Announcements

• First project and second homework have been posted.
  a) loadI 1024 => r0  first instruction in all test cases
  b) r0 is not an allocatable register (not part of “k”) 
  c) Latency of outputAI is 3

• Homeworks: No late submissions; however, you may ask for an extension in class

• Paper on register allocation on web site (8th International Conference on Compiler Construction, CC’99, March 1999)
ILOC simulator + ILOC files

(1) Command line option for sim
   -i NUM0 NUM1 ... NUMn Treats NUM1 through NUMn as integer data items and initializes data memory by writing them into consecutive words of memory, starting at the address NUM0.

(2) Inserted “loadI 1024 => r0” to testblock and reportblock files; all spill code has to use loadAI and storeAI instructions; you should use two feasible registers for top-down, and no feasible registers for bottom-up
Lexical Analysis

Read EaC: Chapters 2.1 - 2.5;
The purpose of the front end is to deal with the input language

- Perform a membership test: \( code \in source \text{ language?} \)
- Is the program well-formed (semantically)?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic
The Front End

Scanner
• Maps stream of characters into words
  → Basic unit of syntax
  → \( x = x + y \) \( ; \) becomes
    \(<\text{id},x>\ <\text{eq},=\>\ <\text{id},x>\ <\text{pl},+\>\ <\text{id},y>\ <\text{sc},;\>\)
• Characters that form a word are its *lexeme*
• Its *part of speech* (or *syntactic category*) is called its *token type*
• Scanner discards white space & (often) comments

Source code → Scanner \(\rightarrow\) tokens \(\rightarrow\) Parser \(\rightarrow\) IR

Speed is an issue in scanning
⇒ use a specialized recognizer
The Front End

Parser

- Checks stream of classified words (parts of speech) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

We’ll get to parsing in the next lectures
Language syntax is specified with *parts of speech*, not *words*

Syntax checking matches *parts of speech* against a grammar

Here is an example context free grammar (CFG) $G$:

1. $goal \rightarrow expr$
2. $expr \rightarrow expr \ op \ term$
3. \hspace{1cm} | \hspace{1cm} term
4. $term \rightarrow number$
5. \hspace{1cm} | \hspace{1cm} id
6. $op \rightarrow +$
7. \hspace{1cm} | \hspace{1cm} –

$G$ in BNF form

\[
G = (S, T, N, P)
\]

$S = goal$

$T = \{ \text{number, id, +, -} \}$

$N = \{ goal, expr, term, op \}$

$P = \{ 1, 2, 3, 4, 5, 6, 7 \}$
Why study lexical analysis?

- We want to avoid writing scanners by hand

Goals:

- To simplify specification & implementation of scanners
- To understand the underlying techniques and technologies
Lexical patterns form a **regular language**

***any finite language is regular***

**Regular expressions** (REs) describe regular languages

Regular Expression (over alphabet $\Sigma$)

- $\epsilon$ is a RE denoting the set $\{\epsilon\}$
- If “a” is in $\Sigma$, then $a$ is a RE denoting $\{a\}$
- If $x$ and $y$ are RE$s$ denoting $L(x)$ and $L(y)$ then
  $\rightarrow x \mid y$ is an RE denoting $L(x) \cup L(y)$
  $\rightarrow xy$ is an RE denoting $L(x)L(y)$
  $\rightarrow x^*$ is an RE denoting $L(x)^*$
  $\rightarrow (x)$ is an RE denoting $L(x)$

**Precedence** is closure, then concatenation, then alternation
### Set Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Union of L and M</strong></td>
<td>$L \cup M = { s \mid s \in L \text{ or } s \in M }$</td>
</tr>
<tr>
<td><strong>Written L \cup M</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Concatenation of L and M</strong></td>
<td>$LM = { st \mid s \in L \text{ and } t \in M }$</td>
</tr>
<tr>
<td><strong>Written LM</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Kleene closure of L</strong></td>
<td>$L^* = \bigcup_{0 \leq i \leq \infty} L^i$</td>
</tr>
<tr>
<td><strong>Written L^</strong>*</td>
<td></td>
</tr>
<tr>
<td><strong>Positive Closure of L</strong></td>
<td>$L^+ = \bigcup_{1 \leq i \leq \infty} L^i$</td>
</tr>
<tr>
<td><strong>Written L^+</strong></td>
<td></td>
</tr>
</tbody>
</table>

