Scanning; Grammars

CS 440: Programming Languages and Translators, Spring 2019
Lecture 8, Wed 2/13

Scanning / Lexing

- Compilers often use regular expressions as part of the "tokenizing" process (take the initial input of characters) and turn it into a sequence of tokens. For example, $xy \leq z$ becomes $id(xy) \ operator(\leq) \ id(z)$.
  - Basically, you have a finite automaton that looks for identifiers or operators or etc.
  - Each path to an accepting state is for a particular kind of token.
  - Build the token and restart the automaton to look for the next token.
  - There aren’t any missing subtleties to look for in this process.
- Building the automaton is fairly mechanical given descriptions of the common tokens found in programming languages (identifier, some id’s are keywords, operator, token, parentheses, other special symbols, …)
  - Identifier is $[A-Za-z_] \ [A-Za-z0-9_]*$
  - Operator is $+ \ | \ - \ | \ * \ | \ / \ | \ ** \ | \ <= \ |$ etc.

- There are lexer-generator programs that take these kinds of descriptions and output a lexer.
  - E.g., lex on Unix.

Next chapter - Grammars, Syntaxes, Derivations etc.

- Given a list of tokens, does it look like a program? What is the structure of this program?
- Parsing takes a grammar describing a language and builds a structure describing the input
  - Program, expression, etc.
- Grammar - set of rules for describing the structure of syntactic constructs in the language.
- To describe a grammar, we need
  - A set of nonterminals (“nonterminal symbols”) $N$. A nonterminal describes a set of syntactic constructs (e.g. noun, verb, …)
  - Set of terminals (“terminal symbols”) $\Sigma$
  - Production rules: Tell us how to take symbols and produce other symbols.
    - Nonterminals have to be expanded to a list of terminals.
    - A sentence is a set of terminals in the language.
    - $Expression \rightarrow Expression \ Operator \ Expression$ - an example of a production
  - A starting nonterminal: a member of $N$, typically $S$.
    - To see if a string of tokens is a sentence, we have to find a way to start with $S$, apply production rules, and end up with the given string of tokens.
Kinds of Grammars

- We usually work with context-free grammars (CFGs)
  - The rules have the form \( \text{Nonterminal} \rightarrow \text{string of nonterminals and/or terminals} \) (possibly \( \epsilon \)).
  - \( \text{Operator} ::= + | - | \text{etc.} \)
  - Advantages are that
    - CFGs are expressive (can describe a lot of the things we need to discuss as parts of a programming language).
    - You can build parsers for a certain subset of CFGs that run deterministically and reasonably quickly.

- Regular Grammars - Same languages as regular expressions and NFAs
  - A nonterminal \( \rightarrow \) a single terminal or a single terminal followed by a nonterminal.
    - \( A \rightarrow b \mid c D \)
  - The nonterminals are states in an NFA; following \( A \rightarrow c D \) means there's an NFA arc from \( A \) to \( D \) labeled \( c \). \( A \rightarrow b \) means there's an arc to an accepting state.
  - Advantages are that
    - Easy to write programs that look for various simple constructions in regular languages (lexers, e.g.)
      - Identifier, floating-point-constant, …
    - Disadvantage -- You can't get languages complex enough to make good programming languages.

- There are also Context-Sensitive Grammars
  - Their rules have the form \( \alpha A \beta \rightarrow \alpha B \beta \) where \( A \) and \( B \) are nonterminals and \( \alpha \) and \( \beta \) are strings of terminals and/or nonterminals. CFGs are CSGs with empty \( \alpha \) and \( \beta \) on lhs.
  - Much more expressive than CFGs, difficult to parse.

- Past that there are General Grammars
  - Any lhs \( \rightarrow \) any rhs
  - Describe any language we know how to describe
  - No parsers