Compiler Construction

Lucas Franceschino

June 16, 2017

Contents

1 Language choice

I chose to implement my compiler in **TypeScript**, which is a super-set of JavaScript. It offers ES6 support[1], static typing, generic typing, interfaces, union types...: it makes JavaScript much stricter and safer.

The choice of that language is motivated mainly by the fact I actually use a lot JS for projects these days. It is a very flexible and nice to use language[2]. It has some "bad parts", but ES6 plus TypeScript makes them disappear.

2 Global design

This compiler is intended to be generic and not linked to SPL. Everything is split in two parts: the generic part and the SPL implementation. These statements are more or less true: for example, code generation jumps from the abstract syntax tree to SSM. Then, here no abstraction part is built, a direct SSM builder is used.

3 Lexer

The lexer is very simple: it processes the whole file and builds up a list of tokens. It does not support streams.

In order to have a generic lexer, I have a few classes that the user needs to extend to implement an actual language:

- **Lexer:** is the base of a lexer. Basically, you extend that class, let's say, `SPL_Lexer`, and then you link tokens to it.
- **Token:** take care of everything related to tokens. When implementing a language, you extend Token as much as you have different tokens to be recognize.
- **DeterministicToken:** a specific kind of `Token`.

3.1 How it works

The lexer takes an input string, then ask every registered tokens to try to match against the string, one by one by order of registration. If a token matches a string, then it is saved and the operation is repeated. Otherwise, we try another token. If no token matches, then, we raise an error.

[1]ES6 is the new version of JS, support `let`, `arrow functions`, `destructuring`, `full class support`.... For more details, see `http://es6-features.org`

[2]`http://bdcampbell.net/javascript/book/javascript_the_good_parts.pdf` and `http://javascript.crockford.com/javascript.html`

3.2 Error management

When an error is raised, it is first constructed by a method in the `Lexer` class: it takes in parameter some position in the original file, some length and a message, and format nicely a colored error.

3.3 Token and deterministic token

When adding a kind of Token, you provide just a regular expression, and optionally a paired token (for instance `(` is paired to `)` and *vice versa*). Example of token declaration (supposing `SPL_Lexer` is a defined lexer):

```
export class Integer extends Token {}
SPL_Lexer.register (Integer, /\b\d+\b/y);
```

Token provides the following methods:

- `match`: as seen previously it tries to match against a string.
- `generateRand`: generate a random string matching the regular expression of the token. This will be very useful to generate random programs.
- `error`: it takes in parameter a message and display an error using the previously described `Lexer` function.

`DeterministicToken` is a normal token, but instead of having a regular expression, it has just a string, that's why it's qualified of deterministic. That is useful for pretty printing.

An `EOF` (*end of file*) token is predefined and appends to the list of produced tokens by the Lexer. Then, when implementing a language, there is an standard way of representing *end of file*.

3.4 Paired token

Since a token can be declared as being a pair, the lexer also checks and links automatically paired tokens. Then, it avoids doing it later on when the parser will need it.

4 Parser

Again, the parser is general, and works for any given grammar.

Note

What I presented in my slides during the first two presentations is "outdated": I rebuilt from the bottom my compiler. Indeed, before, the grammar was defined by a file using an intuitive representation looking like:

```
SPL       Decl+, 'eof'
Decl      VarDecl | FunDecl as content
VarDecl    'var' | Type as type, 'id' as name, i '=', Exp as exp,
      i ';'
FunDeclT   i '::', FunType
RetType    Type | 'Void' #drop
...
```

But I chose to drop it, because it built some non-typed data. Now every rule as its own class, and everything is typed checked, hence everything is much more stable and exploitable.

4.1 Architecture

Basically, everything is run by 3 classes:

- `Parser`: global parser class.
- `ParserRule`: represents a grammar rule. Each grammar rule is represented by extending that class.
- `ParserRuleStep`: represents a step of a grammar rule.

4.2 Declaring a rule

Here is the declaration of `VarType`, with `SPL_Parser` a class extending `Parser`:

```
export class VarDecl extends ParserRule{
  @SPL_Parser.addStep()      type: LexSPL.Var | Type;
  @SPL_Parser.addStep()      name: LexSPL.Id;
  @SPL_Parser.addIgnoreTokenStep( LexSPL.Equal )
             .addStep()      exp: Exp;
  @SPL_Parser.addIgnoreTokenStep( LexSPL.Semicolon )
}
```

The decorators `@SPL_Parser.add*` are there to declare that an attribute is intended to receive some tokens or some constructs. This way, it is easy to define a grammar rule, and to use it later on. Moreover, TypeScript is able to type check everything, and also to auto complete while dealing with such entities.

