COMP0012 Lexing and Parsing Coursework

Submission Deadline

Monday 1st March 2021 @ 16:00 UK time

The goal of this parsing coursework is to build a lexer and parser for the $\tilde{Z}$ programming language. Use JFlex 1.8.2 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 any kind of collaboration, including sharing your own tests with others, is strictly forbidden. You will get a single mark, comprising 8% of your mark for the COMP0012 module. Please submit your work (`JFLex` and `CUP` specifications) before Monday 1st March 2021 @ 16:00 UK time.

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	**bool**, **int**, **rat**, **float**, **char**, **str**
Aggregate Data Types	**seq**

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 $\tilde{Z}$ program is a list of declarations or definitions for global variables, data types, and functions. This list cannot be empty: it must ends with a definition for the special `main` function. The execution of a $\tilde{Z}$ program starts from `main`.

§3 An **identifier** in $\tilde{Z}$ 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.

2.1 Data Types

§4 Table 1 defines $\tilde{Z}$'s builtin data types. A **boolean** value is either `T` (true) or `F` (false) and has type **`bool`**. `T` and `F` cannot be used as identifiers.

§5 **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`.

§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 **Sequences** (type **`seq`**) are ordered containers of elements. Its declaration must specify the type of its elements. For instance, `seq<int> l := [1,2,3]`, and `seq<char> s := ['f', 'o', 'o']`. 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'] ]`. Sequences have nonnegative length. Applying the built-in operator **`len`** to a sequence `s` returns its length. The empty list is `[]`.

§8 **Strings** (type **`str`**) are sequences of characters that $\tilde{Z}$ natively defines. String literals are syntactic sugar for character sequences and are wrapped in `""`, so `"abc"` is `[ 'a', 'b', 'c']`.

§9 $\tilde{Z}$ sequences support the standard **indexing** syntax and the indices of `s` range from `0` to `len(s)-1`. The expression `s[i]` returns the element in `s` at `i`. For the string `s := "hello world"`, `s[len(s)-1]` returns '`d`'.

§10 Sequences in $\tilde{Z}$ also support **slicing**: `s[i:j]` returns another sequence that is a subsequence of `s` starting at `s[i]` and ending at `s[j-1]`. Given `a := [1,2,3,4,5]`, `a[1:3]` is `[2,3]`. When the start index is not given, the slice starts from index `0` of the original sequence (*e.g.*, `a[:2]` is `[1,2]`). Similarly, when the end index is not given, the slice 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.

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

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.

§11 $\tilde{Z}$ supports Python-like list comprehensions. A list comprehension is surrounded by square brackets and uses the builtin **range** function. They have the form `[x * 2 for x of range(10)]` where **for** and **of** are keywords. `range` is a function that generates an integer sequence. It can take up to three arguments, but must be called with at least one. These arguments are either integer literals, or variables, with the first argument being the starting point, the second the terminating point, and the third the increment. This is similar to the `range` function in Python. `range(10)` builds an integer sequence starting from 0; `range(1, 10)` builds from 1; and `range (1, 20, step)` builds a list from 1 to 20 in increments of step, a user-defined variable. An optional boolean valued filter, starting with **if**, can also be used `[x * 3 for x of range(100), if x / 2 = 5]`. The **if** expression must follow a comma.

§12 **Dictionaries** (type **dict**) are collections of `(key, value)` pairs. When declaring a dictionary, one must specify the type of its keys and values. For example, for **dict<int,char>** `d`, the keys are integers and the values, characters. Use **top** to define a dictionary that allows any type of keys or values: **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 operator **len** returns the number of `(key, value)` pairs. The operator `+` concatenates two dictionaries, reporting an error when they share a key.

