Compilation's proramming assignement
JavaScript compiler

12 janvier 2023

Attention : This document may change along the semester, especially with the sanitary crisis. But those changes will always be in your advantage.

1 General recommendations

This project consists in compiling some fragments of javascript into an add-hoc assembly language.

You are free to use any programming language to write the compiler, and any externary tools (for the parser generation). Initial material will only be available on C, Java, OCaml and Haskell, but you should be able to find equivalents for other languages on the internet.

However, there is one tool the you must absolutely use : a GIT repository. You are free to use any git server, but we recommend the git server of the university. Do not forget to give reading rights to the teachers (you may also give us writing rights so that we are able to add comments). We strongly recommend you to familiarize yourself with GIT's branching feature, because we will use it heavily.

Indeed, you will have to create two delivery branches : the "parser" and "master" (or "compiler") branches. Only those delivery branches will be evaluated, and the content of each tag present in those branches is predetermined, any mistake will cost points (and potentially nullify your remaining versions). We also recommend you to use (at least) two other working branches, one to work on the parser, and the other to work on the compiler.

The "parser" branch will only contain (sources of) a program that takes as input a JS-program and says if it is correct or not. The "master" branch will only contain (sources of) a program that takes as input a JS-program and generate assembly code. You are allowed to make intermediate commits on those branches, but each of them should be fully functional.

We will present several increasingly larger fragment of JS (think of it as versions), for each fragment, you are invited to perform one tagged commit in the parser branch that "recognizes" this fragment, and one tagged commit in the "master" branch that compiles this fragment. The fragments are called *Fragment* $i.j$, and are described latter, each fragment is part of a "large fragment", here the "large fragment" i.

Each tagged version must :

— contain a README with a short description of the project, a list of required softwares, how to compile it, and how to run it,
— contain a set of tests that verify each feature separately,
— be annotated,
— be named `p_i_j` for the parser of fragment $i.j$, and `c_i_j` for the compiler of fragment $i.j$,
— contain a tag description that precise who did what in the specific update.

"Parser" and "master" branches should only contain the required tagged versions, wit eventually a few bug-fix commits. At the time of the evaluation, we will only look at those versions, with, eventually, the exception of the lasts commits of working branches. For those commits in working branches, please try to maintain a document containing a todo-list with everything that remains to be done before the next "release".

Warning : Non-required features are forbidden in "parser" and "master" branches. If you want to add some feature, please use a supplementary branch (that we may look at, may attribute bonus points for, but we cannot promise anything).

2 Parser

sources The tagged commits should only contain sources and the README. The README should explain how to compile the parser. The commands required for this compilation should be available from the terminal of a Linux with the correct packages. [1]

input/output The expected input of the resulting parser should be a file containing a program written in the correct fragment of javascript. The output should be a sentence validating the input program or refusing it. The error message do not need to be detailed, but bonus points may be attributed if details are given.

3 Compiler

sources The tagged commits should only contain sources and the README. The README should explain how to compile the compiler. The commands required for this compilation should be available from the terminal of a Linux with the correct packages. [2]

input/output The expected input of the resulting parser should be a file containing a program write in the correct fragment of javascript. The output should either be an error message (the one from the parser) or a file containing the compiled version of the input. The compiled version should be executable in the furnished virtual machine.

4 Description of fragments

4.1 Large Fragment 0 : Starting (4pts)

See Section 5.0.1 to get the grammar of the *Large Fragment* 0.

Fragment 0.0 **(given)** It can only treat arithmetical expressions :

- A unique base type : `Number`. Normally, JS's numbers correspond to floats, [3] but for now, only integers should be recognized.
- We use a minimal set of arithmetical operations ("`+`", "`-`" and "`*`"). You have to respect priorities !
- No variable are considered.
- A program can only be of the form `"expression ;"`, for example `2+3+5;` or `2*(3+2);`. Do not forget the semicolon !
- Authorized assembly instructions : `AddiNb`, `MultNb`, `NegaNb`, `SubsNb`, `CsteNb`, `Halt`

Fragment 0.1 **(TP)** Everything from *Fragment* 0.0

- plus a modulo operation `(_%_)` ;
- You have to respect priorities and associativity !

1. This is just a way to say that instructions like "export the project in ide A version V with X and Y plugins installed and run it throw the ide" are not accepted.
2. This is just a way to say that instructions like "export the project in ide A version V with X and Y plugin installed and run it throw the ide" are not accepted.
3. In javascript, there is no integers, only floats.

