COMP207P Compilers Lexing and Parsing Coursework

Submission Deadline

Friday 2nd March 2018 @ 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-20160615, 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 must work on this coursework individually and will get a single mark, comprising 10% of your mark for the COMP207P module. Please submit your work (`JFLex/CUP` specifications) before Friday 2nd March 2018 @ 11:55PM.

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

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

Despite our 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 accompanying your submission. Second, you can ask stakeholders for clarification.

1 Interpreting the Specification

Throughout your career in IT, you will have to contend with interpreting and understanding specifications. If you fail a test case because of some ambiguities in the specification, you must go on to explain why it is a problem and justify how you decide to resolve it. When marking, we will consider the issues you discover; if your justification is sound, you will get full marks for the relevant test cases.

We have numbered each paragraph, table and program listing in this specification. For each issue that you find, make a note in the following format:

```
Paragraph: 72
Problem: it is not clear whether we can omit both the start and end indices
         in sequence slicing, like "foo = bar[:]".
Our solution: this is possible in many other languages that support list
              slicing (like Python), so our compiler accepts this syntax.

Paragraph: 99
Problem: the spec has an assignment using the equality operator,
        "foo == bar;".
Our solution: we think this is a mistake, and our compiler does not accept
            this statement.
```

Call this file **`ambiguities.txt`** and turn it along with your implementation of the parser.

[1] Section 4 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	**char**, **bool**, **int**, **rat**, **float**
Aggregate Data Types	**dict**, **seq**, **set**
Other Data Types	**thread**, **function**

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

2 The $\tilde{Z}$ Language

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

§2 A program in $\tilde{Z}$ consists of a list declarations; the last declaration is for a **main** function. This declaration list defines global variables, new data types, and functions, but cannot be empty: it must have a **main** at the end.

§3 $\tilde{Z}$ 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.

§4 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.

§5 Table 1 summarises $\tilde{Z}$'s builtin data types, in which **function** is typically used along with lambda expressions discussed in Section 2.1 and **thread** is a special type for multithreading detailed in Section 2.3.

§6 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`.

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

§8 **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`.

§9 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 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]`. For a dictionary, the natively-defined property `len` returns the number of `(key, value)` pairs.

§10 **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']`. **String literals** are syntactic sugar for character sequences, so `"abc"` is `[ 'a', 'b', 'c']`. 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, "four";`. The zero length list is `[]`.

§11 $\tilde{Z}$ sequences support the standard **indexing** syntax. For any sequence `s`, the natively-defined property `len` returns the length of `s` and the indices of `s` range from 0 to `s.len-1`. The expression `s[index]` returns the element in `s` at `index`. For the sequence **seq<char>** `s := "hello world"`, `s[s.len-1]` returns '`d`' and `s.len` returns '`11`'.

§12 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:a.len-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.

§13 $\tilde{Z}$ also includes a data type for **sets**. A set (type **`set`**) is an unordered collection with no duplicate elements. The elements of a set must have the same type, which has to be specified beforehand. Unlike that for **`seq`**, you cannot use **`top`** when declaring a set. Curly braces `{}` are used to initialise sets. For instance, **`set<seq<char >>`** `names := {"Ana", "Ben"}` creates a set whose elements are names. The empty set is `{}`.

§14 The basic use of sets is membership (**`in`**) testing. For example, with **`set<int>`** `a := {0, 1}` and **`set<int>`** `b := {1, 2}`, the expression `2` **`in`** `a` is `F` and `2` **`in`** `b` is `T`. Sets also support mathematical operations like union (`|`), intersection (`&`), and difference (`\`). Reusing the definitions of sets `a` and `b`, `a & b` evaluates to `{1}` and `a \ b` evaluates to `{0}`.

2.1 Declarations

§15 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 "**`alias`** `old_name new_name ;`" can appear in a declaration list and allows the use of `new_name` in place of `old_name`. Below are some examples of type declarations.

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

§16 A function declaration is

```
fdef return_type id (formal_parameter_list) { body } ;
fdef id (formal_parameter_list) { body } ;
```

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 declarations, if any, followed by statements. The return type of the function is `return_type` and is omitted when the function does not return a value.