One can define optional steps or repeated steps in the following way:

```
@SPL_Parser.addStep() statements: Stmt[]; // one or more Stmt
@SPL_Parser.addStep() statements?: Stmt[]; // any number of Stmt
@SPL_Parser.addStep() annotation?: Annot; // zero or one
    annotation
```

It relies on TypeScript type annotation, but the sad thing is that Type reflection at run-time is not completely supported by TypeScript[3]. Basically, TypeScript can generate (on compilation time) some decorators to give run time information about types, but only for basic build-in ones: that library is not

[3]blog.wolksoftware.com/decorators...

mature at all, so I made a little tweak to get it works. I use Grunt (a JavaScript task runner) to pre compile all my files, spot where `addStep` is used and adds type information explicitly.

From a `Parser`, it is possible to add steps using either `addStep` or `addIgnoreTokenStep` decorator[4]. A step can be a Token or an instance of some ParserRule. An IgnoreTokenStep takes in parameter a `DeterministicToken` (as defined in section 3).

You can notice that `addIgnoreTokenStep` is not bind to a class attribute. It makes sense: we don't want to have to name a field for recognizing `;` for instance: `;` is just a separator. Such a token does not contain any information, whereas an integer token would actually contain actual important information.

Here comes the notion `DeterministicToken`: it is deterministic, so that it is useless to store any information about any instance of such a token. But, while pretty-printing, we need to write something: there is a reason to have a semi-colon in a grammar, for instance.

4.3 Pretty printer

To pretty print an instance of a certain `ParserRule`, it just needs to print the `DeterministicToken`, to print the content of other tokens, and to recursively apply the same process to contained instances of `ParserRule` (if any).

There is the default way of generating the string representation of some `ParserRule`, but it can be overridden. Moreover, the indentation is managed using two decorators: `@ppIdent` and `@ppNewLine`. `@ppIdent` says that the string should be indented by one lever more, while `@ppNewLine` says the string has to treated as a "block" more than as an inline thing.

4.4 Random program generator

A `ParserRule` contains every information necessary to generate a random string corresponding to it. To do so, we just browse all declared steps, and generate random strings for each of them, and create an instance of that rule with the newly generated sub-rules or sub-tokens. If a step is optional, then we decide randomly to generate something or not, same thing if a step can be repeated.

When all of that is done, just pretty print the resulting `ParserRule`.

4.5 Parsing

As previously described, implementing a parser means:

1. have a class extending `Parser`, say `SPL_Parser`;
2. have one class per rule extending `ParserRule`.

The parsing is initiated by `SPL_Parser` from a token list. It gets the first declared rule, and makes it parse the token list. The token list is actually a `TokenDeck`.

[4]https://www.typescriptlang.org/docs/handbook/decorators.html

4.5.1 TokenDeck

A `TokenDeck` contains a list of tokens. It has a list-like behaviour, but always keeps the whole list. It just keeps track of the position of the current token. The idea is to pass this `TokenDeck` from `TokenRule` to `TokenRule`, and in the same time, keep the track of the attempts. If the main rule can't match anything, then it is very easy to find the most advanced attempts and display it.

4.5.2 `Parser` internal parsing

Parsing a `TokenDeck` *deck* from a `ParserRule` *pr* means that we want *deck* to "fit in" *pr*. *deck* being recognized by *pr* means that each steps of *pr* can parse sequentially *deck*.

A step is a list of possible things to recognize. If any of them can recognize *deck*, then *deck* is recognized by the step.

4.6 Grammar analysis & optimization

Using the previously described method to parse a `TokenDeck` is not very efficient: indeed, since it's all about recursive finding and backtracking, there are some situations where parsing is very inefficient.

In order to speed up the process, we optimize the grammar rules.

4.6.1 Left recursion

Since left recursion leads to probable infinite loops, we need to detect it and avoid it.

Before actually optimizing the grammar, we check for left recursion. If such a rule is found, the parser throws an exception.