— in addition, numbers can now be any float. For now, we only require integer notation (e.g. : `34`, `-12`, `0045`) and fixed-point notations (e.g. : `3.4`, `-1.2`, `0.045`).
— Additional Authorized assembly instruction : `ModuNb`.

Fragment 0.2 Everything from *Fragment* 0.1 plus Boolean manipulations :
— One can use `True` and `False` as constant expressions.
— We add the binary operator `(_==_)` that takes two numbers or two Boolean and produce a Boolean.
— We add the Boolean negation `(!_)`.
— We add the order operators `(_>_)`, and `(_>=_)` that takes two numbers and produce a Boolean.
— Behavior of non well typed expressions are not specified for now (you can return whatever you want).
— You have to respect priorities and associativity !
— Additional Authorized assembly instructions : `Equals`, `NotEql`, `LoEqNb`, `GrEqNb`, `LoStNb`, `GrStNb`, `Not`, `CsteBo`

4.2 Large Fragment 1 : More expressions and small commands (+2pts = 6pts)

Fragment 1.0 Everything from *Fragment* 0.2 plus the floating point notations : numbers can be written `1.215e25` or `.485e-42` or `485e-42`. We also add the `NaN` constant.
No new assembly instructions are authorized, but you can use `NaN` or floating point notations with the instruction `CsteNb`.

Fragment 1.1 Everything from *Fragment* 1.0 plus the possibility write several expressions in sequence. For exemple, one can write `2+3;42;2==5;`. Each expression followed by a "`;`" is a command, and each command have to be executed.
The `Drop` command is autorised, but not mandatory.

Fragment 1.2 Everything from *Fragment* 1.1 plus a new the `import` command. For simplicity, we require the command `import malib;` to look for a file `malib.jsm` containing assembly code and copy it inside the produced assembly.
No new assembly instructions are authorized.

Fragment 1.3 Everything from *Fragment* 1.2 plus the Boolean "and" : it consists in adding the Boolean operator `(_&&_)`. Be careful to correctly follow the behavior of the operator from JS : the second argument is executed only if the first retrun `True` otherwise you should jump. [4].
You have to respect priorities and associativity !
Additional authorized assembly instructions : `Jump`, `ConJmp`.

Fragment 1.4 Everything from *Fragment* 1.0 plus the comments : Comments can be inserted anywhere and should not change the resulting compiled program. They can be inlined (of the form `// my comment`) or multilined (of the form `/* my comment */`).
No new assembly instructions are authorized.

4.3 Large Fragment 2 : Varliables (+2pts = 8pts)

Fragment 2.0 Everything from *Fragment* 1.3 plus variables :
— Variables begin with a lower-case letter and can be composed of letters (lower and uper), numbers and underscores "_".

4. On Booleans only, casts are not defined yet

— Variable do not need do be declared.
— Variables are instantiated via the **expression** [5] `x = expr;` where `expr` is any expression. Those variables are always global.
— You have to respect priorities and associativity ! (use the console if you do not know them).
— Variables can be used inside expressions.

Additional authorized assembly instruction : `SetVar`, `GetVar`.

Fragment 2.1 Everything from *Fragment* 2.0 plus an optimization : if an expression does not contain any variables (nor objects or functions), perform the evaluation at compile-time and replace the whole expression by a constant. This fragment do not imply any change in the lexer or the parser. For example, the program `2*10+x==3*8;` should be compiled

```
CsteNb 20
GetVar x
AddiNb
CsetNb 24
Equals
```

No new assembly instruction are authorized.

4.4 Large Fragment 3 : Loops (+2pts = 10pts)

Fragment 3.0 Everything from *Fragment* 2.1 plus the conditional :
— A program is now a sequence of commands, while a command is either an expression `expr;` or
— a new command `If(_) _ Else _`. Be careful that the first hole has to be an expression, while the two others are commands.

No new assembly instruction are authorized.

Fragment 3.1 Everything from *Fragment* 3.0 plus the `do_while_` loop. No new assembly instructions are authorized.

Fragment 3.2 Everything from *Fragment* 3.1 plus the aggregated commands :
— We add the empty command "`;`".
— In addition, we can compress a sequence of commands into a unique command by applying brackets `{com1 ... comk}`.

