Your lab will need to

(1) read in a file of ILOC operations,
(2) check them for validity,
(3) convert them into some internal representation,
(4) perform renaming, allocation & assignment (next lecture), and
(5) write the resulting code out to stdout.

Today’s lecture uses Lab 1 as an extended example to explore the functionality of a compiler’s “front end”

This lecture provides a quick tour of a compiler front end, without the theory. We will go back and cover the theory in the next 3 weeks. (EaC2e, Chapters 2 & 3).
The Front End

Scanner looks at every character
- Converts stream of chars to stream of classified words:
  - `<part of speech, word>`
  - We sometimes call a classified word a “token”
- Efficiency & scalability matter

Parser looks at every token
- Determines if the stream of words forms a sentence in the source language
- Fits words to some syntactic model, or grammar, for the source language

COMP 412, Fall 2017
The Front End

**Parser** controls the process.
- Parser has a set of grammar rules that define the source language & guide its recognition
- Parser invokes scanner to get tokens as needed
- Parser invokes Semantic Elaboration to perform analysis that is deeper than syntax
  - *e.g.*, build an IR, lay out storage, check types, or compute

**Scanner** operates incrementally
- Batch scanner would require an unbounded buffer (*doesn’t scale* †)
- Scanner may need to look beyond end of current word

† One of the lab 1 test codes has 128,000 lines of input.
The Scanner / Parser Interface

The scanner has a simple interface

• Call to the scanner returns the next classified word
  – A <category, lexeme> pair, where the lexeme can be explicit or implicit
• In Lab 1, the scanner can (should?) discard comments

Efficiency in handling strings

• Strings are big, ugly, awkward, and slow \((See \ § \ 7, \ EaC2e)\)
  – Require space proportional to their length
  – Comparisons require (worst case) \(O(\text{string length})\)
• Integers are compact and fast
  – Space is constant, comparisons are simple & supported by the hardware

When possible, scanners should convert strings to integers

• Manipulate the integers, limit use of the string to printed output
• Implies an efficient map from integer to string
Before we can build either a scanner or a parser, we need a definition of the input language.

- Definition must be formal — that is, mathematically well formed
- Definition must be operational — that is, it must lead to a piece of software that recognizes both valid & invalid inputs

Given the division of labor between scanner & parser, we want a similar split in the definition

- Microsyntax or lexical structure specifies valid spelling for each part of speech
- Syntax or grammar specifies how parts of speech fit together to form sentences

This structure explains one of the key differences between a programming language and a natural language. In a programming language, each word maps to exactly one part of speech. In a natural language, a word can have multiple parts of speech.
### Example: Lab 1

In Lab 1, the input is written in a subset of ILOC

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Meaning</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>load $r_1$ =&gt; $r_2$</td>
<td>$r_2 \leftarrow \text{MEM}(r_1)$</td>
<td>3</td>
</tr>
<tr>
<td>store $r_1$ =&gt; $r_2$</td>
<td>(\text{MEM}(r_2) \leftarrow r_1)</td>
<td>3</td>
</tr>
<tr>
<td>loadI $c$ =&gt; $r_2$</td>
<td>$r_2 \leftarrow c$</td>
<td>1</td>
</tr>
<tr>
<td>add $r_1$, $r_2$ =&gt; $r_3$</td>
<td>$r_3 \leftarrow r_1 + r_2$</td>
<td>1</td>
</tr>
<tr>
<td>sub $r_1$, $r_2$ =&gt; $r_3$</td>
<td>$r_3 \leftarrow r_1 - r_2$</td>
<td>1</td>
</tr>
<tr>
<td>mult $r_1$, $r_2$ =&gt; $r_3$</td>
<td>$r_3 \leftarrow r_1 \ast r_2$</td>
<td>1</td>
</tr>
<tr>
<td>lshift $r_1$, $r_2$ =&gt; $r_3$</td>
<td>$r_3 \leftarrow r_1 &lt;&lt; r_2$</td>
<td>1</td>
</tr>
<tr>
<td>rshift $r_1$, $r_2$ =&gt; $r_3$</td>
<td>$r_3 \leftarrow r_1 &gt;&gt; r_2$</td>
<td>1</td>
</tr>
<tr>
<td>output $c$</td>
<td>prints (\text{MEM}(c)) to stdout</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>idles for one cycle</td>
<td>1</td>
</tr>
</tbody>
</table>

ILOC is an abstract assembly language for a simple RISC processor. We will use it at times as an IR and at other times as a target language.