4.6.2 Determine the n-th token to be accepted by a rule

I tried to make a function determining what tokens can be accepted by a parser at the n-th position, just by looking at the rule definitions, and I actually spent quite a lot of time on it.

The idea was to use a tree, with all the possible acceptable tokens on the n-th level of that tree. I used a sort of shadow tree: it was actually updating sub-trees only when necessary, and keeping everything as references, to handle recursion (since the rules are themselves recursive).

But after a few days of full work on that idea, I just got some almost working algorithms, but I always ended up having issues, like some tokens were missing from some places in the tree.

I eventually decided to stop this experiment there, since it took me too much time already.

4.6.3 First token

Instead of being able to predict the n-th possible tokens, I chose just to look up for the first possible tokens, which is very straightforward.

Then, instead of trying each different possibility in a rule, it is now possible to test only the one with a matching first tokens.

To optimize that, I use hash-tables from token kinds to possible rules for each step of each rule. Then, given a token, we get a list of possible rules to apply, and then we avoid testing anything but necessary things.

4.6.4 Paired tokens or how to have easy multiple errors handling

As explained in section 3.3, a Token can be linked to another one to declare a pair. Since then, the parser treats these tokens as pair, hence, an opening something has to match with a closing something. The parser actually uses that fact.

Indeed, when trying to parse in the context of some `ParserRule` *pr*, if the parser detects a pair token, it matches its closing token, extracts[5] the section and parses it independently.

This only happens if - in the context of *pr* - there is only one possible grammar rule that can recognize the pair expression.

For instance, parsing `(3) + 4` when the only rule that can be applied is `Parenthesis = '(' Exp ')'`, `(3)` is extracted and parsed independently. Here, we know it has to match. If it doesn't match, we know that what was inside the parenthesis is wrong. Then, we raise an error, without stopping parsing.

4.6.5 Paired token optimization

Consider the following expression `((...(3)...))`: we don't want the parser to do the usual backtracking thing here. Indeed, the rule `Parenthesis = '(' Exp ')'` consists of exactly 3 steps : opening, inside rule, closing. If the inside rule is not of this "opening inside closing" form, then we can just parse the expression in a linear way, but extracting `3`.

4.6.6 Same thing without pairing: `getInsideTokens`

A `ParserRule` implements a method `getInsideTokens` that lists all the possible tokens inside it, but the one on the sides.

Using this information, in some context where some rules $r_1, r_2, \ldots, r_n$ could apply on one "entry" token t_1, and where all of these rules end with a token t_2,

Then, if some rules $r_1, r_2, \ldots, r_n$ have o as opening tokens, and c as closing token, but without c being in the set of the `insideTokens`, then we can extract the value and "bootstrap" the parsing operation just as described previously.

Again, error occurring inside this extracted tokens will be reported without stopping the parser.

[5]The extraction is immediate and in linear time, as explained in section 3.4, the matching is done previously by the lexer

Hence, no misleading parsing error is generated: we never assume anything about failing rules.

4.7 Grammar pretty-print: SPL example

`ParserRule` can look-up its own definition and pretty print the grammar rule defined using class extention. Using that, it gives an easier form of the grammar and hence an easier way of reading / "debugging" the grammar.

For instance, here is the grammar used for `SPL`:

```
SPL:           Decl, EOF
Decl:          VarDecl | FunDecl
VarDecl:       'var' | Type, Id, '=', Exp, ';'
VarOrFunDecl:  VarDecl | FunDecl
FunDecl:       Id, '(', FArgs, ')', FunDeclT, '{', VarDecl,
    FunDecl, Stmt, '}'
FunDeclT:      '::', FunType
RetType:       Type | 'Void'
FunType:       Type, '->', RetType
TupleType:     '(', Type, ',', Type, ')'
ListType:      '[', Type, ']'
Type:          BasicType|LambdaType|VarType|TupleType|ListType|Id
VarType:       '(', 'var', Id, '::', VarType_C | 'any', ')'
LambdaType:    '(', FunType, ')'
VarType_C:     Type
FArgsOpt:      Id, ','
FArgs:         FArgsOpt, Id
Else:          'else', Block
If:            'if', '(', Exp, ')', Block, Else
Block:         BlockAcc | Stmt
BlockAcc:      '{', Stmt, '}'
While:         'while', '(', Exp, ')', Block
Assign:        Id, Field, '=', Exp, ';'
Call:          FunCall, ';'
Ret:           'return', Exp, ';'
Stmt:          Assign | If | While | Call | Ret
ExpVar:        Id, Field
ExpOp2:        ExpNOp2, BinaryOperator | '-', Exp
ExpOp1:        '!' | '-', Exp
ExpPar:        '(', Exp, ')'
ExpTuple:      '(', Exp, ',', Exp, ')'
Exp:           ExpOp2 | ExpNOp2
ListCst:       '[', Exp, ']'
ExpNOp2:       ExpOp1 | ExpPar | ExpTuple | Integer | Bool |
    FunCall | ListCst | ExpVar
Field:         '.', FieldOperators, Field
FunCall:       Id, '(', ActArgs, ')'
ActArgsOpt:    ',', ActArgs
ActArgs:       Exp, ActArgsOpt
```

