A. Basics of Compilers

Compiler phases

Front end is less 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 = y z * 25 ; }
  - Look at `if ( x >= 0 ) { id = y z * 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 trees 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)

B. Basics of Languages
  - Symbol - can be a character, can be a larger token (e.g. \texttt{if} (a keyword), \texttt{xyz} (an id)
    - Might just be \textit{id} with \textit{xyz} attached as a property.
  - String notation — often use \textit{w} to name a string (sequence of symbols)
    - Concatenate using juxtapositioning: \textit{w1 w2}
    - $|w|$ is length of string
    - Empty string $\epsilon$ has length 0, $\epsilon w = w \epsilon = w$ for all \textit{w}
  - Prefix / suffix of string — beginning or ending subsequence of a string
    - "abc" is an improper prefix / suffix of itself
  - Alphabet: $\Sigma = \text{a (finite) set of symbols}$
    - $\Sigma^*$ (sigma star) is the (infinite) set of finite-length strings of symbols from $\Sigma$ (including $\epsilon$).
    - "Kleene" star - zero or more copies of item
  - A language $L$ is a subset of $\Sigma^*$. (I.e., a set of strings.)
    - Might be described using recognition ("Is this string in $L$?")
    - Or generation ("Here are strings in $L$")

C. 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: $L(\text{reg expr})$
    - Often infinite but generally still easily describable
    - E.g. sequence of a's of length 2 – 7; sequence of b's of length $\leq 5$, strings of a's and b's with at least three a's (such as \texttt{aaa}, \texttt{bbbbbaaba}, \texttt{bbbbbbbbbbababbbabb}).

D. Syntax of Regular Expressions
  - Exist different families / styles, but what follows is pretty typical.
  - The empty string denotes $\epsilon$
  - A single character or symbol from $\Sigma$ stands for that character
• Usually escape sequences \n, \t, \r, etc.
• A sequence of reg exprs (concatenation). E.g., abc
• The OR of reg exprs (alternation). E.g., abc | cd
  • Also seen is plus sign: abc + cd
• 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, aaa, aaaa, etc.
  • (ab)* stands for ε, ab, abab, ababab, etc.
• A parenthesized regular expression (parens used for grouping)
  • Precedences: star stronger than concatenation stronger than alternation.
  • Examples:
    • a b | c d * means (ab) | (c (d*))
    • (a b | c d)* e* means (( (ab) | (cd) )*) (e*)
• Escaped metacharacters \|, \(, \), \* to get vertical bar, left paren, etc.

E. Other Popular Regular Expressions
• Postfix Kleene plus (one or more occurrences of subexpression)
• 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}
• \d for any decimal digit (i.e., \d means \{0−9\}).
• [^symbols] — any symbol from Σ except for the named ones.
  • E.g. [\^ \t] — not a space nor a tab (note space between ^ and \t)
  • Two nonempty strings separated by spaces or tabs
    • [\^ \t]+[\t]+[\^ \t]+
• dot (period) - any single character ε Σ
• ^ (caret) and $ - to indicate start/end of string (if matching just part of a string is allowed)
  • E.g., ^ab$ matches the one string ab whereas ab matches any string that contains ab as a
    substring, such as zzabz. (We might have an assumption that all strings are surrounded
    implicitly by ^...$ or assume they must be explicit.)
• Numeric escape codes: Notation varies but you might see \x{nnnn} or \unnnn for the character
  whose Unicode representation is hex nnnn. E.g., → is \u2192.

