

CSE105 (spring 2008): Homework 3

Instructor: Daniele Micciancio

Due on Tuesday, May 20, 2008, noon.

Solutions to the homework should be **typed up**, printed out, and turned in at the beginning of class. A homework submission **page** will appear shortly. The files must be named *exactly* as shown in the problem description.

1 Context-Free Grammars

- a. Give a context-free grammar that generates the language

$$L_1 = \{a^i b^j c^k \mid i = j \text{ or } j = k \text{ where } i, j, k \geq 0\}.$$

For example, the strings *aabb*, *abc*, *aaa* are in L_1 , while *bcc*, *abbc*, *cab* are not in L_1 . You can use JFLAP to experiment with your context-free grammar, but you are not required to submit any JFLAP file. Type up your final solution in the written portion of the assignment.

- b. Give a context-free grammar that generates the language

$$L_2 = \{0^{x_1} \# 0^{x_2} \# \dots \# 0^{x_k} \mid k \geq 0, \text{ each } x_i \text{ is an integer } \geq 0, \text{ and } x_i = x_j \text{ for some } i \neq j\}.$$

In other words, the language L_2 describes all strings of 0's separated by #, where there's at least one pair of consecutive 0's having equal length. For example *00#00*, *0#00#000#00*, and *000#00#0#000* are in L_2 , while *0#00#000*, *00*, and *#00* are not in L_2 . Type up your grammar and turn it in the written portion of the assignment. As for part a., you can use JFLAP to experiment with the grammar, but not JFLAP file is required as part of your solutions.

2 Push-Down Automata

Give a push-down automaton that recognizes the language

$$L_3 = \{a^i b^j c^k \mid i \neq j \text{ or } j \neq k \text{ where } i, j, k \geq 0\}.$$

For example, the strings *a*, *bc*, *abbc* belong to L_3 , but ϵ , *abc*, *aabbcc* do not belong to L_3 . Build the PDA in JFLAP, and name the file *hw3-2.jff*. Do not automatically convert a CFG into an automaton; please directly design the PDA yourself.

3 Pretty-Printing Lisp

Purpose

The goal of this programming assignment is to understand how regular expressions and context-free grammars can be used to translate a program written in one language into a program written in another. You will

produce a program that on input some **Lisp** code outputs the same code formatted to be more easily readable. We have provided code to get you started. Your task is to modify that code.

Doing this is a two-step process. First, the lexer uses regular expressions to match chunks of text and turn them into tokens. These tokens have some associated type and optionally a value. For example, the integer -45 in the input stream would become a token of type **INTEGER** with a value of -45 . Furthermore, most lexers have the ability to execute some programmer defined code upon recognition of a token. For this assignment, we are going to use the lexer generator **JFlex**.

The second step is the parsing step. The parser reads a stream of tokens and checks them against a context-free grammar. The parser generator we are going to use is the Constructor of Useful Parsers, **CUP**. CUP requires that the programmer write the grammar in **Backus-Naur form**. You will note that this is slightly different than the production rules seen in class or in the book. For most cases the only real change is replacing \rightarrow with $::=$. Like lexer generators, parser generators will also allow the programmer to define some chunk of code to execute when a valid grammatical construct is found.

Good Sources of More Information

You are highly encouraged to read the following useful, but slightly old, links:

1. [JFlex and CUP](#)
2. [Creating a Calculator Using JFlex and CUP](#)
3. [JFlex User's Manual](#)
4. [CUP User's Manual](#)

The first two links are essential in getting a basic familiarity with these tools, so please read before starting the assignment!

Starting Point

Download [hw3.zip](#). This starting point contains 7 java files, 3 jar files, 2 bash scripts, 1 JFlex file, and 1 CUP file as well as a directory of tests.

The 3 jars are required for generating the lexer and parser as well as running programs which use the parser. The `build.sh` script simplifies the task of generating `Lexer.java`, `Parser.java`, and `sym.java` as well as compiling all of the java files.

The `runtests.sh` script both provides an example of how to actually run the resultant application while dealing with the annoying classpath issues as well as running each of the files `tests/test*.lisp` through the application and comparing the result with the expected results in the corresponding `tests/test*.expected` file. If the `.expected` file does not exist, then the Lisp code is invalid and should not be accepted by the parser.

Assignment Details

You will be required to make a number of modifications to the starting point detailed below. You should not create any files beyond the ones listed here although you are allowed to modify the Java files and the JFlex and CUP files as much or as little as you wish. The only thing that you must keep is the ability to run your program on a file `foo` by running the following:

```
java -cp ./java-cup-11a-runtime.jar Main foo
```

1. Modify `Lexer.jflex` to support both signed integers and floating point numbers. To do this, you will need to modify the macros for `{Integer}` and `{FloatingPoint}`, and write the appropriate lexical rules. Notice that the tokens for these are already defined in `Parser.cup`.

Integers are specified with an optional + or - sign followed by a nonempty sequence of digits. Floating point numbers are recognized by an optional + or - sign followed by an integer, a period, an integer, and optionally an e followed by a signed integer. For example, +5.3e-8, -274.0e75, and 1.0 are all floating point numbers. Note that while +5 is a valid Lisp number, `new Integer("+5")` will throw an exception, so make sure you handle that. Also, watch out so that you express a dot (.) properly when designing the regular expression. When printing numbers, they should appear exactly as returned by `toString()` of the `Integer` and `Float` Java classes. [Printing is already handled for you in the `Atom` class.]