No new assembly instruction are authorized.

4.5 Large Fragment 4 : Functions (+1pts = 11pts)

Fragment 4.0 Everything from *Fragment* 3.1 plus the functions call. You can use imports to get preregistered functions.
Additional authorized assembly instructions : `StCall`, `SetArg` and `Call`.

Fragment 4.1 Everything from *Fragment* 4.0 excepts that keywords are now lower-case [6] and variables can be lower- or upper-case (but cannot be one of the key-words).
No new assembly instruction are authorized.

Fragment 4.2 Everything from *Fragment* 4.1 plus function declarations. No hosting should be performed for now : a function is declared where its code is (like in C).
Additional authorized assembly instructions : `NewClot`, `DclArg` and `Return`.

5. One can thus write `2 + (x = 5);` (try it on your browser console !)
6. Excepts for `NaN` that stays like this.

4.6 Large Fragment 5 : Dynamic types (+2pts = 13pts)

Fragment 5.0 Everything from *Fragment* 4.2 plus the implicit casts on Boolean operators : When a number is used as argument of a Boolean operator or a condition, a cast is performed as in the real JavaStript.
Additional authorized assembly instructions : `Case`, `TypeOf`, `Noop`, `Swap`, `BoToNb`.

Fragment 5.2 Everything from *Fragment* 5.1 plus a new type `Undefined`, with a unique value (and constant) `undefined`. Do not forget to deal with the types conversions.
Additional authorized assembly instructions : `CsteUn`.

Fragment 5.1 Everything from *Fragment* 5.0 plus the implicit casts on the arithmetical operators : When a Boolean is used as argument of an arithmetical operator a cast is performed as in the real JavaStript. [7]
Additional authorized assembly instructions : `NbToBo`.

Fragment 5.2 Everything from *Fragment* 5.1 but with functions that are now values (more exactly they are closures). Closures should not be casted, if used in an operator waiting for another type, you have to write a code that crash. [8]
Additional authorized assembly instruction : `Error`

Optional Fragment 5.3 Everything from *Fragment* 5.2 plus a little bit of type inference : in some cases we can know the returned types of operators : the constants, the arithmetic operators (`+, -, *, %`). [9] Some other have a return type that depends on its sons : the ternary operator and the `||`. While, for now, we consider that we do not know the type of the variables. Perform a semantic analysis that add the types of the expression whose type is known. Then simplify the casts one expressions of known types.
No new assembly instruction are authorized.

4.7 Large Fragment 6 : hosting and functional (+2pts = 15pts)

Fragment 6.0 Everything from *Fragment* 5.2 plus the variable declarations :
- We add a new command `let _;` where the hole is a variable name.
- The declared variable will have the value `undefined`.
- The declaration can be done anywhere, it is not hosted for now.

Additional authorized assembly instruction : `DclVar`

Fragment 6.1 Everything from *Fragment* 6.0 but with a correct hosting for the `let` : variables that are declared outside any block or functions are hosted at the beginning of the program, those that are declared inside a function are hosted at the beginning of the innermost function and those that are declared in a block (i.e. in-beetwin `{...}`) are hosted at the begining of the block. In addition, a programe with two `let`-declaration for the same variable in the same block should be refused by the compiler.
No new assembly instruction are authorized.

Fragment 6.2 Everything from *Fragment* 6.1 plus the lambda expressions.
No new assembly instruction are authorized.

7. Auxiliary functions may be useful here...
8. In JavaScript, functions are casted into the string of their code, this is a dangerous behavior as it exposes the code to agents that should not be able to access it.
9. in fact the + can return a String if given a String as first argument, but we will not implement the string this year.

Optional Fragment 6.3 Everything from *Fragment* 6.2 but with a correct hosting : variables and functions that are declared outside any functions are hosted at the beginning of the program, those that are declared inside a function are hosted at the beginning of the innermost function.
No new assembly instruction are authorized.

4.8 Large Fragment 7 : Record objects (+1.5pts = 16.5pts)

Fragment 7.0 Everything from *Fragment* 6.2 plus an object `null`. Know that `null`, the empty object `{}`, `undefined` and undeclared variables are very different and behave differently. Be careful to always have the correct behavior.

