CS415 Compilers

Lexical Analysis
Part 4

Syntax Analysis
Part 1

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
• Third homework has been posted.

• First homework has been graded.

• Reminder: First project deadline extension: Tuesday (03/02) for code, and Friday (03/05) for report.
Automating Scanner Construction

RE $\rightarrow$ NFA (Thompson’s construction)
- Build an NFA for each term
- Combine them with $\varepsilon$-moves

NFA $\rightarrow$ DFA (subset construction)
- Build the simulation

DFA $\rightarrow$ Minimal DFA
- Hopcroft’s algorithm

DFA $\rightarrow$ 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
\[ a \ (b \mid c)^* : \]

Applying the subset construction:
Applying the subset construction:

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s_0)</td>
<td>(q_0)</td>
<td>(q_1, q_2, q_3, q_4, q_6, q_9)</td>
<td>none</td>
</tr>
<tr>
<td>(s_1)</td>
<td>(q_1, q_2, q_3, q_4, q_6, q_9)</td>
<td>none</td>
<td>(q_5, q_8, q_9, q_3, q_4, q_6)</td>
</tr>
<tr>
<td>(s_2)</td>
<td>(q_5, q_8, q_9)</td>
<td>none</td>
<td>(s_2)</td>
</tr>
</tbody>
</table>
| \(s_3\)    | \(q_7, q_8, q_9\) | none | \(s_2\) | \(s_3\)

**Final states**

\( a(b | c)^* : \)
The DFA for \( a \ (b \mid c)^* \)

- Ends up smaller than the NFA
- All transitions are deterministic
Automating Scanner Construction

RE $\rightarrow$ NFA (*Thompson’s construction*)
- Build an NFA for each term
- Combine them with $\varepsilon$-moves

NFA $\rightarrow$ DFA (*subset construction*)
- Build the simulation

DFA $\rightarrow$ Minimal DFA
- Hopcroft’s algorithm

DFA $\rightarrow$ RE (*not really part of scanner construction*)
- All pairs, all paths problem
- Union together paths from $s_0$ to a final state
How do we know whether two states encode the same information?

$q_1$ and $q_2$ are not equivalent. “$w$” is a witness that they are not equivalent.

Intuition: Two states are equivalent if for all sequences of input symbols “$w$” they both lead to an accepting state, or both end up in a non-accepting state.
The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state
The Big Picture

• Discover sets of equivalent states
• Represent each such set with just one state

Two states are equivalent if and only if:

• $\forall a \in \Sigma$, transitions on $a$ lead to equivalent states \hspace{1cm} \text{(DFA)}$
• if $a$-transitions to different sets $\implies$ two states must be in different sets, i.e., cannot be equivalent
The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:

- \( \forall a \in \Sigma, \text{ transitions on } a \text{ lead to equivalent states } \) (DFA)
- if \( a \)-transitions to different sets \( \Rightarrow \) two states must be in different sets, i.e., cannot be equivalent

A partition \( P \) of \( S \)

- Each state \( s \in S \text{ is in exactly one set } p_i \in P \)
- The algorithm iteratively partitions the DFA’s states
Details of the algorithm

- Group states into maximal size sets, **optimistically**
- Iteratively subdivide those sets, as needed
- States that remain grouped together are equivalent

**Initial partition, \( P_0 \), has two sets:** \( \{F\} \) & \( \{Q-F\} \) \hspace{1cm} (D = (Q, \Sigma, \delta, q_0, F))

**Splitting a set ("partitioning a set \( s \) by \( a \")**

- Assume \( q_a, q_b \in s \), and \( \delta(q_a, a) = q_x \), & \( \delta(q_b, a) = q_y \)
- If \( q_x \) & \( q_y \) are not in the same set, i.e., are considered equivalent, then \( s \) must be split
  \( \rightarrow q_a \) has transition on \( a \), \( q_b \) does not \( \Rightarrow \) \( a \) splits \( s \)
The algorithm

\[
P \leftarrow \{ F, \{Q-F\}\}
\]

while (P is still changing)

\[
T \leftarrow \{\}
\]

for each set \( S \in P \)

\[
T \leftarrow T \cup \text{split}(S)
\]

\[
P \leftarrow T
\]

\text{split}(S): \quad \text{for each } a \in \Sigma

\quad \text{if } a \text{ splits } S \text{ into } S_1, S_2, \ldots \text{ then}

\quad \quad \text{return } \{S_1, S_2, \ldots\}

\quad \text{else return } S

Why does this work?

- Start off with 2 subsets of \( Q \) \{F\} and \{Q-F\}
- \textbf{While} loop takes \( P_i \rightarrow P_{i+1} \) by splitting 1 or more sets
- \( P_{i+1} \) is at least one step closer to the partition with \(|Q|\) sets
- Maximum of \(|Q|\) splits

Note that

- Partitions are \textbf{never} combined

This is a fixed-point algorithm!
Back to our DFA Minimization example

Then, apply the minimization algorithm

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Split on</th>
</tr>
</thead>
<tbody>
<tr>
<td>{s_1, s_2, s_3} {s_0}</td>
<td>a</td>
</tr>
<tr>
<td>{s_1, s_2} {s_0}</td>
<td>b</td>
</tr>
<tr>
<td>{s_1} {s_0}</td>
<td>c</td>
</tr>
</tbody>
</table>

To produce the minimal DFA

We observed that a human would design a simpler automaton than Thompson’s construction & the subset construction did.

