LINGI2132

Languages and Translators

Emilyen Laffineur

Teacher :
Nicolas Laurent

Assistant:
Mathieu Jadin & Alexandre Dubray

Ecole Polytechnique de Louvain
Universtié catholique de Louvain
Belgium
Scholar year : 2020-2021

Contents

Chapter 1

Introduction

This class is about compiler. The goal of compiler is to implement a programming language. It take source code and make it executable (or pass it to a program that execute it). We may think that language like Python or JavaScript does not use compilers however they use it as well! Even if they are dynamical. Note that this class is not about programming paradigms, it's truly about implementing languages in general.

We will learn about compilers and their parts, allowing us to understand how languages works. We will also learn some useful patterns (Domain Specific Language (DSL), trees, interpreter pattern).

Chapter 2

Compilation Pipeline

Here is the classical compiler pipeline :

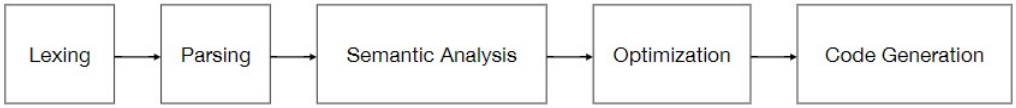

Figure 2.1: Compiler pipeline

We will go through each step.

2.1 Lexing

Definition 2.1.1 (Lexing)**.** It's a lexical analysis, tokenization, scanning. It take the text and produce tokens (= "words"). e.g : foo = bar + 42 will produce |foo| = |bar| + |42|.

2.2 Parsing

Definition 2.2.1 (Parsing)**.** It take the output of lexing (tokens) and extract it's structure (by producing an Abstract Syntax Tree (AST)). It also catches syntax errors (like missing parenthesis around if in Java). Other kind of errors (like something that is not defined by the language) is not a syntax error, thus, the parser cannot catch it.

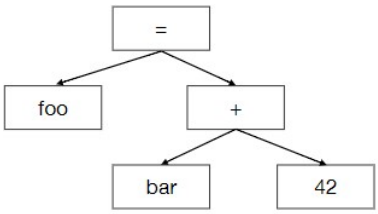

Figure 2.2: Abstract Syntax Tree

Note that lexing is optional while parsing (called scanner-less parser) for others parsing is actually lexing AND parsing.

2.3 Semantic Analysis

It consists of a series of check that is perform on the AST, two mains are Name binding et type checking.

2.3.1 Type checking

Definition 2.3.1 (Type checking)**.** Verify that the type passed is what is expected.

```
1 void print(String str)
2 print(42)
```

Definition 2.3.2 (Type inference)**.** Let the compiler guess the type of the variable given the context.

Note that type checking is inference. When :

```
1 String x = "foobar"
```

is used, it first need to guess what is "foobar", then check the consistency with the type.

Sometimes, we need to know the return value's type of a function to perform the check. That's what Name Binding is.

2.3.2 Name Binding

Definition 2.3.3 (Name binding)**.** Allow us to infer the type of a variable given the return type of a function.

Note that it becomes much more complex with OOP.

2.3.3 Rest

It also performs flow check (missing return for example), access control (public, private, etc.).

2.4 Optimization

Actually, the pipeline we saw in figure 2.1 it's only the old school C compiler. New C compiler, do optimization on the produced machine code (LLVM, Low Level Virtual Machine)! Do not get confuse, it does not use a virtual machine at all... LLVM generate IR (intermediate representation), optimize it and then generate machine code.

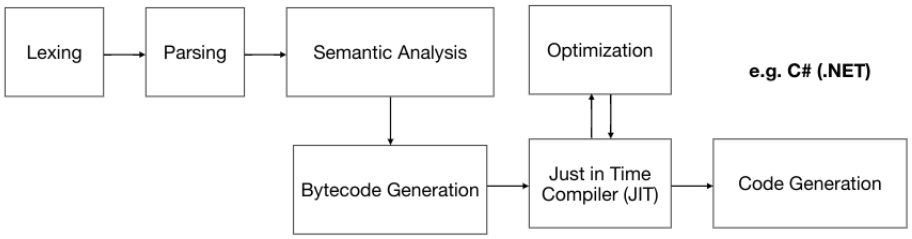

Figure 2.3: C# pipeline

This one generate bytecode (Java name), that use usable but that won't be directly use by the machine. After the generation of the bytecode, it is passed to a Just In Time Compiler (JIT).