5 Type inference

The type inference system was designed to be fully independent from SPL, but I had to make some compromise, and then it was oriented to make it work well later on for generating SSM code. Some behaviors are influenced by SSM.

5.1 Architecture

The type inference relies on two concepts: Contexts and Types.

5.2 Dealing with `ParserRule`

We need a way to make a correpondance between instances of `ParserRule` and type information. In each context, there is a reference to an unique mapping from `ParserRule` to some type and some context or to `undefined` (if the `ParserRule` has no type).

5.3 Context

A context represents - more or less - a scope. Hence, it has a list of identifiers (strings) that points to some `ParserRule`.

It also has a list of variable types, which are named types that can take some concret types.

5.3.1 Variable declaration

When declaring a variable, we add a new identity and link it to a `ParserRule`. Then, the type of that variable will be the type of the `ParserRule`, resolved using the mapping described in 5.2.

Also, it is possible to give the context some hints about what the value is: is that an argument or a local value? Also, we can decide whether the variable is a reference or as a direct value[6].

5.3.2 Get a variable by name

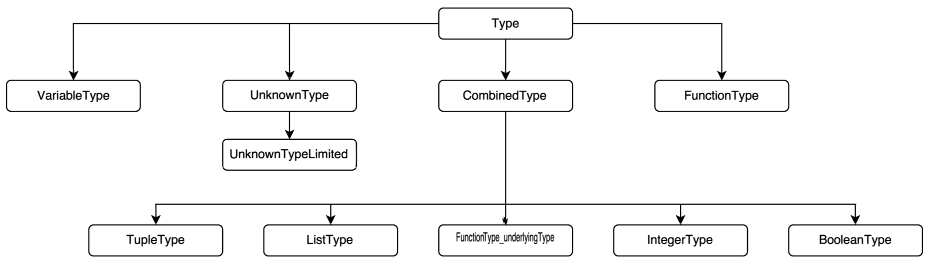
Each time we want to access a variable, we need to provide an identifier and a `ParserRule`. That `ParserRule` is the node that needs the variable. With this information, the context is alterated with the information that some `ParserRule` access to some variable.

Moreover, each time a variable is fetched, the context first checks if it is a local variable: if it is not the case, it just asks its parent. In that second case, it also updates a list of "outside variables" used in the context. Hence, we keep track of nested variable access, in order to implement closure easily later.

[6]Here, clearly, it's done to make SSM generation easier

5.4 Types

Here is the diagram of the types:



5.4.1 `Type` class

This class is abstract and corresponds to the normal scheme of a type. It has an unifying method, equality between types, an order, and some other handy functions.

5.4.2 Unknown type

This class represent a type we don't know anything about. At the first unifying with any type t, it will unify and merge the unknown type with t. It has an *UnknownTypeLimited* variant, which is basically the same, but it's narrowing the type at some list of other types. In that case, we know more about the types.

5.4.3 CombinedType

This is the most important type. It's an abstract type that can contain any number of inner types: this is represented as a list of `Type` (there are no constraints about the length of that list). When this type is extended, then it will create another type.

A `CombinedType` type a unifies with another `CombinedType` if and only if:

- a and b come from the same class;
- a and b have the same number of stored types;
- stored types in a and b have to unify pairwise.

5.4.4 TupleType

`TupleType` extends `CombinedType`. The constructor needs two arguments, a left type and a right type: it stores these types in its inner type list (from `CombinedType`).

5.4.5 ListType

`ListType` has a constructor taking just one type: the one contained in the list. It pushes this type in the inner type list (from `CombinedType`).

