# *Compilers, Languages, Regular Expressions*

*CS 440: Programming Languages and Translators, Spring 2020*

## *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 `i f ( x > = 0 ) { i d = y z * 2 5 ; }`
  - 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 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
- **Language-specific** code optimization
  - Dead code elimination if true then stmt1 else stmt2 ⇒ stmt1

**Middle end (may not exist)**

- **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. `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)
  - **Concatenate** using juxtapositioning: *w1 w2*
    - $|w|$ is length of string
  - **Empty string** ε has length 0, $\varepsilon\ w = w\ \varepsilon = w$ 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).
  - "Kleene" star – zero or more copies of item.
- 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*. ")

## *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*(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 (such as `aaa`, `bbbbbaaba`, `bbbbbbbabbbbabbbabb`).

 

### *D. Syntax of Regular Expressions*

- Exist different families / styles, but what follows is pretty typical.
- ε denotes the empty string
- A single character or symbol from Σ stands for that character; usually also 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*) (using the empty string)
- `[`*concatenated symbols*`]` — any symbols in some set. E.g., `[0123456789]` instead of (`0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`)
- `[`$symbol_1$`-`$symbol_2$`]` 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 ∈ 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 `\u`*nnnn* 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. It's harder to say things like "not > 3"; we're doing it by specifying all the length 1, 2, and 3 strings
  - `b* a b* | b* a b* a b* | b* a b* a b* a b*`
- 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).
  - E.g., can't do string of `a`'s and `b`'s with more `a`'s than `b`'s
    - `a`, `aab`, `aba`, `baa`, `aaab`, `aaba`, `abaa`, `baaa`, `aabb`, `abab`, `abba`, `baab`, `baba`, `...`
  - But we can do string of `a`'s and `b`'s of length ≤ 5 with more `a`'s than `b`'s. (It's not pretty: We specify all the strings of lengths 1 through 5 with the property).
    - ε | `a` | `aab` | `aba` | `baa` | `aaab` | `aaba` | … (you get the idea)

 

# *Activity Questions, Lecture 6*

### *Compilers and Languages*

1. What's the difference between parsing and lexing? Why do both?
2. What are Σ and Σ* How are the different? How large are they?
3. What is a language over Σ*? What is the smallest language? The largest language?

### *Regular Expressions*

1. How do we denote concatenation and alternation in regular expressions?
2. In regular expressions, what are the differences between `(`, `)`, `+`, and `*` versus `\(`, `\)`, `\+`, and `\*` ?

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

3. Integer constants ≥ 0 that don't begin with a leading zero except for `0` itself. E.g., `0`; `1`; `15` but not `01`.
4. Same as in the previous question but with an optional minus sign.
5. Octal constants ≥ 0 that begin with a leading zero, including `0`. E.g., `0`; `000`; `07`; `077`; `89` but not `089`.
6. 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). E.g., `1.0`; `1.5`; `0.0`; `00.0` but not `1.` or `.5` .
7. Floating point constants as in the previous question but can omit digits before or after the dot (but not both). E.g., `1.` and `.5` are now included.
8. Exponent designations of the form `E` or `e` followed by optional `+` or `-` and then a natural number as in Question xxx. (We could optionally concatenate one of these onto an integer or floating-point constant.)

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

 

# *Solutions to Activity Questions*

## *Compilers and Languages*

1. Lexical analysis breaks up the input into small tokens (e.g. keywords, integer constants, operators, …). Parsing takes the output of the lexer and checks it for correctness relative to some language. We do both because lexing is straightforward and it makes parsing easier.
2. Σ is the alphabet of our language (e.g., `a`, `b`, `c`, …) and Σ* is the set of all finite-length strings from Σ (e.g., {the empty string ε, `a`, `b`, `c`, …, `aa`, `ab`, `ac`, …, `ba`, `bb`, `bc`, ….., `aaa`, … …, `aaaa`, …...}. Σ is finite but Σ* is infinite (countably infinite = countable using natural numbers 1, 2, …).
3. A language is any subset of Σ*; the smallest one is ∅, the largest one is Σ* itself. (Both of these are pretty boring as languages go.)

## *Regular Expressions*

1. Concatenation is denoted by juxtapositioning (writing strings next to each other: `ab cd` = `abcd` ). Alternation is denoted by vertical bar or plus sign between the alternatives.
2. The versions with backslashes, `\(`, `\)`, `\+`, and `\*`, denote the actual characters (left parenthesis, etc). Without backslashes, these have special meaning: `(`…`)` are used for grouping (similar to how we use them in (2+2)*4); `+` and `*` are postfix operators for Kleene plus and Kleene star (*exp*`+` is one or more occurrences of the *exp*; *exp*`*` is zero or more occurrences of *exp*, so *exp*`+` = *exp exp*`*`).

3. `0|[1-9]\d*`
4. `-?0|[1-9]\d*`
5. `0[0-7]*`
6. `\d+\.\d+`
7. `\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` )
8. `[eE][+-]?(0|[1-9]\d*)`

 