Compilation’s programming assignment
JavaScript compiler

3 mars 2022

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 programing 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 recommand you to familiarise yourself with GIT’s braching 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 recommand 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” branche 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 (des-)assembly code. You are allowed to make intermediat commits on those branches, but each of them should be fully functional.

We will present several increasinly larger fragment of JS (think of it as versions), for each fragment, you are invited to perform one tagged commit in the parser branche that “reconises” this fragment, and one tagged commit in the “master” branche that compiles this fragment. The fragments are called Fragment $i.j$, and are described latters, each fragment is part of a “large fragment”, here the “large fragment” $i$. 

1
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-therquired 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 attribut 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.[]

**input/output** The expected input of the resulting parser should be a file
containing a programme written in the correct fragment of javascript. The output
should be a sentence validating the input programme or refusing it. The error
message do not need to be detailed, but bonus points may be atributed 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.[]

---

1. 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.
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.
input/output  The expected input of the resulting parseur should be a file containing a programme written 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, but for now, only integers should be recognised.
— We use a minimal set of arithmetical operations (’+’, ’-’ and ’*’). You have to respect priorities!
— No variable are considered.
— A programme can only be of the form ”expression ;”, for example 2+3+5; or 2*(3+2);. Do not forget the semicolon!
— Authorised 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!
— 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 authorised 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.
— Behaviour of non well typed expressions are not specified for now (you can return whatever you want).
— You have to respect priorities and associativity!
— Additional authorised assembly instructions : Equals, NotEql, LoEqNb, GrEqNb, LoStNb, GrStNb, Not

3. In javascript, there is no integers, only floats.
4.2 Large Fragment 1 : More expressions (6pts)

**Fragment 1.0** Everything from Fragment 0.2 plus the floating point notations: numbers can be written \(1.215e25\) or \(4.85e-42\) or \(485e-42\). We also add the NaN constant.

**Fragment 1.1** Everything from Fragment 1.0 plus the comments: Comments can be inserted anywhere and should not change the resulting compiled program. They can be uniline (of the form `// my comment`) or multiline (of the form `/* my comment */`). No new assembly instructions are authorised.

**Fragment 1.2** Everything from Fragment 1.1 plus the ternary operator `_?_: _` that takes three arguments, evaluate the first, if it is a boolean the operator execute the code of the second argument, else it executes the code of the third argument: this is like an if_then_else but inside expressions. Warning: in the AST, the ternary operator should be a ternary node with 3 sons. Additional authorised assembly instructions: Jump, ConJmp.

**Fragment 1.3** Everything from Fragment 1.2 plus the boolean “or” and “and”: it is a small fragment consisting in adding the boolean operator `_||_`. Be careful to correctly follow the behaviour of the operator from JS. You have to respect priorities and associativity! No new assembly instructions are authorised.

4.3 Large Fragment 2 : Commands (8pts)

**Fragment 2.0** Everything from Fragment 1.3 plus variables:
- Variables begin with a lower-case letter and can be composed of letters, numbers, underscores “_”.
- Variable do not need to be declared.
- Variables are instanciated via the expression `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 authorised assembly instruction: SetVar, GetVar.

**Fragment 2.1** Everything from Fragment 2.0 plus the conditional:
- A programme 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.

---

4. On Booleans only, cast are not defined yet
5. One can thus write \(2 + (x = 5)\); (try it on your browser console!)
— You can optionally add the If_Then_ command; it is not required because parsing both conditional can be very difficult depending of the choosen parser generator.\footnote{6. Easy with LR parsers generators, but difficult for LL parser generators.}
— No new assembly instruction are autorised.

\textbf{Fragment 2.2} Everything from \textit{Fragment 2.1} plus the agregated commands :
— We add the empty command “;”.
— In addition, we can compact a sequence of commands into a unique command by applying brakets \{com1 ... comk\}.
— No new assembly instruction are autorised.

4.4 Large Fragment 3 : Loops (10pts)

\textbf{Fragment 3.0} Everything from \textit{Fragment 2.2} plus the do\_while\_ loop. No new assembly instructions are autorised.

\textbf{Fragment 3.1} Everything from \textit{Fragment 3.2} plus the \textit{while(\_)}\_ loops. No new assembly instructions are autorised.

\textbf{Fragment 3.2} Everything from \textit{Fragment 3.1} plus an optimisation : if an expression do not contain variable (nor object or function), 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.