What is done until JIT is done at compilation time, JIT and the following is done while running the program!

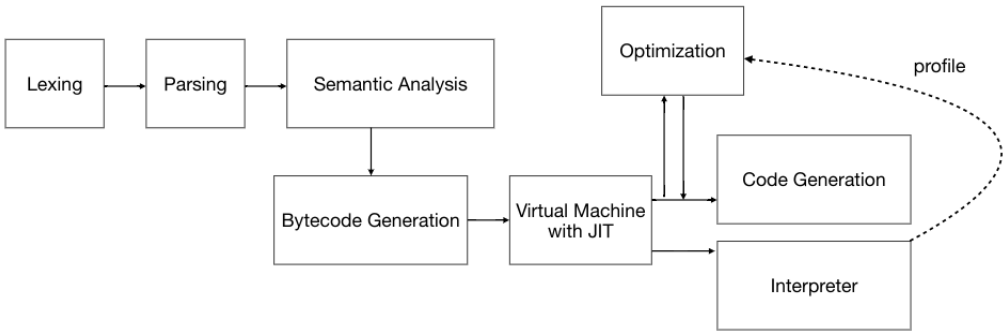

Figure 2.4: Java and Python pipeline

That's the architecture that most recent languages uses. The bytecode is passed to a VM executing a JIT. This VM can generate machine code (code generation) or pass it to in interpreter which will run a runtime profile that is much better for optimization! That the reason why Java can be as fast (or faster) the C. Pypy (Python), Java, TruffleRuby use this kind of architecture!

We could also not optimize it! We could directly use the Tree-Walk interpreter, Ruby before 1.9 did that. CPython (standard one), and Ruby generate the bytecode and run it in a VM.

2.4.1 Tree-Walk Interpreter

Let's take the following program :

```
1 var foo = 42 + 52
2 print(foo)
```

The AST is the following :

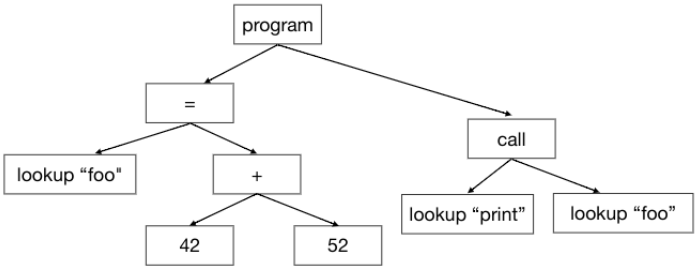

Figure 2.5: Resulting AST

(Note that print has is own AST). We have the scope, containing the value of foo saved in the store and the definition of print. The procedure in order to execute is recursive, it will lookup for foo, perform the addition, store it. Lookup at the print definition, lookup for foo and foo will be passed as a parameter to print.

2.5 Code Generation

It could be machine or bytecode generation. Example of Machine Code (e.g. x64) :

```
1 int square(int num) {
2     return num*num
3 }
```

Becomes :

```
square(int):
    push rbp
    mov rbp, rsp
    mov DWORD PTR [rpb−4], edi
    mov eax, DWORD PTR [rpb−4]
    imul eax, eax
    pop rbp
    ret
With optimization it becomes :
    mov eax, edi
    imul eax, edi
    ret
```

While bytecode generation is:

```
public class Hello {
    int square(int num) {
        return num∗num
    }
}
```

