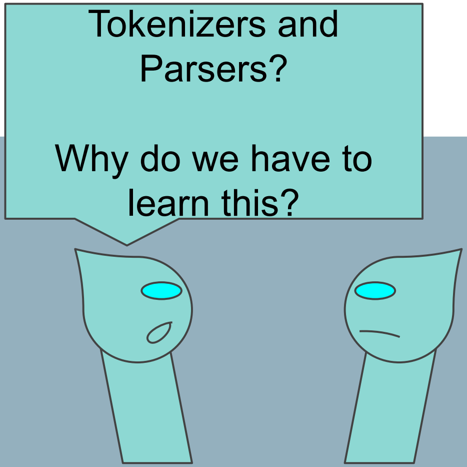
Australian National University

Lab 6 - Tokenizer and Parser

COMP2100/6442



Australian National University

Reminders

First Steps:
1) Fork the comp2100-lab-06 repo to your namespace
2) Fill out the Integrity.md file to allow your Gitlab CI to pass the compliance job ✓

If your CI does not pass with two green ticks then it doesn't count as an attempt and you will get **zero** marks for the lab.

You can see whether your code passes the marking tests by clicking the ✓ in your fork then going to the "Tests" tab (as seen in Lab 1).

Lab assignments have a **HARD DEADLINE**. This means that no late submissions will be accepted (extensions exempted). Your last submission before the deadline is the one that will be marked.

In this lab you are only allowed to edit the code indicated in the comments.

Do not import packages outside of the standard java SE package. The list of available packages can be found here: https://docs.oracle.com/en/java/javase/17/docs/api/index.html

Introduction and Agenda

In this lab you will be building your own calculator!
This calculator will take in a mathematical string, e.g. “1 + 2”, then tokenize it and evaluate it.

To achieve this, you will need to accomplish the following two tasks:
Task 1 : Complete the ‘next()’ method of Tokenizer.java
Task 2 : Complete the four parse methods found in Parser.java

However, before going into the tasks we would like to address a common question amongst students who first see this lab:
“Why are we learning about tokenizers and parser?”
It seems some students believe these concepts are irrelevant… The truth is: we are surrounded by tokenizers and parsers!

Why Learn About Tokenizers

Tokenization
Tokenization breaks up raw text into ‘tokens’ oftentimes more useful to a program than just the raw text.

Here are just a few uses of tokenization:

1. Security (so you are not sending around raw text).
2. Machine learning (divides information into meaningful pieces).
3. Search engines (tokenizes user search input).
4. And more!

Note that in this lab, tokens are Java objects. That is, the INT token holding value ‘15’ is actually an instance of the class Token whose type is INT and value is the string “15”.

Why Learn About Parsers

Parsing
The general definition of parsing is: “analysing a string of symbols according to the rules of a formal grammar” (from Wikipedia).

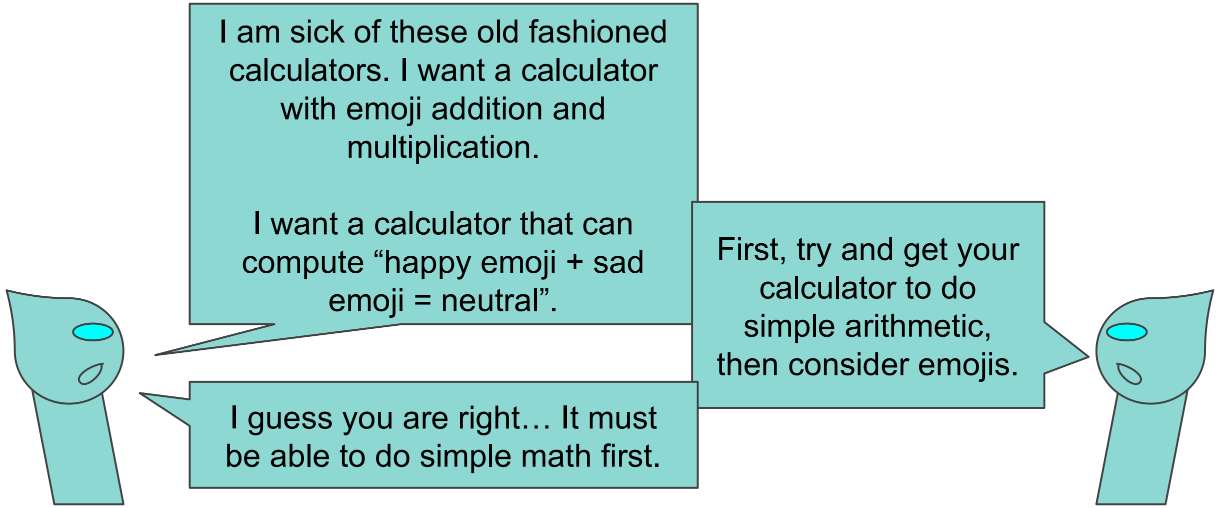
An alternative (which is more alike what we will be doing in this lab) is:
Parsing takes in a series of symbols (tokens in our case) conforming to a grammar and separates them into more easily processed components.

