716.077 DP Compiler Construction - Programming Assignment

Alexander Perko,
Christopher Liebmann, Sebastian Puck
Institute for Software Technology, Graz University of Technology

April 1, 2024

Compiled on 01/04/2024 at 7:24pm

Contents

Introduction

During this semester you will build a `Compiler` for a simple object oriented language, which we like to call `Jova`, in `Java` using the `ANTLR 4`[1] parser generator. `Jova` incorporates, for example, class declarations, inheritance, conditional statements, loops, as well as method calls. Please see Section **Jova** for more details.
Your compiler should be able to translate a `Jova` source program into `Java` bytecode, which then can be converted into executable `Java` class files via the `Jasmin`[2] assembler. Figure 1 illustrates the different phases of a compiler. The phases which will be part of your exercise are highlighted in Grey.

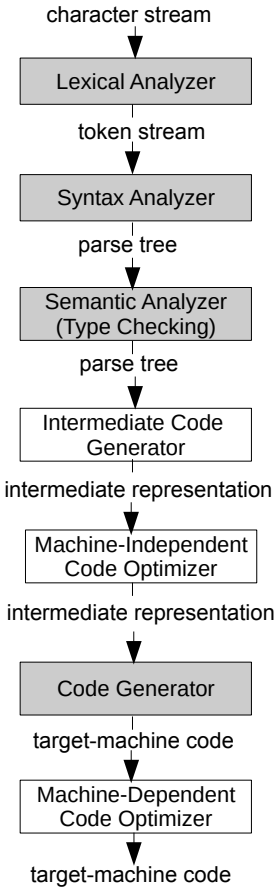

Figure 1: Compiler

[1] https://www.antlr.org/
[2] https://jasmin.sourceforge.net/

Schedule

The following table provides an overview of your tasks and the individual delivery deadlines:

	Task 1 *Lexer & Parser & Class Analysis*	**Task 2** *Type Checker*	**Task 3** *Code Generator*
Release & Tutorial	7^{th} of March, 10:45-12:15, HS i8	11^{th} of April, 10:45-12:15, HS i8	8^{th} of May, 14:30-16:00, HS i8
Q & A	11^{th} of April, 10:45-12:15, HS i8	8^{th} of May, 14:30-16:00, HS i8	6^{th} of June, 10:45-12:15, HS i8
Deadline	12^{th} of April, 23:59	17^{th} of May, 23:59	7^{th} of June, 23:59
Interviews	12^{th} to 14^{th} of June		

General rules

- Course Communication
  - For general questions, please use the TeachCenter Forum.
  - The reading of the forum and TeachCenter is **mandatory**, i.e., all information stated there is binding.
  - **Only for private organizational** questions you may reach us via the mailing list (`cc@ist.tugraz.at`).
- Preliminaries
  - You have to create an account on `GitLab` (**https://git-students.ist.tugraz.at**) using **TUG SSO** before the registration deadline.
  - You have to build groups of 4(!) students and register online using the web interface (see the TeachCenter for details).
  - Once you have created an account and registered for a group, we cannot deregister you anymore from the design practical, i.e., you will receive a grade for this course.
  - **Deadline:** 11^{th} of March, 23:59.
- Submission
  - You have to deliver tasks 1-3 via `GitLab` (note that you do not have to work with Git, but only have to deliver your submissions using `GitLab`).
  - For the duration of the assignments, you need a `GitLab` account. After registering via **TUG SSO**, the project will be created for you **after the end of the group registration**.
  - To submit your implementation, it must be in the correct branch and thus the branches `task1`, `task2`, `task3` must be used to submit the respective tasks. Therefore, submitting task `taskN` should be possible by using `git checkout -b taskN` followed by `git push -u origin taskN`. Afterwards, with `git checkout main` you can go back to the `main` branch.
  - You are allowed to hand in each programming task 24 hours after the deadline. Please write an e-mail to `cc@ist.tugraz.at` **before the regular deadline** is up containing your group number and stating that you would like to have this deadline extension. Note that if you do take advantage of this, you will receive a deduction of 25 % of your points for this task. There will be no further extensions of the deadlines.
  - Make sure your file structure is correct as a deviation will lead to a deduction of up to 100 % of the points for the corresponding task.

- Not executable/compilable submissions, submissions not creating any output and submissions stuck in an infinite loop will lead to 0 points fort the corresponding task.
- Discussion of the tasks between the lecture participants is welcome, but **plagiarism is unacceptable** and will lead to a negative marking of the course. Negative marks due to plagiarism are recorded in the TUGRAZOnline database.
  Do not include any personal information (name, student ID, mail-addresses etc.) in your source code since your files will be uploaded to an external server in order to cross-check against other submissions from this year and the previous years.

- Mandatory Interviews
  - There is a mandatory interview after the completion of task 3.
  - Absence without valid (and verifiable) excuse (e.g. illness) during the mandatory interviews will result in 0 points for the entire course.
  - Every team member has to be able to answer questions about the tasks and explain the source code. If a team member is not able to do so it will result in a deduction of up to 100% of the achievable points for this particular student.

- Grading / Private Test Framework
  - Your submissions will be graded via our private test suite. These tests will not be published and differ from the public tests of the framework.
  - We will test your solutions using `JDK21`, `ANTLR 4.13.1` and `Jasmin` 2.4. It is highly recommended that you develop and test your solutions with the same software versions, since a not working solution due to incompatible software versions results in 0 points for the respective task.
  - You will be able to test your solutions on a private test runner with limited access to the results near the end of the deadline.
    This runner comes with **no guarantees** from our side. We might add or remove tests for grading and we do not provide feedback concerning failing private tests before the actual deadline. Furthermore, the test runner might go offline at irregular times or due to overload shortly before the deadline.
  - Do not solely rely on the public tests to check whether your implementation is working correctly. Create your own example programs to test every important aspect of your implementation. You can add your custom tests to the test-folder.

- Implementation Requirements

  Since we are going to grade your submissions automatically make sure your implementation meets the following minimum requirements:
  - Program Entry Points
    Our test system will instantiate the class `Jovac` and call one of the following methods for the respective task: `task1(..)`, `task2(..)`, or `task3(..)`. Make sure your implementation can be called via these methods.
  - Compilation Error Classes
    You can find predefined classes[3] for every compilation error and warning you have to report during your implementation in the framework. These classes also contain predefined error messages, which you can print via the `JovaErrorPrinter` class.
    It is **mandatory** to use these classes for reporting errors in Task 1 and 2. You can find more information in the respective task or in the framework.
  - Error Reporting
    You will have to implement specific getter-methods of the `Jovac` class which return a collection of the found errors and warnings. Our test system will call these getter-methods at the end of every test to check for reported errors. Make sure that it is

[3]These classes are located in the package `at.tugraz.ist.cc.error`.

possible to access all collected errors and warnings specific to the regarding task via these getters after the respective method for the task (i.e., `task1(..)`, `task2(..)` or `task3(..)`) has been executed.

