National University of Singapore
CS4212 Project Assignment 2
AY 2021/2022 Semester 1
Due Date: Sunday, 17th October 2021 (23:59 Hrs)

This assignment aims to complete the construction of the front end of a JLite compiler. Extend the parser you constructed in Assignment 1 with the following:

1. Semantic analysis to ensure that input JLite programs are semantically sound.
2. Intermediate code generation for JLite programs.

Contact the instructor if you have no confidence in your Assignment 1 parser and/or lexer.

1 Your Tasks

Given a JLite program as input, your developed system is required to generate two pieces of information:

1. **Static Checker**: Your code will perform static semantics checking of JLite programs and report any encountered errors.
2. **Intermediate Code Generator**: Your code will generate IR3-code for those JLite programs which successfully pass your *static checker* tests.
3. **Submission**: You are to submit your entire code in a self-contained ZIP archive, with your name and student ID as part of the ZIP's filename. Your code should work with Python 3.8.10 without requiring any changes or additional package installations.

1.1 Static Checker

The *Static Checker* should assume that JLite follows the **static scope** discipline, and perform the following static checks on JLite programs:

1. **Distinct-Name Checking**: This ensures that:
   (a) No two fields in a class can have the same name.
   (b) No two classes can be declared in a program with the same (class) name.
   (c) No two parameters in a method declaration can have the same name.
   (d) No two methods within a class declaration can have the same name.

   > You have *the option for allowing* **overloaded methods** *to occur in a class declaration.* You will however need to accompany your code with an explanation describing how you handle overloaded methods. There is a **bonus of 5 marks** for correct handling of overloaded method declaration, *and clear description of how you handle it*.

2. **Type Checking**: This makes sure that uses of program constructs conform to how the constructs have been declared. The rules defining the soundness of type checking is described by a set of typing rules, which is presented in Appendix A.

For submitting your code for assessment, your static checker needs to:

1. Produce syntax trees – for JLite programs that pass your checker – for use by the intermediate code generator;
2. Report, as clearly as possible, any compilation errors arisen due to semantic errors in the JLite programs. (There is no need for you to modify the existing parser to handle syntactic errors.);
3. Produce necessary type information required for use by intermediate code generation as well as the backend of the compiler.

Relating to the last point above, you can either produce type information as a separate data structure, or you can attach this information to the parse tree.

1.2 IR Generator

The second program you are going to construct, called "IR3-Gen", takes as input the (abstract) syntax tree that has passed static checking and that contains type information. IR3-Gen aims to generate three-address code of the form called IR3.

We normally think of the code generated to be a string of characters. However, in this project, your IR3-Gen does not directly generate the character string representing the intermediate code. Rather, it generates a data structure as specified by the IR3 grammar, as shown in Appendix B, with examples in Appendix C. As a data structure, we can further take it as input and transform it to some low-level machine code (as will be done in Assignment 3). Generating code from the IR3 data should not be difficult – you just need to use alter the pretty-printing functions you did for Assignment 1.

2 Deliverable

Testing Your Programs

We have provided some JLite programs for you to do your preliminary testing. Nevertheless, you are required to create *at least six* sample programs to test your product. Appendix C provides a sample JLite program and the corresponding IR3 code.

Submission of your product

Please submit a ZIP archive, with your name and student ID as part of the ZIP's filename, containing the following files to `LumiNUS CS4212` website under the "`Programming Assignment 2 Submissions`" folder.

1. The ZIP file should at the top level contain a single folder bearing your name and student ID. In that folder should be your complete project sources, including test files and expected output for these tests.
2. We will test your submission using Python 3.8.10, so you should ensure your code works with this version without requiring any changes, or additional package installations. Your sources should include the following files:

(a) The files from the previous assignment: `lex.py` and `parse.py`;

(b) `ast.py` – class definitions for the abstract syntax tree;

(c) `ir3.py` – class definitions for the intermediate code (IR3) tree;

(d) You may organise the static checker and the IR3 generator however you wish (describe your choice in the README), but you *must* have a file `gen.py` with the following behaviour.

Invoking `./gen.py program.j` should (a) print the errors returned by your static checker (if such errors exist) or (b) pretty-print the IR3 for the given program (if the program is semantically valid).

3. A subfolder called `test/` within the main folder. This should include at least 6 pairs of files—*created by you*—to test your code, each pair consisting of the input program `TESTNAME.j` and the expected (correct) output `TESTNAME.j.gold`.

