LINFO2132 Project Statement

Project Overview

The goal of the project is to implement your own compiler for an imperative language (defined by us) from scratch in Java. You will implement the compiler in several phases (corresponding to the phases of the compiler pipeline, i.e., lexing, parsing, etc.). For each phase, we will give you the description of what we expect from you and the deadline by which you must submit your solution. Each phase will be evaluated. All phases together will count towards 60% of your final grade.

Specific instructions on how to submit on Inginious will be given later.

The first phase: The lexer

Your first job will be to implement a lexer for the programming language. You can find an example that shows most of the syntactic elements of that language in the file "code_example.lang". Looking at the example program, you will notice the following syntactic elements:

- We only use characters corresponding to the printable characters in the ASCII standard plus the space character, tab ('\t' in Java), and newline ('\n' in Java).
- Comments start with //
- Identifiers, i.e., names of variables, functions, and types: they can contain letters and digits and underscores, but they have to start with a letter or underscore, i.e., the following words are valid identifiers: abc, abc123, _abc_, _12s.
- Keywords: final, struct, def, for, while, if, else, return
- Values for the different base types:
    - Natural numbers (32-bit, corresponding to int values in Java)
    - Float numbers (32-bit, corresponding to float values in Java). You only need to consider the decimal point (e.g., 3.24) and you don't have to implement lexing for numbers in scientific notation (e.g. 4.5e-2).
    - Strings. They are enclosed in double quotation marks and can contain any printable character, space characters, and escaped characters, i.e., special notations for some characters, namely \n (newline), \\ (backslash), and \" .
    - Boolean values: true and false.
- A variable can be declared immutable (`final` keyword), and all variables are pre-typed ("int a" and not "a int").
- Various special symbols: =, +, -, *, /, %, ==, <>, <, >, <=, >=, (,), {, }, [,], . (dot), &&, ||; (semicolon), and the comma.
- Symbols **might** be separated by space, newline and tabulator characters (the so-called *whitespace*). For instance, these two inputs would generate the same sequence of symbols: `var x int = 12;`, `var x int=12;`

Your implementation of the Lexer must respect these four points:

1. You must pass a Reader object representing the input to the Lexer constructor.
2. You must implement a method `getNextSymbol()` which returns the next symbol in the sequence of symbols lexed from the input.
3. Each returned symbol must be printable. All symbols should have concise et precise names.
4. We will run your code with the command “javac main.java -lexer filepath”. The argument “-lexer” is optional and enables a debug mode where the output of the lexer is printed. The argument “filepath” will be the file used as input in your main.

Apart from that, you are free to add anything you find necessary (other classes, methods, etc.). Some general advice:

- It's up to you how you want to represent the end of the input. You could let getNextSymbol() return a special symbol or just default class that represents null.
- You can also freely decide how you want to represent symbols: as a single Java class, as multiple subclasses (for each token type), etc.
- It will be much more convenient for the parser if getNextSymbol does not return symbols for whitespaces and comments.

Error Handling

During the whole compilation process, multiple errors might arise. It is important to understand when each of these errors must be thrown at the right time.

For instance, nothing forbids the programmer to write something like `var a int = “Hello” / 2;`

This is syntactically correct, but semantically wrong. It is thus not the role of the lexer or parser to throw such error.

The role of the lexer is to transform the input file into a sequence of symbols. Hence the only errors you must handle are unrecognized tokens (e.g., a “@” character).

What you must not do

We expect from you that you write the lexer by hand. It is <u>forbidden</u> to

- use lexer or parser generation tools or the code generated by them,
- use lexer and parsing tools and libraries (external or in the JDK, such as the Scanner, Pattern and Matcher classes),
- copy solutions (or parts of them) from your fellow students or from the Internet.

The second phase: The parser

The goal of this phase is to produce an Abstract Syntax Tree (AST) from the output of the Lexer. You must transform the sequence of symbols produced by the Lexer into a tree structure that represents your program. You must respect the following guidelines:

- The AST must respect the syntax of the language.
- If the sequence of symbols does not respect it, then you must report an error. For example `int a 2` is syntactically incorrect (the `=` is missing).
- You must not report semantic errors (e.g. `int a = “Hello”`)
- Implement the parser in a class Parser with the lexer as constructor parameter. You should have a method `getAST()` that returns the root of the AST.

Your implementation of the Parser must respect these two points:

1. Each AST node must be printable. All nodes should have concise et precise names.
2. We will run your code with the command “javac main.java -parser filepath”. The argument “-parser” is optional and enables a debug mode where the output of the parser is printed in the form of a tree. The argument “filepath” will be the file used as input in your main. For example, the printed output of “Int x = 1 + 2” would be: (note that variations can exist with your nodes; for example, “ArithmeticOperator, +” might be “AddOperator”; the important is that the role of each node is understandable)

Expr

    Type, Int

    Identifier, x

    AssignmentOperator

    Expr

        Integer, 1

        ArithmeticOperator, +

        Integer, 2

Some tips:

- You might see a design problem/bug in your Lexer when implementing the AST generation. Do not hesitate to refactor your code. This is true for the entire project and you should not be afraid to change some previous parts in order to facilitate the overall coding of your compiler.
- You’ll need to code a lot of boilerplate code. Try to use abstract classes, interface, etc. as efficiently as possible.
- You should re-use the test cases defined for the Lexer. This will ease the test part.
- You should also create new test cases base on what is syntactically (in)correct.
- If you implement the `equals(Object o)` method in your objects, you can use the assert methods from JUnit with your custom objects (e.g., `assertEquals(expectedRoot, myRoot)`)

Third phase: Semantic analysis

At the end of the previous phase, you should have produced, from the input file, a syntactically correct AST. Hence, you know that the input file respects the syntax of the language. But this does not tell you if all the operations you do in the file are correct from the semantic point of view. For instance, the syntax of the language does not prevent you from assigning an integer to a float variable. Such errors are called “semantic errors” because they are linked to the semantic of the language.

