CS 415: Lecture 4

- Lexical Analysis
- Syntax analysis, part 1

Project

- Some small changes to both project1.pdf and project1.tar.gz
  - Re-download or re-print project1.pdf
  - Change in project1.tar.gz not significant; if you see a 1998 due date in the README file, ignore it 😊
- Any questions?
Review of NFA -> DFA Conversion

- What is $e$-closure($T$)?
- What is move ($T$, $a$)?
- Can you think of a brute-force approach to constructing a DFA given an NFA?

Conversion Algorithm

Assume all states in NFA are unmarked initially.
Let $S = e$-closure(start state of NFA).
Let $D = \{S\}$.
while $\exists$ an unmarked state $T \in D$ do
    Mark $T$;
    $\forall$ input symbols $a$ do
    \{ $U = e$-closure (move ($T$, $a$));
    if $U \not\in D$ then {add unmarked $U$ to $D$};
    $D_{trans}(T, a) = U$;
    \}
endwhile
Possible Problems

- Theoretically, for NFA having $n$ states can get DFA with $2^n$ states
  - Fortunately, this typically doesn’t happen in practice.
- Token is recognized if the ending state of the DFA contains an original final state of the NFA.
  - In case of choice, use final state which represents the earliest rule in the list of productions for tokens

NFA to DFA Conversion

- DFA derived is not the most efficient (smallest possible), but is usually of practical size
- There are ways of obtaining an optimal DFA by minimizing the numbers of states
NFA -> DFA vs. NFA Simulation

- Recall that we used an intuition from how one would simulate the execution of an NFA to derive the NFA -> DFA conversion, how come we don't just simulate the NFA's execution directly?
  - Question of space vs. time trade-off
  - Simulating NFA requires computing $\epsilon$-closure() and move() at runtime
  - Transforming to DFA requires more space to keep states (possible $2^n$)

Practical FAs

- Encode transitions as a table
  - Each column is an input symbol
  - Each row is a state
  - Entry at $(s, i)$ is state to transition to when in state $s$ and see input $i$
- Scanner has to try to find longest match in input to a possible token
  - May have to look beyond end of token to do this!
Syntax Analysis

- Parser
  - Perform context-free syntax analysis
  - Guide the context-sensitive analysis
  - Construct intermediate representation
  - Produce meaningful error messages
  - Attempt error correction

Context Free Grammar

- Context-free syntax can be specified with context-free grammars
- A context-free grammar $G$ is a four-tuple $(T, NT, S, P)$, where
  1. $T$ is the set of terminal symbols in the grammar. For our purposes, the set of terminals is equivalent to the set of tokens returned by the lexical analyzer.
  2. $NT$ is a set of syntactic variables that denote sets of strings occurring in the language. These are used to impose a structure on the grammar useful for syntax analysis.
  3. $S$ is a distinguished nonterminal ($S \in NT$) that denotes the entire set of strings in $L(G)$. This is called the start symbol.
  4. $P$ is a set of productions that specify the way that terminals and non-terminals can be combined to form strings in the language. Each production consists of a nonterminal, a produce operator, and a string of nonterminals and terminals.
Backus-Naur Form (BNF)

- Grammars are often written in BNF notation.
- The following grammar defines simple expressions over identifiers and numbers:

  \[
  \begin{align*}
  \text{goal} & :: = \text{expr} \\
  \text{expr} & :: = \text{expr} \text{op} \text{exp} | \text{number} | \text{id} \\
  \text{op} & :: = + | - | * | / \\
  \text{nonterminal} & :: = \text{string1} | \text{string2} \text{is a shorthand for} \\
  \text{nonterminal} & :: = \text{string1} \\
  \text{nonterminal} & :: = \text{string2}
  \end{align*}
  \]

Why Context-Free Grammars?

- Precise and easy-to-understand (?) syntactic specification of programming languages.
- Give structure to language.
- Easier to maintain (e.g., adding new language features).
- Can automatically construct efficient parsers for certain (sub)classes of context-free grammars.
CFG vs RE

- The previous list looks very similar to "why REs" list. Why not just use REs?
- Every language that can be described by an RE can also be described by a CFG
  - Example: let's construct a CFG for (a|b)*abb
- CFG can describe languages that REs cannot
  - Example: can you write an RE to describe the language that contains "all strings of balanced parentheses?"
    - Here's the CFG: S -> (S) | ε
- Grammars that generate regular sets (languages described by REs) are called regular grammars
- Regular grammars ⊆ context-free grammars

Scanning vs. Parsing

- Typically, regular expressions are used to classify identifiers, numbers, keywords
- Context-free grammars are used to count
  - Brackets - (), begin - end, if - then - else
  - Encodes structure - expressions
Complexity of Parsing

- Regular grammars - DFAs - $O(n)$
- CFGs - Early's algorithm - $O(n^3)$
- LR and LL grammars - $O(n)$
- CSGs - LBAs - P-space complete
- Unconstrained - Turing machines

Chomsky's hierarchy

Next Time

- Read ASU 4.3 & 4.4