COMP0012 Compilers Lexing and Parsing Coursework

Submission Deadline

Monday 1st April 2019 @ 11:55PM

The goal of this COMP0012 parsing coursework is to build a lexer and parser for the $\tilde{Z}_{sec}$ 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 any kind of collaboration, including sharing your own tests with others, is strictly forbidden. You will get a single mark, comprising 10% of your mark for the COMP0012 module. Please submit your work (`JFLex/CUP` specifications) before Monday 1st April 2019 @ 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.

2 The $\tilde{Z}_{sec}$ Language

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

§2 A program in $\tilde{Z}_{sec}$ consists of a list of declarations which defines global variables and new data types as well as functions. This list cannot be empty: it must have a definition for a **`main`** function.

§3 $\tilde{Z}_{sec}$ has two types of **comments**. First, any character, other than a newline, following `#` to the end of the line is a comment. Second, any character, 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 $\tilde{Z}_{sec}$ is a **security typed** language[3]. Every primitive type must be labelled as either *low* or *high*[4]. A low label is declared with `L` and a high with `H`. Information is allowed to *flow* from `L` to `H`, but not *vice versa*. Security labels allow the compiler to check whether *confidential* information has been leaked publicly. Checking that the information flow is correct is *not* part of the coursework. We let ***S*** range over `H` and `L` in the remainder of this specification.

§6 A **character** is a single letter, punctuation symbol, or digit wrapped in `''` and has type **`char`** ***S***. 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`** ***S***.

§8 **Numbers** are integers (type **`int`** ***S***), rationals (type **`rat`** ***S***), or floats (type **`float`** ***S***). 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 **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, `l : seq<int` ***S*** `> := [1,2,3]`, and `str : seq<char` ***S*** `> := [ 'f', 'r', 'e', 'd', 'd', 'y']`. You can use the **`top`** ***S*** keyword to specify a sequence that contains any type, writing `s : seq<top` ***S*** `> := [ 1, 1/2, 3.14, [ 'f', 'o', 'u', 'r'] ]`. The zero length list is `[]`.

§10 $\tilde{Z}_{sec}$ 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']`. String literals are treated as *low* by default: they do not require a security annotation ***S***. For the sequence `s : seq<char L> := "hello world"`, `s[len(s)-1]` returns '`d`' and `s[len(s)-1]` returns '`d`' and `len(s)` returns `11`.

§11 Sequences in $\tilde{Z}_{sec}$ 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.

[3] See `https://goo.gl/XN8ebs` and `https://goo.gl/ZawHTW`.

[4] Language-Based Information-Flow Security, Sabelfeld and Myers, 2003.

Primitive Data Types	**bool** *S*, **int** *S*, **rat** *S*, **float** *S*, **char** *S*
Aggregate Data Types	**seq**
Literal	`string`

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

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

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

§12 Table 1 defines $\tilde{Z}_{sec}$'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 $\tilde{Z}_{sec}$. The **in** operator checks whether an element (key) is present in a sequence, as in `2 in [1,2,3]` 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 i^{th} entry in `s` and "`len(s)`" returns the length of `s` as defined in the discussion of sequences and their indexing above.

2.1 Declarations

§13 The syntax of field or variable declaration is "`id : type`". 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. The security label of a new data type is the *least upper bound* of the labels of its constituent fields. It is not declared by the programmer but calculated during the compiler's semantic analysis (*i.e.* not in this coursework).

§14 For readability, $\tilde{Z}_{sec}$ 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`.

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

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

§15 Listing 2 shows the syntax of function declarations in $\tilde{Z}_{sec}$. Specifically, a function's `name` is an identifier. The `formal_parameter_list` separates parameter declarations, which follow the variable/field declaration syntax `id : type`, with commas. A function's body *cannot be empty*; it consists of local variable declarations, if any, followed by statements. The return type of a function is `returnType`; it is omitted when the function is **main**, or does not return a value.

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

Listing 2: $\tilde{Z}_{sec}$ 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`** `S>`

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

2.2 Expressions

§16 $\tilde{Z}_{sec}$ 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.

2.3 Statements

§17 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 one **`else`** branch.

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`**
	`loop` `body` **`pool`**
	`break` `N; # N is optional and defaults to 1.`
	`return` `expression ;`

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

§18 Variables may be initialised at the time of declaration: "`id : type := 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.

§19 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.

§20 The **`if`** statement behaves like that in the C family language. The unguarded **`loop`** statement is the *only* loop construct in $\tilde{Z}_{sec}$. 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 **`loop`** and **`break`**.

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

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

```
a : seq<int L> := [1, 2, 3];
b : seq<int L> := [4, 5, 6];
i : int L := 0;
j : int L := 0;
loop
 if (2 < i) then
  break;
 fi
 loop
  if (2 < j) then
   break; # break to the outer loop
  fi
  if (b[j] < a[i]) then
   break 2; # break out of two loops
  fi
  j := j + 1;
 pool
 i := i + 1;
 j := 0;
pool
```

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

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

3 Error Handling

§22 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: $\tilde{Z}_{sec}$ compiler error message format.

§23 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

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

fdef seq<top> reverse (inseq : seq<top L>) {
      outseq : seq<top L> := [];
      i : int L := 0;
      loop
         if (l.len < i) then
            break;
         fi
         outseq := inseq[i] :: outseq;
         i := i + 1;
      pool
  return outseq;
} : seq<top L>;
```

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

generated.

4 Submission Requirements and Instructions

§24 Your scanner (lexer) must

- Use `JLex` (or `JFlex`) to automatically generate a scanner for the $\tilde{Z}_{sec}$ 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.

§25 Your parser must

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

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

§27 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.

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

- using JFlex 1.6.1

Figure 1: UCL's late submission penalties.

Modules at levels 4, 5 and 6	Component Mark/ Grade	
	40.00-100.00% / A-D	1.00-39.99% / E
Up to 2 working days late	Deduction of 10 percentage points, but no lower than 40.00% / Grade D	No Penalty
2-5 working days late	Mark capped at 40.00% / Grade D	No Penalty
More than 5 working days late	Mark of 1.00% / Grade E	Mark of 1.00% / Grade E

- using Cup version 11b-20160615
- using Java SE 8 Update 201
- on CentOS 7.6

§29 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.

§30 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.

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

§32 The deadline for this coursework is Monday 1st April 2019 @ 11:55PM. We strictly adhere to UCL Academic Manual - Chapter 4: Assessment Framework for Taught Programmes and therefore impose the university's late submission penalties, shown in Figure 1. However, the automatic testing system 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 David Kelly `<uclcompiler2019@gmail.com>`.

§33 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. Though the automatic testing system will be running after the release of this coursework (*i.e.* you will be able to push and receive test results), we will *ignore* submissions before Monday 11th March 2019 @ 11:55PM. So please submit *at least once AFTER Monday 11th March 2019 @ 11:55PM*.

5 Automatic Testing Guidelines

Your coursework should be developed using *git*. 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:

```
12345678 <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/
    comp0012/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 Zheng `<z.gao.12@ucl.ac.uk>`.