In this phase you must analyze your AST and report any semantic errors, which are mainly type errors. Our language is strongly typed and we do static type checking, i.e., most of the type checking happens during the compilation. For type equivalence, you can use named equivalence (easy) or structural equivalence (more difficult).

You should at least handle the following cases. Each case must lead to a program crash with a custom error message with the main function returning a value different from 0 and not null (e.g. “System.exit(2)”). We assign a keyword to each case which must appear in your error message.

- The type of variable, value and constant are correct. The right-hand side (rhs) of an assignment should have the same type as declared in the left-hand side (lhs); keyword: “TypeError”
- Structure declarations are correctly registered as new types for variables and cannot overwrite existing types (e.g. “struct int”, “struct while”, or overwrite a previously defined structure); keyword: “StructError”
- Binary operators operate on the correct types: same numeric type for arithmetic operations, boolean for comparison operators. Same for unary operators. Keyword: “OperatorError”
- Function calls have the correct parameter types (including record constructor). Keyword: “ArgumentError”
- Loop and if constructs have boolean conditions. Keyword: “MissingConditionError”
- Return statements return the correct type in functions. Keyword: “ReturnError”
- The scopes are handled correctly. This means that when an identifier is used you should check that it is in the current scope, or a parent scope. Shadowing of global variables by function parameters is allowed. An error can arise if a variable is called before being defined, or more generally if a variable is used out of scope. Keyword: “ScopeError”

Your implementation of the Semantic Analysis must respect these four points:

1. Your program should throw an error that includes the associated keyword for each above cases. The program must return a value other than 0 and not null in these cases.

2. You must create at least one test for each of the points above and check if the correct error is thrown. We will check and run these tests. If needs be, you can write a short report to show how to run them, but it should be as simple as possible.

Some tips:

- If the first part was done correctly, a traversal of the AST should be able to handle most of the points above. However, it might happen that the design of your AST is not optimal (which is normal!). If you find that the structure of your AST is not well suited for the semantic analysis part, we advise you to refactor it. If you do not manage to handle all these points by the deadline, try to still submit something as it will always positively impact your final grade and refactor after, for the next phase, if necessary.
- Remember that after the semantic analysis, you'll have to generate code based on the AST. A good structure is thus essential.
- You're also free to generate more errors as you wish, try to be creative! A good compiler is not only able to correctly generate code based on its input but must also give a clear explanation of why the compilation failed. And it should point out the errors in the source file. These errors must be checked by additional tests, or we will not see/grade them.

Submissions and grading

All submissions will be done on Inginious. For all submissions, we expect a zip file containing your current code, and a report (two separate tasks).

- The code contains:
    - Source files of your current progress.
    - Test files for each part of your project (for the current deadline; e.g. only one test file at the first deadline for the Lexer). These test files should be well-organized and easy to run, as we can use them to check your progress.
- The report contains:
    - A concise explanation of the current progress of your compiler. If you have partially fulfilled the requirements for a part (e.g., you have done the parsing but miss some language features) you can also explain what the problems are (technical problems, you do not need to explain that you did not have the time)
    - The implementation choices you made. It's important to explain not only what you decided to do, but also the impact this has on the code structure, the efficiency of your compiler etc. You have a lot of freedom in this project, we expect you to make decisions and **justify** them in the reports. You don't have to explain everything, only important implementation choices (and concisely).

    - The quality of your report (clarity, conciseness, organization, quality of writing, respect of the instructions) will be part of the grading. Be as direct as possible: no flowery language, and an image is worth a thousand words.

The grading and expected deliverables are as follows. Exact percentages are subject to changes.

- Lexer 15%: code
- Parser 20%: code and report
- Semantic analysis 20%: code and report
- Code generation 30%: code and report
- Extra features: 15%

Notice that 15% is reserved for extra features. Before the last deadline, you will have time to either: i) Finish the project if you are behind ii) add some extra features to your language (for the 15 extra percents). You should focus on extra features only when you are done with all previous phases.

You are free to enhance your compiler in any way you find interesting. This can be with new language features, optimization in the code generation, less restriction in the source file etc.

If you are not sure whether some of your ideas are doable, do not hesitate to contact the teaching assistants.

Resources

You can find on Teams:

- A file with all the language's features.
- A skeleton of gradle project with the signature for the Lexer (you must keep the project as a gradle project)

Deadline

- Lexer: Monday March 4, 23:59
- Parser: Monday March 25, 23:59
- Semantic analysis: Monday April 15, 23:59
- Code generation: Monday May 20, 23:59 (instructions coming)