COMP207P Compilers Lexing and Parsing Coursework

Submission Deadline

Friday 4th March 2016 @ 11:55PM

The goal of this 207P Compiler coursework is to build a lexer and parser for the $\tilde{Z}$ programming language. Use JFlex 1.6.1 and Cup version 11b-20151001, using *only* the specified versions, to automatically generate code for your scanner and parser[1]. This coursework broadly requires writing regular expressions covering all legal words in the language (`Lexer.lex`), and a context free grammar describing its rules (`Parser.cup`).

You can work on this coursework individually or in groups of up to three. In either case, you will get a single mark, comprising 10% of your mark for the COMP207P module. Please submit your work (`JFLex/CUP` specifications) before Friday 4th March 2016 @ 11:55PM.

Detailed submission instructions are given at the end of the document.

> "There will always be noise on the line."
>
> —Engineering folklore

Despite my best efforts, this project description contains errors[2]. Part of this coursework is then to start to develop the skills you will need to cope with such problems now. First, you think hard about issues you discover (of which only a small subset will be errors in this problem statement) and make a reasonable decision and *document* your decision and its justification in a README accompanying your submission. Second, you can ask stakeholders for clarification.

1 The $\tilde{Z}$ Language

You are to build a lexer and a parser for $\tilde{Z}$.

A program in $\tilde{Z}$ consists of a list declarations, one of which is a **`main`** function. This declaration list defines global variables and new data types as well as functions, but cannot be empty: it must have a **`main`**.

$\tilde{Z}$ has two types of **comments**. First, any text, other than a newline, following `#` to the end of the line is a comment. Second, any text, including newlines, enclosed within `/# ... #/` is a comment, and may span multiple lines.

An **identifier** starts with a letter, followed by an arbitrary number of underscores, letters, or digits. Identifiers are case-sensitive. Punctuation other than underscore is not allowed.

A **character** is a single letter, punctuation symbol, or digit wrapped in `''` and has type **`char`**. The allowed punctuation symbols are space (See `http://en.wikipedia.org/wiki/Punctuation`) and the ASCII symbols, other than digits, on this page `http://www.kerryr.net/pioneers/ascii3.htm`.

The **boolean constants** are `T` and `F` and have type **`bool`**.

Numbers are integers (type **`int`**), rationals (type **`rat`**), or floats (type **`float`**). Negative numbers are represented by the '`-`' symbol before the digits. Examples of integers include `1` and `-1234`; examples of rationals include `1/3` and `-345_11/3`; examples of floats are `-0.1` and `3.14`.

A **Dictionary** (type **`dict`**) is a collection of `(key, value)` pairs, with the constraint that a key appears at most once in the collection. When declaring a dictionary, one must specify the type of the keys and values. For example, **`dict<int,char>`** `d = {key1:val1, key2:val2,...}`; here, the keys must be integers and the values, characters. Use the special type keyword **`top`** to define a dictionary that allows any type for a key or value: **`dict<int,top>`** `d = {1:1, 2:'c', 7:3/5, {1:T}}`. An empty dictionary is `{}`. The

[1] Section 3 explains why I have imposed these constraints on the permitted versions of these tools.

[2] Nonetheless, this document is already a clearer specification than *any* you will find in your subsequent careers in industrial IT.

Primitive Data Types	**bool**, **int**, **rat**, **float**, **char**
Aggregate Data Types	**dict**, **seq**

Table 1: $\tilde{Z}$ data types.

Operator	Defined Over	Syntax
Boolean	**bool**	`!`, &&, `||`
Numeric	**int**, **rat**, **float**	`+ - * / ^`
Dictionary	**dict**	**in**, `d[k]`, `len(d)`
Sequence	**seq**	**in**, `::`, `len(s)`, `s[i]`, `s[i:j]`, `s[i:]`, `s[:i]`
Comparison	Numeric	`< <=`
	Boolean, Numeric	`== !=`

Table 2: $\tilde{Z}$ operators.

assignment `d[k] = v` binds `k` to `v` in `d`. If `d` already contains `k`, `k` is rebound to `v`; if not, the pair `(k, v)` is added to `d` and accessed by `d[k]`.

Sequences (type **seq**) are ordered containers of elements. Sequences have nonnegative length. A sequence has a type: its declaration specifies the type of elements it contains. For instance, **seq<int>** `l = [1,2,3]`, and **seq<char>** `str = [ 'f', 'r', 'e', 'd', 'd', 'y']`. As with **dict** above, you can use the **top** keyword to specify a sequence that contains any type, writing **seq<top>** `s = [ 1, 1/2, 3.14, [ 'f', 'o', 'u', 'r'] ];`. The zero length list is `[]`.