   For programs that do not pass the static checks, the `.gold` file should contain the error messages reported by your compiler, and for programs that do pass the static checks, it should contain the pretty-printed IR3 code.

   You can either delete your parser tests (from Assignment 1) from the `test/` folder or place them in a `test/parse` folder.

4. A document named `[YourFullName]_pa2_readme.md` within the main folder describing the design of your compiler, the file structure of your submission, and any important information you would like to share with us and which you think might earn you more marks (e.g., an explanation of how you implemented method overloading or how you have insightful error messages).

   Marks will be deducted if you do not fully identify yourself in the submission.

A Typing Rules for JLITE Programs

A.1 Class Descriptor, Type Environment and Legitimate Types

The *type environment* comprises two entities: *class descriptor* $\mathcal{C}$ and *local environment* Γ.

1. *Class Descriptor* $\mathcal{C}$ maps a class name to a triple consisting of: its class name, the types of each of its field declarations, and the signatures of each of its method declarations.

$$\begin{array}{rcl} \mathcal{C} & :: & \langle \mathit{cname} \rangle \ \rightarrow \ (\langle \mathit{fds} \rangle \times \langle \mathit{msigs} \rangle) \\ \langle \mathit{fds} \rangle & = & \langle \mathit{id} \rangle \ \rightarrow \ T \\ \langle \mathit{msigs} \rangle & = & \langle \mathit{id} \rangle \ \rightarrow \ ([T] \times T) \end{array}$$

2. *Local environment* Γ maps variables declared locally to their types.

$$\Gamma \quad :: \quad \langle \mathit{id} \rangle \ \rightarrow \ T$$

The type class T is declared as follows:

$$T \ = \ \langle \mathit{cname} \rangle \cup \{int, bool, string, void\}.$$

where $\langle \mathit{cname} \rangle$ ranges over all class names declared.

We begin type checking of a JLITE program P by filling in the class descriptor with all type information declared in the program; we call this process *initialize*(P). This includes class names, class attribute declarations and method signatures occurred in each class. On the other hand, the initial local environment contains empty information (or it may contain some global type information, such as the types of the primitive operations).

In addition, we assume that all types *referred to* in the program must be *legitimate*; That is, they can be found in the type class T; failing so will lead to type errors at compile time.

A.2 Type Checking A JLITE Program

A JLITE program is well typed if the type checker returns `isOK` upon return.

$$\frac{\begin{array}{c} P \ = \ \mathit{mainC} \ C_1 \cdots C_n \qquad \Gamma = [\,] \\ \mathcal{C} = \mathit{initialize}(P) \qquad \langle \mathcal{C}, \Gamma \rangle \ \vdash \ \mathit{mainC} \ \texttt{isOK} \\ \langle \mathcal{C}, \Gamma \rangle \ \vdash \ C_i \ \texttt{isOK} \quad \forall i \in \{1, \ldots, n\} \end{array}}{\vdash \ \mathit{mainC} \ C_1 \cdots C_n \ \texttt{isOK}} \quad \text{[JLITE-Pgm]}$$

A.3 Type Checking Classes

Before type checking all the methods declared in a class, the local environment is set to include respectively the types/signatures associated with the attributes/methods declared in the class, and the types for the special identifiers $this$.

$$\frac{\begin{array}{c}((a_1 \mapsto t_1, \ldots, a_n \mapsto t_n), (ms_1, \ldots, ms_k)) \;=\; \mathcal{C}(c) \\ ms_i = (m_i \mapsto ([t_{i1}; \ldots; t_{ip}],\ tr_i)) \quad \forall i \in \{1, \ldots, k\} \\ \Gamma = [this \mapsto c, a_n \mapsto t_n, \ldots, a_1 \mapsto t_1, ms_k, \ldots, ms_1] \\ \langle \mathcal{C}, \Gamma \rangle \;\vdash\; md_i \;\texttt{isOK} \quad \forall i \in \{1, \ldots, k\}\end{array}}{\langle \mathcal{C}, \_ \rangle \;\vdash\; \texttt{class}\; c\; \texttt{\{}t_1\ a_1; \ldots; t_n\ a_n;\ md_1; \ldots, md_k\ \texttt{\}}\; \texttt{isOK}} \quad \text{[CDecl]}$$

A.4 Environment Operations

We define here several notations describing the operations on a local environment, Γ.