F. Examples of Regular Expressions
• 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”
  • Not if: i followed by not f or non-i followed by anything
  • \(^i[a-eg-z] \| \^[i][a-z]\) matches all two character strings except for 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
  • \([0-9]+(e[0-9]+)\) ?
• More examples:
  • Sequence of a’s and b’s with at least 3 a’s
    • b*ab*ab*a\{ab\}^*
    • First three a’s take care of the minimum 3, the \{ab\}^* takes care of any remaining a’s
  • Sequence of a’s and b’s with exactly 3 a’s
    • b*ab*ab*a matches strings with exactly 3 a’s ending in a (not aaab, for example).
  • 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 \(x (; x)\)^*;
• Can’t do comparisons of arbitrary length (can only compare up to some fixed number of cases).
  • String of a’s and b’s with more a’s than b’s
    • a, aab, aba, baa, aaab, aaba, abaa, baaab, abab, abba, baba, ...
    • But can do string of a’s and b’s of length ≤ 5 with more a’s than b’s
  • Brute force method: alternation with every possible case
  • \(\varepsilon | a | aab | aba | baa | aaab | aaba | ... \) (you get the idea)

-------------------------------------------------------------------------------- ended here 2019-09-11

Added 9/13:

G. Code for Simple Regular Expressions with just Concatenation and Alternation
• Let’s look at a concrete implementation of regular expressions. We’ll start off with just concatenation and alternation and worry about Kleene star some other time.
• We’ll start with the following datatype for patterns (reg exprs) and a \texttt{match} function that tries to match a pattern with the head of a sequence of items
-- Regular expressions with only concatenation and alternation [9/14]
data Pattern a
    = P_const a
    | P_or [Pattern a]
    | P_and [Pattern a] deriving (Eq, Read, Show)

match :: Eq a => Pattern a -> [a] -> Maybe [a]
....

The P_const Pattern Constructor; General Remarks on Pattern Matching

• The basic pattern is a constant (P_const) and we can combine them with alternation (P_or) and conjunction (P_and). For example, the reg expr ab|c would be represented using
  
  p = P_or [ P_and [P_const 'a', P_const 'b'], P_const 'c' ]

• Since this is a pattern on characters, the specific type for match is Pattern Char -> [Char] -> Maybe [Char], so match p string = Nothing (if the match fails) or Just string (if it succeeds). On success, the match routines removes the matching substring from the head of the input and returns a string with the leftover suffix.

• For example:
  
  match p "abcd" == Just "cd" -- the ab in ab|c matched, leaving cd
  match p "cxyz" == Just "xyz" -- the c in ab|c matched, leaving xyz
  match p "qrst" == Nothing -- both ab and c failed to match
  match p "acde" == Nothing -- the a matched but b didn't match cde
  -- and c didn't match acde

• Note we're not restricted to patterns of characters, we can have patterns on any type that supports equality testing, such as numbers, strings, ordered pairs, ...
  
  match (P_const "hi") ["hi", "there"] == Just ["there"]

The P_or Pattern Constructor

• The P_Or constructor takes a list of patterns and tries them one after another against the input list. The first match ends the search and match returns the result of that match. If none of the patterns match, match returns Nothing. The Just leftover list comes from whichever match succeeded.

• Some examples:
  
  por = P_or [P_const 1, P_const 2]
  match por [1,2,3] == Just [2,3] -- (1 or 2) matched the leading 1
  match por [2,3] == Just [3] -- (1 or 2) matched the leading 2
  match por [3,2] == Nothing -- (1 or 2) doesn't match leading 3

  por2 = P_or (map P_const "abc")
  match por2 "axy" == Just "xy"
  match por2 "bcd" == Just "cd"
match por2 "ccd" == Just "cd"
match por2 "def" == Nothing

por3 = P_or (map P_const ["hello","goodbye"])
match por3 ["hello", "and", "goodbye"] == Just ["and", "goodbye"]
match por3 ["goodbye", "and", "hello"] == Just ["and", "hello"]
match por3 ["aloha"] == Nothing

- To implement P_or, first we look at the list of patterns: If it’s empty, the match fails. Otherwise we try to match the head of the pattern list against the input. If that succeeds, we’re done. If it fails (returns Nothing), we recursively search the same input using P_or on the tail of the pattern list.