§17 $\tilde{Z}$ features **lambda expressions** to support functional programming. A lambda expression consists of a comma-separated list of formal parameters enclosed in two `|`, the arrow token `->`, and a body, which consists of a single expression or a statement block. Unlike **`fdef`**, parameters in lambda expressions must be untyped. In a lambda expression, you must enclose the block in braces `{}`. For example, you may use the **`return`** statement:

```
|p| -> { return p.age >= 18; }
```

Alternatively, if you specify a single expression, then $\tilde{Z}$ evaluates the expression and returns its value:

```
|p| -> p.age >= 18
```

Lambda expressions provide another way to define functions, as the following code listing shows:

```
function cube := |a| -> a * a * a;
int x := 3;
int y := cube(x);
```

Operator	Defined Over	Syntax
Boolean	**bool**	`!`, &&, `||`
Numeric	**int**, **rat**, **float**	`+`, −, `*`, `/`, `^`
Dictionary	**dict**	**in**, `d[k]`
Sequence	**seq**	**in**, `::`, `s[i]`, `s[i:j]`, `s[i:]`, `s[:i]`
Set	**set**	**in**, &, `|`, `\`
Comparison	Numeric	`<`, `<=`, `>`, `>=`
	Boolean, Numeric	`==`, `!=`

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

`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.

2.2 Expressions

§18 Table 2 lists the builtin operators in $\tilde{Z}$. Specifically, "`!`" denotes logical not, "&&" logical and, "`||`" logical or, "`==`" referential equality, and "`!=`" inequality, as is typical in the C language family. The **`in`** operator checks whether an element (key) is present in a sequence or a set (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 or set: `3 in [[1],[2],[3]]` is `F`, or false. "`::`" operator denotes concatenation, "`s[i]`" returns the $(i+1)^{th}$ entry in `s` and "`s.len`" returns the length of `s` as defined in the discussion of sequences and their indexing above.

§19 Most $\tilde{Z}$ 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 are expressions; the actual parameters of function calls are also 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.

2.3 Statements

§20 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. When appended with a semicolon, a function call becomes a statement.

§21 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. The following listing 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 ];
```

§22 The input statement **`read`** `var;` reads a value from the standard input and stores it in `var`. The output statement **`print`** prints the evaluation of its expression parameter; the expression parameter cannot be empty.

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`
	`break N; # N is optional and defaults to 1.`
	`forall ( item in iterable ) do body od`
	`return expression ;`

Multithreading	`thread thread_id := body;`

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

```
seq<int> a := [1, 2, 3];
seq<int> b := [4, 5, 6];
int i := 0;
int j := 0;
while (i <= 2) do
  while (j <= 2) do
    if (b[j] < a[i]) then
      break 2; # break out of two loops
    fi
    j := j + 1;
  od
  i := i + 1;
  j := 0;
od
```

Listing 1: $\tilde{Z}$ loop example.

§23 The **`if`** and **`while`** statements behave like those in the C family languages. In any **`if`** statement, there can be zero or more **`elif`** branches, and either zero or one **`else`** branch. To exit a **`while`** loop, one can use **`break`** `N`, usually coupled with an **`if`** statement; the *optional* argument `N` is a positive integer that specifies the number of nested **`while`** loops to exit and defaults to `1`. The use of **`break`** statement is forbidden outside a **`while`** loop. Listing 1 shows how to use **`break`**. The iterable in **`forall`** is either a sequence, a set, or a dictionary; `item` is bound to each element in order for a sequence and in an arbitrary order for a set or a dictionary. Listing 2 shows how to use **`forall`**.

§24 $\tilde{Z}$ enables **multithreading** through the **`thread`** type. You must associate a block of code with each thread variable at declaration. When control reaches a thread declaration, a new thread is created to start executing the code in the associated block; the original thread returns from this assignment immediately and resumes execution with the next statement. You may call the built-in function `wait` to wait for a particular thread to finish. The calling thread will block until the parameter thread is finished. Listing 3 creates two threads, `t1` and `t2`, each of which will print a sentence before the main thread terminates.

§25 Listing 4 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.

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

Listing 2: $\tilde{Z}$ iterables example.

