Build your own language
LINGI2132

2020-2021

1 Context

Through the year, you will build your very own language by group of 2 (we won't allow smaller or larger groups). We do not impose limits on what you can do, but we require that you support **at least** the following set of features

- Literals: Integer (at least 32-bits signed precision, `1`, `-999`, ...), Strings (`"Hello"`), boolean
- Comments (e.g, `//`, `#`)
- Variable definitions (e.g., `var` or `let`)
- Simple arithmetic (at least `+`, `-`, `*`, `/`, `%`) with priority of operations (i.e., `1 + 2 * 3` is equivalent to `1 + (2 * 3)`).
- Conditions (e.g., `if`). You should be able to compare: two integers; two variables; one integer and one variable; two boolean values and compositions of these (at least the 'and', 'or' and 'not' operators)
- Loops (e.g., `while`)
- Function definition (e.g., `function`, `def`, `defun`, etc): they must be able to take parameters and return a value.
- Array and map access (e.g., `array[index]`, `map[key]`) as well as function call (e.g., `foo(1, 2)`)
- Printing to stdout (e.g., `System.out.println`, `print`) (will typically be a function)
- a way to pass string parameters to the program: it could be a `main` method or could be implicit `$1`, `$2`, ... parameters (at least 9 parameters must be supported)
- A way to parse strings representing numbers to integers

Note that you might have a brillant idea that contradicts these requirements. In that case you can send us an email (include the whole teaching team) and we will discuss together to see if we can adapt these requirements for your use case.

The project will be divided in four parts:

1. The parsing with the AST generation
2. The semantic analysis
3. The interpretation of the code/ code generation
4. The final submission with extra features (more info in Section 7)

Here is an overall calendar for the whole project:

Part	Start	end
Parsing	S3	Friday S6
Semantic analysis	Monday S7	Friday S9
Interpreter & Final submission		Friday S13

Each part will have its own deadline and will be graded independently. The two last parts have the same deadline to give you more time and flexibility but they are graded independently as well. We expect you, **at each part**, to do **at least all the required features**. For instance, in the first project we expect you to generate the AST for all the features listed above, but we encourage you to do more if you have time. On the other hand, if you did more features in the first part, we **do expect that you first handle the basic features for the second part**. We will not grade the additional features if you do not give us the minimum set of expected features.

You might add/change features as time pass (it is normal, your language will evolve!), do not worry **we will not decrease retroactively your grade**. Basically it means two things: i) You can still add functionnalities (new grammar rule) even if we are at the last part of the project ii) If you could not finish all the requirements for a given part, finish them for the next part to avoid losing more points.

We encourage you to work incrementally, start with small and easy things, then work up the more difficult/complex part of your language. It does not make sense to start looking how you will implement conditions if you do not have booleans.

2 The design of your language

The first task we ask you to do is to think about your language (this part is obviously not graded, but important). Here is a list (**non-exhaustive**) of things you can decide beforehand (remember that you can still change during the course of the semester if you want!)

- Is your language statically or dynamically typed?
- How do you handle errors (e.g., array out of bounds, used of uninitialized variable, division by zero)?
- What are the default values for your types? Do you allow to separate the variable initialization and declaration?
- What nice **extra** features do you want to have (list comprehension, streams, reactive programming)?

It is nice to have advanced features, but try to keep your to-do list manageable and see at the end if you have time to add more things.

3 Part 1. Grammar and Parsing

The first part of the project is about Parsing. That is, your parser should perform two tasks

1. Recognizing valid input for your language. For instance in java, the following strings will be correctly parsed `int a = 2;`, `System.out.println("valid")`, `a + 1 < 2`.
2. Generate the AST for the input strings.

More precisely, you should

- Define the syntax of your language (e.g., do you want ; at the end of statements? How do you declare variables, functions and arrays?, etc.)
- Write you parser using the Autumn library

  - Write the rules for so that your language can be correctly recognized
  - For each rules push, if necessary, AST nodes (you have to build you own classes for that!) on the stack of values

- Write a comprehensive test suite (see Section 6).

4 Part 2. Semantic analysis

In this part you will write the semantic analysis part of your project using the Uranium. This means that you should check things such as

- A variable is used after it is declared
- If you language is statically typed, check the type of the assignment
- That the correct number of argument is passed to a function call
- etc.

Uranium will use the AST you produced in the first part of the project so it is important that you handle that correctly. Note however that your AST structure might be incomplete (for good reasons, you can not anticipate everything!), you should complete it in this part to fit your need.

5 Part 3 & 4. Intepreter & Final submission

5.1 Part 3. Interpreter

Finally, you will write a walk-tree interpreter for your language. In this step you should write a walker that go through your AST and execute the appropriate code. There is no library on which you should rely this time, but you should watch the video on tree-walk interpreter before starting implementing your interpreter. In this stage you will also handle the errors that could not be caught previously (e.g., a division by zero).

