Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue&Thu 12:00-13:15 @SH 212, Lab: Fri 10:00-11:15/11:30-12:45 @EB 103

Lab 1: A Tour of a Simple Compiler Frontend

Learning Objectives Upon successful completion, students will be able to:

- Explain the structure of a typical compiler frontend and the role of each component.
- Describe the interface between adjacent components, and the typical internal representations for the associated data.
- Be familiar with the basic Java language mechanisms relevant to compiler implementation.

How Does the Lab Work?

The labs of this course are designed to be self-guided. Just follow the instructions in the handouts. Instructor and/or TA are available for answering any questions during the lab sessions. For some labs, you may need extra time out the labs sessions to complete.

For this lab, your main task is to understand the external behavior of each component of a compiler frontend and the internal representations of program data. There is a sequence of units, 00 – 04. Walk through the units in order. In each unit, observe the provided programs as instructed, and do the exercises. While some of the implementation details are presented through out the units, and you may be asked to look inside the programs for further information, there is no requirement that you understand every line of the code.

Preparation

Log in to a Linux Lab machine and verify that you have Java 8 in your environment. Type `'java -version'`, and see if it shows version 1.8 or above:

```
linux> java -version
java version "1.8.0_05"
...
```

If not, you need to run `addpkg`:

```
linux> addpkg
```

which will show a list of available software packages. Use the cursor to select `java8`, then save and exit. You need to logout and re-login to see the change.

Download from D2L the file `lab1.zip` and unzip it. Enter the directory `lab1`. You'll see a set of sub-directories (00–04), each contains some Java programs that implement components of a compiler frontend. The driver program is called `Compiler.java`.

To compile the provided programs, enter a specific sub-directory, and do

```
linux> javac Compiler.java
```

To test a program with an input, do

```
linux> java Compiler <inputfile>
```

The Source Language

For this lab, we are compiling a very simple expression language. It is for describing the basic forms of *regular expressions*, and we'll call it *RE*.

RE's alphabet consists of just lower-case letters, and its operators are *alternate* (`|`) and *repeat* (`*`). (A third operation, *concatenate*, does not have an explicit operator.) In addition, *parentheses* are used to group components. RE expressions are all single-line expressions. Every RE expression must end with a newline character (`\n`).

RE's token definition and grammar definition (in Extended BNF notation) are shown here:

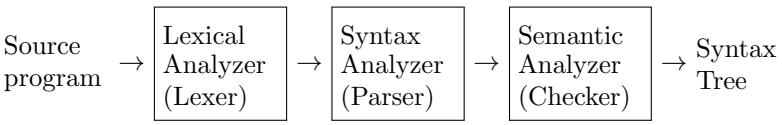
```
<LETTER: ['a'-'z']>
<ALTER:  '|'>
<REPEAT: '*'>
<LPAREN: '('>
<RPAREN: ')'>
<END:    '\n'>
```

```
Expr   -> Alter '\n'
Alter  -> Concat [ '|' Alter ]
Concat -> Repeat [ Concat ]
Repeat -> Atom [ '*' ]
Atom   -> '(' Alter ')' | <LETTER>
```

In the grammar, the square brackets `[,]` are EBNF's symbols for optional components, and the unquoted `|` is EBNF's alternation operator.

Compiler Frontend Review

Recall the typical structure of a compiler frontend:



The implementation of the RE language follows this structure.

00. File I/O in Java

The program `Echo.java` illustrates the handling of simple file input in Java. A file name is provided as a command-line argument. The program opens the file, repeatedly reads a character from the file and prints it out, until the end of the file. It then closes the file.

"from 00/Echo.java"

```
import java.io.*;

public class Echo {
  public static void main(String [] args) throws Exception {
    if (args.length == 1) {
      FileReader input = new FileReader(args[0]);
      int c, charCnt = 0;
      while ((c = input.read()) != -1) {      // read() returns -1 on EOF
        System.out.print((char)c);
        charCnt++;
      }
      input.close();
      System.out.println("Total: " + charCnt + " characters");
    } else {
      System.err.println("Need a file name as command-line argument.");
    }
  }
}
```

Note that we use Java's `FileReader` class to handle the input. This is because our input file is assumed to be a text file. If there is a need to read a binary file, then we need to switch to the `FileInputStream` class. There is no explicit character encoding mentioned in the program, which means the system default encoding (UTF-8) is used.