Fragment 7.1 Everything from *Fragment* 7.0 plus the record objects.
Remark : In JavaScript, an attribute of a record object can have any name, even a key-word ; this feature is not required but you can try to add it.
Additional authorized assembly instruction : `GetObj`, `NewObj`, `SetObj`

Fragment 7.2 Everything from *Fragment* 7.1 plus the keyword `this` that can be used inside the declaration of a record object and refers to the object itself (for this simply add a variable in the context). Remark : this is a "java-like" version of the `this`, in JavaScript its behavior is much more complex.

4.9 Large Fragment 8 : Exceptions (+1.5pts = 18pts)

Fragment 8.0 Everything from *Fragment* 7.2 with a sanitarization of the stack : You will find cases where your stack is polluted by unused expressions, for example in `x=42; x+1; 25;`. In this fragment, the stack should be sanitarized. Remark : there is no change required to the parser here.
Additional authorized assembly instructions : `Drop`, `SetVaD`.

Fragment 8.1 Everything from *Fragment* 8.0 plus the `try_catch` and `throw` structures. You are now authorize to use any assembly instruction you'd like.

4.10 Large Fragment 9 : Classes (+2pts = 20pts)

Fragment 9 Everything from *Fragment* 8.1 plus the classes with the expected behavior.

4.11 Optional Fragments

Optional fragments can be performed in any order once everything else is completed. They are not required to get 20, but it will be difficult without any of them.

Optional Fragment **Lambda expression** Add the lambda expressions (relatively easy fragment.)

Optional Fragment **Strings** Adding stings is not very complex,[10] but it adds yet another type which multiply the number of cases you have to try in the cast. That is why you will absolutely have to use auxiliary functions here. Also, be careful of the nasty behavior of the operator "+".

Optional Fragment **Finally** Add the `try_catch_finally` operator.

10. Strings are a base type in the given machine, but it is not standard, and the situation is much more complex with an encoding

***Optional Fragment* Inference** In some cases, we can infer the type of some variable. Try to use such a semantic analysis to later perform some optimizations.

***Optional Fragment* Error** Find a way to obtain readable and useful message for your errors.

***Optional Fragment* Other** You can implement additional operators of you choice :
- `_++` operator,
- Tuples,
- Break,
- Switch,
- Tabs,
- Partial static typing,
- Optimizations,
- Declaring multiple variables,
- Declaring and initializing a variable.
- ...

5 Grammars to implement

We are giving the grammar of every fragment. These grammars cannot be use directly in your parser generator because we do not consider associativity and priority. To add them, you will have to work by yourself but you can use the javascript console in you favorite browser, you can also use some official references.

Recall that <NUMBER>, <BOOLEAN>, <IDENT>, <MIDENT> are tokens representing respectively numbers, Booleans, names (for variables and functions) and class names (begin by a maj).

5.0.1 Grammar of *Large Fragment* 0

```
<command>    ::=   <expression> ;
<expression> ::=   <NUMBER> | <BOOLEAN>
                 | (<expression>)
                 | <expression> + <expression>
                 | <expression> - <expression>
                 | <expression> * <expression>
                 | <expression> % <expression>
                 | <expression> == <expression>
                 | <expression> > <expression>
                 | <expression> >= <expression>
                 | ! <expression>
                 | - <expression>
```

5.0.2 Grammar of *Large Fragment* 1

From now on, in each fragment, we write changes in red :

```
<program>    ::=   ϵ | <command> <program>

<command>    ::=   <expression> ;
                 | Import <IDENT> ;

<expression> ::=   <NUMBER> | <BOOLEAN>
                 | (<expression>)
                 | <expression> + <expression>
                 | <expression> - <expression>
                 | <expression> * <expression>
                 | <expression> % <expression>
                 | <expression> == <expression>
                 | <expression> > <expression>
                 | <expression> >= <expression>
                 | <expression> && <expression>
                 | ! <expression>
                 | - <expression>
```

5.0.3 Grammar of *Large Fragment* 2

```
<program>    ::=   ϵ | <command> <program>

<command>    ::=   <expression> ;
                 | Import <IDENT> ;

<expression> ::=   <NUMBER> | <BOOLEAN> | <IDENT>
                 | <IDENT> = <expression> ;
                 | (<expression>)
                 | <expression> + <expression>
                 | <expression> - <expression>
                 | <expression> * <expression>
                 | <expression> % <expression>
                 | <expression> == <expression>
                 | <expression> > <expression>
                 | <expression> >= <expression>
                 | <expression> && <expression>
                 | ! <expression>
                 | - <expression>
```