Which becomes (stacked base instead of registry based as before):

```
public class Hello {
    public <init>()V
      L0 LINENUMBER 1 L0
       ALOAD 0
       INVOKESPECIAL java/lang/Object.<init> ()V
       RETURN
      L1
       LOCALVARIABLE this LHello; L0 L1 0
       MAXSTACK = 1
       MAXLOCALS = 1
      square(I)I
       L0
         LINENUMBER 3 L0
         ILOAD 1
         ILOAD 1
         IMUL
         IRETURN
       L1
         LOCALVARIABLE this LHello; L0 L1 0
         LOCALVARIABLE num I L0 L1 1
         MAXSTACK = 2
         MAXLOCALS = 2}
```

2.6 Optimizations

As we have seen, we can optimize on tree or on target code. They are two types of optimization that are the base of everything :

- Inlining : pulling a function in another one
- Partial evaluation : propagating known information (e.g constant-folding)

Some other exists like loop unrolling, etc.

2.6.1 Inlining

Take the following program :

```
1 var foo = false
2 int a(){
3     if (foo) return somethingLongAndBoring()
4     else return b(true)+1
5 }
6 int b(boolean bar) {
7     return bar ? 42 : 52
8 }
```

If we optimize the function a. It we online somethingLongAndBoring and b. However, if we do this and do not enter the condition, this could be bad for cache locality. Indeed, it will lead to a cache miss, with a lot of code, the time to retrieve the actual code will be huge. However, we know that foo is always false so we can get rid of the first condition! Then we can inline b. Thanks to the inlining of b, then constant folding it will lead on only this :

```
1 int a(){
2     return 43;
3 }
```

Which is much more efficient! Thanks to that we can see that there is a complementary between constant folding and inlining.

Chapter 3

Formal Grammars

3.1 Language vs Grammar

Definition 3.1.1 (Grammar)**.** The (formal) definition (*description*) of a language

Definition 3.1.2 (Language)**.** A (potentially infinite) *set* of sentences, in programming language, a sentence is a source file, REPL expression, etc.

Definition 3.1.3 (Alphabet)**.** Composed of tokens, lexemes.

Let's take an example which is JSON. The language is all valid JSON expression. Sentences : e.g.

```
{
    "version": 17,
    "bundles": [
        { "name" : "org.graalvm.component.installer.Bundle" },
        { "name" : "org.graalvm.component.installer.commands.Bundle" },
        { "name" : "org.graalvm.component.installer.remote.Bundle" },
        { "name" : "org.graalvm.component.installer.os.Bundle" }
    ]
}
```

Figure 3.1: JSON Sentence

Grammar :

```
VALUE   ::= STRINGLIT / NUMBER / OBJECT / ARRAY
OBJECT  ::= "{" (PAIR ("," PAIR)* )? "}"
PAIR    ::= STRINGLIT ":" VALUE
ARRAY   ::= "{" (VALUE ("," VALUE)* )? "}"
```

Alphabet (Tokens)

Figure 3.2: JSON Grammar

3.2 Grammar notation

3.2.1 PEG

Parsing Expression Grammar. The "?" denotes optional, "*" means 0 or more times.

```
VALUE   ::= STRINGLIT / NUMBER / OBJECT / ARRAY
OBJECT  ::= "{" (PAIR ("," PAIR)* )? "}"
PAIR    ::= STRINGLIT ":" VALUE
ARRAY   ::= "{" (VALUE ("," VALUE)* )? "}"
```

Figure 3.3: PEG notation

PEG is a grammar formalism and a notation.

Definition 3.2.1 (Formalism)**.** Mathematical system to define the language (set of sentences)

Definition 3.2.2 (Notation)**.** A way to denote a grammar to be interpreted by the formalism.

3.2.2 EBNF and BNF

Note that (E)BNF is a notation for CFG (Context-Free Grammars).