- Static Data Structures
  Since our test system is based on JUnit tests be careful with static data structures. Make sure to manually reset them at the beginning of your implementation.

- Bonus Points
  - You can earn additional points by implementing one or several of the bonus tasks.
  - Bonus points are capped with 10 % of the regular task points meaning no matter how many bonus tasks you implement you can at most get 10 bonus points in total.
  - To get the bonus points you need to provide your own private test programs, extend the `JUnit` test, and document your changes in the `README` file.
  - In addition, bonus points are only counted towards your grade if you actively participated in all tasks and get at least half the points on Task 3.
  - In case you decide to implement a bonus task, you do not have to pursue the feature in the upcoming assignments. However, you cannot receive points retroactively for previous tasks.

Framework

You can clone the framework from our upstream repo. This repo can be found on `https://git-students.ist.tugraz.at/cc24/framework`.

This framework contains among others

- a skeleton `Jovac` class and predefined error classes for testing,
- a `Main.java` file,
- `JUnit` test files,
- example input and output files,
- a `README` file and
- gradle (build) files (use `gradlew tasks --all` for information).

Upload this repo to your own repository.

```
$ git remote rename origin upstream
$ git remote add origin <your-repo-url>
$ git push -u origin main
```

Do not change the gradle files. For testing your submission we will utilize the `compileTestJava` task. Furthermore, we will add additional material such as new classes for the release of new assignments to the upstream repository.

File/folder hierarchy

Your `GitLab` repository must have the following structure (**Please take care that the required files are in the correct folders!**):

- `build.gradle.kts`
- `settings.gradle.kts`
- `readme.txt`

- src/
  - main/
    - antlr/**/Jova.g4
    - java/**/*.java
  - test/

Do not change the package `at.tugraz.ist.cc.error`, since the containing classes are used for automated testing!

readme.txt

The file `readme.txt` must contain:

- full names, matriculation numbers and e-mail addresses of all team members as well as your group number
- general remarks related to your implementation in clear and short sentences as well as known limitations of the implementation
- a list of all implemented bonus tasks where each bonus task is listed by its id (can be found within the assignment sheet where the related bonus task is described). For example, if the bonus tasks `bonus task 1` and `bonus task 2` have been implemented, it is necessary to list them at the readme's respective section as - `bonus task 1` and - `bonus task 2`, each in a separate line.
- A description and the file names for the self-written test cases

Jova

`Jova` is a simple object oriented language inspired by `Java`. Please see Listing 1 for an example program written in this language.

Listing 1: Example Jova program

```
Person {
  string name;
  int printInfo() {
    print("Person:\n");
    print("Name: ") + print(name) + print("\n");
    return 0;
  }
}

Student : Person {
  int matr_num;
  int printInfo() {
    print("Student:\n");
    print("Name: ") + print(name) + print("\n");
    print("Matr.Num.: ") + print(matr_num) + print("\n");
    return 0;
  }
}

Employee : Person {
  string office;
  int printInfo() {
    print("Employee:\n");
    print("Name: ") + print(name) + print("\n");
    print("Office: ") + print(office) + print("\n");
    return 0;
  }
}

Test {
  int main() {
    // init new Student
    Student s;
    s = new Student;
    s.name = "SpongeBob";
    s.matr_num = 50191;

    // init new Employee
    Employee e;
    e = new Employee;
    e.name = "Squidward";
    e.office = "The Krusty Krab";

    // print info
    s.printInfo();
    e.printInfo();
    return 0;
  }
}
```

When compiling and running the program from Listing 1, the following lines are printed:

```
% java -cp . Test
Student:
Name: SpongeBob
Matr.Num.: 50191
Employee:
Name: Squidward
Office: The Krusty Krab
```

Jova Type System

In `Jova` there are three kinds of types:

- Builtin Types
  - **`int`**, i.e., integer numbers
  - **`bool`**, i.e., boolean values (**`true`**, **`false`**)
  - **`string`**, i.e., character strings
- Null Type
  - **`nix`**, i.e., type of the expression null[4]
- User Defined Types (Class Types)
  - all object types, e.g., `Student` or `Test` in Listing 1

[4] `nix` is the Styrian equivalent to `null`.

1 Task 1

In the first part of this task you will create a lexical and syntax analyzer for the `Jova` language.

1.1 Lexical and Syntax Analysis

Resolve all **TODOs** in the `Jova` grammar in Appendix A and adapt the supplied `Jova.g4` grammar file in the framework. Generate a lexer and parser in `Java` with `ANTLR` and check a source program for lexical or syntax errors. Lastly, report any compilation error found.

Tasks

1. Create the appropriate lexer token for integer numbers. This will be essential in the syntactical analysis. The following table contains examples of valid and invalid integers:

Valid integer	Invalid integer
`0, 1, 40423`	`01, 003426, 3E2, -42`

2. Create the `param_list` rule used to define the syntax of a method's parameters. For each parameter within the parameter list, there is the parameter's `type` followed by its `ID`. Furthermore, note that parameters are separated by a comma (`,`). The following listing contains examples of valid and invalid parameter lists:

Listing 2: Examples for valid and invalid Jova parameter lists

```
...
// examples for valid parameter lists
int valid1(bool a) { ... }
int valid2(bool a, int b, string c) { ... }
int valid3() { ... }

// examples for invalid parameter lists
int invalid1(, bool a) { ... }
int invalid2(bool a, int b, ) { ... }
int invalid4(bool, int, string) { ... }
int invalid5(a, b, c) { ... }
...
```

3. Create the `decl` rule for declaring local variables and fields. When declaring a variable, its `ID` should be preceded by its `type`. Moreover, the rule must also support the declaration of multiple variables where a comma (`,`) serves as delimiter and the end must be indicated by a semicolon (`;`). Valid and invalid examples for declarations can be found in the following listing:

Listing 3: Examples for valid and invalid Jova declarations

```
...
// examples for valid declarations
bool a;
int a, b;

// examples for invalid declarations
, bool a;
bool a, b, ;
bool a, int b;
...
```

4. Complete the `expr` rule. `expr` should match a single operand or an arbitrary (potentially nested) binary expression of the form:

```
expr   → expr (<OPERATOR> expr)*
       → (ADDOP | NOT) expr
```

<OPERATOR> should be a ADDOP, ASSIGN, RELOP, MULOP, AND, OR or DOT. In addition you should structure the rule according to operator precedence and ensure the appropriate associativity.
The assign operator is right-associative: `a = b = 42` is equal to `a = (b = 42)`. Every other binary operator is left-associative: `a + b + c` is equal to `(a + b) + c`. Thus, expressions on the left-hand side within a chain of operators of the same precedence shall be computed first. Operators abide to the following precedence:

DOT > NOT / unary ADDOP > MULOP > ADDOP > RELOP > AND > OR > ASSIGN

Whereby DOT will be resolved first and ASSIGN last. The resulting parse tree should represent these rules.

