CSE 504: Compiler Design

Lexical Analysis

Pradipta De
pradipta.de@sunykorea.ac.kr
• Lexical Analysis or scanning phase of front-end of a compiler
  – Recognize tokens using Finite Automata
  – Express token classes using Regular Expressions
  – Generate Scanners from Regular Expressions
  – Implement Scanners
Lexical Analyzer or Scanner or Recognizer?

- Scanner transforms a stream of characters into a stream of tokens
  - Aggregates characters to form words
  - Applies set of rules to determine if valid word
  - If valid, assign a syntactic category (e.g. integer)
How to implement a scanner?

• Automatic tools for scanner generation exist
  – Input mathematical description of language’s lexical syntax; outputs code for scanner
  – Scanners can be hand-crafted as well
Recognizing Words

Recognize word “new”

\[
c \gets \text{NextChar}();
\]
\[
\text{if } (c = 'n')
\]
\[
\text{then begin;}
\]
\[
c \gets \text{NextChar}();
\]
\[
\text{if } (c = 'e')
\]
\[
\text{then begin;}
\]
\[
c \gets \text{NextChar}();
\]
\[
\text{if } (c = 'w')
\]
\[
\text{then report success;}
\]
\[
\text{else try something else;}
\]
\[
\text{end;}
\]
\[
\text{else try something else;}
\]
\[
\text{end;}
\]
\[
\text{else try something else;}
\]

Simple transition diagram works as a recognizer
Recognizing Multiple Words

Recognize word “new” and “not”

Recognize word “new” “not” “while”

Transition diagrams are abstractions of the code that would implement a scanner

Transition diagrams are mathematical objects, called finite automata
Finite Automata

• FA is a formalism for recognizers that has,
  – a finite set of states (S)
  – finite alphabet set (Σ)
  – a transition function (δ)
    • δ(s,c) is the transition function which maps one state on to the next state based on the input alphabet
  – a start state (s₀)
  – Set of (one or more) accepting states (S_A)
Finite Automata for the \{new, not, while\} recognizer

\[ S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\} \]

\[ \Sigma = \{e, h, i, l, n, o, t, w\} \]

\[ \delta = \left\{ \begin{array}{l}
    s_0 \xrightarrow{n} s_1, \quad s_0 \xrightarrow{w} s_6, \quad s_1 \xrightarrow{e} s_2, \quad s_1 \xrightarrow{o} s_4, \quad s_2 \xrightarrow{w} s_3, \\
    s_4 \xrightarrow{t} s_5, \quad s_6 \xrightarrow{h} s_7, \quad s_7 \xrightarrow{i} s_8, \quad s_8 \xrightarrow{l} s_9, \quad s_9 \xrightarrow{e} s_{10}
\end{array} \right\} \]

\[ s_0 = s_0 \]

\[ S_A = \{s_3, s_5, s_{10}\} \]

An FA accepts a string \( x \) if and only if, starting in \( s_0 \), the sequence of characters in the string takes the FA through a series of transitions that leaves it in an accepting state when the entire string has been consumed.
More on FA

• A string composed of characters $x_1x_2x_3\ldots x_n$ is accepted by a FA iff,
  $\delta(\delta(\ldots \delta(\delta(s_0,x_1),x_2),x_3)\ldots x_{n-1}),x_n) \in S_A$

• FA encounters an error while processing a string $\Rightarrow$ lexical error or syntax error

• The entire input string is consumed, but did not reach a final state, $s_e \Rightarrow$ lexical error
Recognizer for unsigned integer

• How do we recognize numbers?
  – May design transition diagram for all numbers 😞