4.5 Large Fragment 4 : Functions (11pts)

\textbf{Fragment 4.0} Everything from \textit{Fragment 3.2} plus the functions. No hosting is required for now : a function can be declared where its code is (like in C). Additional autorised assembly instructions : \texttt{Lamba, DclArg, SetArg, Call} and \texttt{Return}.

\textbf{Fragment 4.1} Everything from \textit{Fragment 4.0} excepts that key-words are now lower-case\footnote{7. Excepts for NaN that stays like this.} and variables can be lower- or upper-case (but cannot be one of the key-words).

4.6 Large Fragment 5 : Dynamic types (13pts)

\textbf{Fragment 5.0} Everything from \textit{Fragment 4.1} 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 JavaScript.\footnote{8. Auxilairy functions may be usefull here.}
Additional autorised assembly instructions : \texttt{Case, TypeOf, Noop, Swap}. 
**Fragment 5.1** Everything from Fragment 5.0 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 JavaScript.

**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.

**Fragment 5.3** Everything from Fragment 5.2 plus a little bit of type inference: in some cases we can know the return 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 semantical analysis that add the types of the expression whose type is known. Then simplify the casts one expressions of known types.

### 4.7 Large Fragment 6: hosting and functional (15pts)

**Fragment 6.0** Everything from Fragment 5.1 plus the variable declarations:
- We add a new command `var _`; 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 authorised assembly instruction: `DclVar`

**Fragment 6.1** Everything from Fragment 6.0 but with a correct hosting: variables and functions that are declared outside any functions are hosted at the beginning of the programme, those that are declared inside a function are hosted at the beginning of the innermost function.

**Fragment 6.2** Everything from Fragment 6.1 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.¹⁰ Additional authorised assembly instruction: `Error`

**Fragment 6.3** Everything from Fragment 6.1 but with a tail-call optimisation: if a return command is returning the result of a function, then then do not write a `Return` instruction and replace the `Call` by a `TlCall` that will not create a new closure so that the called function will directly return from the outer function.

---

⁹ In fact the + can return a String if given a String as first argument, but we will not implement the string this year.

¹⁰ In JavaScript, functions are casted into the string of their code, this is a dangerous behaviour as it exposes the code to agents that should not be able to access it.
4.8 Large Fragment 7 : Record objects (16.5pts)

*Fragment 7.0* Everything from *Fragment 6.3* 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.

*Fragment 7.1* Everything from *Fragment 7.0* 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 behaviour.

*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 behaviour is much more complex.

4.9 Large Fragment 8 : Exceptions (18pts)

*Fragment 8.0* Everything from *Fragment 7.2* with a sanitisation 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 sanitised. Remark: there is no change required to the parser here.

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

4.10 Large Fragment 9 : Classes (20pts)

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

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 strings is not very complex 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 behaviour of the operator “+”.

11. 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 Finaly   Add the try\_catch\_finaly operator.

Optional Fragment Classes   This year, any attempt to treat classes is purely optional. You can look at the different instructions available to see what can be done.

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

Optional Fragment Error   Find a way to obtain readable message for you compiling errors.

Optional Fragment Other   You can implement additional operators of your choice:
- ++ operator,
- Tuples,
- Break,
- Swith,
- Tabs,
- Partial static typing,
- Optimisations,
- Declaring multiple variables,
- Declaring and initialising a variable.
- ...

5 Grammars to implement

We are giving the grammar of every fragment. These grammars cannot be used 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 your favorite browser, you can also use some official references.

Recall that <NUMBER>, <BOOLEAN>, <IDENT> are tokens representing respectively numbers, booleans, and names (for variables and functions).
5.0.1 Grammar of Large Fragment 0

<commande> ::= <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:

<commande> ::= <expression> ;
<expression> ::= <NUMBER> | <BOOLEAN>
               | (<expression>)
               | <expression> + <expression>
               | <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

\[<\text{programme}> ::= \epsilon | <\text{commande}> <\text{programme}>\]

\[<\text{commande}> ::= <\text{expression}> ;\]
\[| ;\]
\[| \{<\text{programme}>\}\]
\[| \text{If (}<\text{expression}>\text{) } <\text{commande}> \text{ Else } <\text{commande}>\]

\[<\text{expression}> ::= <\text{NUMBER}> | <\text{BOOLEEN}> | <\text{IDENT}>\]
\[| <\text{IDENT}> = <\text{expression}> ;\]
\[| (<\text{expression}> )\]
\[| <\text{expression}> + <\text{expression}>\]
\[| <\text{expression}> - <\text{expression}>\]
\[| <\text{expression}> * <\text{expression}>\]
\[| <\text{expression}> \% <\text{expression}>\]
\[| <\text{expression}> == <\text{expression}>\]
\[| <\text{expression}> < <\text{expression}>\]
\[| <\text{expression}> <= <\text{expression}>\]
\[| ! <\text{expression}>\]
\[| - <\text{expression}>\]
\[| <\text{expression}> ? <\text{expression}> : <\text{expression}>\]
5.0.4 Grammar of Large Fragment 3

