Lexical Analysis

Syntax Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• First homework solutions available on sakai under Resources tab

• Third homework has been posted. Due date: Tuesday, February 26

• First project has been posted. Due dates: CODE Monday March 4; REPORT Wednesday March 6

• Midterm: Wednesday, March 13, in class; closed book, closed notes, 80 minutes
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

- Regular Expression (RE)
- Non-Deterministic Finite Automaton (NFA)
- Deterministic Finite Automaton (DFA)
Thompson’s construction produces

The Cycle of Constructions

Abbreviated Register Specification
The subset construction builds

\[ \text{This is a DFA, but it has a lot of states ...} \]

The Cycle of Constructions
The DFA minimization algorithm builds

This looks like what a skilled compiler writer would do!

The Cycle of Constructions
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

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

Of course, this would generate a DFA ...

If REs are so useful ...

*Why not use them for everything?*
Not all languages are regular

\[ RL's \subset CFL's \subset CSL's \]

You cannot construct DFA’s to recognize these languages

- \[ L = \{ p^kq^k \} \]
  (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)
What can be so hard?

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

SheepNoise → 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 \( \text{(set of strings in } L(G)) \)
- \( N \) is a set of non-terminal symbols \( \text{(syntactic variables)} \)
- \( T \) is a set of terminal symbols \( \text{(words)} \)
- \( P \) is a set of productions or rewrite rules \( \text{(} P : N \rightarrow (N \cup T)^* \text{)} \)

\[
L(G) = \{ w \in T^* \mid S \Rightarrow^* w \}
\]
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>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td><em>SheepNoise</em></td>
</tr>
<tr>
<td>2</td>
<td><em>baa</em></td>
</tr>
</tbody>
</table>

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

And so on ...
To explore the uses of CFGs, we need a more complex grammar $G$:

- Such a sequence of rewrites is called a *derivation*.
- Process of discovering a derivation is called *parsing*.

We denote this derivation: $\text{Expr} \Rightarrow^* \text{id} - \text{num} \ast \text{id}$

$x - 2 \ast y \in L(G)$ ?
Derivations

- At each step, we choose a non-terminal to replace
- Different choices can lead to different derivations

Two derivations are of interest
- **Leftmost derivation** — replace leftmost NT at each step; generates left sentential forms \((\Rightarrow^*_{lm})\)
- **Rightmost derivation** — replace rightmost NT at each step; generates right sentential forms \((\Rightarrow^*_{rm})\)

These are the two systematic derivations

*(We don’t care about randomly-ordered derivations!)*

The example on the preceding slide was a leftmost derivation
- Of course, there is also a rightmost derivation
- Interestingly, the resulting parse trees may be different
<sentence> ::= <subject> <verb> <rest>

Example:

time flies like an arrow
fruit flies like a banana.
More Syntax Analysis (top-down parsing)

Read EaC: Chapter 3.3