Compilation's proramming assignement
JavaScript compiler

8 mai 2021

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

1 General recomandations

This project consists in compaling 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.

Howeover, there is one tool the you must absolutly use : a GIT repository. You are free to use any git server, but we recommand 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 recomand 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.

Each tagged version must :

— contain a readme with a short description of the project, a list of required softwears, how to compile it, and how to run it,
— contain a set of tests that verify each feature separatelly,
— 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" baranches should only contain the required tagged versions, wit eventually a few bug-fix commits. At the time of the evaluation, we willonly 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 "realease".

Warning : Non-requirered features are forbiden in "parser" and "master" branches. If you want to add some feature, please use a suplementary 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 shoud explain how to compile the parser. The commands required for this compiation should be available from the terminal of a linux with the correct packages. [1]

input/output The expected input of the resulting parseur should be a file containing a programme writen in the correct fragment of javascript. The output should be a sentense validating the input programme or refusing it. The error message do not need to be detailed, but bonnus points may be atributed if details are given.

3 Compiler

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

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 writen 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 (5pts)

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

- A unic base type : `Number`. Normally, JS's numbers correspond to floats, [3] but for now, only integers should be reconised.
- 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 semicolumn !
- Autorised assembly commands : `AddiNb`, `MultNb`, `NegaNb`, `SubsNb`, `CsteNb`, `Halt`

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

- plus a division 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 autorised assembly command : `DiviNb`.

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 autorised assembly commands : `Equals`, `NotEql`, `LoEqNb`, `GrEqNb`, `LoStNb`, `GrStNb`, `Not`

3. In javascript, there is no integers, only floats.

4.2 Large Fragment 1 : Variables (7pts)

Fragment 1.0 Everything from *Fragment* 0.2 plus variables :

- Variables begin with a lower-case letter and can be composed of letters, numbers, underscores "_". Edit : Le tiret "-" n'apparaissent pas dans les variables JS (c'était une typo hérité d'un vienux projet)
- Variable do not need do be declared.
- Variables are instanciated via the command `x = expr;` where `expr` is any expression. Those variables are always global.
- Variables can be used inside expressions.
- A programme is now a sequence of commands, while a command is either a variable instanciation `x = expr;` or an expression `expr;`
- Additional autorised assembly commands : `SetVar`, `GetVar`.

Fragment 1.1 Everything from *Fragment* 1.0 plus the `_++` operator. You have to respect priorities. Edit : Je ne me suis pas rendu compte que remonter, cette année, l'incrémentation aussi tôt dans le projet rendait la génération de code difficile. Nous seront donc très clément sur celle-ci, par contre le parsing doit être parfaitement réalisé.

Fragment 1.2 Everything from *Fragment* 1.1 plus the comments : Comments can be inserted anywhere and should not change the resulting compiled programme. They can be uniline (of the form `// my comment`) or multiline (of the form `/* my comment */`). No new assembly commands are autorised.

Fragment 1.3 Everything from *Fragment* 1.2 plus the floating point notations : numbers can be writen `1.215e25` or `.485e-42` or `485e-42`. We also add the `NaN` constant.

4.3 Large Fragment 2 : Conditionals (9pts)

Fragment 2.0 Everything from *Fragment* 1.2 plus the conditional :

- We add a new command `If(_) _ Else _`. Be carefull that the first hole has to be an expression, while the two others are commands.
- You can optionally add the `If_Then_` command ; it is not required because parsing both conditional can be very dificult depending of the choosen parser generator. [4]
- Additional autorised assembly commands : `Jump`, `ConJmp`.

Fragment 2.1 Everything from *Fragment* 2.0 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 commands are autorised.

4. Easy with LR parsers generators, but difficult for LL parser generators.

Fragment 2.2 Everything from *Fragment* 2.1 plus the Boolean "or" and "and" : it is a small fragment consisting in adding the two boolean operators `(_&&_)` and `(_||_)`. Be carefull to corectly follow the behaviour of the operator from JS. [5]. You have to respect priorities and associativity ! No new assembly commands are autorised.

4.4 Large Fragment 3 : Loops (11pts)

Fragment 3.0 Everything from *Fragment* 3.1 excepts that variable asignments `x = expr` is now an expression. One can thus write `2+(x=5);` (try it on your browther console !) You have to respect priorities and associativity !

Fragment 3.1 Everything from *Fragment* 3.0 plus the `do_wile_` loop. No new assembly commands are autorised.

Fragment 3.2 Everything from *Fragment* 3.1 plus the `for(_;_;_)_` and `wile(_)_` loops. No new assembly commands are autorised.

Fragment 3.3 Everything from *Fragment* 3.2 excepts that key-words are now lower-case [6] and variables can be lower- or upper-case (but cannot be one of the key-words). Warning : the current version of the mini-JS-machine may not support this feature well, it should be updated during the semester.

4.5 Large Fragment 4 : Dynamic types Loops (13pts)

Fragment 4.0 Everything from *Fragment* 4.3 plus the inplicit 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 autorised assembly commands : `Case`, `TypeOf`, `Noop`, `Swap`.