2. Modify the macro for matching identifiers in `{Identifier}`. An identifier consists of any sequence of letters, numbers, +, -, *, /, <, =, >, &, ?, or period (.) with the condition that it must contain at least one character that isn't a number and isn't a period. A printed identifier should appear exactly as it does in the input. [Printing is already handled for you in the `Atom` class.]
3. Add support for comments in the macro `{Comment}`. Comments start with a semicolon and extend to the end of the line. Use `{LineTerminator}` to refer to the end of the line. Comments should be added to the rule for `WhiteSpace` so that they are ignored along with the rest of the white space. This means that comments will *not* appear in our output. For example,

```
-5.3      ; This is a floating point number
```

should produce only a `FLOAT` token with the value `-5.3` and should be printed as follows.

```
-5.3
```

4. We will come back to the lexer in a while. For now, let's start editing `Parser.cup`. As you may have noticed, currently the grammar accepts only a single expression. We are going to expand the grammar so that it accepts more complicated expressions. Change the `exprs` nonterminal so that it allows an arbitrary number of expressions (`expr`)—including 0—rather than the single `expr` it allows currently. So the input

```
+1.0 abc -86 "foobar"
```

should produce 4 tokens and be printed as follows:

```
1.0
abc
-86
"foobar"
```

5. The only type of expression (`expr`) that is currently defined is an atom. Atoms are numbers, strings, and identifiers. The grammar for atoms, along with the `Atom` java class, have already been completed, so there is no need to change these components.

We would now like to expand the grammar to handle an additional sort of expression: an “S-expression.” There are two types of S-expressions: functions and special forms. Functions are easier, so we will start with them and get to special forms later.

There are two forms of functions which will be printed slightly differently so it makes sense to separate them into two rules. You will need to add new nonterminals to the grammar at this point. The simple form of a function is

$$(\langle identifier \rangle \langle arguments \rangle)$$

where $\langle identifier \rangle$ is a valid identifier and $\langle arguments \rangle$ is a (possibly empty) sequence of expressions. For example, the following are all examples of functions:

```
(foo)
(+ 3 4 5)
(bar (foo) (< -5 (foo)))
```

When printing these, the identifier should come right after the opening parenthesis. If there are any arguments, they should be aligned with one per line with the first being one space after the identifier. The closing parenthesis should come immediately after the last argument. So, the example above should be printed as follows:

```
(foo)
(+ 3
  4
  5)
(bar (foo)
     (< -5
      (foo)))
```

To implement the indentation, the `Expr` java interface has a method `printWithIndentation()` which passes the number of spaces subsequent lines in the expression should be indented. Use this variable wisely to control the indentation whenever we move to the next line.

The second form functions can take is

$$(\langle Sexpression \rangle \langle arguments \rangle)$$

where $\langle Sexpression \rangle$ is any valid S-expression. In this case, the argument expressions should be aligned with the S-expressions. For example

```
((which-op + - * /) (foo) 3)
```

should be printed as follows:

```
((which-op +
  -
  *
  /)
 (foo)
 3)
```

6. Now, let's expand the grammar to handle special forms (a type of S-expression). We will implement `if`, `quote`, `quote`'s shorter form (`'`), and `defun`.

You need to go back and modify `Lexer.jflex` to add support for the tokens `if`, `quote`, `'`, and `defun`. Since these tokens do not have values associated with them, we can add them in a manner analogous to the left and right parentheses. Make sure to give different tokens to `quote` and `'`. Also you will need to add the corresponding symbols as terminals to `Parser.cup`.

You should now add rules for two types of `if` which have the special form

$$(\text{if } \langle test \rangle \langle thenClause \rangle [\langle elseClause \rangle])$$

where $\langle test \rangle$ and $\langle thenClause \rangle$ are expressions and $[\langle elseClause \rangle]$ is an optional expression. The $\langle test \rangle$ should be printed with a single space before it on the same line as the `(if`. The other two clauses should be indented 2 spaces farther than the `(if`. For example,

```
(if (foo a b) (set c 5))
(if (foo a b) (+ a b) (- a b))
```

should be printed as follows:

```
(if (foo a
    b)
    (set c
        5))
(if (foo a
    b)
    (+ a
      b)
    (- a
      b))
```

The next special form is `quote`. There are two ways to input it, but both should print the same way. The syntax is

```
(quote (<exprs>))
'(<exprs>)
```

where in both cases, $\langle exprs \rangle$ is a possibly empty sequence of expressions. Each expression should be on its own line (except for the first which is on the line with the opening parenthesis and the last which is on the line with the closing parenthesis) and be aligned. For example,

```
(quote (a b c))
'(a b c)
```

should be printed as follows:

```
'(a
  b
  c)
'(a
  b
  c)
```

The last special form is `defun` which is used to define new functions. The syntax is

```
(defun <identifier> (<arguments>) <body>)
```

where $\langle identifier \rangle$ is a valid identifier, $\langle arguments \rangle$ is a possibly empty sequence of valid identifiers, and $\langle body \rangle$ is a possibly empty sequence of expressions. The $\langle identifier \rangle$ and $\langle arguments \rangle$ should be printed on the same line as the `(defun` separated by a space. The $\langle body \rangle$ expressions should be printed each on their own line (except for the last which is printed on the same line as the closing parenthesis) and indented 2 spaces farther than the `(defun`. For example,

```
(defun min (x y) "Min function" (if (< x y) x y))
```

should be printed as follows:

```
(defun min (x y)
  "Min function"
  (if (< x
      y)
      x
      y))
```

What to Turnin

You need to turn `Lexer.jflex`, `Parser.cup`, and all of the `.java` files except for `Lexer.java`, `sym.java`, and `Parser.java` which are generated automatically. The homework submission page is at <https://www-cse.ucsd.edu/~scheckow/cse105/submit.php>.