For details on Lab 1 ILOC, see the lab handout, the ILOC simulator document and Appendix A in EaC2e. (ILOC appears throughout the book.)

Note that ILOC is case sensitive. The ‘I’ in “loadI” must be an uppercase letter and the others must be lowercase letters.

‘c’ represents a constant.

The same ILOC subset (syntax & meaning) will be used in Lab 3, with different per-operation latencies. You should plan to reuse your ILOC front end in Lab 3.
The Syntax of Lab 1 ILOC

The ILOC-subset operations fit one of five patterns or rules

An operation is

\[
\begin{align*}
\text{load} & \quad r_1 \to r_2 & \text{MEMOP} \\
\text{store} & \\
\text{loadI} & \quad c \to r_2 & \text{LOADI} \\
\text{add} & \\
\text{sub} & \\
\text{mult} & \\
\text{lshift} & \\
\text{rshift} & \quad r_1, r_2 \to r_3 & \text{ARITHOP} \\
\text{output} & \quad c & \text{OUTPUT} \\
\text{nop} & \\
\end{align*}
\]

• Scanner must recognize words & assign categories (or parts of speech)
• Parser must discover sentences in the stream of categorized words
The Syntax of Lab 1 ILOC

Rewriting the rules in a more standard notation

\[
\begin{align*}
\text{operation} & \rightarrow \text{MEMOP} \quad \text{REG} \quad \text{INTO} \quad \text{REG} \\
| & \quad \text{LOADI} \quad \text{CONSTANT} \quad \text{INTO} \quad \text{REG} \\
| & \quad \text{ARITHOP} \quad \text{REG} \quad \text{COMMA} \quad \text{REG} \quad \text{INTO} \quad \text{REG} \\
| & \quad \text{OUTPUT} \quad \text{CONSTANT} \\
| & \quad \text{NOP}
\end{align*}
\]

\[
\begin{align*}
\text{block} & \rightarrow \text{block} \quad \text{operation} \\
| & \quad \text{operation}
\end{align*}
\]

Where

- *Italics* indicates a syntactic variable
- ‘→’ means “derives”
- ‘|’ means “also derives”
- **All CAPITALS** indicate a category of word (e.g., *INTO* contains one word, “=>”)
- Categories may contain > 1 word

These rules form a *grammar*. See the digression on page 87 in EaC2e titled “Backus-Naur Form”.

COMP 412, Fall 2017
The Microsyntax of Lab 1 ILOC

Microsyntax is just the spelling of the words in the language

<table>
<thead>
<tr>
<th>Category</th>
<th>Specific Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMOP</td>
<td>load, store</td>
</tr>
<tr>
<td>LOADI</td>
<td>loadI</td>
</tr>
<tr>
<td>ARITHOP</td>
<td>add, sub, mult, lshift, rshift</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>output</td>
</tr>
<tr>
<td>NOP</td>
<td>nop</td>
</tr>
<tr>
<td>CONSTANT</td>
<td>A non-negative, base 10 number</td>
</tr>
<tr>
<td>REGISTER</td>
<td>‘r’ followed by a non-negative, base 10 number</td>
</tr>
<tr>
<td>COMMA</td>
<td>‘,’</td>
</tr>
<tr>
<td>INTO</td>
<td>“=&gt;”</td>
</tr>
</tbody>
</table>

- Categories are dictated by the needs of the grammar
- Scanner finds words and determines their categories (or “token type”)

For simplicity’s sake, assign a small integer to each of these categories (e.g., 0 to 8). Then you can use an array of strings, statically initialized, to efficiently convert the integer to a string for debugging or final output.

Do not use a map or a dictionary. The array reference is a couple of instructions. The overhead on the function call is more like 30 or 40 instructions, before the real work of computing a hash and looking up the value in the table. (See § B.4 in EaC2e)
Scanning Lab 1 ILOC

How do we write a scanner for these syntactic categories?

• All of your training, to date, tells you to call some support routine
  – fscanf() in C, the Scanner class in Java, the extract operator (>>) in C++, a
    regex library in Python (or Java, or C++), ...

• COMP 412 is about implementing languages, so in COMP 412, you WILL
  (that means you MUST) use character-by-character input
  – Character-by-character algorithms have clarity & they expose complexity

You may not use a regular expression library or built-in facility.*
Scanning Lab 1 ILOC

How do we write a scanner for these syntactic categories?