The P_and Pattern Constructor
- With P_and, we try to match a sequence of patterns against the input. First we match the head pattern against the input; if that fails, then we fail. If it succeeds (returns Just leftover), we continue the match using the tail of the pattern list on the leftover input. So we stop as soon as the first pattern fails or when we’ve used up all the patterns and succeeded.
- Here are some examples:
  pabc = P_and $ map P_const "abc"  -- look for "a" then "b" then "c"
match pabc "abcd" == Just "d"  -- "d" left after dropping "a", "b", "c"
match pabc "ab" == Nothing  -- "a" and "b" ok but matching "c" fails

match (P_and [pabc,pabc]) "abcabcz" == Just "z"
  -- match "abc", get Just "abcz", match "abc" and get Just "z"
match (P_and [por2,por2]) ["hello","goodbye","okay?"] == Just ["okay?"]
  -- 1st por2 matches "hello", 2nd matches "goodbye"
match (P_and [por2,por2]) ["hello","nope"] == Nothing
  -- 1st por matches "hello" but 2nd por doesn’t match "nope"

Skeleton code for match
- The code for match is considerably shorter than the explanation above of what it does.
- The skeleton below contains code for the P_const and P_or constructors but not P_and. (You’re to write that part for homework.)

- Match takes a pattern and input and tries to match the pattern with a prefix of the input. If it can’t, it returns Nothing. If it does find a matching prefix, then it returns Just of the leftover suffix.
  match :: Eq a => Pattern a -> [a] -> Maybe [a]

- For P_const val, check head of list for == val and return Just the tail of the input if we find it.
  match (P_const _) [] = Nothing
  match (P_const x) (h:t) | h == x = Just t
\[ | \text{otherwise} = \text{Nothing} \]

- For \texttt{P\_or}, try matching the head pattern; against the input if that succeeds with \texttt{Just suffix}, then we succeed the same way. If the head pattern fails to match, we recursively try to match the \texttt{P\_or} of the remaining patterns. If we run out of patterns, fail.

\[
\begin{align*}
\text{match } (\texttt{P\_or } []) \text{ vals} &= \text{Nothing} \\
\text{match } (\texttt{P\_or } (p:ps)) \text{ vals} &= \text{case match } p \text{ vals of} \\
& \quad \text{Nothing} \to \text{match } (\texttt{P\_or } ps) \text{ vals} \\
& \quad \text{ok } @ \text{(Just leftover)} \to \text{ok}
\end{align*}
\]

- (\ldots \texttt{P\_and} code omitted\ldots )
**Activity Questions, Lecture 6**

Give a regular expression for each of the following kinds of strings.

1. Integer constants ≥ 0 that don't begin with a leading zero except for 0 itself.
2. Same as (1) but with an optional minus sign.
3. Octal constants ≥ 0 that begin with a leading zero, including 0.
4. Floating point constants that include a dot and at least one digit before and after the dot. You can include constants that consist of only zeros (plus that dot).
5. Floating point constants that include a dot and at least one digit before or after the dot (or both).
6. Floating point constants that have an optional exponent of the form \(E\) followed by optional + or - and then an integer constant that doesn't begin with 0 (except for 0 itself). Also allowed are integer constants followed by a (required) exponent. A constant can also have a leading minus sign.

There are any number of other examples — find some and post them on Piazza!

**Solutions to Activity Questions**

1. \(0|\[1-9]\d*\)
2. \(-?0|\[1-9]\d*\)
3. \(0\[0-7]*\)
4. \(\d+\.\d+\)
5. \(\d+\.\d* | \.\d+\) (the first alternate matches numbers that begin with a digit, such as 12.3 and 12. and the second is for numbers that begin with a dot, such as in .1234)
6. \(-?(\(\d+\.\d* | \.\d+)E[+-]?0|\[1-9]\d*)|0|\[1-9]\d*E[+-]?0|\[1-9]\d*)\)

Breaking this down, \(\d+\.\d* | \.\d+\) is for a floating point constant with dot, \(E[+-]?0|\[1-9]\d*)\) is an optional exponent, and \(0|\[1-9]\d*E[+-]?0|\[1-9]\d*)\) is an integer constant with a required exponent. Surrounding the whole thing with –? (.....) adds the optional leading minus sign.