EBNF

Extended Backus-Naur Form. "[]" means optional, "{}" means 1 or more times, so in order to make "0 or more" with have to combine symbols.

```
VALUE   ::= STRINGLIT | NUMBER | OBJECT | ARRAY
OBJECT  ::= "{" [ PAIR [{ “,” PAIR }] ] "}"
PAIR    ::= STRINGLIT ":" VALUE
ARRAY   ::= "{" (VALUE [{ " ," VALUE }] ] "}"
```

Figure 3.4: EBNF notation

BNF

Backus-Naur Form. It is at the base of EBNF, it does not contain "{}" or "[]", so it uses recursion and a lot more rules. Note that ϵ is the mathematical notation for "nothing"

```
VALUE   ::= STRINGLIT | NUMBER | OBJECT | ARRAY
OBJECT  ::= "{" PAIRS? "}"
PAIRS?  ::= ε | PAIRS
PAIRS   ::= PAIR TPAIRS
TPAIRS  ::= ε | "," PAIRS
PAIR    ::= STRINGLIT ":" VALUE
ARRAY   ::= "[" VALUES? "]"
VALUES? ::= ε | VALUES
VALUES  ::= VALUE TVALUES
TVALUES ::= ε | "," VALUES
```

Figure 3.5: BNF notation

3.2.3 Summary

CFG and PEG are different formalism. In some case, PEG and CFG can be the same (like in JSON) because it is a very simple definition. By it is not true in general. Sometimes, we can use PEG notations like +, *, ? in CFGs (coming from REGEx, which are related to CFGs.). Actually we can use any notation that we want, we just need to define it!

Chapter 4

Parsers

Definition 4.0.1 (Parser)**.** A parser is a program that :

- Accepts/rejects input a sentence in the language (recognizer)
- Extracts a(n) (abstract) syntax tree

4.1 Parsing Tools

A parsing tool lets you create parsers. We can denote two kinds : Generator an Library. Note that is just a generalization. The main distinction to retain is the generation vs interpretation.

Parsing tools are often called "parsers", it is acceptable, but incorrect, parsing tools create and/or run parsers.

Parsing tools are compilers!

- Language = grammar notation
- Code generation (parser generators) or interpretation (parsing libraries) (depending on the used tool)
- Domain Specific Language (DSL) (depending on how we define it. DSL is opposed to Turing-Complete language like Java, Ruby, Python, etc. A DSL could be XML, JSON, Grammar, SQL, etc.)

4.1.1 Parsing Generator

Note : in the following figure, all solid boxes denote programs.

Definition 4.1.1 (Parsing Generator)**.** A Parsing Generator is :

- Outputs a parser program (typically source code)
- Typically a command line tool
- Typically uses own grammar language

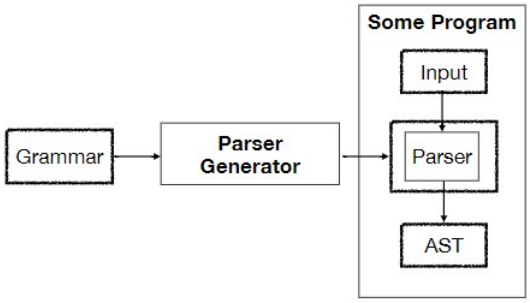

Figure 4.1: Parser Generator

4.1.2 Parsing Library

Definition 4.1.2 (Parsing Library)**.** A Parsing Library is :

- Let us "interpret" a grammar
- Grammar typically defined with a DSL.

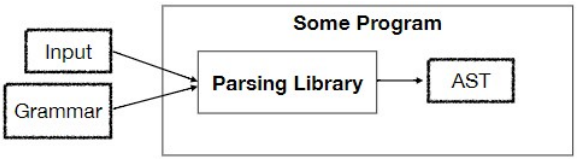

Figure 4.2: Parser Library

Note that input AND grammar could be in the "Some Program" box.