Minimizing that DFA produces the one that a human would design!
Another Example Register Specification

Start with a regular expression

\( r0 \mid r1 \mid r2 \mid r3 \mid r4 \mid r5 \mid r6 \mid r7 \mid r8 \mid r9 \)

The Cycle of Constructions

\[ \text{RE} \rightarrow \text{NFA} \rightarrow \text{DFA} \rightarrow \text{minimal DFA} \]
Thompson’s construction produces

\[
\begin{align*}
& r_0 \\
 & r_1 \\
 & r_2 \\
 & \vdots \\
 & s_0 \\
& r_8 \\
& r_9 \\
\end{align*}
\]

**The Cycle of Constructions**
Abbreviated Register Specification

The subset construction builds

This is a DFA, but it has a lot of states ...

The Cycle of Constructions

RE → NFA → DFA → minimal DFA
The DFA minimization algorithm builds

This looks like what a skilled compiler writer would do!

\[ r \quad 0,1,2,3,4, \quad 5,6,7,8,9 \]

\[ s_0 \quad s_f \]

---

**The Cycle of Constructions**

\[ \text{RE} \rightarrow \text{NFA} \rightarrow \text{DFA} \]

\[ \text{minimal DFA} \]
Advantages of Regular Expressions

- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — an expression grammar

\[
\begin{align*}
Term & \rightarrow [a-zA-Z] ([a-zA-Z] | [0-9])^* \\
Op & \rightarrow + | - | * | / \\
Expr & \rightarrow ( Term Op )^* Term
\end{align*}
\]

Of course, this would generate a DFA ...

If REs are so useful ...

*Why not use them for everything?*
Limits of Regular Languages

Not all languages are regular

RL’s ⊂ CFL’s ⊂ CSL’s

You cannot construct DFA’s to recognize these languages

• \( L = \{ p^k q^k \} \)  
  (parenthesis languages)

• \( L = \{ wcw^r \mid w \in \Sigma^* \} \)

Neither of these is a regular language

But, this is a little subtle. You can construct DFA’s for

• Strings with alternating 0’s and 1’s
  \(( \varepsilon \mid 1 ) (01)^* ( \varepsilon \mid 0 )\)

• Strings with and even number of 0’s and 1’s

• Strings of bit patterns that represent binary numbers which are divisible by 5 (homework)
Poor language design can complicate scanning

- **Reserved words are important**
  
  if then then then = else; else else = then  
  
  *(PL/I)*

- **Insignificant blanks**
  
  do 10 i = 1,25  
  do 10 i = 1.25  
  
  *(Fortran & Algol68)*

- **String constants with special characters**
  
  newline, tab, quote, comment delimiters, ...  
  
  *(C, C++, Java, ...)*

- **Limited identifier “length”**
  
  *(Fortran 66 & PL/I)*
Parsing
(Syntax Analysis)

EAC Chapters 3.1 - 3.2
Review: The Front End

Parser
- Checks the stream of **words** and their **parts of speech** (produced by the scanner) for grammatical correctness
- Determines if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code
The process of discovering a derivation for some sentence

- Need a mathematical model of syntax — a grammar $G$
- Need an algorithm for testing membership in $L(G)$
- Need to keep in mind that our goal is building parsers, not studying the mathematics of arbitrary languages

Roadmap
1. Context-free grammars and derivations
2. Top-down parsing
   - LL(1) parsers, hand-coded recursive descent parsers
3. Bottom-up parsing
   - Automatically generated LR(1) parsers
Specifying Syntax with a Grammar

Context-free syntax is specified with a context-free grammar

\[
\text{SheepNoise} \rightarrow \text{SheepNoise} \ baa \\
| \ baa
\]

This CFG defines the set of noises sheep normally make.

It is written in a variant of Backus-Naur form.

Formally, a grammar is a four tuple, \( G = (S,N,T,P) \)

- \( S \) is the start symbol (set of strings in \( L(G) \))
- \( N \) is a set of non-terminal symbols (syntactic variables)
- \( T \) is a set of terminal symbols (words or tokens)
- \( P \) is a set of productions or rewrite rules (\( P: N \rightarrow (N \cup T)^* \))

\[
L(G) = \{ \ w \in T^* \ | \ S \Rightarrow^* w \}
\]
We can use the *SheepNoise* grammar to create sentences

→ use the productions as *rewriting rules*

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td><em>SheepNoise</em></td>
<td>—</td>
<td><em>SheepNoise</em></td>
</tr>
<tr>
<td>2</td>
<td>baa</td>
<td>1</td>
<td><em>SheepNoise  baa</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td><em>SheepNoise  baa  baa</em></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>baa  baa  baa*</td>
</tr>
</tbody>
</table>

*And so on ...*
More Syntax Analysis (top-down parsing)

Read EaC: Chapter 3.1 - 3.3