Exercise Copy `Echo.java` to a new file `Copy.java`. Modify the new program to take two file names as command-line arguments, one input file and one output file. Modify the code to copy the content of the input file to the output file, one character at a time. Don't forget that the class name also needs to be changed from `Echo` to `Copy`. Compile and test your program.

01. Lexer Implementation

The program `Lexer.java` implements a lexer for the RE language. As a typical lexer, it reads characters from an input file one at a time, and generates tokens as output.

Since a token is associated with multiple attributes, such as token code, lexeme, line and column numbers, it is typically represented by a *record* (in the Java case, a *class*). In our lexer, the token class definition is shown here:

"from 01/Lexer.java"

```
static class Token {
  int code;   // code
  char lex;   // lexeme
  int pos;    // position

  public Token(int code, char lex, int pos) {
    this.code = code; this.lex = lex; this.pos = pos;
  }
  public String toString() {
    return String.format("(%02d) [%d]\t%s", pos, code, (lex=='\n'? "\\n" : lex));
  }
}
```

The main lexer routine is `nextToken()`. It is the interface between the lexer and the parser. Each time this routine is called, the next token from input is returned.

"from 01/Lexer.java"

```
static Token nextToken() throws Exception {
  int c = nextChar();
  while (isSpace(c))
    c = nextChar();
  if (isLetter(c))
    return new Token(LETTER, (char)c, pos);
  switch (c) {
  case '|':  return new Token(UNION, (char)c, pos);
  case '*':  return new Token(REPEAT, (char)c, pos);
  case '(':  return new Token(LPAREN, (char)c, pos);
  case ')':  return new Token(RPAREN, (char)c, pos);
  case '\n': return new Token(END, (char)c, pos);
  }
  throw new Exception("(" + pos + ") Lexical Error: Illegal character " +
                      ((c==-1)? "EOF" : (char)c));
}
```

As an example, for the input RE expression:

```
(ab | c* | bac*) (d a)* x
```

The lexer will produce the following sequence of tokens, provided that `nextToken()` is repeatedly called until `\n` is reached:

```
(01) [4]    (                (15) [3]    *
(02) [1]    a                (16) [5]    )
(03) [1]    b                (18) [4]    (
(05) [2]    |                (19) [1]    d
(07) [1]    c                (21) [1]    a
(08) [3]    *                (22) [5]    )
(10) [2]    |                (23) [3]    *
(12) [1]    b                (25) [1]    x
(13) [1]    a                (27) [0]    \n
(14) [1]    c
```

In addition to returning tokens, the `nextToken()` routine is also responsible for detecting *lexical errors* and for skipping whitespace characters. For the simple RE language, there is only one type of lexical error possible: `illegal characters`. In next week's lectures and lab, we'll see more types of lexical errors. Whitespace characters serve as delimiters between tokens. They themselves do not correspond to any tokens, and should be skipped.

Exercise

1. Write an RE expression with an illegal character embedded in it. Save the expression in a file and run the compiler with it. Does the compiler catch the error?
2. Write an RE expression with plenty whitespace characters embedded in it. Save the expression in a file and run the compiler with it. Does the compiler produce the correct sequence of tokens?

02. Parser Implementation

The file `Parser.java` contains a bare-bone parser for the RE language. In this version, the parser does not generate any output. It simply verifies that the input expression is correct according to the language's grammar. If not, it reports a *syntax error* and halts.