- *Environment Lookup*: Given $\Gamma \;=\; [(v_1, t_1); \ldots; (v_n, t_n)]$, looking up the type of a variable/attribute x from Γ is defined by:

$$\begin{array}{rcll} \Gamma(x) & = & \texttt{match}\ \Gamma\ \texttt{with} & \\ & & \mid\ ((v, t) :: \Gamma') & \rightarrow\ \texttt{if}\ (v = x)\ \texttt{then Some}\ t \\ & & & \quad \texttt{else}\ \Gamma'(x) \\ & & \mid\ [\,] & \rightarrow\ \texttt{None} \end{array}$$

- *Environment Augmentation*: When we wish to add more associate pairs into the existing environment Γ, we always add them to the front of the association list. Given $\Gamma \;=\; [(v_1, t_1); \ldots; (v_n, t_n)]$, we have

$$\begin{array}{rcl} \Gamma[\,] & = & \Gamma \\ \Gamma[v'_1 \mapsto t'_1] & = & \Gamma'\ \texttt{where} \\ & & \Gamma' \;=\; [(v'_1, t'_1); (v_1, t_1); \ldots; (v_n, t_n)] \\ \Gamma[v'_1 \mapsto t'_1, v'_2 \mapsto t'_2, \ldots, v'_m \mapsto t'_m] & = & (((\Gamma[v'_1 \mapsto t'_1])[v'_2 \mapsto t'_2]) \cdots)[v'_m \mapsto t'_m] \end{array}$$

A.5 Type Checking Method Declaration

When type checking a method declaration, the local environment is augmented with the types of the parameters declared in the method, **and the return type** of the method, associated with and unique special identifier Ret.

$$\frac{\begin{array}{c}\texttt{Some}\ ([t_1; \cdots; t_n],\ t_0) \;=\; \Gamma(m) \\ \langle \mathcal{C}, \Gamma[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n, Ret \mapsto t_0] \rangle \;\vdash_S\; S \;:\; t \\ t \;=\; t_0\end{array}}{\langle \mathcal{C}, \Gamma \rangle \;\vdash\; t_0\ m\texttt{(}t_1\ v_1, \ldots, t_n\ v_n\texttt{)}\ S\ \texttt{isOK}} \quad \text{[MDecl]}$$

A.6 Type Checking Statements

Type checking of statements is specified by the judgment: $\langle \mathcal{C}, \Gamma \rangle \;\vdash_S\; S \;:\; t$.

$$\frac{}{\langle \mathcal{C}, \Gamma \rangle \;\vdash_S\; \varepsilon \;:\; void} \quad \text{[Empty statement]}$$

$$\frac{\langle \mathcal{C}, \Gamma[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n] \rangle \;\vdash_S\; S \;:\; t}{\langle \mathcal{C}, \Gamma \rangle \;\vdash_S\; \texttt{\{}\ t_1\ v_1; \ldots; t_n\ v_n;\ S\ \texttt{\}} \;:\; t} \quad \text{[Block]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \;\vdash_S\; S \;:\; t}{\langle \mathcal{C}, \Gamma \rangle \;\vdash_S\; S; \;:\; t} \quad \text{[Seq-base]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_S S_1 : t_1 \qquad \langle \mathcal{C}, \Gamma \rangle \vdash_S S_2 : t_2}{\langle \mathcal{C}, \Gamma \rangle \vdash_S S_1 ; S_2 : t_2} \quad \text{[Seq]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_E e : t_e \qquad \Gamma(v) = \texttt{Some}\, t_v \qquad t_e = t_v}{\langle \mathcal{C}, \Gamma \rangle \vdash_S v = e : \mathit{void}} \quad \text{[VarAss]}$$

$$\frac{\begin{array}{c}\langle \mathcal{C}, \Gamma \rangle \vdash_E e_1 : c_1 \qquad \langle \mathcal{C}, \Gamma \rangle \vdash_E e_2 : t_2 \\ \langle fds, \_ \rangle = \mathcal{C}(c_1) \qquad \texttt{Some}\, t_a = fds(a) \qquad t_a = t_2\end{array}}{\langle \mathcal{C}, \Gamma \rangle \vdash_S e_1.a = e_2 : \mathit{void}} \quad \text{[FdAss]}$$