The following listing contains examples of valid and invalid expressions:

Listing 4: Extamples for valid and invalid Jova expressions

```
1  ...
2  // examples for valid expressions
3  1;
4  1 + 2;
5  a;
6  a + 1 > 2 + 3 < 10;
7  a.b == b() && c != 0;
8  a = b();
9
10 // examples for invalid expressions
11 *1;
12 a 1;
13 1 a();
14 a + b c;
15 ...
```

Hint: Create 1 alternative production in you grammar file for every operator-kind.

5. Complete the **id_expr** rule's portion responsible for parsing the arguments of a method invocation. Arguments are represented by a list of expressions (**expr**). A comma (**,**) should delimit individual arguments. Valid and invalid examples are demonstrated in the following listing:

Listing 5: Examples for valid and invalid Jova method calls

```
1  ...
2  // examples for valid method calls
3  a();
4  a(b, c);
5  a(b(c + 1));
6
7  // examples for invalid method calls
8  a(, b);
9  a(b, c, );
10 a(1 + 2 b);
11 ...
```

6. Modify the provided grammar file `Jova.g4` in the framework to match the grammar you created in the previous tasks. Do not modify the header of the grammar file! Generate the `JovaLexer` and `JovaParser` with `ANTLR` from this file.
   Implement the following methods of the class `Jovac.java` in the package `at.tugraz.ist.cc` in the `src` directory:

   - `public void task1(String file_path)`
     Invoke the generated lexer and parser on the `.jova` input file indicated by the parameter `file_path`.

     By default these classes will print any found lexical or syntax errors to `stderr`. Intercept these generated error messages so that they are no longer printed to `stderr`. Instantiate a new

     - `LexicalError`[5] for every intercepted lexical error, and a
     - `SyntaxError` for every intercepted syntax error

     with the error message, line number and character position of the intercepted errors. Store these objects so that they are accessible via the matching getter-methods after the parsing process.

     Write a summary of all found errors to `stdout` including the line number and character position of the errors. You can use the methods of the provided `JovaErrorPrinter`[6] class to print the expected error summary.
   - `public Collection<LexicalError> getLexicalErrors()`
     This method returns all intercepted `LexicalError`s in the order they were reported by the `JovaLexer`. Furthermore, it returns an empty collection if no errors were reported.
   - `public Collection<SyntaxError> getSyntaxErrors()`
     This method returns all intercepted `SyntaxError`s in the order they were reported by the `JovaParser`. Furthermore, it returns an empty collection if no errors were reported.

7. Adapt the `README` file as described above.

1.2 Class Structure Analysis

For the second part of this task you will have to traverse the generated syntax tree of the input program, which your parser has created as output in the first part. Furthermore, you will have to report any found `SemanticError`s by instantiating the correct subclasses.
Note that for Task 1 we are only concerned about the **declarations of classes, fields and methods**. You will check the method bodies (i.e., statements, expressions and local variable declarations) in Task 2. The relevant semantic rules for the structure of `Jova` classes are informally described in the following subsections.

Scoping

`Jova` is lexically scoped. Thus, name bindings are valid within the block they are declared, i.e., the program (block) as well as the class and method body. Each of these blocks has its own scope, which can contain class, field, method and local variable (see Task2) declarations respectively.

Identifiers

Class and field names can only be declared once in the respective scope. Classes can be used as type before they are declared in the source file.
Report a `IDDoubleDeclError` if a name is declared more than once in a given scope. Furthermore, report a `IDUnknownError` if a class name is used, which is not declared in the global scope.

[5] Both classes are located in the package `at.tugraz.ist.cc.error.lexandparse`.
[6] This class is located in the package `at.tugraz.ist.cc.io`.

Listing 6: Examples - ID double declaration errors

```
Krusty {
  int krab;
  string krab;
  //      ^ line 3 - Double declaration of identifier: 'krab'.
}
...

   Krusty { int fish; }
// ^ line 8 - Double declaration of identifier: 'Krusty'.
```

Listing 7: Examples - Undefined ID error

```
  ...
  UndefBurger cookBurger() {
//^ line 2 - Usage of unknown identifier: 'UndefBurger'.
    ...
  }
  ...
```

Methods

In `Jova` a *method signature* is defined to be the combination of the method name and its parameter types (in the declared order). A method signature can only be used once in a given class scope to define a method. Report a `MethodDoubleDefError` if a method with the same method signature is defined more than once in a given class scope.

Listing 8: Examples - Method double definition errors

```
Krusty {
  int cookBurger(int patty, bool sauce) {
  ...
  }

  bool cookBurger(int patties, bool ketchup) {
  //    ^ line 6 - Double definition of method 'cookBurger(int, bool)'.
  ...
  }
}
```

Furthermore, report a `IDDoubleDeclError` if a parameter name is used more then once in the same method signature.

Listing 9: Example - Parameter double declaration error

```
  ...
  int cookBurger(int patty, string patty) {
  //                                ^ line 2 -  Double declaration of
  //                                            identifier: 'patty'.
  ...
  }
```

Built-in Functions

The following built-in functions are usable throughout a `Jova` program without the need of a class instance:

- `int print(int)`[7]: prints an integer to stdout.

[7]The `print`-methods are expected to return `0`.

- `int print(bool)`: prints a boolean to stdout.
- `int print(string)`: prints a character string to stdout.
- `int readInt()`: reads an integer from stdin.
- `string readLine()`: reads the next line from stdin.

It causes a `MethodDoubleDefError` when a method with one of these signatures is define in a class.

Listing 10: Example - Print "protection"

```
1 Krusty {
2   int print(int a) {
3   //   ^ line 2 - Double definition of method 'print(int)'.
4     return a;
5   }
6 }
```

Note that, these functions are translated to equivalent calls to standard library functions in Task 3.

Inheritance

`Jova` supports single inheritance, i.e., a `Jova` class can be derived from exactly one other class. In case the inheritance hierarchy contains a cycle, a `CyclicInheritanceError` is reported.