• All of your training, to date, tells you to call some support routine
  – `fscanf()` in C, the Scanner class in Java, the extract operator (>>) in C++, a
    regex library in Python (or Java, or C++), ...

• COMP 412 is about *implementing* languages, so in COMP 412, you **WILL**
  (that means you **MUST**) use `character-by-character` input
  – Character-by-character algorithms have clarity & they expose complexity

You may not use a regular expression library or built-in facility.

If you want to use regex, implement the regex routines yourself.
  → COMP 412 is about implementation of languages & their runtimes
  → In the next two weeks, you will learn enough to build a great regex library

You can look at the simulator’s front end, but you cannot reuse it
  → In fact, it uses flex and bison, which are not allowed in this lab
How do we write a scanner for these syntactic categories?

• All of your training, to date, tells you to call some support routine
  – fscanf() in C, the Scanner class in Java, the extract operator (>>) in C++, a
    regex library in Python (or Java, or C++), ...

• COMP 412 is about implementing languages, so in COMP 412, you WILL (that means you MUST) use character-by-character input
  – Character-by-character algorithms have clarity & they expose complexity

Let’s look at an easy one.
How do we recognize a “NOP”?  
• The spelling is ‘n’ followed by ‘o’ followed by ‘p’
• The code is as simple as the description
  – Cost is $O(1)$ per character
  – Asymptotically optimal

```c
  c ← next character
  If c = ‘n’ then {
    c ← next character
    if c = ‘o’ then {
      c ← next character
      if c = ‘p’
        then return <NOP,"nop">
      else report error
    }
    else report error
  }
  else report error
```
We can represent this recognizer as an automaton

- Execution begins in the start state, \( s_0 \)
- On an input of ‘n’, it follows the edge labeled ‘n’, and so on, ...
- Implicit transition to an error state, \( s_e \), on any unexpected character
- Halts when it is out of input
- States drawn with double lines indicate success (a “final” state)

\[
c \leftarrow \text{next character}
\]
\[
\text{If } c = 'n' \text{ then } \{
\text{c } \leftarrow \text{next character}
\text{if } c = 'o' \text{ then } \{
\text{c } \leftarrow \text{next character}
\text{if } c = 'p' \text{ then return } \langle \text{NOP, "nop"} \rangle
\text{else report error}
\}
\text{else report error}
\}
\text{else report error}
\]

\( O(1) \) per input character, if branches are \( O(1) \)

**Transition Diagram for “nop”**

---

COMP 412, Fall 2017
Recognizing the rest of the finite categories is (relatively) easy

Any character
Transitions to $s_e$ are implicit from every state

§ 2.5 in EaC2e discusses how to convert a transition diagram into code. The reference implementation uses a table-driven scanner.
Recognizing the unbounded categories is conceptually harder

Consider unsigned integers — a \textsc{constant} in Lab 1 ILOC

\begin{verbatim}
c ← next character
n ← 0
if c = '0'
    then return <n,CONSTANT>
else if ('1' ≤ c ≤ '9') then {
    n ← atoi(c)
    c ← next character
    while ('0' ≤ c ≤ '9') {
        t ← atoi(c)
        n ← n * 10 + t
    }
    return <n,CONSTANT>
} else report error
\end{verbatim}

- Recognizes a valid unsigned integer of arbitrary magnitude
- Cycle in transition diagram admits an unbounded set of numbers

Any character

0 \ldots 9 means any letter from 0 to 9

Transitions to $s_e$ are implicit from every state

Unsigned integers are an infinite, but \textit{recursively enumerable} set.
Scanning Lab 1 ILOC

REG follows from CONSTANT

Registers are simple
- ‘r’ followed by an unsigned integer
- We concatenate a recognizer for ‘r’ onto the front of the one for unsigned integer
- Resulting recognizer is still $O(1)$ cost per character, with very low constant overhead (if done well)

We can add the recognizers for REG and CONSTANT to the recognizer on slide 12 by combining the start states & merging the edges for ‘r’.

One recognizer gets all the words in Lab 1, at $O(1)$ cost per letter.