Fragment 4.1 Everything from *Fragment* 4.0 plus the inplicit 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. Once again, be carefull of the behaviour of the operators `_&&_` and `_||_`.

Fragment 4.2 Everything from *Fragment* 4.1 with a sanitarisation of the stack . You will find cases where your stack is poluted by unused expressions, for example in `x=42; x+1; x++; 25;`. In this fragment, the stack should be sanitarised. Reamark : there is no change required to the parser here.
Additional autorised assembly commands : `Drop`, `SetVaD`.

5. On Booleans only, caste are not defined yet
6. Excepts for `NaN` that stays like this.
7. Auxiliary functions may be usefull here...

Fragment 4.3 Everything from *Fragment* 4.2 plus a new type `Undefined`, with a unique value (and contant) `undefined`. Do not forget to deal with the types conversions.

4.6 Large Fragment 5 : Functions (15pts)

Fragment 5.0 Everything from *Fragment* 4.3 plus the functions :
- Recursivity is not required.
- Additional autorised assembly commands : `Lambda`, `DclArg`, `SetArg`, `Call` and `Return`

Fragment 5.1 Everything from *Fragment* 5.0 but recursivity of functions is now required

4.7 Large Fragment 6 : hosting and functional (17pts)

Fragment 6.0 Everything from *Fragment* 5.1 plus the variable declarations :
- We add a new command `var _;` where hole is a variable name.
- The declared variable will have the value `undefined`.
- The declaration can be donne anywhere, it is hosted, which means that it will happend at the begining of the programme.
- The functions are also hosted.
- Additional autorised assembly command : `DclVar`

Fragment 6.1 Everything from *Fragment* 6.0 but with a correct hosting : variables and functions that are declared into a functions are not hosted at the begining of the programme, but at the begining of the 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, please write a code that crash. [8] Additional autorised assembly command : `Error`

Fragment 6.3 Everything from *Fragment* 6.2 plus the lambda expressions.

4.8 Large Fragment 7 : Record objects (18pts)

Fragment 7.0 Everything from *Fragment* 6.3 plus the record objects. Remark : In JavaScript, an atribut of a record object can have any name, even a key-word ; this feature is not required but you can try to add it.

8. In JavaScript, functions are casted into the string of their code, this is a dangerous behaviour as it exposes the code to peaople that should not be able to access 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 carefull 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 (20pts)

Fragment 8 Everything from *Fragment* 7.2 plus the `try_catch` and `throw` structures. You are now autorise to use any assembly command you'd like.

4.10 Optional Fragments

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

Optional Fragment **Finaly** Add the `try_catch_finaly` operator.

Optional Fragment **Classes** This year, any atempt to treat classes is purelly optional. You can look at the diferent instructions available to see what can be done

Optional Fragment **TailRec** In some cases, you can replace a `Call` by a `TCall`, try to understand which cases, and to implement it.

Optional Fragment **Inference** In some cases, we can infer the type of some expression (for example if there is no variable). Try to use such a semantical analysis to later peform some optimisations.

Optional Fragment **Error** Find a way to optain readable message for you comiling errors.

Optional Fragment **Other** You can implement aditional operators of you choice :

- Ternary operator,
- String (+ escaped charaters),
- 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 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 favourit browser, you can also use some official references.

Recall that <NUMBER>, <BOOLEAN>, <IDENT> are tokens representing respectivelly 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 :

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

<commande> ::= <expression> ;
| <IDENT> = <expression> ;

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

Edit :

le ++ ne s'applique maintenant qu'à une variable, ce qui est plus simple

5.0.3 Grammar of *Large Fragment* 2

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

<commande> ::= <expression> ;
| <IDENT> = <expression> ;
| ;
| {<programme>}
| If (<expression>) <commande> Else <commande>

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

5.0.4 Grammar of *Large Fragment* 3

<programme> ::= <commande> | <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>
| <IDENT> ++

Edit : j'avais oublié les parenthèse autours du second argument du do-while.

5.0.5 Grammar of *Large Fragment* 4

<programme> ::= <commande> | <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>
| 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> ++

5.0.6 Grammar of *Large Fragment* 5

<programme> ::= <commande> | <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>
| <IDENT> ++
| <IDENT> (<arguments>)

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

5.0.7 Grammar of *Large Fragment* 6

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

<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> ::= 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>
| <IDENT> ++
| <IDENT> (<arguments>)
| function (<dec_args>) { <programme> }
| (<dec_args>) => <expression>

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

5.0.8 Grammar of *Large Fragment* 7

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

<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> ::= 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>
| <IDENT> ++
| <IDENT> (<arguments>)
| function (<dec_args>) { <programme> }
| (<dec_args>) => <expression>
| {<cont_objet>} | Nul
| <expression> . <IDENT> | <expression> . <IDENT> ++

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

<cont_object> ::= ϵ | <IDENT> : <expressions>
| <IDENT> : <expressions> , <cont_objet>

Edit : l'opérateur "." qui permet de récupérer un attribut d'un record avait été oublié