COMP207P Compilers Lexing and Parsing Coursework

Submission Deadline

Friday 10th March 2017 @ 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 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 10th March 2017 @ 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 $\tilde{Z}$ 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, 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`**.

§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 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`.

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

§7 **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`.

§8 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, `d : dict<int,char> := {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: `d : dict<int,top> := {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.

§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> := [1,2,3]` , and `str : seq<char> := [ '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 `s : seq<top> s := [ 1, 1/2, 3.14, [ 'f', 'o', 'u', 'r'] ];`. The zero length list is `[]`.

§10 $\tilde{Z}$ 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`. String literals are syntactic sugar for character sequences, so `"abc"` is `[ 'a', 'b', 'c']`. For the sequence `s : seq<char> := "hello world"`, `s[s.len-1]` returns '`d`' and `s.len` returns '`11`'.

§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: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.

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]`
Sequence	**seq**	**in**, `::`, `s[i]`, `s[i:j]`, `s[i:]`, `s[:i]`
Comparison	Numeric	`<`, `<=`
	Boolean, Numeric	`=`, `!=`

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

§12 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; "`=>`" denotes implication. Note that "`=`" is referential equality, *not* 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 "`s.len`" 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. 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`.

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

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

§14 For function declarations, each formal parameter follows the variable/field declaration syntax, `id : type`. 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` and is omitted when the function does not return a value. In Listing 2, the function's `name` is an identifier.

2.2 Expressions

§15 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`. $\tilde{Z}$ features a conditioned function call expression. One *must* prefix a predicate, a boolean expression enclosed by `?`, to a function call. If the boolean expression evaluates to `F`, the call immediately returns **`null`**, a literal, without invoking the function; otherwise, the function is executed. The parameters of function calls are expressions that, in the semantic

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

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

`p.age + 10`	Assumes "`p: person;`" previously declared
`b - ?T?foo(?T?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.

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

§16 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 **`loop`**, terminate with a semicolon. The return statement appears in a function body, where it is optional.

Assignment	`var := expression ;`
Input	**`read`** `var ;`
Output	**`print`** `expression ;`
Function Call	`? expression ? 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: $\tilde{Z}$ statements.

§17 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.

§18 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. When appended with a semicolon, a function call becomes a statement. Calling a function whose predicate evaluates to `F` equals `NOP`.

§19 The **`if`** statement behaves like that in the C family language. The unguarded **`loop`** statement is the *only* 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 **`loop`** and **`break`**.

§20 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.

```
a : dict<int, char> := { 1:'1', 2:'2', 3:'3' } ;
b : int := 10;
c : string := "hello world!";
d : person := "Shin", "Yoo", 30;
e : char := 'a';
f : seq<rat> := [ 1/2, 3, 4_2/17, -7 ];
g : int := ?pred?foo(); /# x gets null if pred evaluates to F;
    otherwise x is assigned with the return value of foo. #/
```

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

```
a : seq<int> := [1, 2, 3];
b : seq<int> := [4, 5, 6];
i : int := 0;
j : int := 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: $\tilde{Z}$ loop example.

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

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

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

3 Error Handling

§21 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.

§22 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

§23 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.

§24 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.

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

§26 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.

§27 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 121

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

§29 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.

§30 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

§31 Only one submission per group is necessary. The deadline for completion of this part of the coursework is Friday 10th March 2017 @ 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 as a group, using the git version control system for collaboration. We have created bare git repositories for each group 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 your group. Your coursework will be marked partially based on the results of this test suite.

The automatic testing system is not yet online, but should be turned on by early next week. You can write and run the tests for your coursework on your local machine in the mean time. We will notify you—and this document will be updated with automatic testing instructions—once we have switched it on.