```
thread t1 := {print "Hello from thread 1!";};
thread t2 := {print "Hello from thread 2!";};
wait(t1);
wait(t2);
print "Multithreaded program ends.";
```

Listing 3: $\tilde{Z}$ multithreading example.

```
fdef seq<int> reverse (seq<int> inseq) {
  seq<int> outseq := [];
  int i := 0;
  while (i < inseq.len) do
    if (inseq[i] == 0) then
      break;
    fi
    outseq := inseq[i] :: outseq;
    i := i + 1;
  od
  return outseq;
} ;

main {                      # Main must be last.
  seq<int> a := [1,2,3];
  seq<int> b := reverse(a); # This is a declaration.
  print b;
} ;
```

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

3 Error Handling

§26 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 5: $\tilde{Z}$ compiler error message format.

§27 The provided `SC` class uses a boolean field `syntaxErrors` of the parser object to decide whether parsing is successful. So please add such a public field to the `Parser` class and set it to **`true`** when a syntax error is generated.

4 Submission Requirements and Instructions

§28 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 3, which specifies the format that will be matched by the grading script.

§29 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.

§30 *Your scanner and parser must work together.*

§31 Once the scanner and parser have been successfully produced using `JFlex` and `CUP`, use the provided `SC` class to test your code on the test files given on the course webpage.

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

- using JFlex 1.6.1
- using Cup version 11b-20160615
- using Java SE 8 Update 162

§33 If your submission fails to build using this Makefile with these versions of the tools, your mark will be zero.

§34 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 emit the correct line and column of the error, as specified in Section 3 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.

§35 Each student must use the automatic testing system (detailed in Section 5) to submit the coursework.

§36 The deadline for completion of this coursework is Friday 2nd March 2018 @ 11:55PM. The maximum mark for coursework handed in up to one working day after the deadline will be 67%, and for up two working days after the deadline will be 33%. Any coursework handed in later than 2 working days after the deadline will automatically receive a zero mark.

5 Automatic Testing Guidelines

Your coursework should be developed using the git version control system. We have created bare git repositories for each of you on Department machines. These repositories run a test suite whenever you push a commit to them, and can email the results of the test run to you. Your coursework will be marked partially based on the results of this test suite.

You can push your work to the repository as many times as you like before the deadline; the results will be emailed to you every time you push. All pushes will be automatically rejected after the deadline, and we will mark the last commit that you pushed. We advise you to plan to push your final commit well before the deadline, and to ensure that you are able to access your repository at least a week before the deadline.

5.1 SSH access to the Computer Science department

Ensure that you can SSH into the department before continuing. Run the following command:

```
ssh YOUR_CS_USERNAME@gitlab.cs.ucl.ac.uk
```

If you do not have access, you are probably using the wrong username: your CS username is different to your UCL username (which you use to log in to Moodle, Portico etc.) *Do not* try to log in more than three times with an incorrect username, as your IP address will be banned by the department. Please visit Tech Support on the 4th floor of the Malet Place Engineering Building to figure out what your CS username is.

5.2 Pushing your work

Once your coursework is building, create a text file called `student.txt` in your repository with a single line containing your student ID, preferred email, and name, in this format:

```
10481459 <grace.hopper@rand.com> Grace Hopper
```

Stage and commit that text file. Next, add your UCL repository as a remote (the command below is a single line):

```
git remote add origin YOUR_CS_USERNAME@gitlab.cs.ucl.ac.uk:/cs/student
   /comp207p/YOUR_CS_USERNAME
```

Then push your repository (`git push origin master`). If all goes well, you should receive an email from "Compilerbot" after a few minutes, which contains your test scores. The git server will reject repositories that do not contain a `student.txt` file or other important files; if your push is rejected, read the error message. The directory structure of your repository should contain at least the following files:

- `student.txt`
- `src/SC.java`

- `src/Parser.cup`
- `src/Lexer.lex`

If you think there is an error with this automatic testing system, please contact Kareem <karkhaz@karkhaz.com>.