Just for the information: this parser is implemented using the *recursive-descent* approach. (The other common approach is called *shift-reduce*.) A characteristic of this approach is that the program structure closely correspond to the grammar rules. Here are some code segments from `Parser.java`:

"from 02/Parser.java"

```
// The main parser routine
//
static void parse(FileReader input) throws Exception {
  Lexer.init(input);              // pass input handle to lexer
  nextToken = Lexer.nextToken();  // get first token ready
  parseExpr();
}

// Expr -> Union '\n'
//
static void parseExpr() throws Exception {
  parseUnion();
  match(Lexer.END);
  verboseMsg("End", nextToken.pos);
}

// Union -> Concat [ '|' Union ]
//
static void parseUnion() throws Exception {
  parseConcat();
  if (nextToken.code == Lexer.UNION) {
    match(Lexer.UNION);
    if (verbose) indent++;
    parseUnion();
```

```
      if (verbose) indent--;
      verboseMsg("Union", nextToken.pos);
    }
  }
```

Exercise

1. Run this parser with a test input, say `test1.in`. You should see just one line at the end: `"Parsing successful"`.
2. Now run the parser again with the "verbose" switch on:

   ```
   linux> java Compiler -v test1.in
   ```

   Now you can see the actions of the parser.
3. Write two RE expressions, each with a distinct syntax error. Save the expressions in two separate files and run the compiler with them. Does the compiler catch both errors?

03. AST Generation

This version of the RE parser adds the construction of an internal representation for the input expression, the *AST (abstract syntax tree)*. For a syntax-correct input, the parser generates a corresponding AST.

The AST node types correspond to the syntax constructs of the input language. In this parser, the AST nodes are defined as a collection of Java classes (in `Ast.java`), one class per AST node type. Note that (1) AST is defined by compiler writers; hence it is not unique to the input language. (2) AST is an abstract representation; hence it does not necessarily keep all the details of the input expression. For instance, parentheses are not represented in the AST. (3) AST is a compiler internal representation. To view it, we have to dump it to an external form.

As an example, here is the definition for one AST node type and its two dumping routines:

"from 03/Ast.java"

```
public static class Union extends Expr {
  private final Expr rand1, rand2;

  public Union(Expr rand1, Expr rand2) {
    this.rand1 = rand1;
    this.rand2 = rand2;
  }

  public String toDot() {
    return hashCode() + " [fillcolor=lightcyan,label=\"|\"];\n"
      + hashCode() + " -> " + rand1.hashCode() + ";\n"
      + hashCode() + " -> " + rand2.hashCode() + ";\n"
      + rand1.toDot() + rand2.toDot();
  }

  public String toString() {
    return "(" + rand1 + "|" + rand2 + ")";
  }
}
```

In the new version of `Parser.java`, each parsing routine now constructs and returns an AST object. Here are a couple of sample routines:

"from 03/Parser.java"

```
// Expr -> Union '\n'
//
private static Ast.Expr parseExpr() throws Exception {
```