5.4.6 FunctionType

The constructor takes any number of input argument types and one output type.

It builds two `FunctionType_UnderlyingType` (extending `CombinedType`): one with arguments and output types in the inner list, the other one only with inputs.

Then, when unifying, it just uses underlying `CombinedType` types own unifications with some twist to guess the output type in case of a generic function for example.

5.4.7 Integer/Boolean type

Integer type does not contain any type information, the constructor takes nothing, and puts nothing on the inner list. Here if an integer type a unifies with some type b, then b is an integer, that's all.

The exact same holds for booleans.

5.4.8 Type order

A type order is a tuple of two numbers. It measures the "concretness".

5.4.9 Type unification

Every type is immutable, in order to avoid conflicts with changing something somewhere and not elsewhere.

When we unify a type a with another one b, we obtain a list of replacements. A remplacement is a tuple of types: (c, d) means c has to be changed to d.

Each time types are unified, we ask the linked context ctx to unify everything itself. First, ctx finds the context ctx_{origin} where the variable to unify was declared first, then ctx_{origin} browses all `ParserRule` to apply the replacements.

5.4.10 Combined type unification

As described previously, a can unify with b only if it comes from the same class, has the same number of inside types, and inside types of a and b unify pairwise.

Here, something bad can happen: if we unify (Int, b) with (a, a) (with a an unkown type and b a variable type), we want to deduce (b, Int). But if we unify first Int with a, we get (a, Int). Then we unify b with a (which is now Int), and this is not working.

To avoid that issue, I sort these unifications according type orders: we unify the less concrete ones in priority.

5.5 Add typing logic to the different `ParserRule`s

I didn't want the type infering interfers with previous definitions of `ParserRule`. Instead of overloading every `ParseRule`, I just define a mapping from `ParserRule` classes to typing logic.

So basically, there is a big list of functions taking care of the typing logic of each `ParserRule` that needs one.

Some `ParserRule` classes don't need any typing logic, for instance `if` or `while`. In this case, the type checker just digs into the `ParserRule` definition and types check sub constructions. Hence, everything is verified, even if nothing is explicitely stated: this way, no any part of the program can be unchecked.

I will explain some typing logic I find interesting.

5.5.1 Function declaration

Typing logic of function declaration is interesting because it needs to figure out if the function is well-typed, if it returns something, if the signature (if present) is making sense, etc.

Pre treatment First, I declare the identifier of the function in the context: this is useful because I want my compiler to support high order functions.

Then I extract the arguments and the annotations to check if it is consistant (same numbers of parameters than annotations)

Verifying returns paths We need to ensure that the function actually returns for all different logical paths. For that, I use the following structure: `{mabye: Ret[], concl: Ret[]}`. Basically, `concl` represent some return statements like `if C then return A else return B`: it will be either A either B. `concl` represents returns like `if C then return A`.

I defined some compose functions taking two structures like that and outputing a new structure:

```
type X = {maybe: Ret[], concl: Ret[]};
let compose = (A:X, B: X) =>
  A.concl.length && B.concl.length ? {maybe: [..A.maybe,..B.maybe
      ], concl: [..A.concl,..B.concl]} :
  A.concl.length == B.concl.length ? undefined : {maybe: [..
      A.maybe,..B.maybe,..A.concl,..B.concl], concl: []};
```

Looking for type cycle dependencies I build a dependency graph to check the dependencies between objects (functions or variables). If an ambiguous situation is spotted, then an error is raised and the user has to annote the function.

Previously, if a cycle was found, I was using other return statements to guess the type, but that was quite buggy. Also, now I'm making a cycle dependency of every resource used inside a function, then I have no troubles when I type check the function body.

Arranging the order of type check The user is allowed to declare a function calling another function defined later on. That is not a problem, but if you ask the type checker to type the first function before typing the second one, it will fail.

The previous step just eliminates situations where we have recursive type dependencies: hence it is not a graph but a tree, we don't have any cycle there. Then we can just order all the resources to type check the most needed things first.

To do so, I use the tree of dependency and I just type check everything from the bottom of the tree to the top. Hence, I'm sure all dependencies will be satisfied.

Check if the output is making sense Executing type checking modified the context, and hence, the output type. If the output type was given by annotation, I just need to check that the result is coherent. If there were no any annotations, then, I check if the output changed to a more concrete one.