The actual text for a word recognized by a FA is called a **lexeme**.
113 is a lexeme of type “unsigned integer”
Implement Recognizer for Unsigned Integer

```c
char ← NextChar();
state ← s₀;
while (char ≠ eof and state ≠ sₑ) do
    state ← δ(state, char);
    char ← NextChar();
end;
if (state ∈ S_A) then report acceptance;
else report failure;
```

\[ S = \{s₀, s₁, s₂, sₑ\} \]

\[ \Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \]

\[ \delta = \begin{cases}
0 & s₀ \rightarrow s₁, \quad s₀ \rightarrow s₂ \\
0-9 & s₂ \rightarrow s₂, \quad s₁ \rightarrow sₑ
\end{cases} \]

\[ S_A = \{s₁, s₂\} \]

---

Implements the \( \delta \) function

The table can be further optimized for space
An identifier: *an identifier consists of an alphabetic character followed by zero or more alphanumeric characters* (definition used in C or Java)
Regular Expressions

• The set of words accepted by a FA, $F$, forms a language denoted by $L(F)$.
• The language, $L(F)$, can be described using a notation called Regular Expressions (RE).
  – Language described by a RE is called a Regular Language

• Regular Expressions and FAs are equivalent, but REs are more intuitive to express specifications of a language
## Notations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union of $L$ and $M$ written $L \cup M$</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>
Language expressed by RE: Example

- $\Sigma = \{a\} \Rightarrow \text{RE: } a$
- $\Sigma = \{\varepsilon\} \Rightarrow \text{set of words } = \{\varepsilon\}$
- $\Sigma = \{n, e, w\}$ and set of strings = \{new\}
  - RE: $n.e.w$ or just $new$
- Language contains two words $new$ and $while$
  - RE: $new \mid while$
- Language contains two words $new$ and $not$
  - RE: $new \mid not$ (alternative, $n(ew \mid ot)$
- If $x$ and $y$ are two RE for $L(x)$ and $L(y)$, then
  - $x \mid y$ denotes RE for all strings in $L(x) \cup L(y)$
  - $xy$ denotes RE for all strings in $L(x)L(y)$
  - $x^*$ denotes RE for all strings in $L(x)^*$
RE to express language token classes

• Identifiers: an alphabetic character followed by zero or more alphanumeric characters
  – ([A . . . Z] | [a . . . z]) ([A . . . Z] | [a . . . z] | [0 . . . 9])*

• Unsigned Integer: either zero or a nonzero digit followed by zero or more digits
  – 0 | [1 . . . 9] [0 . . . 9]*

• Unsigned real number:
  – (0|[1 . . . 9][0 . . . 9]*)(ε|.[0 . . . 9]*)

• Quoted character strings in Java or C:
  – “(^)”* → uses the complement operator
RE to express language token classes

Letter : (a|b|c ...|z|A|B|C| ... |Z)
Digit : (0|1|2| ... |9)
Identifier: Letter ( Letter | Digit)*

Integer : (+ | - | ε ) ( 0 | (1|2|3| ... |9) (Digit*))
Decimal : Integer . Digit*
Real : (Integer | Decimal) E (+|-| ε) Digit*
Complex : (Real . Real )

• Trying to be more specific with a RE can lead to complex expressions → complex FAs → more states to store
  • Does FA with more states take longer to execute than one with less states ?

• The cost of operating an FA is proportional to the length of the input, not to the length or complexity of the RE that generates the FA.
  • The cost of FA operation remains one transition per input character
Overview: From RE to Scanner

- Construct a non-deterministic finite automata (NFA) that recognizes an RE
  - uses $\varepsilon$-transitions
- Construct a Deterministic Finite Automation (DFA) to simulate the NFA
- Minimize the number of states in a DFA
- Generate scanner code
Non Deterministic Finite Automata

NFA: An FA that allows transitions on the empty string, $\varepsilon$, and states that have multiple transitions on the same character

$\varepsilon$-transition: a transition on the empty string $\varepsilon$, that does not advance the input

FAs that recognize strings “m” and “n” … how to recognize string “mn”

Which strings are accepted by NFA: Allows one or more $\varepsilon$-transitions between any two characters in a string
There can be multiple transitions from one state, unlike in a DFA

**Model of NFA**

1. Each time the NFA must make a nondeterministic choice, it follows the transition that leads to an accepting state for the input string, if such a transition exists.  
   ➔ Guess correctly at each transition to reach accept state

2. Each time the NFA must make a nondeterministic choice, the NFA clones itself to pursue each possible transition ➔ concurrent execution

**Configuration of an NFA:** the set of concurrently active states of an NFA
Regular Expression to NFA

Thomson’s construction
• Provides a template to construct a NFA corresponding to a single letter RE
• A transformation on NFAs that model the effect of each basic operator
  (alternation, concatenation, and closure)
Although NFA has more states, the construction can be easily automated.