2.2 Expressions

§13 Applications of the operators listed in Table 2 result in expressions. Table 3 exemplifies $\tilde{Z}$ expressions. Specifically, "`!`" denotes logical not, "&&" logical and, and "`||`" logical or, as is typical in the C language family. In $\tilde{Z}$, "`=`" is referential equality and "`:=`" is the assignment operator. The `?` operator checks whether a key is present in a dictionary, as in `2 ? (1:"one", 2:"two")`, and returns a boolean. The "`+`" operator over sequences denotes concatenation. Field accesses are expressions and have the syntax `id.field`. Parentheses "`()`" enforce precedence.

§14 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.

2.3 Declarations & Definitions

§15 The syntax of variable declaration is "`type id;`". Variables may be initialised at the time of declaration: "`type id := exp;`".

§16 A data type definition is

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

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

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

where `declaration_list` is a comma-separated list of variable declarations. These variables are called *fields*. For newly defined 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.

§17 For readability, $\tilde{Z}$ supports type aliasing: the statement "**alias** `old_name new_name;`" allows using `new_name` in place of `old_name`.

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

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

§18 Listing 2 shows the syntax of function definitions in $\tilde{Z}$. Specifically, a function's `name` is an identifier. The `formal_parameter_list` separates parameter declarations, which follow the variable declaration syntax `type id`, with commas. A function's body *cannot be empty*; it starts with declarations or definitions for local variables, followed by other statements. The return type of a function, `return_type`, can be omitted when the function does not return a value. The `main` function does not return a value; that is, **return** statements in `main` cannot have a value.

2.4 Statements

§19 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** and **while**, terminate with a semicolon. The return statement appears in a function body and is optional. In any **if** statement, there can be zero or one **else** branch.

Assignment	`var_id := expression ;`
Input	`read var_id ;`
Output	`print expression ;`
Function Call	`function_id ( expression_list ) ;`
	`if ( expression ){ body }`
	`if ( expression ){ body } else { body }`
Control Flow	`if ( expression ){ body }`
	`  elif ( expression ){ body }`
	`  else { body }`
	`while { body }`
	`break N; # N is optional and defaults to 1.`
	`return expression ;`
Multithreading	`thread thread_id := { body };`

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

§20 The statement **read** `var_id;` reads a value from the standard input and stores it in `var_id`; the statement **print** `exp;` prints evaluation of `exp`, followed by a newline.

§21 The **if** statement behaves like that in the C family language. In any **if** statement, there can be zero or more **elif** branches, followed by either zero or one **else** branch.

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

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

§22 The unguarded **`while`** statement is the *only* explicit loop construct in $\tilde{Z}$. To exit a loop, one must use **`break`** `N`, usually coupled with an **`if`** statement; the *optional* argument `N` is a positive integer that specifies the number of nested loops to exit and defaults to `1`. The use of **`break`** statement is forbidden outside a loop. Listing 4 shows how to use **`while`** and **`break`**.

```
seq<int> a := [1, 2, 3];
seq<int> b := [4, 5, 6];
seq<int> c := [x * 2 for x of range(10)];
int i := 0;
int j := 0;
while {
  if (2 < i) {
    break;
  }
  while {
    if (2 < j) {
      break; # break to the outer loop
    }
    if (b[j] < a[i]) {
      break 2; # break out of two loops
    }
    j := j + 1;
  }
  i := i + 1;
  j := 0;
}
```

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

§23 $\tilde{Z}$ 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 5 creates two threads, `t1` and `t2`, each of which will print a sentence before the main thread terminates.

§24 Listing 6 shows an example program in $\tilde{Z}$: it defines a function to reverse a sequence.

3 Outputs and Error Handling

§25 Your parser will be tested against a test suite of public test cases provided via Moodle and private ones. This testing is scripted; so it is important for your output to match what the script expects.

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

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

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

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

Listing 6: $\tilde{Z}$ example program.

§26 The test cases 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.

§27 The provided `SC` class uses a boolean field `syntaxErrors` of the parser object to decide whether parsing is successful. So please find such a public field in 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 `JFlex` (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. Section 3 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 1.8.2
- using Cup version 11b-20160615
- using Java SE 8
- on CentOS 7.6

§33 If your submission fails to build using this makefile with these versions of the tools and on the specified operating system, 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 the tests.