$$\frac{\begin{array}{c}\langle \mathcal{C}, \Gamma \rangle \vdash_S S_1 : t_1 \qquad \langle \mathcal{C}, \Gamma \rangle \vdash_S S_2 : t_2 \\ \langle \mathcal{C}, \Gamma \rangle \vdash_E e : \mathit{bool} \qquad t_1 = t_2\end{array}}{\langle \mathcal{C}, \Gamma \rangle \vdash_S \texttt{if}\, e\, \texttt{then}\, S_1\, \texttt{else}\, S_2 : t_2} \quad \text{[Cond]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_E e : \mathit{bool} \qquad \langle \mathcal{C}, \Gamma \rangle \vdash_S S : t}{\langle \mathcal{C}, \Gamma \rangle \vdash_S \texttt{while}\, e\, \texttt{\{}\, S\, \texttt{\}} : t} \quad \text{[While]}$$

$$\frac{\Gamma(v) = \texttt{Some}\, t \quad t \in \{\mathit{int}, \mathit{bool}, \mathit{string}\}}{\langle \mathcal{C}, \Gamma \rangle \vdash_S \texttt{readln}\, v : \mathit{void}} \quad \text{[Read]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_E e : t \quad t \in \{\mathit{int}, \mathit{bool}, \mathit{string}\}}{\langle \mathcal{C}, \Gamma \rangle \vdash_S \texttt{println}\, e : \mathit{void}} \quad \text{[Print]}$$

$$\frac{\begin{array}{c}\langle \mathcal{C}, \Gamma \rangle \vdash_E e_i : t'_i \qquad \forall i \in \{1, \ldots, p\} \\ \texttt{Some}\, ([t_1; \ldots; t_p], t_r) = \langle \mathcal{C}, \Gamma \rangle(m) \\ t'_i = t_i \qquad \forall i \in \{1, \ldots, p\}\end{array}}{\langle \mathcal{C}, \Gamma \rangle \vdash_S m(e_1, \ldots, e_p) : t_r} \quad \text{[SLocalCall]}$$

$$\frac{\begin{array}{c}\langle \mathcal{C}, \Gamma \rangle \vdash_E e_i : t'_i \qquad \forall i \in \{1, \ldots, p\} \\ \langle \mathcal{C}, \Gamma \rangle \vdash_E e_0 : c_0 \quad \langle \_, \_, \Gamma_{MS} \rangle = \mathcal{C}(c_0) \\ \texttt{Some}\, ([t_1; \ldots; t_p], t_r) = \Gamma_{MS}(m) \\ t'_i = t_i \qquad \forall i \in \{1, \ldots, p\}\end{array}}{\langle \mathcal{C}, \Gamma \rangle \vdash_S e_0.m(e_1, \ldots, e_p) : t_r} \quad \text{[SGlobalCall]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_E e : t \qquad t' = \Gamma(\mathit{Ret}) \qquad t = t'}{\langle \mathcal{C}, \Gamma \rangle \vdash_S \texttt{return}\, e : t'} \quad \text{[Ret-T]}$$

$$\frac{\Gamma(\mathit{Ret}) = \mathit{void}}{\langle \mathcal{C}, \Gamma \rangle \vdash_S \texttt{return} : \mathit{void}} \quad \text{[Ret-Void]}$$

A.7 Type Checking Expressions

Type checking of expressions is specified by the judgment: $\langle \mathcal{C}, \Gamma \rangle \vdash_E e : t$.

$$\frac{v \in \langle \mathit{Id} \rangle \cup \{\texttt{this}\} \qquad \Gamma(v) = \texttt{Some}\ t}{\langle \mathcal{C}, \Gamma \rangle \vdash_E v : t} \quad \text{[Id]}$$

$$\frac{}{\langle \mathcal{C}, \Gamma \rangle \vdash_E \texttt{1} : \mathit{int}} \quad \text{[Integers]}$$

$$\frac{}{\langle \mathcal{C}, \Gamma \rangle \vdash_E \texttt{true} : \mathit{bool}} \quad \text{[Booleans]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_E e : c \qquad \langle\, \mathit{fds}, \_ \,\rangle = \mathcal{C}(c) \qquad \texttt{Some}\ t_a = \mathit{fds}(a)}{\langle \mathcal{C}, \Gamma \rangle \vdash_E e.a : t_a} \quad \text{[Field]}$$

