Chapter 1

The Opal Compiler

Jorge ►*I think that the first thing that we need is a description of the Opal model, with the different classes that take part in it. Then we can explain how the compilation and decompilation work. And also explain the intermediate representation. Finally, how to extend and change the compilation process.*◄

Opal is a Smalltalk to Bytecode compiler for Pharo. This project was initiated to replace the original compiler, which slowly evolved from the one developed in the 80s. It was designed in one process Scanner/Parser. The result is that the old compiler is hard to understand, to extend, and completely unadapted to the new needs. ~~That's why the Opal project started~~ →This is why a new flexible, configurable and adaptable compiler was needed, Opal fulfill this need..

The Opal compilation process, is built around 3 steps, from source code to the bytecode.

1. Source code to abstract syntax tree,
2. Abstract syntax tree to intermediary representation,
3. Intermediary representation to bytecode.

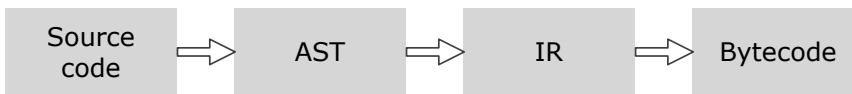

Figure 1.1: Opal complete compilation process.

Loading Opal

Right now we should load it to be able to execute the code snippets used in this chapter.

```
Gofer new
   squeaksource: 'OpalCompiler';
   package: 'ConfigurationOfOpalCompiler';
   load.

(Smalltalk at: #ConfigurationOfOpalCompiler) load.
```

1.1 Build of the Abstract Syntax tree

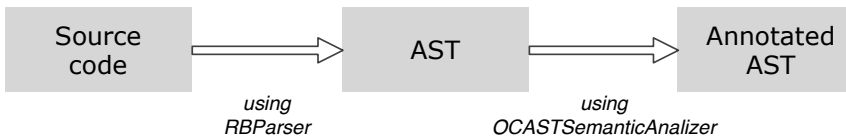

Figure 1.2: Source to AST Annotated Jorge ►*what is RBParser and OCASTSemantic-Analyzer? we never explained that*◄

The Abstract Syntax Tree (AST) is a tree representation of the source code. The AST is easy to manipulate and scan it. Jorge ►*Should we assume that the reader knows what the visitor pattern is? at least we should have a reference.*◄. JB ►*We should define what is the design pattern visitor but maybe not here, is not really specific to AST it's specific to ours implementation*◄. The AST used by Opal comes from Refactoring Engine Jorge ►*Include reference to the Refactoring Engine chapter.*◄. It uses RBParser to generate ASTs, this step verifies the syntax. The structure of an AST is a simple tree. Evaluate and explore the following expression:

```
t := RBParser parseExpression: '1 + 2'.
"explore it"
```

Stéf ►*so what do we see? we should add t something* ◄ Using the message parseExpression: we get an AST representing the expression.

Let's try another example

```
RBParser parseExpression: 'one plus: two'.
"inspect it"
```

```
RBParser parseExpression: 'one plus: two plus: three'.
"inspect it"
```

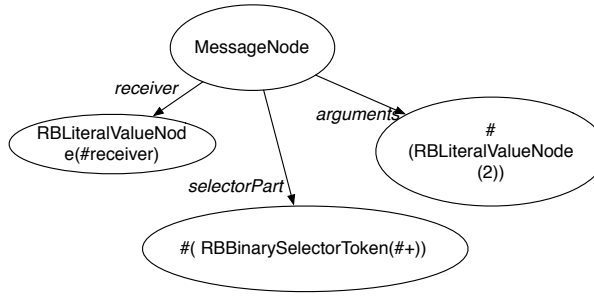

Figure 1.3: Generated tree for ’1 + 2’

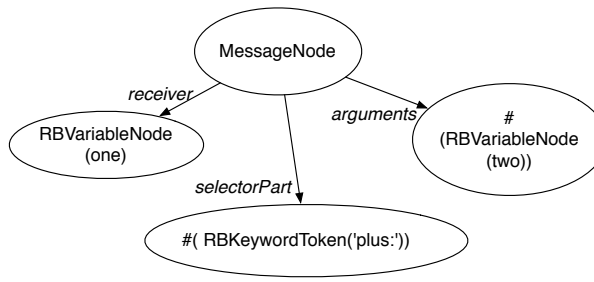

Figure 1.4: Generated tree for ’one plus: two’

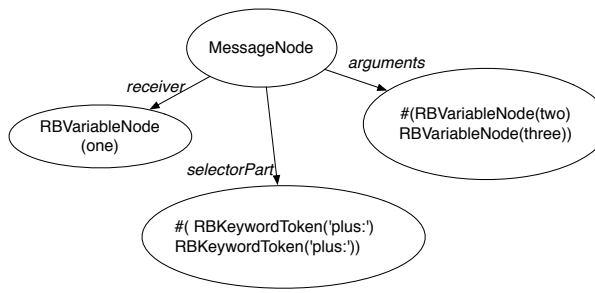

Figure 1.5: Generated tree for ’one plus: two plus: three’