$\tilde{Z}$ sequences support the standard **indexing** syntax. For any sequence `s`, the operator `len(s)` : **seq** $\rightarrow \mathbb{N}$ returns the length of `s` and the indices of `s` range from 0 to `len(s)-1`. The expression `s[index]` returns the element in `s` at `index`. String literals are syntactic sugar for character sequences, so `"abc"` is `[ 'a', 'b', 'c']`. For the sequence **seq<char>** `s = "hello world"`, `s[len(s)-1]` returns 'd' and `s[len(s)-1]` returns 'd' and `len(s)` returns '11'.

Sequences in $\tilde{Z}$ also support **sequence slicing** as in languages like Python or Ruby: `id[i:j]` returns another sequence, which is a subsequence of `id` starting at `id[i]` and ending at `id[j]`. Given `a == [1,2,3,4,5]`, `a[1:3]` is `[2,3,4]`. When the start index is not given, it implies that the subsequence starts from index 0 of the original sequence (*e.g.*, `a[:2]` is `[1,2]`). Similarly, when the end index is not given, the subsequence ends with the last element of the original sequence (*e.g.*, `a[3:]` is `[4,5]`). Finally, indices can be negative, in which case its value is determined by counting backwards from the end of the original sequence: `a[2:-1]` is equivalent to `a[2:len(a)-1]` and, therefore, is `[3,4,5]`, while `s[-2]` is `4`. The lower index in a slice must be positive and smaller than the upper index, after the upper index has been subtracted from the sequences length if it was negative.

Table 1 defines $\tilde{Z}$'s builtin data types. In Table 2, "`!`" denotes logical not, "&&" logical and and "`||`" logical or, as is typical in the C language family. Note that "`==`" is referential equality and "`=`" is the assignment operator in $\tilde{Z}$. The **in** operator checks whether an element (key) is present in a sequence (dictionary), as in `2 in [1,2,3]` or `2 in {1:"one", 2:"two"}`, and returns a boolean. Note that **in** only operates on the outermost sequence: `3 in [[1],[2],[3]]` is `F`, or false. "`::`" operator denotes concatenation, "`s[i]`" returns the $\texttt{i}^{th}$ entry in `s` and "`len(s)`" returns the length of `s` as defined in the discussion of sequences and their indexing above.

1.1 Declarations

The syntax of field or variable declaration is "`type id`". A data type declaration is

```
tdef type_id { declaration_list } ;
```

where `declaration_list` is a comma-separated list of field/variable declarations. Once declared, a data type can be used as a type in subsequent declarations. For readability, $\tilde{Z}$ supports type aliasing: the directive

```
fdef returnType name (formal_parameter_list) { body } ;
```

Listing 2: $\tilde{Z}$ function declaration syntax.

`p.age + 10`	Assumes "`person p;`" previously declared
`b - foo(sum(10, c), bar()) == 30`	Illustrates method calls
`s1 :: s2 :: [1,2]`	Assumes `s1` and `s2` have type **`seq<int>`**

Table 3: $\tilde{Z}$ expression examples.

"**`alias`** `old_name new_name ;`" can appear in a declaration list and allows the use of `new_name` in place of `old_name`.

```
alias seq<char> string;
tdef person { string name, string surname, int age };
tdef family { person mother, person father, seq<person> children };
```

Listing 1: $\tilde{Z}$ data type declaration examples.

Function In Listing 2, the function's `name` is an identifier. Each formal parameter follows the variable/field declaration syntax, `type id`. The formal parameter list is comma-separated list of parameter declarations. A function's body consists of local variable declarations, if any, followed by statements. The return type of the function is `returnType`, or **`void`** when the function does not return a value. Note that **`void`** is not a type, so declaring a variable as **`void`**, such as "**`void`** `id`" is not permitted in $\tilde{Z}$.

1.2 Expressions

$\tilde{Z}$ expressions are applications of the operators defined above. Parentheses enforce precedence. For user-defined data type definitions, field references are expressions and have the form `id.field`. Function calls can be either a statement or an expression; their parameters are expressions that, in the semantic phase (*i.e.* not this coursework), would be required to produce a type that can unify with the type of their parameter. Table 3 contains example expressions.

1.3 Statements

In Table 4, `var` indicates a variable. An `expression_list` is a comma-separated list of expressions. As above, a body consists of local variable declarations (if any), followed by statements. Statements, apart from **`if-else`**, **`while`**, and **`forall`**, terminate with a semicolon. The return statement appears in a function body, where it is optional. In any **`if`** statement, there can be zero or more **`elif`** branches, and either zero or one **`else`** branch.

Variables may be initialised at the time of declaration: "`type id = init ;`". For newly defined data types, initialisation consists of a sequence of comma-separated values, each of which is assigned to the data type fields in the order of declaration. Listing 3 contains examples.

```
dict<int, char> a = { 1:'1', 2:'2', 3:'3' } ;
int b = 10;
string c = "hello world!";
person d = "Shin", "Yoo", 30;
char e = 'a';
seq<rat> f = [ 1/2, 3, 4_2/17, -7 ];
```

Listing 3: $\tilde{Z}$ variable declaration and initialization examples.