This particular recognizer allows r01, r02, and so on. The recognizer for integer constants on the previous slide did not allow leading zeroes.
These DFAs do not allow leading zeroes

DFA\(s\) that allow leading zeroes are simpler and smaller

Either way, the cost is \(O(1)\) per character
Need to merge these DFAs with the big DFA for the rest of ILOC

**Note the state numbers**

- States tie into the big **DFA**
  - States & transitions in **gray** are in the earlier **DFA**
- Register hangs off the ‘r’ for “rshift”
- Constant hangs off the start state

These two **DFAs** will extend the big **DFA** to handle register names and constants.
Comments are easy to handle

• A comment begins with “//” and includes any character up to the “end of line” sequence
  – Windows uses “\r\n”
  – Linux & MacOS use ‘\n’

• Your code will be graded on a linux system, so you need to get the EOL thing right

• Comment handler should invoke the scanner, recursively, from $s_3$

Differences in EOL trip up even the most experienced developers. For example, do not try to write a shell script on a Windows machine & copy it to clear. The error messages from the shell will be less than helpful.
Scanning Lab 1 ILOC

This seems pretty simple. How do real compilers scan their input?

• The approach used in the previous slides is built on the theory of regular languages and their relationship to deterministic finite automata (DFA).
  – Deep theoretical foundations
  – General techniques

• Automata based scanners have been used in industry and research since the 1960s
  – Same techniques form the basis for “regular expression” search in editors, wildcard matching in shells, and regex libraries in programming languages
  – Pervasive technology

Next week, we will see how to construct DFAs automatically from specifications written as regular expressions.

Tools such as lex, flex, and the regular expression libraries in common programming languages are based on these techniques.

For Lab 1, merge the transition diagrams on 17, 21, & 22, then implement with one of the schemes in § 2.5.
ILOC has a particularly simple format (as do most assembly codes)

- The opcode determines the syntax of the rest of the line
  - Dictated by the simple mechanism used to decode operations in hardware
  - Once we see an ARITHOP, we know that the rest of the operation has form
    \[ \text{REG COMMA REG INTO REG} \]

- We can build a simple parser as a switch statement based on opcode

```c
switch (category) {
  case ARITHOP:
    if (ParseREG() == FALSE)
      then report error
    if (ParseCOMMA() == FALSE)
      then report error
    ...
    if (ParseREG() == FALSE)
      then report error
}
```

To be useful, the parser must build a representation of the parsed instructions, presumably in the missing else clauses.
ILOC has a particularly simple format (as do most assembly codes)

In C, we might do something even simpler

```c
switch (category) {
    case ARITHOP:
        if (((ParseREG()) && ParseCOMMA() && ParseREG()) &&
            ParseINTO() && ParseREG()) == FALSE)
            then report error
        else {
            /* build the structure to represent this op */
        }
    break;
    case MEMOP:
        ...
}
```

• More concise, although we sacrifice some error reporting ability
  – Cannot tell user where within the line the error occurred
  – Often, isolating the error to a specific line might be enough

• Avoids the long, drawn out series of if-then-else constructs
Is this how real compilers parse their inputs?

- No. Assembly languages have a particularly simple structure.
- Real programming languages require more complicated mechanisms
  - For example, you cannot match opening and closing brackets this way
  - The simple syntax of Scheme would defeat this kind of ad hoc technique
- But, the simple structure of assembly makes it easy to parse

- We will spend three weeks on rigorous parsing techniques
  - Both top-down and bottom-up parsers \((\text{recursive descent, LL}(1), \text{LR}(1))\)
  - You will become well versed in both parsing and techniques to automate the construction of parsers

However, we cannot wait that long to start Lab 1, so get to work with these ad hoc techniques
Building an Intermediate Representation

Scanning & Parsing tell us if the input is well formed. To work on the code, your lab needs a concrete representation for it.

- The design of an IR has major consequences for the rest of the compiler
  - Compiler cannot easily manipulate what it does not represent
  - Different data structures have different cost structures
    → Common operations should be cheap & easy

For Lab 1 (& Lab 3), the IR can be almost be as simple as the pictures *(wait for next class†)*

† Before designing an IR, try to understand the whole problem.
Once the input is scanned, parsed, and encoded into some appropriate IR, what should your lab do with it?

- The basic terminology of register allocation
- Sheldon Best’s algorithm (a.k.a. “bottom-up local”)
- Maybe a hint or two

To prepare, read § 13.1, 13.2 (skip 13.2.3) and 13.3.2 in EaC2e