Listing 11: Example - Cyclic inheritance

```
1 Krusty: Krab {
2 //       ^ line 1 - Cyclic inheritance between 'Krusty' and 'Krab'.
3   ...
4 }
5
6 Krab: Krusty {
7   ...
8 }
```

Furthermore, all defined fields and methods of the superclass are also defined within the scope of the subclass. A field declaration in the subclass with the same name as a field of the superclass is *hiding* the declaration of the superclass. Thus, no error is reported in such a case and the hiding field declaration is visible inside the subclasses scope.
Analogically, method definitions with same signature but a **different return type** are *hiding* definitions with the same signature of the superclass. However, method definitions with the same signature **and** return type are *overriding* the definitions of the superclass. In such a case an `OverrideWarning` is reported at the position of the overriding method.

Listing 12: Example - Inheritance

```
1  Krusty: Krab {
2    int customers; // hides field declaration of superclass
3    int cookBurger(string name) {...}
4    //   ^ 3:6 - Method 'cookBurger(string)' overrides method of superclass.
5    bool withFries() {...} // hides method definition of superclass
6  }
7
8  Krab {
9    string customers;
10   int cookBurger(string bruger_name) {...}
11   string withFries() {...}
12 }
```

Error Reporting

To report an error or warning instantiate the appropriate error class[8] via supplying the *correct* arguments to the respective constructor. Save the created instances so that they can be accessed after the semantic analysis via the getter-methods.

Semantic errors are reported from top to bottom and left to right of the source file, i.e., an error in line 1 comes before an error in line 2 and an error in line 3 at character position 1 comes before an error in line 3 at character position 10 in the error collection.
You can skip the check of the entire body of a class if you have found an error in its declaration.

Tasks

1. Extend and implement the following methods of the class `Jovac.class` in the package `at.tugraz.ist.cc` in the `src` directory:

   - `public void task1(String file_path)`
     Extend your implementation from the first part of Task 1. In case any lexical or syntax errors are found, do not perform any further analysis and abort the compilation process via returning from this method. If no errors where found during parsing, traverse the resulting parse tree and check the class structure of the `Jova` program. You can use a listener or visitor implementation generated by `ANTLR` to do so. You can skip the traversal of the method bodies for now, as you will check them in Task 2.
     Check whether the input program conforms to the semantic rules of the `Jova` language which are explained above. Report any semantic errors or warnings that you find via instantiating the appropriate `SemanticError` or `JovaWarning` classes and save them in the expected order.
     Write a summary of all errors and warnings found during the analysis to `stdout`, including the line number (and in case of warnings character position). You can use the methods of the provided `JovaErrorPrinter` class to print the expected error summary.
   - `public Collection<SemanticError> getSemanticErrors()`
     This method returns all reported `SemanticError`s in the specified order. It returns an empty collection if no errors were reported.
   - `public Collection<JovaWarning> getWarning()`
     This method returns all reported `JovaWarning`s in the specified order. It returns an empty collection if no errors were reported.

2. Adapt the `README` file as described above.

Notes and Hints

- You may rewrite rules of the provided `Jova` grammar. However, the rewritten grammar has to be **equivalent** to the given grammar from the assignment description, i.e., the resulting lexer/parser has to **accept** and **reject** the same set of inputs as a lexer/parser generated from the provided grammar would.
- For debugging your grammar implementation you can use the `testRig`[9] gradle task or (in case you are using IntelliJ IDEA with the `ANTLR` 4 plugin) the `ANTLR` preview tab. This will give you a visual representation of the constructed parse tree.
- The correct handling of the operator precedence will be essential in the upcoming tasks, i.e., correctly implementing it in Task 1 will help you in the next assignments.

[8] You can find all classes for semantic errors and warnings in the package `at.tugraz.ist.cc.error.semantic`.
[9] `gradlew testRig -PfileName=<file>`

- You can inspect the provided reference outputs for this task to check whether your implementation works correctly for the given public tests. Note that, the reported error messages of the generated lexer/parser may differ due to how you structure your grammar, i.e., deviating error message may not indicate an actual error in your grammar.
- Design and implement a data structures (**symbol table**) to hold the type information for identifiers or symbols. One possible design approach would be to construct a symbol table in accordance to the structure of the scopes of the different blocks in a `Jova` program.
  You will also need the symbol table for Task 2 and 3. Thus, you may want to design this data structure with that in mind, e.g., add support for local variables and type information.
- You may need to traverse the syntax tree (or specific parts of it) several times to perform the wanted analysis.
- For the semantic errors and warnings make sure that the **first** error in the input file is correctly reported.

2 Type Checking

In the second task you will finish the semantic analysis of a `Jova` program by adding type checking to your compiler. To achieve this, you will have to traverse the method body branches of the syntax tree. Same as for Task 1.2, you will have to report any found `SemanticError`s and `JovaWarning`s by instantiating the correct subclasses.
The remaining `Jova` semantic and type checking rules are informally described in the following subsections.

Jova Types

`Jova` supports 3 built-in types:

- `int` (32-bit signed integer) - e.g.: `0`, `1`, `-42` (via unary operator), `123`
- `bool` - `true`, `false`
- `string` - e.g.: `"Hello World!"`, `"CC-Points:  110/100."`

In addition, the language supports the definition of classes, which can be instantiated via the `new`[10] operator. Moreover, `Jova` supports the null type via the expression `nix`.
Variables and fields can be associated with a type via a declaration. Furthermore, every expression (`expr`) has a type which depends on the type(s) of its components.

Scoping for Method and Inner Blocks

Similar to classes each method block has its own scope, which can contain local variable declarations respectively. Note that, local variables include method parameters in this document.
Report a `IDDoubleDeclError` if a local variable is declared more than once in the method's scope. Furthermore, report a `IDUnknownError` if a name (class, field, or variable) is used, which is not declared or visible in the current method's scope and a `MethodUnknownError` if a method is invoked which has not been defined in the current class scope.
Note that the inheritance rules, which are defined in Task 1.2, also apply here and have to be checked accordingly in case fields or methods are accessed.

Listing 13: Examples - ID double declaration errors

```
1   ...
2   int cookBurger() {
3      int patties;
4      bool patties;
5    //      ^ line 4 - Double declaration of identifier: 'patties'.
6      ...
7   }
8   ...
9    int cookFries() {
10     undefVar = 41;
11   //^ line 10 - Usage of unknown identifier: 'undefVar'.
12     ...
13   }
14   ...
15   int cookPizza() {
16     undefMethod();
17   //^ line 16 - Call to unknown method: 'undefMethod()'.
18     ...
19  }
20  ...
```

`if` and `while` blocks do **not** have their own scope in `Jova`. If a variable declaration is found inside an `if` or `while` block, report a `CannotDeclVarError` at the position of that declaration.

[10] The `new` operator calls the default constructor of the respective class.