*These definitions should be well known*
Examples of Regular Expressions

**Identifiers:**

\[ \begin{align*}
\text{Letter} & \rightarrow (a|b|c \ldots |z|A|B|C \ldots |Z) \\
\text{Digit} & \rightarrow (0|1|2 \ldots |9) \\
\text{Identifier} & \rightarrow \text{Letter} (\text{Letter} \mid \text{Digit})^* 
\end{align*} \]

**Numbers:**

\[ \begin{align*}
\text{Integer} & \rightarrow (+|-|\varepsilon) (0|1|2|3 \ldots |9)(\text{Digit}^*) \\
\text{Decimal} & \rightarrow \text{Integer} \cdot \text{Digit}^* \\
\text{Real} & \rightarrow (\text{Integer} \mid \text{Decimal}) \cdot e (\pm|\varepsilon) \text{Digit}^* \\
\text{Complex} & \rightarrow (\text{Real} \pm \text{Real}) 
\end{align*} \]

*Numbers can get much more complicated!*
Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer.

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions.

\[ \Rightarrow \text{We study REs and associated theory to automate scanner construction!} \]
Consider the problem of recognizing ILOC register names

\[ Register \rightarrow r \ (0|1|2| \ldots | 9) \ (0|1|2| \ldots | 9)^* \]

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)

Recognizer for \textit{Register}

Transitions on other inputs go to an error state, \( s_e \)
DFA operation

- Start in state $S_0$ & take transitions on each input character
- DFA accepts a word $x$ iff $x$ leaves it in a final state ($S_2$)

So,

- $r17$ takes it through $s_0, s_1, s_2$ and accepts
- $r$ takes it through $s_0, s_1$ and fails
- $a$ takes it straight to error state $s_e$ (not shown here)
To be useful, recognizer must turn into code

Char ← next character
State ← s₀

while (Char ≠ EOF)
    State ← δ(State, Char)
    Char ← next character

if (State is a final state )
    then report success
else  report failure

Skeleton recognizer

<table>
<thead>
<tr>
<th>δ</th>
<th>r</th>
<th>0,1,2,3,4, 5,6,7,8,9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s₁</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₁</td>
<td>sₑ</td>
<td>s₂</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₂</td>
<td>sₑ</td>
<td>s₂</td>
<td>sₑ</td>
</tr>
<tr>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
</tbody>
</table>

Table encoding RE

Example (continued)
To be useful, recognizer must turn into code

Char ← next character
State ← $s_0$

while (Char ≠ EOF)
    State ← $\delta$(State,Char)
    perform specified action
    Char ← next character

if (State is a final state) then report success
else report failure

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>r</th>
<th>0,1,2,3,4,5,6,7,8,9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td></td>
<td>start</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>add</td>
<td>error</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>add</td>
<td>error</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

Skeleton recognizer

Table encoding RE
r Digit Digit* allows arbitrary numbers

- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31?

Write a tighter regular expression

\[ \text{Register} \rightarrow r \ ( (0|1|2) \ (Digit \ | \ \varepsilon) \ | \ (4|5|6|7|8|9) \ | \ (3|30|31) ) \]

\[ \text{Register} \rightarrow r0|r1|r2 | \ldots | r31|r00|r01|r02 | \ldots | r09 \]

- Produces a more complex DFA
  - Has more states
  - Same cost per transition
  - Same basic implementation
The DFA for

\[ \text{Register} \rightarrow r \ ( (0|1|2) \ (\text{Digit} \ | \ \varepsilon) \ | \ (4|5|6|7|8|9) \ | \ (3|30|31) ) \]

- Accepts a more constrained set of registers
- Same set of actions, more states
Tighter register specification (continued)

Table encoding RE for the tighter register specification

<table>
<thead>
<tr>
<th>δ</th>
<th>r</th>
<th>0,1</th>
<th>2</th>
<th>3</th>
<th>4-9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s₁</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₁</td>
<td>sₑ</td>
<td>s₂</td>
<td>s₂</td>
<td>s₅</td>
<td>s₄</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₂</td>
<td>sₑ</td>
<td>s₃</td>
<td>s₃</td>
<td>s₃</td>
<td>s₃</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₃</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₄</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₅</td>
<td>sₑ</td>
<td>s₆</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₆</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
</tbody>
</table>