Here are just a few uses of parsing:
1. Compilers (oftentimes used to breaks data into small components but has more uses in compilers).
2. Interpreters (parses the source code).
3. Webpages (html and xml is parsed)
4. And more!

In the task 2 slides, we have provided a step by step explanation of the parsing process to help you understand how this all works.

Reinventing the Calculator

One day, our alien friend set out to make their own calculator. In the process however, they would need to learn about tokenizers and parsers.


I am sick of these old fashioned calculators. I want a calculator with emoji addition and multiplication. I want a calculator that can compute “happy emoji + sad emoji = neutral”.
First, try and get your calculator to do simple arithmetic, then consider emojis.
I guess you are right… It must be able to do simple math first.

Task 1: Tokenizing Input

Your goal in this task is to complete the ‘next()’ method inside Tokenizer.java.

You can view tokenizing as simply a matter of pattern matching:

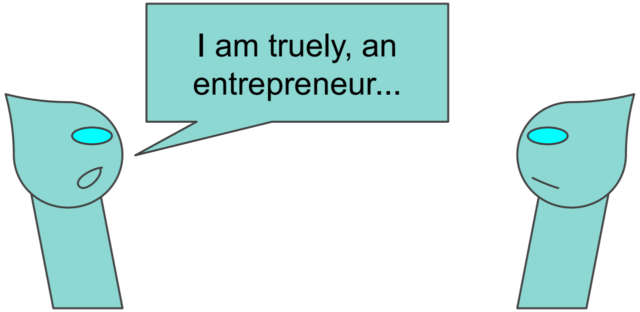
- ‘+’ returns Token.Type.ADD with value ‘+’
- ‘-’ returns Token.Type.SUB with value ‘-’
- ‘123’ returns Token.Type.INT with value ‘123’
- IllegalTokenException if the character does not match any token type.

For example: ‘1 + 2’ becomes: INT(1) ADD INT(2).

To help you both see this in action and test your code, we have provided a main method in Tokenizer.java which will output your tokenized input!

I can put in anything for it to tokenize! (with some exceptions).

Task 1 Complete



Task 2: Parsing the Tokens

You must complete task 1 before you can complete task 2.
The overall objective of this task is to implement a parser for the following grammar:

```
<exp> ::= <term> | <term> + <exp> | <term> - <exp>
<term> ::= <factor> | <factor> * <term> | <factor> / <term>
<factor> ::=  <coefficient> | <coefficient> !
<coefficient> ::= <unsigned integer> | ( <exp> )
```

This takes the form of completing the three parse methods within Parser.java. If any series of tokens provided does not conform to the grammar, you must return an IllegalProductionException.

For a better understanding of what is happening here, we have provided a step by step explanation of the parsing process that you can follow in the next couple slides.

Note that the symbols: “exp”,”term”,”factor” and “coefficient” in this grammer don’t necessarily correspond with their formal meanings in mathematics.

Task 2: Step by Step Explanation of Parsing (Part 1)

Let's first define three things:

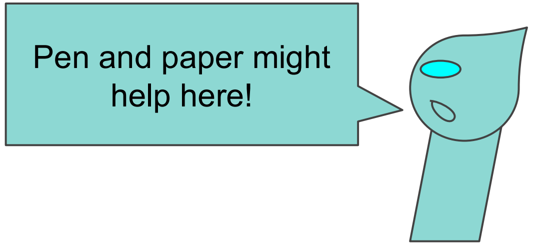
1. Grammar: set of (recursive) rules used to generate patterns of strings.
2. Non-terminal symbol: any symbol that can be replaced with other symbols according to a production rule.
3. Terminal symbol: any symbol which cannot be replaced with other symbols (has no production rule).

