CS415 Compilers

Lexical Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Review: Three-pass Compiler

- Front end maps legal source code into IR
- Middle end analyzes IR and transforms IR
- Back end maps IR into target machine code

Typically, front end is $O(n)$ or $O(n \log n)$, while middle end and back end are NPC
The purpose of the front end is to deal with the input language

• Perform a membership test: code ∈ source 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

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
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

Represent words as indices into a global table

Specifications written as “regular expressions”
Lexical patterns form a *regular language*

***any finite language is regular***

Regular expressions (REs) describe regular languages

Regular Expression (over alphabet $\Sigma$)

- $\varepsilon$ is a RE denoting the set $\{\varepsilon\}$
- If $a$ is in $\Sigma$, then $a$ is a RE denoting $\{a\}$
- If $x$ and $y$ are REs denoting $L(x)$ and $L(y)$ then
  - $x | y$ is an RE denoting $L(x) \cup L(y)$
  - $xy$ is an RE denoting $L(x)L(y)$
  - $x^*$ is an RE denoting $L(x)^*$

Precedence:

(1) closure
(2) concatenation,
(3) alternation

Ever type “rm *.o a.out”?
### Set Operations

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

These definitions should be well known
Examples of Regular Expressions

Identifiers:

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

Numbers:

\[\text{Integer} \rightarrow (+\mid-\mid\varepsilon) (0\mid 1\mid 2\mid 3 \ldots \mid 9)(\text{Digit}^*)\]
\[\text{Decimal} \rightarrow \text{Integer} \cdot \text{Digit}^*\]
\[\text{Real} \rightarrow (\text{Integer} \mid \text{Decimal}) \cdot (\pm \mid \varepsilon) \text{Digit}^*\]
\[\text{Complex} \rightarrow (\text{Real}, \text{Real})\]
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.

⇒ 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 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 encoding RE
To be useful, recognizer must turn into code

Char ← next character
State ← s₀

while (Char ≠ EOF)
    State ← δ(State,Char)
    perform specified action
    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
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
- \(\rightarrow Register \rightarrow r0|r1|r2| \ldots |r31|r00|r01|r02| \ldots |r09\)

Produces a more complex DFA
- Has more states
- Same cost per transition
- Same basic implementation
What if we need a tighter specification?

$r \text{Digit Digit}^*$ allows arbitrary numbers

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

Write a tighter regular expression

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

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

Produces a more complex DFA

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

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

- 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>$\delta$</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>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_5$</td>
<td>$s_4$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_4$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_5$</td>
<td>$s_e$</td>
<td>$s_6$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_6$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

Runs in the same skeleton recognizer
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 \( \varepsilon \)-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
More Regular Expressions

- 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

- 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

  \(\text{Cons} \rightarrow (b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z)\)

  \(\text{Cons}^* a \text{ Cons}^* e \text{ Cons}^* i \text{ Cons}^* o \text{ Cons}^* u \text{ Cons}^*\)

- 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

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

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

Cons → (b|c|d|f|g|h|j|k|l|m|n|p|q|r|s|t|v|w|x|y|z)
Cons* a Cons* e Cons* i Cons* o Cons* u Cons*

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

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

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

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 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 automate the RE→DFA construction
• We can paste together NFAs with $\varepsilon$-transitions
Relationship between NFAs and DFAs

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
\[ \rightarrow Obviously \]

NFA can be simulated with a DFA \[ (less\ obvious) \]
• 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
  (define all parts of speech)
• You could build one in a weekend!
Key idea

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

NFA for $a$

NFA for $ab$
Key idea
- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order

NFA for a

NFA for ab

NFA for a | b
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

NFA for $a$

$$
\begin{array}{c}
S_0 \\
S_1
\end{array} \xrightarrow{a} \begin{array}{c}S_2 \\
S_3
\end{array} \xrightarrow{a} S_4
$$

NFA for $ab$

$$
\begin{array}{c}
S_0 \\
S_1
\end{array} \xrightarrow{a} \begin{array}{c}S_2 \\
S_3
\end{array} \xrightarrow{\varepsilon} \begin{array}{c}S_4 \\
S_5
\end{array} \xrightarrow{b} S_6
$$

NFA for $a \mid b$

$$
\begin{array}{c}
S_0 \\
S_1
\end{array} \xrightarrow{a} \begin{array}{c}S_2 \\
S_3
\end{array} \xrightarrow{b} S_4
$$
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

NFA for $a$

NFA for $ab$

NFA for $a | b$
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

NFA for $a$

NFA for $ab$

NFA for $a \mid b$

NFA for $a^*$
Key idea

- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε 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

\[ S_0 \xrightarrow{\varepsilon} S_1 \xrightarrow{a} S_2 \xrightarrow{\varepsilon} S_5 \]
NFA for a | b

\[ S_0 \xrightarrow{\varepsilon} S_1 \xrightarrow{a} S_3 \]
NFA for a*
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

NFA for $a$

NFA for $ab$

NFA for $a \mid b$

NFA for $a^*$
Key idea

- NFA pattern for each symbol and each operator
- Each NFA has a single start and accept state
- Join them with ε moves in precedence order
Example of Thompson’s Construction

Let’s try $a \ (b \mid c)^*$

1. $a, b, \text{ & } c$

2. $b \mid c$

3. $(b \mid c)^*$
Example of Thompson’s Construction

Let’s try \( a ( b | c )^* \)

1. \( a, b, \) & \( c \)

2. \( b | c \)

3. \( ( b | c )^* \)
4. $a (b \mid c)^*$

Of course, a human would design something simpler ...

But, we can automate production of the more complex one ...
NFA $\rightarrow$ DFA with Subset Construction

Need to build a simulation of the NFA

Two key functions
- $\text{Move}(s_i, a)$ is set of states reachable from $s_i$ by $a$
- $\varepsilon$-closure($s_i$) is set of states reachable from $s_i$ by $\varepsilon$

The algorithm:
- Start state derived from $s_0$ of the NFA
- Take its $\varepsilon$-closure $S_0 = \varepsilon$-closure($s_0$)
- Take the image of $S_0$, move($S_0$, $a$) for each $a \in \Sigma$, and take its $\varepsilon$-closure
- Iterate until no more states are added

Sounds more complex than it is...
Syntax Analysis

Read EaC: 3.1 – 3.3

Next class