The number of states can be reduced.
NFA to DFA

• DFA must simulate the NFA

• Two key functions:
  – move\( (s_i, a) \) : set of states reachable from \( s_i \) on input \( a \)
  – \( \varepsilon \)-closure\( (s_i) \) : subset of states reachable from \( s_i \) using input \( \varepsilon \)

• Subset construction technique:
  – Begin with start state, \( n_0 \), of NFA
  – Find \( \varepsilon \)-closure\( (n_0) \) \(\Rightarrow\) a set of states, \( N_0 \)
  – Take the input alphabet, \( a \), and move\( (N_0, a) \) and take the \( \varepsilon \)-closure of the new states
  – Iterate till no more states are left in the NFA
Example of Subset Construction

(a) NFA for “a(b | c)” (With States Renumbered)

\[
\begin{array}{c}
\bullet n_0 \xrightarrow{a} n_1 \xrightarrow{\epsilon} n_2 \xrightarrow{\epsilon} n_3 \xrightarrow{\epsilon} n_4 \xrightarrow{b} n_5 \xrightarrow{\epsilon} n_6 \xrightarrow{c} n_7 \xrightarrow{\epsilon} n_8 \xrightarrow{\epsilon} n_9
\end{array}
\]

\[
\begin{array}{c}
\epsilon \\
\end{array}
\]

(b) Iterations of the Subset Construction

<table>
<thead>
<tr>
<th>Set Name</th>
<th>DFA States</th>
<th>NFA States</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q_0</td>
<td>d_0</td>
<td>n_0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{n_1, n_2, n_3} {n_4, n_6, n_9}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- none -</td>
</tr>
<tr>
<td>q_1</td>
<td>d_1</td>
<td>{n_1, n_2, n_3, n_4, n_6, n_9}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- none -</td>
</tr>
<tr>
<td>q_2</td>
<td>d_2</td>
<td>{n_5, n_8, n_9, n_3, n_4, n_6}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- none -</td>
</tr>
<tr>
<td>q_3</td>
<td>d_3</td>
<td>{n_7, n_8, n_9} {n_3, n_4, n_6}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- none -</td>
</tr>
</tbody>
</table>

(a) Resulting DFA

\[
\begin{array}{c}
d_0 \xrightarrow{a} d_1 \xrightarrow{b} d_2 \xrightarrow{c} d_3 \xrightarrow{b} d_1
\end{array}
\]
Subset Construction

q₀ ← ε-closure({m₀});
Q ← q₀;
WorkList ← {q₀};

while (WorkList ≠ Ø) do
    remove q from WorkList;
    for each character c ∈ Σ do
        t ← ε-closure(Delta(q, c));
        T[q, c] ← t;
        if t ≠ Q then
            add t to Q and to WorkList;
        end;
    end;
end;

DFA from subset construction:
- Each qᵢ ∈ Q is a state in the DFA
- If qᵢ contains an Accepting state of the NFA, then qᵢ is an Accepting state of DFA
- Transitions are based on T by observing the mapping from qᵢ
- Start state: q₀ (all states reachable by ε-transition from n₀)
Points on Subset Construction

- The set $Q$ in the algorithm contains elements, $q_i$
  - Each $q_i$ is a subset of $N$ (states in NFA) $\Rightarrow q_i \in 2^N$

- When algorithm terminates, each $q_i \in Q$ corresponds to a state, $d_i$ in the DFA