5.6 Note

`FunctionType` is just a normal type and can be used everywhere. `VariableType` make polymorphism and generic programming possible.

Hence, the type inference is strong enough to handle higher order functions.

6 SSM Code generation

Using everything done before in the type checker makes it quite easy.

6.1 Scopes

A scope correspond to all the (functions and variables) declarations of a function. Each declaration records exactly one new value on the stack and will be considered as the $n-th$ local value.

A value can either be a basic type (integer, boolean) either a composed type (tuple, list). A basic type is stored directly on the stack as a value, while a composed one store a pointer to the heap.

Hence, when you access to a local variable, you can get either a pointer either a value, according the expected type (known at compiler time).

But since we handle closure, we also need (that is explained later on) a way to store a simple value or a pointer to a complex value in the heap. Hence, a local variable can also be a pointer to a value.

The local variable positions are calculated by the `Context`. A method in `Context` gives information about any stored values, and compute the local position (relative to `MP`). It handle the case of parameters, local values, and exteral values linked by closures.

The scope is organized on the stack in the following way (on the left are represented the local positions):

-n'-n	Argument 1	
	...	
	Argument n	
-n'	Outside variable 1	
	...	
-1	Outside variable n’	
0	Previous PC	MP
1	Previous MP	
2	Local variable 1	
	...	
n''	Local variable n”	
n''+1	"Working" area	SP

6.2 Closure support

When a function references a variable from some parent scope, I lift that variable to the heap: instead of having value (i.e. if it’s an integer) or a pointer (i.e. if it’s a list) in the local values, we will have a pointer to either a value or another pointer in the heap. That allows the value to be existing outside the local scope of some functions.

Then, I use a structure to store these pointers along with the address of the closure.

6.3 Function pointer structure

This is basically just a package with some hidden parameters (the closure "outside variables") and with the PC. It has the following form:

Number of outside variables
Outside variable 1 address
...
Outside variable n address
Function address

6.4 Declaring a variable

There are two scenarios here:

The variable is lifted to the heap Here, we push on the stack the variable expression result - which is a single value (that can be a pointer or a value) - then we store it somewhere in the heap, and the resulting pointer will be our value, stored in locals (on the stack, then).

The variable is not lifted We do almost the same thing, but we skip the part where we put the resulting value on the heap. Here, we want the value to be stored directly on the stack.

6.5 Setting a variable

Again, two scenarios:

6.5.1 Lifted variable

Before setting a new value in some variable, we have to jump to the pointer specified by the variable in locals. Then, we can set the value. Here is the precise way of doing it:

- Push the new value on the stack;
- Save MP to RR;
- Load the variable pointer from locals;
- Set MP with top value of stack;
- Store in local (which is now the location pointed by the variable) the new value (which is on the stack);
- Restore registers.

6.5.2 Not lifted variable

We just store the new value to locals.

6.6 Consulting a variable

To get a variable in position p, we need:

If it's a lifted variable Load using a local referencing an address (with `ldla` p)

Otherwise Simply load a local value (with `ldl` p)

6.7 Declaring a function

When we declare a function, we need to write on the stack the corresponding `Function pointer structure` (defined in 6.3), and to store it somewhere. After doing so, the resulting address will be on the locals, just like any other variable.

6.8 Calling a function

Let f (a function pointer structure) be in the locals, then executing f means:

- Load the value pointed by f: we retrive the number of outside variables n
- Load the $n+1$ next values after f: we retrive the outside variable pointers along with the program address
- Save the program address in RR (we don't want it to stay there on the stack)
- Push to the stack all the parameters
- Save current PC on the stack
- Store the value of RR into PC (i.e. branching)

6.9 Global scope

I consider the global scope as a normal scope, as a normal function. So, basically, every top levels objects (function or variable declarations) behave as it was inside one big function.

Hence, I don't need to rebuild some different dependencies analysis or anything: all objects belongs to some locals scope, but this "big fonction".

Then, on the top level, on the global level, we have only one object, hence, no dependencies issue can occurs.

7 Conclusion

This course was very interesting for me, since it was the first time I built an actual compiler. I previously made some small parser to handle simple languages, but nothing serious.

Working on that compiler was very interesting, even if it was quite time consuming. I feel much more confortable with how a compiler is working, now.