CS 164 Programming Languages and Compilers **Spring 2019**

Discussion 1/28: Lexical Analysis

Instructor: Paul N. Hilfinger GSIs: Nikhil Athreya, Vivant Sakore

1 Stages of Compilation

1. Compilers are conceptually broken down into multiple phases, each of which carry out transformations on a source program towards the compiler's ultimate goal; the generation of a target program. The result of each phase is an intermediate representation that facilitates the implementation of subsequent phases (or, if it is the last phase, the result is the target program). The first phase in a compiler, the lexical analyzer or *lexer*, reads an input stream of characters and converts it into a stream of *tokens* to be sent to the *parser* for syntactic analysis.

   (a) Convert the following Java source code into a sequence of tokens.

```
// n! = 1 * 2 * 3 * .... * (n-1) * n
public static int factorial(int n) {
    int result = 1;
    for(int i = 2; i <= n; i++)
        result *= i;
    return result;
}
```

   Answer:

```
KEYWORD("public") KEYWORD("static") KEYWORD("int") ID("factorial") LPAREN
KEYWORD("int") ID("n") RPAREN LCURLY KEYWORD("int") ID("result") ASSIGN
INTLIT("1") SEMI FOR LPAR KEYWORD("int") ID("i") ASSIGN INTLIT("2") SEMI
ID("i") LEQ ID("n") SEMI ID("i") OP("++") RPAREN ID("result") OP(*=)
ID("i") SEMI KEYWORD("return") ID("result") SEMI RCURLY
```

2. What are the rest of the stages of compilation? List them in order and write a brief description about what they and what they take as input and what they output.

   (a) Parsing: Takes in a stream tokens, outputs an abstract syntax tree (AST), and checks to make sure the program is syntactically correct (matching parentheses, well-formed expressions, etc.)

(b) Semantic Analysis: Takes in an AST, and makes sure the program makes sense semantically, does type checking as well. Returns a decorated AST (with the corresponding types of variables/expressions/etc. or a IR (intermediate representation) for optimization.

(c) (Optional) Optimization: Takes in an AST or an IR, and generates an optimized AST or IR.

(d) Code generation: Takes in an AST or IR and generates code for the specific backend which this compiler is targetting.

(e) (Optional) Optimization: Takes in the backend code and performs optimizations on that, leaving optimized back-end code.

(f) i. Executable generation: Takes in assembly files, assembles them into object files, and links object files to form an executable.

ii. Virtual Machine: Can interpret VM IR code or a decorated AST.

2 Regular Expressions

The lexer needs to scan and identify finite character sequences that match a pattern corresponding to a particular token (the matching strings are also called *lexemes*). Patterns of finite strings can be codified using *regular expressions*, each of which specify what is called a *regular language*. Kleene's theorem states that any regular language can be recognized by a finite state automaton (FSA) and any language that is recognized by an FSA is regular.

1. What language is denoted by each of the following regular expressions? Try writing down a few simple strings and give a concise description of the language.

   (a) `[_a-zA-Z][_a-zA-Z0-9]*`

   Answer: Identifiers

   (b) `((`ϵ`|a)b*)*` (Can you simplify this expression?)

   Answer: All strings containing a and b characters, including the empty string. Simplified: `[ab]*`

2. Write a regular expression for the following languages:

   (a) All strings of uppercase letters, where the letters are in ascending lexicographic order (empty string allowed).

   Answer: `A*B*...Z*`

   (b) The following four strings: October 8th, October 8, Oct 8th, Oct 8 (Be as concise as possible with your regex).

   Answer: `Oct(ober)?\s8(th)?`

(c) Even binary numbers without leading 0's. (Read from left to right; i.e.,"1101" is 13 and odd).

Answer: `1[01]*0 | 0`

(d) Non-empty binary numbers differing in the first and last bits.

Answer: `(0+1+)+ | (1+0+)+` or `0[01]*1 | 1[01]*0`

3. Check whether a string of nested open- and close- parentheses are balanced.

   This is not possible with classical regexes. However, there are regex dialects that support this. Intuitively, the reason why regexes can't match this type of string is because they need to somehow "remember" what string of parentheses they have seen so far (and keep track of the number of open- and close- parentheses) and regexes do not have this kind of power. We will get into this more with context-free grammars (CFGs) later in the course.

3 Regular Expressions with Backreferences

1. Suppose you have a large spreadsheet, each row of which has a `firstname`, `lastname`, and `email` column. You want to find all people who use an email address of the form `firstname+lastname@gmail.com`. Write a regular expression using backreferences that describes this pattern.

   Answer: `(\p{L}+),(\p{L}+),\1\+\2@gmail\.com`