<programme> ::= $\epsilon$ | <commande> <programme>

<commande> ::= <expression> ;
| ;
| {<programme>} if (<expression>) <commande> else <commande>
| do <commande> while (<expression>)
| while (<expression>) <commande>
| for (<expression> ; <expression> ; <expression>) <commande>

<expression> ::= <NUMBER> | <BOOLEEN> | <IDENT>
| (<expression>)
| <IDENT> * <expression>
| <expression> + <expression>
| <expression> - <expression>
| <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

```plaintext
<programme> ::= ϵ | <commande> <programme>

<commande> ::= <expression> ;
| ;
| {<programme>}
| if (<expression>) <commande> else <commande>
| do <commande> while (<expression>)
| while (<expression>) <commande>
| for (<expression>; <expression>; <expression>) <commande>
| function <IDENT> (<decl_args>) { <programme> }
| 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>
| ! <expression>
| - <expression>
| <expression>?<expression>:<expression>
| <IDENT> (<arguments>)

<arguments> ::= ϵ | <expression> | <expression>, <arguments>
```
5.0.6  Grammar of Large Fragment

<programme> ::= ε | <commande> <programme>

<commande> ::= <expression> ;
| {<programme>}
| if (<expression>) <commande> else <commande>
| do <commande> while (<expression>)
| while (<expression>) <commande>
| for (<expression>; <expression>; <expression>) <commande>

| function <IDENT> (<decl_args>) { <programme> }
| return <expression> ;

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

<expression> ::= <NUMBER> | <BOOLEEN> | <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>
| <expression> ? <expression> : <expression>
| <IDENT> ( <arguments> )

<arguments> ::= ε | <expression> | <expression>, <arguments>
5.0.7 Grammar of Large Fragment 6

\[
<\text{programme}> ::= \epsilon | <\text{commande}> <\text{programme}>
\]

\[
<\text{commande}> ::= <\text{expression}> ;
\]

\[
\{ <\text{programme}> \}
\]

\[
\text{var} <\text{IDENT}> ;
\]

\[
\text{if} ( <\text{expression}> ) <\text{commande}> <\text{else} <\text{commande}>
\]

\[
\text{do} <\text{commande}> \text{while} ( <\text{expression}> )
\]

\[
\text{while} ( <\text{expression}> ) <\text{commande}>
\]

\[
\text{for} ( <\text{expression}> ; <\text{expression}> ; <\text{expression}> ) <\text{commande}>
\]

\[
\text{function} <\text{IDENT}> (<\text{decl_args}>) \{ <\text{programme}> \}
\]

\[
\text{return} <\text{expression}> ;
\]

\[
<\text{decl_args}> ::= \epsilon | <\text{IDENT}> | <\text{IDENT}>, <\text{decl_args}>
\]

\[
<\text{expression}> ::= <\text{NUMBER}> | <\text{BOOLEEN}> | <\text{IDENT}>
\]

\[
\text{undefined}
\]

\[
(<\text{expression}>)
\]

\[
<\text{IDENT}> = <\text{expression}>
\]

\[
<\text{expression}> + <\text{expression}>
\]

\[
<\text{expression}> - <\text{expression}>
\]

\[
<\text{expression}> * <\text{expression}>
\]

\[
<\text{expression}> \% <\text{expression}>
\]

\[
<\text{expression}> == <\text{expression}>
\]

\[
<\text{expression}> < <\text{expression}>
\]

\[
<\text{expression}> <= <\text{expression}>
\]

\[
<\text{expression}> ||= <\text{expression}>
\]

\[
<\text{expression}> && <\text{expression}>
\]

\[
<\text{expression}> !<\text{expression}>
\]

\[
- <\text{expression}>
\]

\[
<\text{expression}>?<\text{expression}> :<\text{expression}>
\]

\[
<\text{IDENT}> (<\text{arguments}>)
\]

\[
\text{function} (<\text{dec_args}>) \{ <\text{programme}> \}
\]

\[
<\text{dec_args}> => <\text{expression}>
\]

\[
<\text{arguments}> ::= \epsilon | <\text{expression}> | <\text{expression}>, <\text{arguments}>
\]
### 5.0.8 Grammar of Large Fragment

