Basics of Compilers

Compiler phases

Front end — not interested in target machine / architecture
  • More related to actual language
  • Lexical analysis - break up input into stream of tokens (id, keyword, etc).
    • instead of if ( x >= 0 ) { id = yz * 25 ; }
    • Look at if ( x >= 0 ) { id = yz * 25 ; }
  • Parsing - make sure structure of input corresponds to programming language
    • if ( expr ) stmt
    • parsing error on if { stmt }
    • Typically produce a parse tree (displays detailed structure of program)
    • expr → term → factor → paren_expr → ( expr )
      → term → factor ...
    • Complicated tress related to how language grammars are designed
  • Semantic analysis
    • Check for undeclared variables, do typechecking, ...
  • Intermediate representation / code generation
    • Typically produce shorter easier-to-work-with representation
      • Expression tree
          *
          / \  
          x   3
      • Simple code (each instruction has ≤ 3 parts)
        • X = Y * Z ; A = X + Y; If A > 0 go to L

Middle end (may not exist)

• Code optimization
  • Language-specific code optimization
    • dead code elimination if true then stmt1 else stmt2 ⇒ stmt1
  • Architecture-independent code optimization
    • lift code out of loops

• If they share an intermediate representation, we can set things up so different language compilers all use the same program for these steps — this would be a "middle end"
Back end (specific to architecture and OS of target)

- **Machine-specific** code optimization
- **Code generation** (register allocation, reordering code in basic blocks)

Basics of Languages

- **Symbol** - can be a character, can be a larger token (e.g. `if` (a keyword), `xyz` (an id))
  - Might just be `id` with `xyz` attached as a property.
- **String notation** — often use `w` to name a string (sequence of symbols)
  - `|w|` is length of string
  - **string concatenation** operation — typically `w1 w2` (juxtapositioning ) or maybe `w1 . w2`.
  - **Empty string** `ε` has length 0, `ε w = w ε = w` for all `w`
  - **Prefix / suffix** of string — beginning or ending subsequence of a string
  - "`abc`" is an improper prefix / suffix of itself
- **Alphabet**: `Σ` = a (finite) set of symbols
  - `Σ*` (sigma star) is the (infinite) set of finite-length strings of symbols from `Σ` (including `ε`).
- **A language** `L` is a subset of `Σ*`. (I.e., a set of strings.)
  - Might be described using recognition ("Is this string in `L`?")
  - Or generation ("Here are strings in `L`")

Regular Expressions

- Heavily used in programming environments (searching text, recognizing or describing patterns)
- **Notation / pattern scheme** for denoting one of a particular set of languages ("regular" languages)
- Each reg expr corresponds to a particular regular language generated by the reg expr
  - Use a reg expr to determine whether a string is in the language or not.
- **Language of a reg expr** is the set of all strings it generates
  - Notation: `L(reg expr)`
  - Often infinite but generally still easily describable
  - E.g. sequence of `a`'s of length 2 – 7; sequence of `b`'s of length ≤ 5
    - strings of `a`'s and `b`'s with at least three `a`'s
    - `aaa, bbbbbabaaba, bbbbbbbabbbbabbabb

Syntax of Regular Expressions

- Exist different families / styles, but what follows is pretty typical.
- The empty string denotes `ε`
- A single character or symbol from `Σ` stands for that character
• Usually escape sequences \n, \t, etc.
• A sequence of reg exprs (concatenation). e.g. abc
• The OR of reg exprs (alternation). E.g., abc | cd
  • also see abc + cd
• A parenthesized regular expression (parens used for grouping)
• The postfix star of a reg expr (Kleene star)
  • Sequence of any number of strings from the reg expr (possibly 0)
  • a* stands for ε, a, aa, aaaa, etc.
  • (ab)* stands for ε, ab, abab, ababab, etc.
• Escaped characters \l, \(, \) often used to stand for characters ∈ Σ
  • They are metacharacters when used as part of describing a regular expression

Other Popular Regular Expressions
• Postfix Kleene plus (one or more occurrences of sub expr)
  • expr ? optional expression — same as ( | expr)
• [concatenated symbols] — any symbols in some set. E.g., [0123456789]
  instead of (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)
• [symbol1 -symbol2] any symbol in some sequence E.g. [0-9] or [a-z] or [a-zA-Z]
• [^symbols] — any symbol from Σ except for the named ones.
  • E.g. [^\t] — not a space nor a tab
    • Two nonempty strings separated by spaces or tabs
      • [^\t]+ [\t]+ [^\t]+
• dot (period) - any single character ∈ Σ
• Also used — binary . for concatenation, + for alternation

Precedences (strong to weak)
• star and plus, then concatenation, then or. [fixed Thu 2/14]
  • E.g. a b | c d * means (ab) | (c (d*))
• (a b | c d )+ e+ means (( (ab) | (cd) )* ) (e+)

Examples
• An identifier is a letter or underscore followed by any number of alphanumeric symbols (including underscore):
  • [a-zA-Z_] [a-zA-Z0-9_]*
  • Can't easily do: A letter or underscore etc. except for the strings if, then, else
    • i[a-zA-Z] | [^i] [a-zA-Z] ← not if
• An integer is a nonempty sequence of digits followed by an optional exponent
- where an exponent is "e" followed by a nonempty sequence of digits
- \[0-9]+(e[0-9]+)\]
- \[(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9\]* (| e (0|1|2|3|4|5|6|7|8|9) (0|1|2|3|4|5|6|7|8|9\)*

- Can't do
- String of a's and b's with more a's than b's
  - a, aab, aba, baa, aaab, abaa, baaa, aabb, abab, abba, baab, baba, ...

- Can do
- Sequence of a's and b's with at least 3 a's
  - b* a b* a b* (a | b)*
    - First three a's take care of the minimum 3, the (a | b)* takes care of any remaining a's

- Sequence of a's and b's with exactly 3 a's
  - b* a b* a b*

- Sequence of a's and b's with at most 3 a's
  - b* a b* | b* a b* a b* | b* a b* a b* a b*
    - notice it's harder to say things like "not > 3"

- A nonempty sequence of x separated by commas
  - x ( , x)*

- A nonempty sequence of x terminated by semicolons
  - (x ; )+ or x ; (x ; )* or x ( ; x)* ;