- At each step in the while loop, each $q$ represents a valid configuration of the NFA
  - Apply $\text{Move}(s,c)$ on the configuration to get the next state of the next state $\Rightarrow \text{Move}(s,c)$ returns $\bigcup_{s \in q_i} \text{Move}(s,c)$

- What is the maximum possible states that the DFA can have?
  - $|2^N|$ distinct states where $N$ is the number of states of NFA
Minimal DFA construction

• Minimize the number of states since smaller scanner needs less memory
  – No impact on execution time

• Detect when two states are equivalent
  – Produces same behavior on any input string ➔ set of paths leading to the states are equivalent
DFA Minimization Basics

- Group states into maximal size sets
- Iteratively subdivide the sets based on the transition graph
- States that stay grouped are equivalent
- Begin with initial partition of all accepting states, $D_A$, and $\{D-D_A\}$
- Split $\{D-D_A\}$ based on the transition due to an input “a”
DFA Minimization Algo

\[
\begin{align*}
T & \leftarrow \{D_A, \ D - D_A\}; \\
P & \leftarrow \emptyset \\
\text{while } (P \neq T) \text{ do} & \\
& \quad P \leftarrow T; \\
& \quad T \leftarrow \emptyset; \\
& \quad \text{for each set } p \in P \text{ do} \\
& \quad \quad T \leftarrow T \cup \text{Split}(p); \\
& \quad \text{end;} \\
& \quad \text{end;}
\end{align*}
\]

\[
\text{Split}(S) \{
\quad \text{for each } c \in \Sigma \text{ do} \\
\quad \quad \text{if } c \text{ splits } S \text{ into } s_1 \text{ and } s_2 \\
\quad \quad \quad \text{then return } \{s_1, s_2\}; \\
\quad \text{end;}
\}
return S;
\]

DFA for \(a(b|c^*)\).

(a) Original DFA
(b) Initial Partition
DFA as scanner for a Language

- Language has multiple syntactic categories
  - Multiple REs to represent each of them
- Construct a single RE for the language by forming \((r_1 \mid r_2 \mid r_3 \mid \ldots \mid r_k)\)
- The scanner constructed using the RE should return the \textit{token} and its \textit{syntactic category}

- Real programs contain multiple words
  - FAs recognize single words \(\Rightarrow\) need some change
  - Use a delimiter to mark end of each word
    - What is the problem with this approach?
  - Find the longest word that matches a syntactic category or RE
    - If ( Reached a state where there is no further transition) then
    - If (this is an accepting state), then word found
    - Else, if (crossed an accepting state), then back up to most recent state
      » Else error
- There can be multiple REs accepting a word \(\Rightarrow\) which RE to pick as the syntactic category
  - One can specify priority among Res
  - The first re has highest priority, while the last re has lowest priority
Table Driven Scanners

- A simple scanner has 4 sections
  - Initialization
  - Scanning loop that models DFA
  - Rollback loop
  - Final section that interprets and reports result

```c
NextWord()
    state ← s₀;
    lexeme ← " ";
    clear stack;
    push(bad);
    while (state≠sₑ) do
        NextChar(char);
        lexeme ← lexeme + char;
        if state ∈ S_A
            then clear stack;
            push(state);
            cat ← CharCat[char];
            state ← 8[state,cat];
        end;
        while(state ≠ S_A and state≠bad) do
            state ← pop();
            truncate lexeme;
            RollBack();
        end;
        if state ∈ S_A
            then return Type[state];
        else return invalid;
    end;
```
Notes on scanner implementation

- Table driven scanner can be optimized further by marking dead-end transitions

- Other techniques for generating scanners
  - Direct-coded scanners
    - A direct coded scanner has specialized code fragment to implement each state of DFA ➔ no two-way table lookup required similar to table-driven scanners
  - Hand-coded scanners
    - Can reduce the overhead of interfacing between the scanner and the rest of the system
Summary

• Finite Automata or Transition Diagrams to recognize words
• Regular Expressions to represent token classes
• Convert from RE to FA
• Automatic scanner generator from FA