In our case, <exp> is a non-terminal symbol but the string '12' is a terminal symbol. Notice as well, that the production rule governing <exp> is recursive as it can create another <exp> symbol.

For something to be parsed, it must conform to the grammar the parse is built for. So let's see how we can check if a string conforms to a particular grammar. Let's say you are given the following string: '12 + 1 * 2'.
How can we use the grammar below to make this string (starting with exp)?

```
<exp> ::= <term> | <term> + <exp> | <term> - <exp>
<term> ::= <factor> | <factor> * <term> | <factor> / <term>
<factor> ::= <coefficient> ! | <coefficient>
<coefficient> ::= <unsigned integer> | ( <exp> )
```

Try this out yourself, the next slide will give you the answer.



Task 2: Step by Step Explanation of Parsing (Part 2)

‘12 + 1 * 2’ can be generated using our grammar in these steps:

1. <exp>
2. <term> + <exp> (using rule 1)
3. <factor> + <term> (using rule 2 and 1)
4. <coefficient> + <factor> * <term> (using rule 3 and 2)
5. 12 + <coefficient> * <factor> (using rule 4,3 and 2)
6. 12 + 1 * <coefficient> (using rule 4 and 3)
7. 12 + 1 * 2 (using rule 4)

At step 7 we no longer have any non-terminal symbols so the process finishes. We were able to use the grammar to form the string. What this tells us is that the string conforms to our grammar and can be parsed.

Task 2: Step by Step Explanation of Parsing (Part 3)

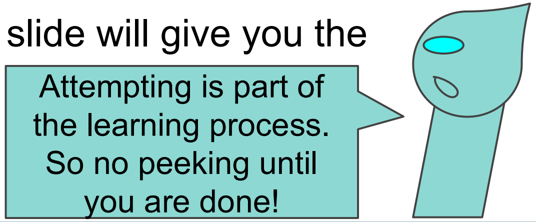
Now that we know the string, ‘12 + 1 * 2’, conforms to our grammar, we will attempt to tokenize it and parse it.

The tokenization process (as we can see with our task 1 tokenizer) provides us with the tokens:

INT(‘12’) ADD INT(‘1’) MUL INT(‘2’)

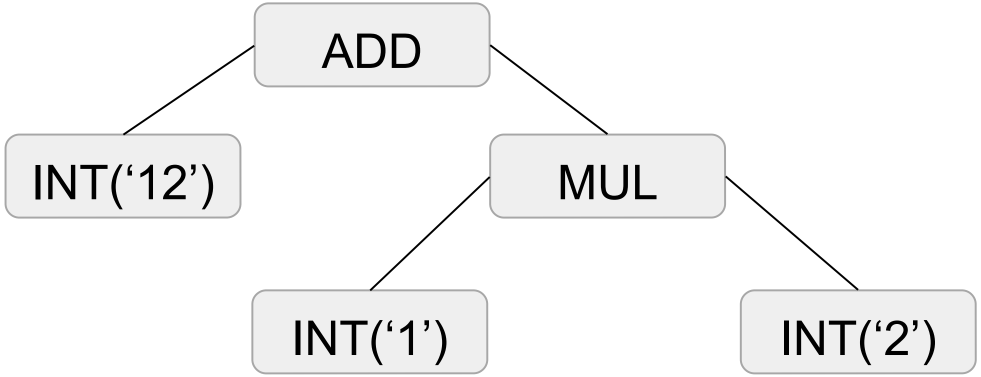
Using these tokens, we can form what is called a ‘parse tree’. In our parse tree, operations connect two values (e.g. ‘+’ connects two integers).

Try to form a parse tree from the above series of tokens yourself. The next slide will give you the answer.


Attempting is part of the learning process. So no peeking until you are done!

Task 2: Step by Step Explanation of Parsing (Part 4)

The series of tokens below form the given parse tree.
INT(‘12’) ADD INT(‘1’) MUL INT(‘2’)



An important note here is that when evaluating a parse tree, the lower operations will be evaluated first. Thus, MUL will be evaluated before ADD.