You can also parse a the code of a method using the message parseMethod: instead of parseExpression:. We will get a methodNode object:

```
RBParser parseMethod: 'xPlusY x + y'.
"inspect it"
```

Stéf ►*so what do we see? we should add t something* ◄

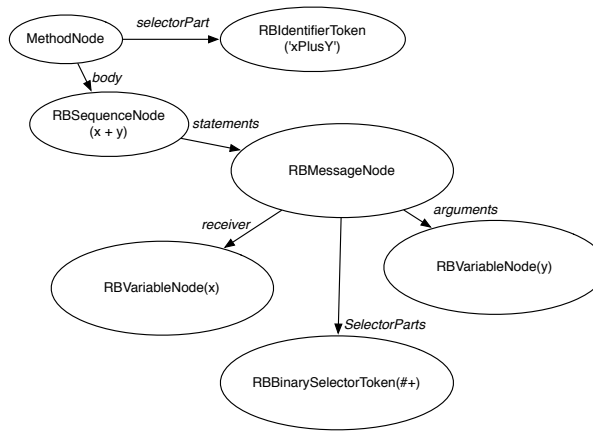

Figure 1.6: Generated tree for 'xPlusY x + y'

Annotating the Abstract Syntax Tree

Once parsed we can perform a semantic analysis of the AST. The goal of a semantic analysis is to add semantic data to the generated tree. One of the key function of the semantic analysis is to bind variables. Because as we saw before the AST only checks the *syntax* of the code. A semantic analysis checks the semantics of the code: if in context of a class, the code is valid. We can identify if a variable is undeclared or used out of scope, or if a message is send to the UI.

The AST is annotated by visiting the graph by two visitors:

- OCASTSemanticAnalyzer performs the variable binding.
- OCASTClosureAnalyzer performs the closure analysis **Jorge** ►*What is the closure analysis?*◄.

Let's try an example:

```
|ast|
ast := RBParser parseExpression: '1 + 2'.

"visit and annotated the AST for the closure analysis "
OCASTClosureAnalyzer new visitNode: ast.
"visit and annotated the AST for the var binding"

OCASTSemanticAnalyzer new
      scope: Object parseScope;
      visitNode: ast.
ast. "inspect It"
```

Jorge ►*Maybe we need to break this into pieces and explain each of them, or enumerate the lines.*◄

Stéf ►*what do we get?*◄

All the data of binding is injected in the AST, when you inspect your AST you can see the value properties is now set to a dictionary. Stéf ►*show it*◄ JB ►*In addition some class need to be compile in a specific way (Context ... need a long form bytecode because the bytecode is used by (Stack and Cog)VM), and it is at this level two the properties is set we should say one word about that* ◄

1.2 Intermediate Representation

JB ►*In many case people will not manipulate IR, we should just explain when they should. You want to change jump, closure or push of the temp. It is a this level, if you want to indirect all the instance var access it's at the level of the AST.we should simply rewrite this part, the IR is for : 1 different bytecode plug, 2 Bytecode optimization(easier to manipulate,more accurate, and it's more coherent), 3 small grain manipulation. We should explain that without going to much in detail.the question is if we don't explain the utilities of IR because is to low level why have a IR in the process of compilation why not directly do bytecode.*◄

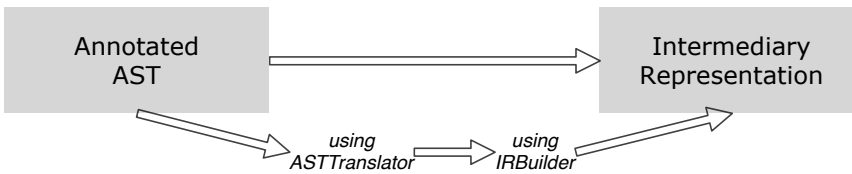

Figure 1.7: AST Annotated to Intermediary Representation Jorge ►*The captions should be a sentence that adds some information otherwise they are meaningless*◄

JB ►*to redo*◄

Stéf ►*we need a code snippet*◄

1.3 Bytecode

Once we have an IR tree, we will transform the IR tree in a bytecode sequence. We apply a new visitor but on the IR tree this time. Since IR is close to bytecode, the visitors visits each node and pushes the corresponding bytecode.

Stéf ►*we need a code snippet*◄

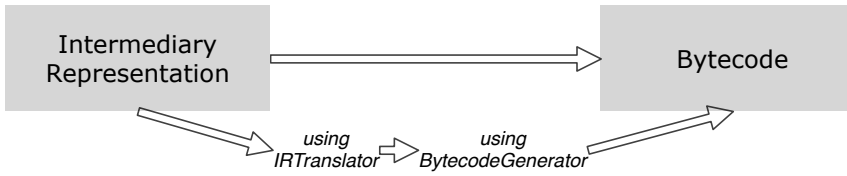

Figure 1.8: Intermediary Representation to Bytecode

1.4 Decompilation

Such the compilation process