CSCI 1260: Compilers and Program Analysis
Steven Reiss
Fall 2017

Lecture 3: Project Overview and Lexical Analysis

I. Project Ideas
   A. Each person/team get to present their ideas
      1. Recruit others if desired
      2. Describe the challenge
   B. Provide feedback

II. Lexical analysis or Scanning
   A. Why do lexical analysis
      1. Specifying a grammar in terms of characters would be messy at best
         a. Different things are significant (white space, comments)
         b. Different level of specification
      2. Generally, splitting a transformation into simpler, independent
         transformations is easier
   B. What is lexical analysis
      1. Input is the source file
         a. What about C/C++
         b. What would you do for a simple preprocessor
            i. Includes
            ii. Other macros
      2. Output is tokens
         a. What is a token – a lexical unit
         b. What does it consist of
            i. Unit identifier (small int, enumeration) describing type of token
            ii. Associated value (for numbers, strings, keywords?, …)
               - Can be more general (RELOP, BOOLCONST)
            iii. Position information for error messages
      3. Lexical analysis is mapping characters to tokens
   C. Properties
      1. Speed (why is this important)
      2. Ease of modification
      3. Ease of writing
4. Ability to assist the parser
D. Several ways of doing this

III. Brute Force scanners
A. Write code to implement the scanner directly
   1. Typically switch on first character
   2. Based on that character, either return token or scan additional characters to build the actual token and value
   3. Character management: get next, check next, put back (or peek)
   4. Can use character classes
   5. Probably the most common (why?)
B. Pros and cons
   1. Can write a very fast analyzer (little overhead)
   2. Hard to modify as language changes (but generally few changes in tokens)
   3. Hard to debug

IV. More formal approach
A. A tokenizer is effectively a large FSA
   1. Input is the characters
   2. Output is tokens
   3. No need for anything fancier
      a. Next token is independent of previous one
      b. Tokens don’t involve balancing parentheses, etc.
   4. Examples
      a. Identifier, integer
B. Why not approach it this way
   1. Write the tokenizer as a large automata
C. Create a table that has characters across the top, states going down
   1. Each box can go to a new state or generate a token
      a. Token generation can call a routine to get the proper value
      b. String from start of token to end is available
   2. Using character classes makes this doable
      a. Keywords are special cases of identifiers
D. Pros and cons
   1. Still reasonably fast
   2. Difficult to specify, change

V. Better formal approach
A. Specify each token separately
1. Specifications are simpler
2. Why should you have to do the merging
   a. Let a tool merge for us
B. How to specify FSAs
   1. Draw them or build tables
   2. Regular expressions
C. Understanding regular expressions
   1. Try to create a regular expression for a floating point number
   2. Try to create a FSA for the same
   3. Creating regex from FSA
D. Regular expressions are NFSAs
   1. Can just create a single NFSA from all the regular expressions
   2. Is this sufficient?
      a. Need to have an action associated with a rule (i.e. build token)
      b. Can the rules be ambiguous?
   3. Ambiguity is inherent in the definitions
      a. 0, ‘.’, 1 versus 0.1
      b. “if,a” versus “ifa”
      c. You generally want the longest match
         i. When wouldn’t you? (errors in strings, comments)
         ii. Languages that require otherwise (Fortran)
      d. Why could this be problematic
         i. x .* y, x as two rules. How far to scan before y?
         ii. This can create a n^2 scanner
         iii. But it generally doesn’t arise in programming languages
E. NFSAs aren’t what you want to implement
   1. Computers are deterministic
   2. Luckily NFSA and DFSA have the same power (recall that)
F. Convert NFSA to DFSA
   1. Can be done automatically
   2. Set-of-states construction
   3. Generates a state table as before
   4. Don’t want to do this by hand however
VI. Tools
   A. Lex/flex
      1. Work for C/C++, designed to work with yacc/bison
         a. Can be used separately
b. Yacc/bison can be used without lex/flex
2. Take a set of definitions
3. Regular expression => action
   a. Action builds the token with type and value
   b. Action has access to a buffer containing the token
   c. Action can also be do nothing (white space, comment)
4. Extended regular expressions allowed
   a. regex1 / regex2 :: regex1 followed by regex2, but ignoring regex2
   b. Can also be done via pushing back characters in the action
5. Actions can set a different starting state
   a. Rules can be restricted to a particular starting state
   b. Different lexical contexts (i.e. preprocessor vs regular, php vs html)
   c. Splitting complex rules into parts
6. Ambiguity in regular expressions
   a. How would you express an identifier in a regular expression
      i. Is this correct
      ii. What about keywords
   b. There are 2 rules for resolving ambiguities
      i. First rule: choose the longer of 2 matches (as before)
      ii. Second rule: First regex=>action has priority over latter ones

B. JAVACC
1. JAVACC is a full parser generator, not just lexical analyzer
   a. But the specification is given in two parts
   b. The first part is essentially lexical specification
2. KIND : <regex>
   a. Kind = TOKEN : produce a token as output
      i. Tokens have type, no value, but include the original text
   b. Kind = SKIP : white space (ignored)
   c. Kind = SPECIAL_TOKEN : comment (ignored but tracked)
   d. Kind = MORE : used for complex expression
      i. Let you specify an internal state and then given additional rules
         for when in that state
   e. Can also specify states, following
C. Other tools exist for other languages
VII. HOMEWORK
A. Qi Xin will teach top-down parsing on Tuesday
B. READ CHAPTER 3.1-3.3
1. Exercises 3.2, 3.5