Task 2: Step by Step Explanation of Parsing (Part 5)

Now let us consider what this parsing would actually look like in code.
In task 2 you are provided with many classes, a few of which will be useful in the next step (the inputs are provided as well):

- AddExp(Term, Exp) returned by parseExp() according to the production rule.
- MulExp(Factor, Term) returned by parseTerm() according to the production rule.
- IntExp(Integer) returned by parseCoefficient() according to the production rule.

Try to use the above to parse the tokens: INT(‘12’) ADD INT(‘1’) MUL INT(‘2’). Keep in mind that we start from parseExp() and build from there.

Australian National University

Task 2: Step by Step Explanation of Parsing (Part 6)

Our series of tokens: INT(‘12’) ADD INT(‘1’) MUL INT(‘2’)
Our returns given the methods:

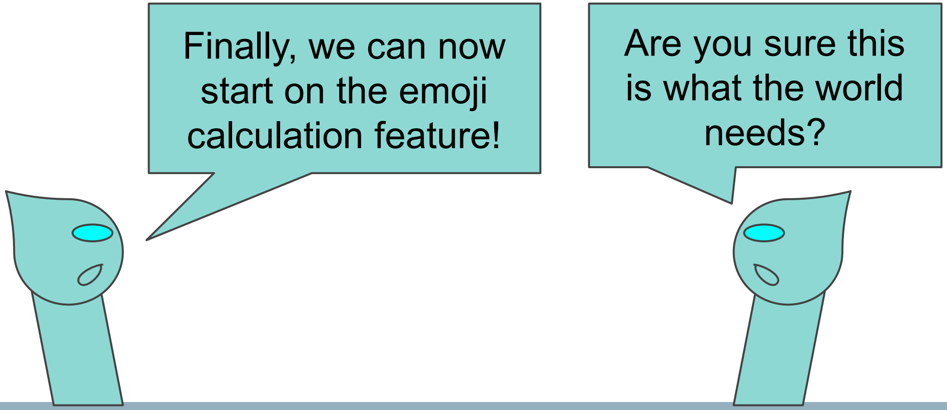
- AddExp(Term, Exp) returned by parseExp() according to the production rule.
- MulExp(Factor, Term) returned by parseTerm() according to the production rule.
- IntExp(Integer) returned by parseCoefficient() according to the production rule.

The step by step:

1. parseExp()
2. AddExp(parseTerm(), parseExp())
3. AddExp(parseFactor(), parseTerm())
4. AddExp(parseCoefficient(), MulExp(parseFactor(), parseTerm())
5. AddExp(IntExp(‘12’), MulExp(parseCoefficient(), parseFactor())
6. AddExp(IntExp(‘12’), MulExp(IntExp(‘1’), parseCoefficient())
7. AddExp(IntExp(‘12’), MulExp(IntExp(‘1’), IntExp(‘2’))

When we finally evaluate this overall expression, the lowest levels (MulExp in this case) will be evaluated first.
That concludes this step by step explanation of parsing. If you are confused by any step, we suggest you ask your tutor.

Task 2 Complete


Finally, we can now start on the emoji calculation feature!
Are you sure this is what the world needs?

Submission Guidelines

Assignment deadline: see the deadline on Wattle (always!)

Submission mode: via your Gitlab fork.

Files marked:

“`Tokenizer.java` and `Parser.java`

Submission guidelines checklist:

1) Forked comp2100-lab-06
2) Commited and pushed your final code to the ‘main’ branch of your fork
3) Passed the CI (i.e., got two green ticks)
4) COMP2100 marker accout has maintainer access to your fork

Violation of the above submission guidelines 1-3 will result in zero marks.

If you can’t work out why your CI isn’t passing, ask your tutor (or post on the Ed forum).

Note that this is a formative assessment and marks are given based on your attempt. We encourage you to collaborate with your tutor and colleagues to complete this task. Don't forget to follow the submission checklist above to ensure the validity of your attempt.

Australian National University

Some Final Words for the Lost

This lab can be hard to get your head around, particularly the parsing section.

Going through and attempting each mini-question in the step by step explanation of parsing may help significantly.

If you are having trouble coding the process, some really good advice is to put a bunch of print statements in your code to see what's happening. Then you can see what needs to be done next.

The tutors are here to help, so make sure to make use of them!