Building a Scanner

*(a lab 1 perspective)*

Comp 412

Copyright 2019, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 1 & 2 in EaC2e, Lab 1 Handout (on web site)
Lab 1

In Lab 1, you will build a front end for ILOC

• Handout and software will be online this afternoon
  – Handout at the course web site under “programming exercises”
  – Software is on CLEAR under ~comp412/students/lab1
  – See also the ILOC VM tutorial under “additional materials on Lectures page

• For each lab, we provide an execute-only reference implementation
  – Shows you how we expect the lab to behave
  – Lets you discover the answers to some of your questions

Notable Points:

• See discussion in lab 1 handout on Honor Code Policy (page 7)
  – Your labs must consist of code that you wrote
  – You may discuss anything with other COMP 412 students
  – You may collaborate on shell scripts and makefiles, but not on the code

• See remarks in lecture 1 on programming language choice
## 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</td>
<td>(r_1 \rightarrow r_2) (r_2 \leftarrow \text{MEM}(r_1))</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>(r_1 \rightarrow r_2) (\text{MEM}(r_2) \leftarrow r_1)</td>
<td>3</td>
</tr>
<tr>
<td>loadI</td>
<td>(x \rightarrow r_2) (r_2 \leftarrow x)</td>
<td>1</td>
</tr>
<tr>
<td>add</td>
<td>(r_1, r_2 \rightarrow r_3) (r_3 \leftarrow r_1 + r_2)</td>
<td>1</td>
</tr>
<tr>
<td>sub</td>
<td>(r_1, r_2 \rightarrow r_3) (r_3 \leftarrow r_1 - r_2)</td>
<td>1</td>
</tr>
<tr>
<td>mult</td>
<td>(r_1, r_2 \rightarrow r_3) (r_3 \leftarrow r_1 \times r_2)</td>
<td>1</td>
</tr>
<tr>
<td>lshift</td>
<td>(r_1, r_2 \rightarrow r_3) (r_3 \leftarrow r_1 &lt;&lt; r_2)</td>
<td>1</td>
</tr>
<tr>
<td>rshift</td>
<td>(r_1, r_2 \rightarrow r_3) (r_3 \leftarrow r_1 &gt;&gt; r_2)</td>
<td>1</td>
</tr>
<tr>
<td>output</td>
<td>(x \rightarrow r_3) (\text{prints MEM}(x)) to stdout</td>
<td>1</td>
</tr>
<tr>
<td>nop</td>
<td>()</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 more details on ILOC, see the lab handout, the ILOC VM tutorial on the web site, and Appendix A in EaC2e. (ILOC appears throughout EaC2e.)

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

‘x’ represents a non-negative integer constant.

The same ILOC subset (syntax & meaning) will be used in Labs 2 & 3, with different. You should plan to reuse your ILOC front end in Labs 2 & 3.
In Lab 1, you will build a **FRONT END** that reads **ILOC**.

It will need to:

1. read in a file of **ILOC** operations,
2. check them for validity, and
3. convert them into some intermediate representation.

You will construct a hand-coded scanner and a simple hand-coded parser.

Today’s lecture focuses on the scanner.

Lectures 2 & 3, together, provide a quick tour of a compiler’s front end, without the theory.

**Do not worry.**
You will see the theory over the next three weeks. (EaC2e, Chapters 2 & 3).
Read the book. It is an old-fashioned MOOC.
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
The Front End

Scanner and Parser collaborate to check the syntax of the input program.

- Scanner maps stream of characters into words (words are the fundamental unit of syntax)
- Scanner produces a stream of tokens for the parser
  - Token is <part of speech, word>
  - “Part of speech” is a unit in the grammar
- Parser maps stream of tokens into a sentence in a grammatical model of the input language
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 tasks that are 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 \(<\text{part of speech}, \text{lexeme}\>) pair, where the \text{lexeme} can be explicit or implicit

• In \text{COMP 412} labs, the scanner can (and should) discard comments

Efficiency in handling strings

• Strings are big, ugly, awkward, and slow
  – Require space proportional to their length
  – Comparisons require (worst case) \(O(\text{string length})\) comparisons

• 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

\textit{Do not store opcodes as strings in your IR}
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</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>$x \rightarrow r_2$</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>$x$</td>
<td>prints $\text{MEM}(x)$ 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 ‘l’ in “loadI” must be an uppercase letter and the others must be lowercase letters.

‘x’ represents a constant.
The Syntax of Lab 1 ILOC

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

<table>
<thead>
<tr>
<th>An operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>is</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>or</td>
</tr>
</tbody>
</table>

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

Rewriting the rules in a more standard notation

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

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

Where

• *Italics* indicates a syntactic variable (or “nonterminal” symbol)
• ‘\(\rightarrow\)’ means “derives”
• ‘\(|\)’ means “also derives”

All CAPITALS indicate a category of word (or “terminal” symbol)

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 2019
The Microsyntax of 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)
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.*
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

\[
\begin{align*}
c & \leftarrow \text{next character} \\
\text{If } c &= \text{‘n’} \text{ then } \\
&\text{If } c &= \text{‘o’} \text{ then } \\
&\text{If } c &= \text{‘p’} \\
&\text{then return } <\text{NOP}, \text{”nop”}> \\
&\text{else report error} \\
&\text{else report error} \\
\end{align*}
\]

- 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)

Transition Diagram for “nop”

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

See § 2.2 in EAC2e
Scanning Lab 1 ILOC

Recognizing the rest of the finite categories is (relatively) easy

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

Start State

COMMA

> INTO

Still $O(1)$ per input character
Recognizing the rest of the finite categories is (relatively) easy

§ 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 CONSTANT in Lab 1 ILOC

```c
void atoi(const char* str);
```

```c
const char* sc = next character;
n = 0
if (c = '0')
    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
```

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

Unsigned integers are an infinite, but recursively enumerable set.
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.
In lab 1, you **should** allow leading zeroes in register names. You may allow them in constants.

**These DFAs do not allow leading zeroes**

![Diagram of DFAs that do not allow leading zeroes](image)

**DFA**s that allow leading zeroes are simpler and smaller

![Diagram of DFAs that allow leading zeroes](image)

Either way, the cost is $O(1)$ per character.
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. Do **not** try to write a shell script or makefile on a Windows machine & copy it to **clear**.

The error messages from the shell will be less than helpful.
Back to that “character-by-character” restriction:

What, precisely, is allowed?

• Your code **may** read the input with an I/O routine that returns a single character, such as `getc()`.
  – C and PYTHON support this kind of I/O. It is a little tricky in JAVA
  – This approach can be slower than using some buffered I/O scheme.

• Your code **may** read a line into a buffer and then process a single character at a time (*refilling on demand, of course*)
  – `getline()` in C, `BufferedInputStream` or `BufferedReader` in JAVA, and using the `read` method of a file object in PYTHON

• Your code **may** read a bounded number of characters into a buffer and then process a single character at a time (*refilling on demand, of course*)
  – The buffer must be bounded to ensure good behavior on large files

• Your code should **not** read the *entire file* into a buffer
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 (Even web application firewall rules ...)

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 common in programming languages are based on these techniques.

For Lab 1, merge the transition diagrams on 17, 18, 19, & 21, then implement with one of the schemes in § 2.5.