Listing 14: Example - `if`- and `while`-Blocks

```
1     ...
2     if(mini) {
3       bool wumbo;
4         //  ^ line 3 - Cannot declare variables at this position.
5       ...
```

Due to the structure of the `Jova` grammar it is possible that an arbitrary expression (`expr`) can be used as statement in a method body. If this is the case, report an `AssignmentExpectedError` at the position of the respective expression.

Listing 15: Example - Illegal Method Body Expressions

```
1     ...
2     41;
3     //^ line 2 - Assignment or method call expected.
4     a = 42;
5     ...
```

One exception is the Jova Print Sequence (see Task 3), which is allowed as a special expression without an assignment.

Shadowing

Declarations of local variables are "shadowing" field declarations with the same name. In such a case, no error is reported. Furthermore, a shadowed field can be accessed via the `this` keyword in the respective class scope. With `this` the current object can be accessed. Thus, the type of `this` is current class.

Listing 16: Examples - Shadowing

```
1 Krusty {
2   int krab;
3   string bob() {
4     string krab; // no error here...
5     int i;
6     i = this.krab; // this.krab accesses the field 'krab'
7     return krab;
8   }
9 }
```

Type Checking

During type checking you will have to check whether every expression or statement conform to the rules described in this subsection.

Integer Boundaries

Check whether used integer constants are between $-2,147,483,648$ and $2,147,483,647$ (inclusive), so that they can be stored in a 4 byte memory location on the JVM. In case a too larger or too small integer number is used, report a `IntegerSizeError` at the position of the respective expression.

Listing 17: Example - Integer Boundary

```
1     ...
2     a = 2147483648;
3     //  ^ line 2 - Integer number too large.
4     ...
```

Integer and Boolean Operators

Table 1 shows the accepted types for the operands as well as the resulting type (i.e., the type, the respective operator expression evaluates to) of the different operations. If an operator is applied to incompatible operand types, a `OperatorTypeError` is reported at the position of the respective operator.

Operator	Operand Type	Result Type
+, -, *, /, %	int	int
Unary +, -	int	int
>, <	int	bool
&&, ||	bool	bool
!	bool	bool

Table 1: Operator Type Rules

Listing 18: Examples - Invalid operand types

```
1  ...
2  a = 1 + true;
3  //     ^ line 2 - Invalid operand type(s) for operator '+'.
4  b = !"hello";
5  //  ^ line 4 - Invalid operand type(s) for operator '!'.
6  ...
```

Equality Operators

The equality operators == and != can be applied to 2 operands of the same built-in type (`int`, `bool`, `string`) to compare them.
In case one of the operands is a class type, the other can either the same (sub-)class type or `nix`.
If non of these combinations apply, a `OperatorTypeError` is reported.
The result type of these operations is `bool`.

Listing 19: Examples - Invalid ==-operator usage

```
1  ...
2  if(1 == "hello")
3  //   ^ line 2 - Invalid operand type(s) for operator '=='.
4  ...
5  MyClass clazz;
6  my_bool = clazz != 42;
7  //              ^ line 6 - Invalid operand type(s) for operator '!='.
8  ...
```

Dot Operator

The dot operator `.` can be apply to a left-hand side operand of a class type to access a field or call a method of that class instance. The result type of the operation is the type of the accessed field or method, i.e., the type of the right-hand side operand.
If the left-hand side type does not have the field or method specified on the right-hand side either a `FieldUnknownError` or `MemberFunctionUnknownError` is reported.

Listing 20: Examples - Invalid .-operator usage

```
1  ...
2  a = 1.abc;
3  //     ^ line 2 - 'int' does not have field 'abc'.
4  ...
```

```
5    b = pizza.pineapple;
6    //        ^ line 5 - 'Pizza' does not have field 'pineapple'.
7    ...
8    c = plankton.getFormular();
9    //           ^ line 8 - 'DrPeterLankton' does not have method
10   //                      'getFormular()'.
11   ...
```

Due to the `Jova` syntax the right-hand side of the `.`-operator can be an arbitrary expression. Check whether the right-hand side operand of a dot-operator `.` is indeed either a field (`ID`) or a method call. If this is not the case, report a `MemberExpectedError` at the position of the operand.

Listing 21: Example - Dot-Operator

```
1    ...
2    half_life = gaben.game;
3    half_life.play();
4    half_life.3;
5    //        ^ line 4 - Field or method call expected.
6    half_life.!fun;
7    //        ^ line 6 - Field or method call expected.
8    ...
```

Assignment Operator

The assignment operator can be used to assign a value (right-hand side operand) to a variable or field (left-hand side operand). The result type is the type of the left-hand side operand.
In case the left-hand side operand is of a built-in type, the right-hand side operand has to be of the same type. Otherwise an `OperatorTypeError` is reported.
In case the type of the left-hand side operand is a class, the type of the right-hand side has to be of the same (sub-)class type, or `nix`. Otherwise a `OperatorTypeError` is reported.

Listing 22: Examples - Invalid assignment operator usages

```
1   ...
2   int i;
3   i = "42";
4  //  ^ line 3 - Invalid operand type(s) for operator '='.
5   ...
6   ClassA a;
7   ClassB b;  // not a subclass of ClassA
8   a = b;
9  //  ^ line 8 - Invalid operand type(s) for operator '='.
10  ...
```

Due to the `Jova` syntax it is possible that the left-hand side operand of an assignment can be something different than a variable or field. Check whether the left-hand side operand of an assignment operator is indeed a variable or a field. If this is not the case, report a `VariableExpectedError` at the position of the left-hand side expression.

Listing 23: Example - Invalid left-hand side of assignment

```
1    ...
2    a = 4;
3    b.c = 2;
4    1 = 2;
5   //^ line 4 - Variable or field expected for assignment.
6    ...
```

Conditions

A condition for an `if` or `while` statement is expected to be of type `bool`. Otherwise an `ConditionTypeError` is reported.

Listing 24: Example - Invalid condition

```
1  ...
2  if("true") {
3  // ^ line 2 - Incompatible type for condition.
4  ...
```

Return Statement

The type of the expression in a `return` statement has to match the return type of the current method. In case the return type is a class type, the same class type, a subtype of the class, or `nix` can be return. If the types do not match a `ReturnTypeError` is reported.

Listing 25: Example - Invalid return

```
1  ...
2  int cookBurger() {
3    return "Krabby Patty";
4 //          ^ line 3 - Incompatible type for return.
5  }
6  ...
```

Note: Due to the `Jova` grammar it is possible that a method body is defined without a `return` statement. If you do not plan on implementing the Return Statement Bonus Task, you can assume that every method has a `return` statement as last effective statement in a method body.