$$\frac{\langle\, \mathit{fds},\ \mathit{ms} \,\rangle \in \mathcal{C}(c)}{\langle \mathcal{C}, \Gamma \rangle \vdash_E \texttt{new}\ c() : c} \quad \text{[New]}$$

$$\frac{\begin{array}{c}\langle \mathcal{C}, \Gamma \rangle \vdash_E e_i : t'_i \qquad \forall i \in \{1, \ldots, p\} \\ \texttt{Some}\ ([t_1; \ldots; t_p], t_r) = \Gamma(m) \\ t'_i = t_i \qquad \forall i \in \{1, \ldots, p\}\end{array}}{\langle \mathcal{C}, \Gamma \rangle \vdash_E m(e_1, \ldots, e_p) : t_r} \quad \text{[LocalCall]}$$

$$\frac{\begin{array}{c}\langle \mathcal{C}, \Gamma \rangle \vdash_E e_i : t'_i \qquad \forall i \in \{1, \ldots, p\} \\ \langle \mathcal{C}, \Gamma \rangle \vdash_E e_0 : c_0 \qquad \langle\, \_,\ \mathit{mds} \,\rangle = \mathcal{C}(c_0) \\ \texttt{Some}\ ([t_1; \cdots; t_p], t_r) = \mathit{mds}(m) \\ t'_i = t_i \qquad \forall i \in \{1, \ldots, p\}\end{array}}{\langle \mathcal{C}, \Gamma \rangle \vdash_S e_0.m(e_1, \ldots, e_p) : t_r} \quad \text{[GlobalCall]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_E e_1 : \mathit{int} \qquad \langle \mathcal{C}, \Gamma \rangle \vdash_E e_2 : \mathit{int} \qquad \mathit{aop} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}}{\langle \mathcal{C}, \Gamma \rangle \vdash_E e_1\ \mathit{aop}\ e_2 : \mathit{int}} \quad \text{[Arith]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_E e : \mathit{int}}{\langle \mathcal{C}, \Gamma \rangle \vdash_E -e : \mathit{int}} \quad \text{[Negation]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_E e_1 : \mathit{string} \qquad \langle \mathcal{C}, \Gamma \rangle \vdash_E e_2 : \mathit{string}}{\langle \mathcal{C}, \Gamma \rangle \vdash_E e_1 + e_2 : \mathit{string}} \quad \text{[String]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_E e_1 : \mathit{int} \qquad \langle \mathcal{C}, \Gamma \rangle \vdash_E e_2 : \mathit{int} \qquad \mathit{rop} \in \{\texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}, \texttt{==}, \texttt{!=}\}}{\langle \mathcal{C}, \Gamma \rangle \vdash_E e_1\ \mathit{rop}\ e_2 : \mathit{bool}} \quad \text{[Rel]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_E e_1 : \mathit{bool} \qquad \langle \mathcal{C}, \Gamma \rangle \vdash_E e_2 : \mathit{bool} \qquad \mathit{bop} \in \{\texttt{||}, \texttt{\&\&}\}}{\langle \mathcal{C}, \Gamma \rangle \vdash_E e_1\ \mathit{bop}\ e_2 : \mathit{bool}} \quad \text{[Bool]}$$

$$\frac{\langle \mathcal{C}, \Gamma \rangle \vdash_E e : \mathit{bool}}{\langle \mathcal{C}, \Gamma \rangle \vdash_E !e : \mathit{bool}} \quad \text{[Complement]}$$

One 'messy' thing I have left out is `null`, which should be a base type for all nullable types. To add it would lengthen this description. So I am leaving it to your discretion. Do the 'natural' thing. One thing to note is that for String, `null` is the empty string as well as the null-object.

B Syntactic Specification of IR3

B.1 Lexical Issues

- ⟨*CName3*⟩ ∈ **Class Names**. It is defined similarly as ⟨*CName*⟩ in JLite.
- ⟨*id3*⟩ ∈ **Identifiers**. This includes ⟨*id*⟩ as defined in JLite. In addition, it also includes `this` (referring to specific object), as well as newly generated temporary variables, which are of the format `(_[a-z]+[0-9]+)`.