4.2 (Abstract) Syntax Tree

4.2.1 Syntax Tree

Remember the notation of figure 3.2 and 3.1. To make it easier for Syntax Trees we will use the figure 3.5. Doing so, we can extract the following ST :

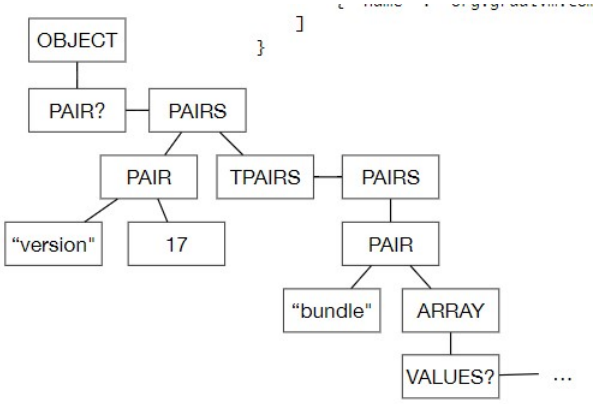

Figure 4.3: Syntax Tree

Syntax Tree is the following : for each phrase in the sentence we follow the grammar. It is the only thing we can do!

4.2.2 Abstract Syntax Tree

Here is the same Syntax Tree but abstracted :

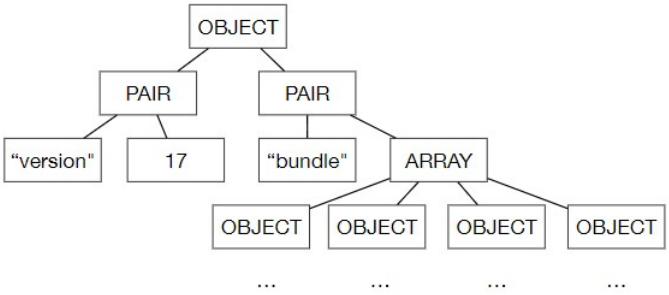

Figure 4.4: Abstract Syntax Tree

As we can see, it is much more clear than the classical Syntax Tree.

4.3 Summary 1

Parsing Tools that allow us to use grammar definition like PEG or EBNF (with more symbols that denotes more explicit stuffs) will generally produce better Syntax Tree than the others. However AST let us decide precisely what we want as node/leaves in the tree. Let's take an example using Java :

```
1 list.forEach(x -> System.out.println(x));
2 list.reduce((x,y) -> x + y);
```

As we can see, the difference is that with 1 identifier we do not need parenthesis. That must be take into account when generating the Syntax Tree. It would lead to the following Syntax Trees :

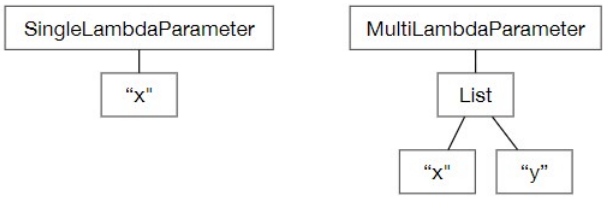

Figure 4.5: First ST

However, as we have seen, parsing is just the start of the pipeline! We have to do a lot of work with it (check for errors, generate bytecode, etc.). With this kind of Syntax Tree, we would have to deal with different kind of node for the same lambda principle. Using AST we can have the following :

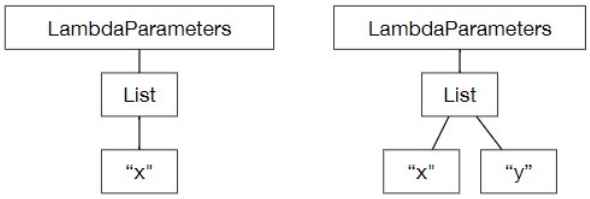

Figure 4.6: Second AST

Which allow us to use the same code to deal with both!

4.4 Coding a parser

General principles :

- One function per symbol (non-terminals & terminals);
- Model in the input as globally accessible character array;
- The current input position is globally accessible;
- Calling a function = attempting to parse the symbol it denotes at the current input position;
- A parsing function returns :
  - true if the symbol was matched, updating the input position past the matched input.
  - false if they failed to match, the input remains unchanged.

See parser.java file. Note that, in order to be sure our implementation really work, we can use the debugger and check for the pos cursor at the end, or to go further we could also inspect the parse tree if we have build one.