Main Method

A method declared as `int main()` can be used as entry point for the execution of a `Jova` program on the JVM. In order to achieve this you will translate this method to `public static void main(String[])` in Task 3. Since `Jova` does not support the `static` modifier, a `main` method is implicitly static. Thus, no fields or methods of the current class can be accessed without an instance. Furthermore, the `this` reference cannot be used.
Check whether a field/method is directly accessed or `this` is being used in a `main` method and report a `MainError` in such a case

Listing 26: Eample - Main Method

```
1   ...
2   int customers;
3   string cookBurger() { return "Krabby Patty"; }
4   int main() {
5      print(customers);
6  //         ^ line 5 - Cannot access a member or 'this' in main method.
7      cookBurger();
8  //  ^ line 7 - Cannot access a member or 'this' in main method.
9      this.cookBurger();
10 //  ^ line 9 - Cannot access a member or 'this' in main method.
11     ...
12  }
13  ...
```

Error Reporting for Expressions

Errors in expressions are expected to be reported in the order of evaluation (i.e., from left to right) in accordance to the operator precedence. Assignments are checked from right to left.

If you have found an error in a compound expression, you can skip the check for the remaining parts of the expression.
Otherwise the error reporting rules from Task 1.2 apply.

Tasks

1. Implement the following methods of the class `Jovac.class` in the package `at.tugraz.ist.cc` in the `src` directory:

   - `public void task2(String file_path)`
     Invoke the implementation of Task 1 on the input file indicated by `file_path`. In case any errors are found during the checks performed in Task 1, do not perform any type checking and abort the compilation process via returning from this method.
     If no errors where found, traverse the resulting parse tree again to perform the remaining semantic analysis steps on the `Jova` program.
     Check whether the input program conforms to the semantic rules of the `Jova` language which are explained above. Report any semantic errors or warnings that you find via instantiating the appropriate `SemanticError` or `JovaWarning` classes and save/add them in the expected order.
     Write a summary of all errors and warnings found during the whole analysis to `stdout`, including the line number (and in case of warnings character position). You can use the methods of the provided `JovaErrorPrinter` class to print the expected error summary.

2. Adapt the `README` file as described above.

3. *Optional*: You may implement one or more of the bonus tasks listed in the following subsection. In case you implement bonus tasks you must provide a detailed description about your solutions (explain how you implemented the bonus task, provide examples and list the names of the corresponding test cases) and **adapt** the `README` file accordingly in order to obtain points.

Notes and Hints

- Before you start: Fix any remaining errors (if you had any) from Task 1!
- Extend the symbol table data structures to hold the type information for local identifiers or symbols. One possible design approach would be to construct a symbol table in accordance to the structure of the scopes of the different blocks in a `Jova` program.
- You may need to traverse the syntax tree (or specific parts of it) several times to perform the wanted analysis.
- If you are unsure which error class you have to instantiate or which arguments you have to pass to a constructor for an error, check the provided examples in the assignment sheet, the comments in the provided error classes, or the reference output in the framework. If you are still unsure, feel free to ask in the forum.
- Make sure to write your own test cases.
- For Task 3 it will be essential that your compiler is capable to identify valid programs, i.e., make sure that your compiler does not identify type checking errors where there are none. Especially the built-in `print`-functions as well as the `main` method will be crucial for achieving points in Task 3.
- For the semantic errors and warnings make sure that the **first** error in the input file is correctly reported.

Bonus Tasks

Return Statement (3P) [id: return statement]

Check whether the last statement in a method body is indeed a return statement. Alternatively, the last statement can also simply be an arbitrary expression which has to conform to the type rules of the expression in a return statement.
Report a `MissingReturnError.java` at the position of the closing bracket of the method body, if the last element in a method body is not a return-statement or expression. If the last element is an `if-then-else`-statement, apply this check recursively to its blocks.

Listing 27: Example - Expression as return statement bonus

```
1  ...
2  int cookBurger() {
3    42;              // valid return
4  }
5  ...
6  int fries() {
7    false;
8  //^ line 7 - Incompatible type for return.
9  }
10 ...
```

Listing 28: Example - Return Statement Bonus

```
1  ...
2  int cookBurger() {
3    int a;
4  }
5 //^ line 4 - Missing return statement.
6  ...
7  int fries() {
8    if(b) {
9      while(b) {    // last statement in this branch is 'while'
10       b = b - 1;
11     }
12   } else {
13     42;
14   }
15 }
16 //^ line 15 - Missing return statement.
17 ...
```

Type Coercions (3P) [id: type coercions]

Implement implicit type conversion of `int` operands for all boolean operators (`&&`, `||`, `!`) and conditions, i.e., expressions of type `int` can be used as operands (or condition) for boolean operators. In addition, it should also be possible to assign an `int` value to a `bool` variable or return an `int` in a method with type `bool`. An `int` expression is internally converted to type `bool` to make it compatible with the respective operator. Report a `CoercionWarning` in case an `int` expression is converted to type `bool`. The integer `0` represent `false`; every other integer `true`.

Listing 29: Example - Type Coercion Bonus

```
1   ...
2   bool isFancy() {
3     bool pinkyUp, hands_down;
4     pinkyUp = 1;
5     //         ^ 4:14 - 'int' coerced to 'bool'.
6     hands_down = 4 && 2;
7     //            ^ 6:17 - 'int' coerced to 'bool'.
8     //                 ^ 6:22 - 'int' coerced to 'bool'.
9   }
10  ...
```

Operator Overloading (1P) `[id: operator overloading]`

Implement operator overloading for the + operator to concatenate strings, by allowing `string` operands to be also accepted by this operation. In case one of the operands is of type `string`, the result type of this operation is changed to `string`. Thus, if one operand is of type `string` and the other of type `int`, the `int` operand is coerced to `string` and a `CoercionWarning` is reported at the position of the `int` operand.

Listing 30: Example - Operator Overloading Bonus

```
1   ...
2   string result;
3   result = "Points: " + 120;
4 //                       ^ 3:24 - 'int' coerced to 'string'.
5   ...
```

3 Code Generation

In the final task you will generate Java bytecode instructions from a `Jova` input program. To construct the executable `.class`-files we will utilize the Jasmin assembler.

Jasmin

Jasmin is a Java bytecode assembler which translates `.j` files to executable `.class` files. A `.j` file contains a description of a single class and its methods, which include listings of Java bytecode instructions. A small tutorial, which includes a description of how Jasmin operates and the different directives you will need, as well as additional helpful links are available on the TeachCenter.
You can use the provided `jasmin.jar` executable in the `libs` directory of the framework:

Convert a Jasmin `*.j` file into `*.class Java` bytecode:
`java -jar /path/to/jasmin.jar <ClassName>.j`
e.g. `java -jar lib/jasmin.jar MyClass.j`
Execute the created `ClassName.class` file:
`java -cp /path/to/source <ClassName>`
e.g. `java -cp outputfolder MyClass`

Alternatively you can also use the `jasmin` or `jasminAll` gradle tasks.