Assignment	`var = expression ;`
Input	**`read`** `var ;`
Output	**`print`** `expression ;`
Function Call	`functionId ( expression_list ) ;`
Control Flow	**`if`** `( expression )` **`then`** `body` **`fi`** **`if`** `( expression )` **`then`** `body` **`else`** `body` **`fi`** **`if`** `( expression )` **`then`** `body` **`elif`** `( expression )` **`then`** `body` **`else`** `body` **`fi`** **`while`** `( expression )` **`do`** `body` **`od`** **`forall`** `( item` **`in`** `iterable )` **`do`** `body` **`od`** **`return`** `expression ;`

Table 4: $\tilde{Z}$ statements.

The statement **`read`** `var;` reads a value from the standard input and stores it in `var`; the statement **`print`** prints evaluation of its expression parameter, followed by a newline. The **`if`** and **`while`** statements behave like those in the C family language. The iterable in **`forall`** is either a sequence or a dictionary; `item` is bound to each element in order for a sequence and in an arbitrary order for a dictionary. Listing 4 shows how to use **`forall`**.

```
seq<int> a = [1, 2, 3];
forall(n in a) do
  print n * 2;
od
```

Listing 4: $\tilde{Z}$ iterables example.

```
main {     # Main is not necessarily last.
  seq<int> a = [1,2,3];
  seq<int> b = reverse(a); # This is a declaration.
  print b;    # This is the required statement.
};

fdef seq<top> reverse (seq<top> inseq) {
  seq<top> outseq = [];
  int i = 0;
  while (i < len(l)) do
    outseq = inseq[i] :: outseq;
    i = i + 1;
  od
  return outseq;
};
```

Listing 5: $\tilde{Z}$ example program.

Listing 5 shows an example program, contain two functions. The function **`main`** is the special $\tilde{Z}$ function where execution starts. $\tilde{Z}$'s **`main`** returns no value.

2 Error Handling

Your parser will be tested against a test suite of positive and negative tests. This testing is scripted; so it is important for your output to match what the script expects. Add the following function definition into the "`parser code`" section of your Cup file, between its `{:` and `:}` delimiters.

```
public void syntax_error(Symbol current_token) {
  report_error(
    "Syntax error at line " + (current_token.left+1) + ", column "
    + current_token.right, null
  );
}
```

Listing 6: $\tilde{Z}$ compiler error message format.

3 Submission Requirements and Instructions

Your scanner (lexer) must

- Use `JLex` (or `JFlex`) to automatically generate a scanner for the $\tilde{Z}$ language;
- Make use of macro definitions where necessary. Choose meaningful token type names to make your specification readable and understandable;
- Ignore whitespace and comments; and
- Report the line and the column (offset into the line) where an error, usually unexpected input, first occurred. Use the code in Section 2, which specifies the format that will be matched by the grading script.

Your parser must

- Use `CUP` to automatically produce a parser for the $\tilde{Z}$ language;
- Resolve ambiguities in expressions using the precedence and associativity rules;
- Print “parsing successful”, followed by a newline, if the program is syntactically correct.

Your scanner and parser must work together.

Once the scanner and parser have been successfully produced using `JFlex` and `CUP`, write a `QC.java` class similar to the `Test.java` seen during lecture, to test your code on the test files provided on the course webpage.

I have provided a makefile on Moodle. This makefile *must* build your project, from source, when `make` is issued,

- on Ubuntu 14.04.3 LTS 64bit
- using JFlex 1.6.1
- using Cup version 11b-20151001
- using Java SE 8 Update 66

If your submission fails to build using this Makefile on Ubuntu 14.04.3 LTS 64bit with these versions of the tools, your mark will be zero.

The provided makefile has a test rule. The marking script will call this rule to run your parser against a set of test cases. This set of test cases includes public test cases provided via Moodle and private ones; they include positive tests, on which your parser must emit "parsing successful" followed by a newline and *nothing else*, and negative tests on which your parser must emil the correct line and column of the error, as specified in Section 2 above.

Your mark will be the number of positive tests cases you correctly pass and the number of negative test cases you correctly fail divided by the total number of test cases.

Group work is optional. Each student/group should, via Moodle, submit a tar ball, or zip file, that contains

- The Makefile with which I have provided you
- The `JFlex` specification `Lexer.lex`
- The `CUP` specification `Parser.cup`
- Any other classes you have defined, if any, using the directory layout the Makefile expects
- To save space, it is not necessary zip up and include the JFlex and Cup jars in the lib directory

Only one submission per group is necessary. The deadline for completion of this part of the coursework is Friday 4th March 2016 @ 11:55PM. Any coursework handed in later than 2 working days after the deadline will automatically receive a zero mark.