5.0.4 Grammar of *Large Fragment* 3

<program> ::= ϵ | <command> <program>

<command> ::= <expression> `;`
| `Import` <IDENT> `;`
| `;`
| `{`<program>`}`
| `If (` <expression> `)` <command> `Else` <command>
| `While` (<expression>) <command>

<expression> ::= <NUMBER> | <BOOLEAN> | <IDENT>
| `(`<expression>`)`
| <IDENT> `=` <expression>
| <expression> `+` <expression>
| <expression> `-` <expression>
| <expression> `*` <expression>
| <expression> `%` <expression>
| <expression> `==` <expression>
| <expression> `>` <expression>
| <expression> `>=` <expression>
| <expression> `&&` <expression>
| `!` <expression>
| `-` <expression>

5.0.5 Grammar of *Large Fragment* 4

<program> ::= ϵ | <command> <program>

<command> ::= <expression> ;
| `import` <IDENT> ;
| ;
| `{`<program>`}`
| `if (` <expression> `)` <command> `else` <command>
| `while (` <expression> `)` <command>
| `function` <IDENT> `(`<decl_args>`) {` <program> `}`
| `return` <expression> ;

<decl_args> ::= ϵ | <IDENT> | <IDENT> `,` <decl_args>

<expression> ::= <NUMBER> | <BOOLEAN> | <IDENT>
| `(`<expression>`)`
| <IDENT> `=` <expression>
| <expression> `+` <expression>
| <expression> `-` <expression>
| <expression> `*` <expression>
| <expression> `%` <expression>
| <expression> `==` <expression>
| <expression> `>` <expression>
| <expression> `>=` <expression>
| <expression> `&&` <expression>
| `!` <expression>
| `-` <expression>
| <IDENT> `(` <arguments> `)`

<arguments> ::= ϵ | <expression> | <expression> `,` <arguments>

5.0.6 Grammar of *Large Fragment* 5

<program> ::= ϵ | <command> <program>

<command> ::= <expression> `;`
| `import` <IDENT> `;`
| `;`
| `{`<program>`}`
| `if (` <expression> `)` <command> `else` <command>
| `while (` <expression> `)` <command>
| `function` <IDENT> `(`<decl_args>`) {` <program> `}`
| `return` <expression> `;`

<decl_args> ::= ϵ | <IDENT> | <IDENT> `,` <decl_args>

<expression> ::= <NUMBER> | <BOOLEAN> | <IDENT>
| `undefined`
| `(`<expression>`)`
| <IDENT> `=` <expression>
| <expression> `+` <expression>
| <expression> `-` <expression>
| <expression> `*` <expression>
| <expression> `/` <expression>
| <expression> `==` <expression>
| <expression> `>` <expression>
| <expression> `>=` <expression>
| <expression> `||` <expression>
| <expression> `&&` <expression>
| `!` <expression>
| `-` <expression>
| <IDENT> `(` <arguments> `)`

<arguments> ::= ϵ | <expression> | <expression> `,` <arguments>

5.0.7 Grammar of *Large Fragment* 6

<program> ::= ϵ | <command> <program>

<command> ::= <expression> ;
| `import` <IDENT> ;
| ;
| {<program>}
| `let` <IDENT> ;
| `if` (<expression>) <command> `else` <command>
| `while` (<expression>) <command>
| `function` <IDENT> (<decl_args>) { <program> }
| `return` <expression> ;

<decl_args> ::= ϵ | <IDENT> | <IDENT> , <decl_args>

<expression> ::= <NUMBER> | <BOOLEAN> | <IDENT>
| `undefined`
| (<expression>)
| <IDENT> = <expression>
| <expression> + <expression>
| <expression> - <expression>
| <expression> * <expression>
| <expression> % <expression>
| <expression> == <expression>
| <expression> > <expression>
| <expression> >= <expression>
| <expression> && <expression>
| ! <expression>
| - <expression>
| <IDENT> (<arguments>)
| `function` (<dec_args>) { <program> }
| (<dec_args>) => <expression>

<arguments> ::= ϵ | <expression> | <expression> , <arguments>

5.0.8 Grammar of *Large Fragment* 7

<program> ::= ϵ | <command> <program>

<command> ::= <expression> ;
| `import` <IDENT> ;
| ;
| `{`<program>`}`
| `let` <IDENT> ;
| `if (` <expression> `)` <command> `else` <command>
| `while (` <expression> `)` <command>
| `function` <IDENT> `(`<decl_args>`)` `{` <program> `}`
| `return` <expression> ;