`.j`-File Structure

Note: For a more in-depth explanation of a Jasmin assembly as well as the needed JVM Bytecode instructions for the expected translations you have to perform, please see the Task 3 tutorial and the additional resources.

Every `.j`-file starts with the `.source`, `.class` and `.super` directives, which define the source from which the assembly was created, the name of the class and the fully qualified name of its superclass. These directives are followed by the classes fields and methods (including the special `<init>` constructor methods), which are defined via `.field` and `.method` respective.
Every method has 2 data structures to perform operations on: the *locals array* and the *operand stack*. The size of these data structures has to be set to the *minimum needed value* using the `.limit` directive for every method.
After these Jasmin directives the bytecode instructions are listed. These instructions represent the correctly translated statements and expressions of the respective `Jova` method body. You can find all relevant instructions you will need for the translation in the Task 3 tutorial and the additional resources.
A Jasmin method descriptions ends with the `.end method` directive.

Minimum Requirements

To achieve any points on Task 3 your compiler has to create a working *base translation* of a `Jova` program. The following points have to be correctly implemented to achieve any points on Task 3, i.e., if 1 or more of these points do not work correctly you may achieve 0 points even though you have implemented other functionalities correctly.
In a nutshell, a fully compiled `Jova` program must be placed in a defined location and be executable on the JVM by invoking the `main`-method on the expected class. Furthermore, a compiled program must be able to print correct output to `stdout` via the print-functions for us to check whether it works as expected.

Output Requirements

For each class in your `Jova` program your compiler has to create a single `.j` file. These files have to be placed in a folder specified by the `out_path` parameter of the `generateCode(..)`-method.

If the path to the output folder does not exist your compiler has to make the entire directory path.

Compilation Example

Assuming the `generateCode(..)`-method is called with the following arguments:
`file_path` = `"test/input/simple.jova"`
`out_path` = `"output/jasmin"`

The compiler creates the directory-path (if non-existent) `"output/jasmin"`. Subsequently, it creates the `"output/jasmin/Simple.j"` and `"output/jasmin/Test.j"` files for the following input file:

Listing 31: Example Input File: `simple.jova`

```
1  Simple {
2    int simple() {
3     return 4;
4    }
5  }
6
7  Test {
8    int main() {
9      return 2;
10   }
11 }
```

Listing 32: Example Output File 1: `output/jasmin/Simple.j`

```
1  .source simple.jova
2  .class public Simple
3  .super java/lang/Object
4
5  .method public <init>()V
6    aload_0
7    invokespecial java/lang/Object/<init>()V
8    return
9  .end method
10
11 .method public simple()I
12 .limit stack 1
13 .limit locals 1
14   iconst_4
15   ireturn
16 .end method
```

Listing 33: Example Output File 2: `output/jasmin/Test.j`

```
1  .source simple.jova
2  .class public Test
3  .super java/lang/Object
4
5  .method public <init>()V
6    aload_0
7    invokespecial java/lang/Object/<init>()V
8    return
9  .end method
10
11 .method public static main([Ljava/lang/String;)V
12 .limit stack 0
13 .limit locals 1
```

```
14    return
15  .end method
```

Main Method

A specific `main`-method has to be provided for a compiled `Jova` program to be executed on the JVM. Translate the following `Jova` method to the correct `main`-method in your Jasmin code:

`int main()` ⇒ `public static main([Ljava/lang/String;)V`

Note that the translated `main`-method is of type `void`, i.e., you have to use the appropriate `return` instruction to return from the `main`-method without a return value.

Make sure that this program entry point is translated correctly, so that a compiled `Jova` program can be executed on the JVM.

Print

Calls to the built-in `print` functions are translated to the following equivalents of the `System/out` object.

- `int print(int)` ⇒ `print(I)V`
- `int print(bool)` ⇒ `print(Z)V`
- `int print(string)` ⇒ `print(Ljava/lang/String;)V`

All calls will need to be invoked using the `invokevirtual` instruction after pushing a reference to `java/lang/System/out` and the desired parameter onto the stack.
As the translated calls do not provide a return value you will need to provide a workaround in your assembly which supports each call to this function with a default return value of `0`.

Make sure all `print` methods **work correctly** as they are used primarily for testing your assignment.

Additional Features

Read

Calls to the predefined `read` functions are translated to calls of the following `java/util/Scanner` methods:

- `int readInt()` ⇒ `nextInt()I`
- `string readLine()` ⇒ `nextLine()Ljava/lang/String;`

A `Scanner` object has to be instantiated with `java/lang/System/in` set as input stream. This has to be done *once* for every method in which a `read`-function is used. Save this object in the respective local array and access the *same* instance for every call to a `read`-function in the current method.

Ignoring Return Values

The returned value of a method call is popped off the stack in case a method-call is used as statement in `Jova`, i.e., is not part of another statement or expression, to prevent a stack overflow.

Listing 34: Example: Ignoring the return value of a method

```
1  ...
2  krab();  // defined as int krab() {}
3  ...
```

Listing 35: Example Translation: Ignoring the return value of a method

```
1  ...
2  invokevirtual Krusty/krab()I
3  pop
4  ...
```

Print Sequence

In `Jova` it is possible to combine several calls to the built-in `print`-functions via the +-operator as a single statement. A print sequence is translated to a sequence of calls to the respective print methods. No return values are pushed and no additions are performed for this special statement:

Listing 36: Print sequence example in Jova

```
1  ...
2  print(true) + print(2) + print ("drei");
3  ...
```

Listing 37: Print sequence example translation

```
1  ...
2  getstatic java/lang/System/out Ljava/io/PrintStream;
3  iconst_1
4  invokevirtual java/io/PrintStream/print(Z)V
5
6  getstatic java/lang/System/out Ljava/io/PrintStream;
7  iconst_2
8  invokevirtual java/io/PrintStream/print(I)V
9
10 getstatic java/lang/System/out Ljava/io/PrintStream;
11 ldc "drei"
12 invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
13 ...
```

Note: If this construct is used as an expression, it should be translated to an normal addition of method calls which results in `0` if executed.

Other Expressions, Statements and Jova Constructs

Translate all other `Jova` expression, statements and constructs to the appropriate sequence of Java bytecode instructions and Jasmin assembly directives. You can find a selection of the necessary instructions in the Task 3 tutorial slides. A link to a complete list of all available instructions can be found on the TeachCenter.
In addition, you can find sample translations in the reference output for the Task 3 public tests. However, these translations are only meant as reference point for one possible translation, i.e., your compiler does not have to generate the exact same instruction or sequence of instructions for a given `Jova` code.