#### <programme>

\[
\text{<programme> ::= \epsilon | <commande> <programme>}
\]

#### <commande>

\[
\text{<commande> ::= <expression> ;}
\]

| {<programme>} |
| var <IDENT> ; |
| if ( <expression> ) <commande> else <commande> |
| do <commande> while (<expression>) |
| while ( <expression> ) <commande> |
| for ( <expression> ; <expression> ; <expression> ) <commande> |
| function <IDENT> (<decl_args>) { <programme> } |
| return <expression> ; |

#### <decl_args>

\[
\text{<decl_args> ::= \epsilon | <IDENT> | <IDENT>, <decl_args>}
\]

#### <expression>

\[
\text{<expression> ::= <NUMBER> | <BOOLEEN> | <IDENT>}
\]

| undefined |
| <IDENT> = <expression> |
| <expression> + <expression> |
| <expression> - <expression> |
| <expression> * <expression> |
| <expression> \% <expression> |
| <expression> == <expression> |
| <expression> < <expression> |
| <expression> <= <expression> |
| <expression> && <expression> |
| ! <expression> |
| - <expression> |
| <expression>?<expression>:<expression> |
| <IDENT> ( <arguments> ) |
| function ( <dec_args> ) { <programme> } |
| (<dec_args>) => <expression> |
| {<cont_objet>} | Nul |
| <expression> , <IDENT> |

#### <arguments>

\[
\text{<arguments> ::= \epsilon | <expression> | <expression>, <arguments>}
\]

#### <cont_objet>

\[
\text{<cont_objet> ::= \epsilon | <IDENT> : <expressions>}
\]

| <IDENT> : <expressions>, <cont_objet> |
5.0.9 Grammar of Large Fragment 8

\[ \texttt{<programme>} ::= \ \epsilon \ | \ \texttt{<commande>} \ \texttt{<programme>} \]

\[ \texttt{<commande>} ::= \ \texttt{<expression> ;} \begin{align*}
& | \{ \texttt{<programme>} \} \\
& | \texttt{var <IDENT> ;} \\
& | \texttt{if ( <expression> ) <commande> else <commande>} \\
& | \texttt{do <commande> while ( <expression> )} \\
& | \texttt{while ( <expression> ) <commande>} \\
& | \texttt{for ( <expression> ; <expression> ; <expression> ) <commande>} \\
& | \texttt{function <IDENT> ( \texttt{<decl_args>} ) \{ <programme> \}} \\
& | \texttt{return <expression>} \\
& | \texttt{try \{ <programme> \} catch ( <IDENT> ) \{ <programme> \}}
\]

\[ \texttt{<decl_args>} ::= \ \epsilon \ | \ <IDENT> | \ <IDENT>, <decl_args> \]

\[ \texttt{<expression>} ::= \ <NUMBER> | \ <BOOLEEN> | \ <IDENT> \\
& | \ \texttt{undefined} \\
& | (\texttt{<expression>} ) \\
& | \ <IDENT> = \ <expression> \\
& | \ <expression> + \ <expression> \\
& | \ <expression> - \ <expression> \\
& | \ <expression> \ * \ <expression> \\
& | \ <expression> \ % \ <expression> \\
& | \ <expression> \ == \ <expression> \\
& | \ <expression> \ < <expression> \\
& | \ <expression> \ <= \ <expression> \\
& | \ <expression> \ &\& \ <expression> \\
& | \texttt{-} \ <expression> \\
& | \ <expression> \ ? \ <expression> : \ <expression> \\
& | \ <IDENT> ( \ <arguments> ) \}
\]

\[ \texttt{<function>} ::= \ (\texttt{<dec_args>} ) \Rightarrow \ <expression> \]

\[ \texttt{<arguments>} ::= \ \epsilon \ | \ <expression> | <expression>, <arguments> \]

\[ \texttt{<cont_object>} ::= \ \epsilon \ | \ <IDENT> : \ <expressions> \\
& | <IDENT> : <expressions>, <cont_objet> \]