As usual you should test your code in depth. We also expect you to provide test cases for the example program defined in Section 6.

5.2 Part 4. Final submission

Once your interpreter is up and running (with some nice tests), your project is roughly finished. But you have time to improve it so here are the things you can still do

- Fix issues of your parsing or semantic analysis. This is the occasion to get some points back (see Section 7).
- Add some additional features to your language. See Section 7 for more information about the grading of the additional features!
- Enhance the lisibility of your code. **We will read your code** so try to make it as clear and nice as possible (for instance try to comment it as you code, it will also help you!)
- Demonstrate the power of your language with more programs that the ones defined in Section 6

You will also have to write a more complete report (see Section 8 for details).

6 Testing

We expect you to test **thoroughfully** your code, for each part. As a rule of thumb, the rules you define for your program should be tested in **isolation** if possible (sometimes it will not be, do not worry).

We will run your tests on INGInious (java 15) and your submission will be considered only if they succeed!. Also note that we will look at your code to see if there is actually some tests.

Moreover we ask you to write the following programs **in your language**

- The FizzBuzz function
- Find the first N (argument of the program) prime numbers
- Print the first N (argument of the program) element of the Fibonacci sequence (use a recursive function)
- A program that takes a serie of Strings as parameters and print them on `stdout` omitting duplicates (use a map)
- A program that takes N integers (arguments of the programs) and print them in increasing order

At each step you should test that these programs parse/are semantically correct and run with the correct output. All your tests, including those on the above programs should be automatically tested by running `gradle test`. We provide you an implementation in Java for these programs, feel free to use these implementation to check the result.

7 Grading

The minimum requirements presented at the beginning account for roughly 2/3 of the total of the points for the project: Each of the 4 parts account for a quarter of these 2/3 account: parsing, semantic analysis, interpreter, and final submission.

You might notice that some design choices lead to less work in some part of the project. Do not worry, this means that the others part will take more time and thus everyone is treated equally.

To get the remaining 1/3 of the points, you need to implement additionnal features to your language. Here is a **non-exhaustive list** of features that might be interesting to add:

- List comprehension
- Pattern Matching
- Generic types
- Lambda functions and closures
- Final variables
- Nullable types
- Object-orientations (e.g. classes that can have subclasses overriding their methods)
- Better handling for the errors/exceptions (e.g., java-style try-catch or go-style defer)
- Better I/O than a simple print function (e.g., functions to read/write to files)

As usual you are free to chose from these features or add others. However, do not underestimate the time it will take you to implements additional features. Moreover we will only grade the additional features **if you have finished all the required features**.

Since you are free to add extra features as you wish during the project, these extra points will be evaluated at the end of the project only in the final submission part.

8 Deliverables

Each part has its own deadline, be cautious about that we will not grade your parser if you do not submit it in time, even if you code it for the semantic analysis! We ask you to provide in each step you source code. Please only include the minimum required files (The gradle files, the sources and your tests (`.java` files). **Please do not include your hidden files, we do not want to have them on our computer when we download your submissions!**).

For the report, we expect you to go over the features of your language and explain them as you go (**provide example of source code in your language**). Explain your choices, the constraints of your language as well as the possibilities it offers. Try to sell us your language!

We require more in the final report (see below).

Here are the deadlines and the expected deliverables

Grammar and Parser We ask you to submit, before **18:00 the Friday 12th March 2021 (S6)**, on **INGInious**:

- Your source code.
- A report of maximum 2 pages, in pdf format, explaining the features of your language. You do not need to provide a formal grammar (in EBNF format, for instance)

Semantic analysis We ask you to submit, before **18:00 the Friday 2nd April 2021 (S9)**, on **INGInious**:

- Your source code
- A report of maximum 2 pages, in pdf format, explaining how you perform semantic analysis in your language.

Interpreter & final submission We ask you to submit, before **18:00 the Friday 14th of May 2021 (S13)**, on **INGInious**:

- Your source code
- A report of maximum 5 pages containing the following pieces of information:
  - A list of the required features actually working and a minimal example for each of them in your language.
  - A list of the extra supported features (cleary separated from the previous list) with, also, a minimal example for each of them.
  - The semantic of each feature (e.g., when you declare a variable, does it have a default value or not).
  - Optionally, interesting observations on your language choices. Please, only tell things that you personally think are interesting or surprising, not things that you think we will approve. Use this section to point things that you think are not obvious in your implementation that we might want to look at and might otherwise miss. This section is NOT for "selling" your work. It is perfectly fine not to have any observations, and you will not be penalized for that.