Tasks

1. Implement the following method of the class `Jovac.class` in the package `at.tugraz.ist.cc` in the `src` directory:

   - `public void task3 (String file_path, String out_path)`
     Invoke the lexer, parser and type checker you have implemented in the previous tasks on the input file indicated by `file_path`. In case any errors are found, report and print these errors as specified in the respective task, do **not** perform any code generation and abort the compilation process via returning from this method.

If no compilation errors were found, create the directory path specified by `out_path` (if it does not already exist).
Generate a <`ClassName`>`.j`-file for every defined class in the `Jova` input program, with <`ClassName`> being the name of the corresponding `Jova` class, in the out-path. Add the translated Jasmin code for the respective class to the `.j`-file as specified in the subsections above. You can use another visitor/listener implementation and traverse the generated syntax tree again to generate the Jasmin code or use your own intermediate representation.

2. Adapt the `README` file as described above.

3. *Optional*: You may implement one or more of the bonus tasks listed in the following subsection. In case you implement bonus tasks you must provide a detailed description about your solutions (explain how you implemented the bonus task, provide examples and list the names of the corresponding test cases) and **adapt** the `README` file accordingly in order to obtain points.

Notes and Hints

- You may need to create a mapping from used `Jova` variables to the appropriate local array indices in a method. Use the symbol table or create a separate data structure to keep track of that mapping.
- You may need information about which class a field or method has been defined in to supply the correct arguments to the respective sequence of "member-access" instructions.
- Comments within your assembly relating the assembly statements to the `Jova` statement can be useful for debugging.
- Information like the currently processed class/method may need to be accessible for the translation.
- Jasmin does not directly provide relational operators like > or ==. Instead you will have to utilize conditional jump statements such as `if_icmpgt` <`label_name`> and `if_icmpeq` <`label_name`> in combination with custom labels and commands to put the desired outcome onto the operand stack.
- A compiled `Jova` program has to be executable by running a `.class` file containing the translated `int main()`-method on the JVM.
- Make sure that the `print`-methods work correctly as these are used primarily for testing of the assignment.

Bonus Tasks

Exit Code (1P) `[id: exit code]`

In case the `Jova main`-functions returns a non-zero exit code, translate the `return` statement in the `public int main()` function to a call to `java/lang/System/exit(I)V`.

Operator Overloading (1+1P) `[id: operator overloading 2]`

Continue the implementation of the Operator Overloading Task and translate the +-operation with strings to an equivalent set of Bytecode instructions. *Hint:* Utilize the `StringBuilder` class. (1P)
In addition, translate the comparison operations (== and !=) for strings to calls to the `equals`-method of the `String` class. (1P)

Return (1P) [id: return 2]

Continue the implementation of the Return Statement Task and translate an expression at the end of a method body to the correct set of instructions to return that value.

User Defined Constructors (1+3P) [id: user defined constructors]

Translate a Jova method with the following type and signature to the special <init> constructor methods in Jasmin to allow the usage of user defined-default constructors for Jova classes (1P):

`int init()` ⇒ `public <init>()V`

In addition, add the following small extension to the new-expression in your ANTLR Grammar to allow the usage of parameterized constructors in Jova via an init-method:

```
expr  → ...
      | KEY_NEW CLASS_ID ( ( ( expr ( , expr )* )?  ) )?
      | ...
```

Listing 38: Example - Constructors

```
1  MyClass {
2    int init() {
3     ...
4    }
5    int init(int a, bool b, string c) {
6     ...
7    }
8  }
9  ...
10   MyClass m1,m2,m3;
11   m1 = new MyClass;   // calls the user-defined default ctor in line 2
12   m2 = new MyClass(); // calls the user-defined default ctor in line 2
13   m3 = new MyClass(1,true,"drei"); // calls custom ctor in line 5
14 ...
```

Return statements inside the constructor's body are skipped during the code generation of a constructor's body.

Warning: Make sure that the default constructor can still be called via the default syntax of the new-expression, i.e., `new ClassName;`, otherwise you risk losing points! (3P)

Appendix

A Jova Grammar

Terminals are displayed in blue, token classes in uppercase and parser rules in lower case letters. (Note that **nix** is the Styrian equivalent to `null`.)

Keywords

KEY_IF	→	**if**
KEY_ELSE	→	**else**
KEY_WHILE	→	**while**
KEY_RETURN	→	**return**
KEY_INT	→	**int**
KEY_BOOL	→	**bool**
KEY_STRING	→	**string**
KEY_NIX	→	**nix**
KEY_THIS	→	**this**
KEY_NEW	→	**new**

Operators

ADDOP	→	+ | -
ASSIGN	→	=
RELOP	→	< | > | == | !=
MULOP	→	* | / | %
OR	→	||
AND	→	&&
NOT	→	**!**
DOT	→	.

Numerical literals

INT	→	**TODO: Define an integer number (see Task 1.1.1)**
fragment DIGIT0	→	[**0**-**9**]
fragment DIGIT	→	[**1**-**9**]

Boolean literals

BOOL	→	**true** | **false**

Identifiers

CLASS_ID	→	UPPERCASE (LETTER | DIGIT0 | _)*
ID	→	LOWERCASE (LETTER | DIGIT0 | _)*
fragment LETTER	→	(LOWERCASE | UPPERCASE)
fragment LOWERCASE	→	[**a**-**z**]
fragment UPPERCASE	→	[**A**-**Z**]

White spaces

WS	→	[_ | **\n** | **\t** | **\r**]

Comment

COMMENT → // -[\n | \r]*

String literals

STRING → " (-[\n | \r | \\ | "] | ESC_SEQ)* "
fragment ESC_SEQ → \\ [n | r | t | " | b | f | ' | \\]

Syntax

program → class_decl+
class_decl → CLASS_ID (: CLASS_ID)? { class_body }
class_body → (decl | method)*
method → type ID (param_list?) block
type → KEY_INT | KEY_BOOL | KEY_STRING | CLASS_ID
param_list → **TODO: create this rule (see Task 1.1.2)**
block → { (decl | if_stmt | while_stmt | return_stmt | expr ;)* }
decl → **TODO: create this rule (see Task 1.1.3)**
if_stmt → KEY_IF (expr) block (KEY_ELSE block)?
while_stmt → KEY_WHILE (expr) block
return_stmt → KEY_RETURN expr ;
expr → literal
| id_expr
| KEY_NEW CLASS_ID
| (expr)
| **TODO: complete this rule (see Task 1.1.4)**
id_expr → ID
| ID (**TODO: complete this rule (see Task 1.1.5)**)
literal → INT | BOOL | STRING | KEY_NIX | KEY_THIS