B.2 Syntactic Specification

⟨*Program3*⟩ → ⟨*CData3*⟩* ⟨*CMtd3*⟩*

⟨*CData3*⟩ → `Data3` ⟨*CName3*⟩ `{` ⟨*VarDecl3*⟩ * `}`

⟨*VarDecl3*⟩ → ⟨*Type3*⟩ ⟨*id3*⟩ `;`

⟨*CMtd3*⟩ → ⟨*Type3*⟩ ⟨*id3*⟩ `(` ⟨*FmlList3*⟩ `)` ⟨*MdBody3*⟩

⟨*FmlList3*⟩ → ⟨*CName3*⟩ `this` ⟨*FmlL13*⟩

⟨*FmlL13*⟩ → `,` ⟨*Type3*⟩ ⟨*id3*⟩ ⟨*FmlRest3*⟩ * | ϵ

⟨*FmlRest3*⟩ → `,` ⟨*Type3*⟩ ⟨*id3*⟩

⟨*Type3*⟩ → `Int` | `Bool` | `String` | `Void` | ⟨*CName3*⟩

⟨*MdBody3*⟩ → `{` ⟨*VarDecl3*⟩ * ⟨ *Stmt3*⟩ + `}`

⟨*Stmt3*⟩ → ⟨*Label3*⟩ `:` | `if (` ⟨*RelExp3*⟩ `)` `goto` ⟨*Label3*⟩ `;` | `goto` ⟨*Label3*⟩ `;`
| `readln (` ⟨*id3*⟩ `)` `;` | `println (` ⟨*idc3*⟩ `)` `;`
| ⟨*Type3*⟩ ⟨*id3*⟩ `=` ⟨*Exp3*⟩ `;` | ⟨*id3*⟩ `=` ⟨*Exp3*⟩ `;` | ⟨*id3*⟩ `.` ⟨*id3*⟩ `=` ⟨*Exp3*⟩ `;`
| ⟨*id3*⟩ `(` ⟨*VList3*⟩ `)` `;`
| `return` ⟨*id3*⟩ `;` | `return` `;`

⟨*RelExp3*⟩ → ⟨*idc3*⟩ ⟨*Relop3*⟩ ⟨*idc3*⟩ | ⟨*idc3*⟩

⟨*Exp3*⟩ → ⟨*idc3*⟩ ⟨*Bop3*⟩ ⟨*idc3*⟩ | ⟨*Uop3*⟩ ⟨*idc3*⟩ | ⟨*id3*⟩ `.` ⟨*id3*⟩ | ⟨*idc3*⟩
| ⟨*id3*⟩ `(` ⟨*VList3*⟩ `)` | `new` ⟨*CName3*⟩ `()`

⟨*Relop3*⟩ → `<` | `>` | `<=` | `>=` | `==` | `!=`

⟨*Bop3*⟩ → ⟨*Relop3*⟩ | `||` | `&&` | `*` | `/` | `+` | `-`

⟨*Uop3*⟩ → `!` | `-`

⟨*VList3*⟩ → ⟨*idc3*⟩ ⟨*VRest3*⟩ * | ϵ

⟨*VRest3*⟩ → `,` ⟨*idc3*⟩

⟨*idc3*⟩ → ⟨*id3*⟩ | ⟨*Const*⟩

⟨*Const*⟩ → `true` | `false` | `INTEGER_LITERAL` | `STRING_LITERAL` | `NULL`

C Sample JLite Program and Corresponding IR3 Code Generated

Following is a sample JLite program:

```
class MainC {

   Void main (){
      Functional fo ;
      Int i;
      Int j ;

      readln(i) ;
      if (i > 0) {
         fo = new Functional() ;
         j = fo.f(i) ;
         println(j) ;
      }
      else {
         println("Error") ;
      }
      return ;
   }
}

class Functional {
Int a;

Int f (Int b){
        return 3;
}

}
```

Following is a possible IR3 code generated for the above-mentioned JLite program.

```
======= CData3 =======

class MainC{
}

class Functional{
  Int a;
}

=======  CMtd3 =======

void main(MainC this){
  Functional fo;
  Int i;
  Int j;
  readln(i);
  If(i > 0) goto 1;
  println("Error");
  goto 2;
 Label 1:
  fo = new Functional();
  j = %Functional_0(fo,i);
  println(j);
 Label 2:
  Return;
}

Int %Functional_0(Functional this,Int b){
  Int _t1;
  _t1=3;
  Return _t1;
}

=====fx== End of IR3 Program =======
```