<decl_args> ::= ϵ | <IDENT> | <IDENT> `,` <decl_args>

<expression> ::= <NUMBER> | <BOOLEAN> | <IDENT>
| `undefined`
| `(`<expression>`)`
| <IDENT> `=` <expression>
| <expression> `+` <expression>
| <expression> `-` <expression>
| <expression> `*` <expression>
| <expression> `/` <expression>
| <expression> `==` <expression>
| <expression> `>` <expression>
| <expression> `>=` <expression>
| <expression> `&&` <expression>
| `!` <expression>
| `-` <expression>
| <IDENT> `(` <arguments> `)`
| `function (`<dec_args>`)` `{` <program> `}`
| `(`<dec_args>`)` `=>` <expression>
| {<object_content>} | `Null`
| <expression> `.` <IDENT>

<arguments> ::= ϵ | <expression> | <expression> `,` <arguments>

<object_content> ::= ϵ | <IDENT> `:` <expressions>
| <IDENT> `:` <expressions> `,` <object_content>

5.0.9 Grammar of *Large Fragment* 8

<program> ::= ϵ | <command> <program>

<command> ::= <expression> ;
| `import` <IDENT> ;
| ;
| {<program>}
| `let` <IDENT> ;
| `if` (<expression>) <command> `else` <command>
| `while` (<expression>) <command>
| `function` <IDENT> (<decl_args>) { <program> }
| `return` <expression> ;
| `try` { <program> } `catch` (<IDENT>) { <program> }

<decl_args> ::= ϵ | <IDENT> | <IDENT> , <decl_args>

<expression> ::= <NUMBER> | <BOOLEAN> | <IDENT>
| `undefined`
| (<expression>)
| <IDENT> = <expression>
| <expression> + <expression>
| <expression> - <expression>
| <expression> * <expression>
| <expression> / <expression>
| <expression> == <expression>
| <expression> > <expression>
| <expression> >= <expression>
| <expression> && <expression>
| ! <expression>
| - <expression>
| <IDENT> (<arguments>)
| `function` (<dec_args>) { <program> }
| (<dec_args>) => <expression>
| {<object_content>} | `Null`
| <expression> . <IDENT>

<arguments> ::= ϵ | <expression> | <expression> , <arguments>

<object_content> ::= ϵ | <IDENT> : <expressions>
| <IDENT> : <expressions> , <object_content>

5.0.10 Grammar of *Large Fragment* 9

```
<program>        ::=  ϵ | <command> <program>

<command>        ::=  <expression> ;
                    | import <IDENT> ;
                    | ;
                    | {<program>}
                    | let <IDENT> ;
                    | if ( <expression> ) <command> else <command>
                    | while ( <expression> ) <command>
                    | function <IDENT> (<decl_args>) { <program> }
                    | return <expression> ;
                    | try { <program> } catch ( <IDENT> ) { <program> }
                    | class <MIDENT> { <class_content> }
                    | class <MIDENT> extends <expression> { <class_content> }

<decl_args>      ::=  ϵ | <IDENT> | <IDENT> , <decl_args>

<class_content>  ::=  ϵ | <IDENT> ; <class_content>
                    | <IDENT> (<decl_args>) { <program> } <class_content>

<expression>     ::=  <NUMBER> | <BOOLEAN> | <IDENT>
                    | undefined
                    | (<expression>)
                    | <IDENT> = <expression>
                    | <expression> + <expression>
                    | <expression> - <expression>
                    | <expression> * <expression>
                    | <expression> / <expression>
                    | <expression> == <expression>
                    | <expression> > <expression>
                    | <expression> >= <expression>
                    | <expression> && <expression>
                    | ! <expression>
                    | - <expression>
                    | <IDENT> ( <arguments> )
                    | function (<dec_args>) { <program> }
                    | (<dec_args>) => <expression>
                    | {<object_content>} | Null
                    | <expression> . <IDENT>

<arguments>      ::=  ϵ | <expression> | <expression> , <arguments>

<object_content> ::=  ϵ | <IDENT> : <expressions>
                    | <IDENT> : <expressions> , <object_content>
```