Runs in the same skeleton recognizer
Constructing a Scanner - Quick Review

- The scanner is the first stage in the front end
- Specifications can be expressed using regular expressions
- Build tables and code from a DFA
Goal

- We will show how to construct a finite state automaton to recognize any RE

Overview:
- Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
  - Requires ε-transitions to combine regular subexpressions
- Construct a deterministic finite automaton (DFA) to simulate the NFA
  - Use a set-of-states construction
- Minimize the number of states
  - Hopcroft state minimization algorithm
- Generate the scanner code
  - Additional specifications needed for details
• All strings of 1s and 0s ending in a 1

\((0 \mid 1)^*1\)

• All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order

\[\text{Cons} \rightarrow (b|c|d|f|g|h|j|k|m|n|p|q|r|s|t|v|w|x|y|z)\]
\[\text{Cons}^* \ e \ \text{Cons}^* \ i \ \text{Cons}^* \ o \ \text{Cons}^* \ u \ \text{Cons}^*\]

• All strings of 1s and 0s that do not contain three 0s in a row:
More Regular Expressions

• All strings of 1s and 0s ending in a 1

\[(0 | 1)^* 1\]

• All strings over lowercase letters where the vowels (a, e, i, o, & u) occur exactly once, in ascending order

\[\text{Cons} \rightarrow (b | c | d | f | g | h | j | k | l | m | n | p | q | r | s | t | v | w | x | y | z)\]
\[\text{Cons}^* \text{ a Cons}^* \text{ e Cons}^* \text{ i Cons}^* \text{ o Cons}^* \text{ u Cons}^*\]

• All strings of 1s and 0s that do not contain three 0s in a row:

\[(1^* (\varepsilon | 01 | 001 ) 1^*)^* (\varepsilon | 0 | 00)\]
Each RE corresponds to a deterministic finite automaton (DFA)
• May be hard to directly construct the right DFA

What about an RE such as \((a \mid b)^* abb\) ?

This is a little different
• \(S_0\) has a transition on \(\varepsilon\)
• \(S_1\) has two transitions on \(a\)
This is a non-deterministic finite automaton (NFA)
Non-deterministic Finite Automata

- An NFA accepts a string $x$ iff $\exists$ a path though the transition graph from $s_0$ to a final state such that the edge labels spell $x$
- Transitions on $\varepsilon$ consume no input
- To “run” the NFA, start in $s_0$ and $\textbf{guess}$ the right transition at each step
  - Always guess correctly
  - If some sequence of correct guesses accepts $x$ then accept

Why study NFAs?
- They are the key to automating the RE$\rightarrow$DFA construction
- We can paste together NFAs with $\varepsilon$-transitions

$$NFA \xrightarrow{\varepsilon} \text{NFA} \quad \text{becomes an} \quad \text{NFA}$$
DFA is a special case of an NFA

- DFA has no $\varepsilon$ transitions
- DFA’s transition function is single-valued
- Same rules will work

DFA can be simulated with an NFA

\[ \text{Obviously} \]

NFA can be simulated with a DFA

\[ \text{(less obvious)} \]

- Simulate sets of possible states
- Possible exponential blowup in the state space
- Still, one state per character in the input stream
To convert a specification into code:
1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators
- Lex and Flex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser  \( (\text{define all parts of speech}) \)
- You could build one in a weekend!
Automating Scanner Construction

**RE → NFA** *(Thompson’s construction)*
- Build an NFA for each term
- Combine them with ε-moves

**NFA → DFA** *(subset construction)*
- Build the simulation

**DFA → Minimal DFA**
- Hopcroft’s algorithm

**DFA → RE** *(Not part of the scanner construction)*
- All pairs, all paths problem
- Take the union of all paths from \( s_0 \) to an accepting state

*The Cycle of Constructions*
Key idea

- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with $\varepsilon$ moves in precedence order

\[ S_0 \xrightarrow{a} S_1 \]

NFA for $a$

\[ S_0 \xrightarrow{a} S_1 \xrightarrow{\varepsilon} S_3 \xrightarrow{b} S_4 \]

NFA for $ab$

\[ S_0 \xrightarrow{\varepsilon} S_1 \xrightarrow{a} S_3 \xrightarrow{\varepsilon} S_4 \]

NFA for $a^*$

Ken Thompson, CACM, 1968
More Lexical Analysis