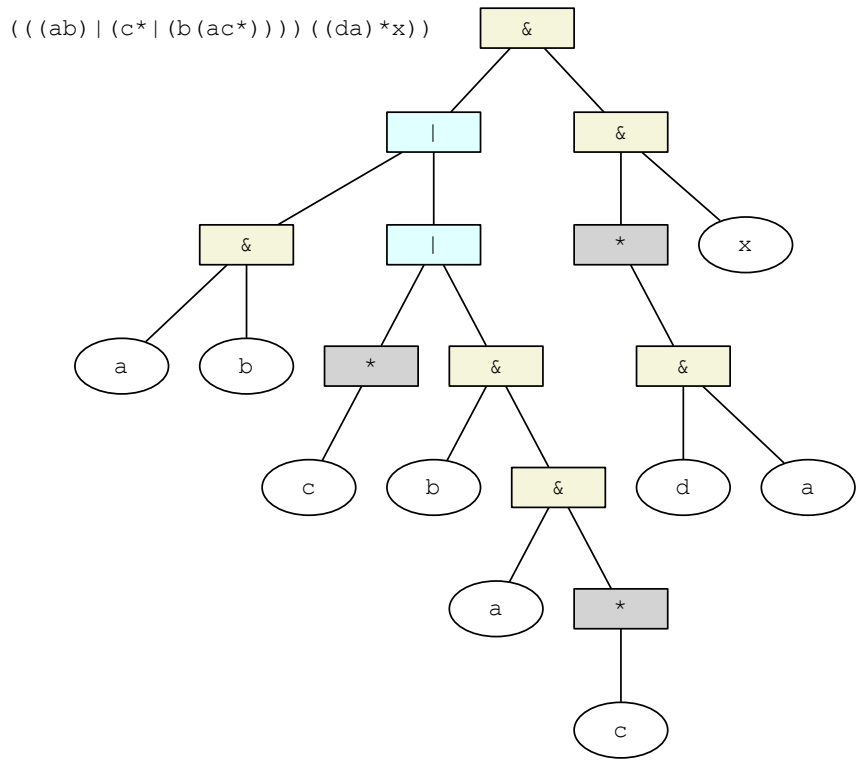
```
    Ast.Expr expr = parseUnion();
    match(Lexer.END);
    return expr;
  }

  // Union -> Concat [ '|' Union ]
  //
  private static Ast.Expr parseUnion() throws Exception {
    Ast.Expr rand1 = parseConcat();
    if (nextToken.code == Lexer.UNION) {
      match(Lexer.UNION);
      if (verbose) indent++;
      Ast.Expr rand2 = parseUnion();
      if (verbose) indent--;
      Ast.Expr union = new Ast.Union(rand1, rand2);
      verboseMsg("Union", union, nextToken.pos);
      return union;
    }
    return rand1;
  }
```

Of the two AST dumping forms, the `.dot` form is a graphic form. When you run this version of compiler with an input, a `.dot` file is automatically generated. To view the graphic AST form, you need to further process the `.dot` file and convert it to a `.pdf` file:

```
linux> dot -Tpdf test1.in.dot -o test1.pdf
```

Here is the result:



Exercise

1. Parentheses in an RE expression serve an important purpose. They help to provide precedence order of operations on subexpressions. However, they are not explicitly represented in an AST. Can you use `test1`'s AST to explain why the information provided by the parentheses in the RE expression is not lost in the AST?
2. Think of an RE expression that would correspond to a very imbalanced AST. Write it in a file, and verify it with the compiler.

04. Static Analysis

After parsing, the next step for a compiler is to perform *static analysis*. This step is to use the input language's semantic constraints to further validate the input program or to compute semantic result from the program.

As an illustration, we add one semantic constraint to the RE language: Distinguishing vowels and consonants in the RE's alphabet, and requiring that an RE expression to have at least one vowel.

Under this constraint, `a*b|cd*e`, would be semantically valid, while `a*b|cd*` would not be, even though both are syntactically valid RE expressions.

Our second example of static analysis is to *compute* a total value for an input RE expression: Each letter occurrence in the expression contributes a value of one. (Two occurrences of the same letter contribute a value of two.) The value of an expression is the total sum of the letter values. In other words, this value represents the total occurrence of letters in an RE expression.

Static analysis is typically implemented as a *distributed computation* over the AST: Every node participates and the final result is collected at the AST's root node. We follow this approach. Here is a modified AST node class, with two computation routines added: `vowelCheck()` and `letterCount()`:

"from 04/Ast.java"

```
public static class Union extends Expr {
  private final Expr rand1, rand2;

  public Union(Expr rand1, Expr rand2) {
    this.rand1 = rand1;
    this.rand2 = rand2;
  }

  public boolean vowelCheck() {
    return rand1.vowelCheck() || rand2.vowelCheck();
  }

  public int letterCount() {
    return rand1.letterCount() + rand2.letterCount();
  }
```

As shown here, the computation for the `Alter` node depends on the results of the same computation at its two children nodes, `rand1` and `rand2`.

Exercise

1. Using `test1`'s AST as a reference, decide whether the computation for `vowelCheck()` and `letterCount()` should be carried out top-down or bottom-up on the AST?
2. Try to annotate each node in `test1`'s AST with the results of `vowelCheck()` and `letterCount()`. You don't need to trace the program, just use your intuition.