§35 We enforce a special testing mechanism: we will run your submission against a negative tests, *only* if it passes the corresponding atomic positive test. The relations are indicated by the names of the tests:

- `p-foo.s` is an atomic test that assesses the language feature `foo`;
- `p-foo-bar.s` is not an atomic test, but it also assesses the feature `foo`;
- `n-foo[-bar].s` depends on `p-foo.s` and assesses the feature `foo`.

§36 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.

§37 Each student must follow the submission instructions detailed in Section 5 to submit the coursework.

§38 The deadline for this coursework is Monday 1st March 2021 @ 16:00 UK time. We strictly adhere to UCL Academic Manual - Chapter 4: Assessment Framework for Taught Programmes and therefore impose the university’s late submission penalties (visit this page for more information). However, COMPILERBOT will reject any submissions 5 working days after the deadline. If you still would like to submit after that, please attach your code in an email to Zheng Gao `<z.gao.12@ucl.ac.uk>`.

§39 To distribute coursework workload more evenly across the academic year, the department has implemented a policy for spacing deadlines. As a result, you are getting more time to complete this coursework than the students from previous years. COMPILERBOT will start accepting submissions from Friday 5th February 2021 @ 16:00 UK time.

5 Automatic Testing Guidelines

Your coursework should be developed using *Git*. We have created bare git repositories for each of you on a department server. Whenever you push a commit to them, a test suite consisting of a set of public (already given to you) and private test cases is run and then the results of the test run will be emailed to you. You can push your work to the repository as many times as you like before the deadline. All pushes will be automatically rejected 5 working days after the deadline. We will mark the *last* commit that you have pushed. We advise you to test access to your repository as soon as possible and to push your final commit well 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@scm4.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 email the Technical Support Group to figure out what your CS username is.

5.2 Pushing Your Work

Having confirmed that you indeed can ssh into a department server, download and decompress `project.tar.gz` we have provided on Moodle. This results in a directory `project` with the following structure:

```
project/
 ├─src/
 ├─lib/
 ├─bin/
 └─tests/
    └─open/
```

Change your working directory to `project` using the `cd` command on Unix-like operating systems. Then initialise `project` into a git repository using `git init`. Add your UCL repository as a remote (the command below is a single line):

```
git remote add origin YOUR_CS_USERNAME@scm4.cs.ucl.ac.uk:/cs/student/
   comp0012/YOUR_CS_USERNAME
```

Develop your lexer (*i.e.* `Lexer.lex`) and parser (*i.e.* `Parser.cup`) in the `src/` subdirectory. Stage and commit all your local changes using `git add` and `git commit`. Once your coursework builds and you are ready to submit, create a text file called **`student.txt`** in the root of your repository with a single line containing your *student number*, *UCL email*, and name, in this format:

```
12345678 <grace.hopper@ucl.ac.uk> Grace Hopper
```

You *MUST* provide the correct student number and UCL email address; otherwise, your submission will be rejected. You *MUST* use the email address containing your name and year of entry (e.g. `f.bloggs.14@ucl.ac.uk`), whereas `UCLID@ucl.ac.uk` (*e.g.* `ucaaxxx@ucl.ac.uk`) is your Windows Live ID, not the actual address.

Next, stage and commit `student.txt`. At this point, your repository should contain at least the following files:

- `student.txt`
- `src/SC.java`
- `src/Parser.cup`
- `src/Lexer.lex`

Push your repository via `git push origin <branch name>`. If all goes well, you should receive an email from COMPILERBOT after a few minutes, which contains your test scores. The server will reject submissions that do not contain a `student.txt` file or other important files; if your push is rejected, read the error message.

If you think there is an error with this automatic testing system, please contact Zheng Gao `<z.gao.12@ucl.ac.uk>`.