As we can see, the parser we wrote is verbose and cumbersome. Also it does not have any bells or whistles, if it fails it just fails, it does not tell us where it fails or why. What we have done is implementing a PEG semantics for the grammar (hint, a* a is empty). Backtracking in PEG is to reset the counter before the start of the sequence. CFG also have a backtracking technique but it is not the same.

We can also add AST to our parser. In order to do that in Java, we just need to add a Dequeue and modify only a few methods. See parser_ast.java for that.

4.5 Parser Combinator

The parser we have implemented from PEG grammar. As we have seen, this parsers have a lot of duplication (each "expression" is handled the same way). The idea to solve that is to build an AST where each node is an "expression" and interpret it.

The idea, is that each combinator is a node/parsing expression in the AST. See file Combinators.java. Using combinators allow us to cut down on verbosity & code duplication (across all grammars and within a specific grammar). However it still have some drawbacks :

- Harder to debug
- It's slower because of megamorphism (we'll come back later on this)

Note that theses issues can be eliminated by using combinators for code generation. Good frameworks (like Autumn) will mitigate usability issues.

Chapter 5

PEG & CFG Semantics

5.1 Recap

Definition 5.1.1 (Grammar)**.** The (formal) definition of a language

Definition 5.1.2 (Language)**.** A (potentially infinite) set of sentences

Definition 5.1.3 (Parser)**.** Recognizes a sentence in the language + extract a syntax tree

Definition 5.1.4 (Parsing Tool)**.** Generate and/or runs parsers

Two types of formalism :

- CFG (Context-Free-Grammar)
- PEF (Parsing Expression Grammar)

Notations :

- (E)BNF (usually for CFG)
- PEG Notation (usually for PEG)

Definition 5.1.5 (Non-Terminal)**.** Things that we define

Definition 5.1.6 (Terminals)**.** Tokens, strings, etc. that cannot be extended further.

5.2 Context-Free Grammars

Usually a CFG is defined as a tuple of 4 components Grammar $= (N, \Sigma, P, S)$:

- N: Non-Terminals
- Σ: Alphabet (Terminals)
- P: Production Rules $(P : N \to (\Sigma \cup N)*)$
- Starting Symbol $(S \in N)$

In order to have a CFG from a "full" grammar, we first need to replace all syntactic sugars by the complete recursive rules, then eliminate all choices by introducing as many rules as we have choices.

5.2.1 Semantics Derivation

- The language defined by a CFG is the set of all sentences that can be derived from its rules
- Start from the start symbol, replace it by the right-hand side of one its production
- At each step, replace a non-terminal from the current string symbol by its definition (until no non-terminals are left in the string)
- Any terminal, the order does not matter

Note that by doing that, we define the language, not a parsing algorithm and we're doing it in a generative way grammar → sentences.

Also, if we would have used characters derivation instead of tokens we would have need to continue to derive which would of potentially lead to an infinite derivation. So, we can say that for a sentence to be part of a language in need to be derivable however we cannot describe the all derivation table because in many case, it would be infinite.

5.3 PEG Semantics

We can see in the slide (9 of PEG - CFG semantics's pdf) the use of a top-down recursive descent parsers that produce production rules (with lookahead operator as an exception).

5.4 PEG vs CFG

5.4.1 Semantics

CFG are generative (their semantics is given by constructing the language set by the grammar through derivation). On the other hand PEG are recognition based : a sentence is in the language defined by a PEG grammar only if it is recognized by the language recognizer. In order to formalize PEG grammar we need to formalize the recognizer for the grammar.

These two approaches are very different in the mathematical point of view. Also, CFG is easier to defined in the mathematical language, on the other hand the PEG is very much related to the practice.

5.4.2 PEG vs CFG

- CFG has unordered choices, while derivating we can pick any non-terminal we want
- PEG has ordered choices, the first matching is the "correct" one

5.4.3 The differences

Unordered and ordered has a consequence for parsing algorithm. PEG does not test anything after a fail (prefix capture) if the parse failed while CFG can make an other choice!

PEG: Single parse Rule

Once a choice have been made, we never visit it again. As repetition can be desugared to choice it has an impact on them too. Repetition are greedy. (A ::= a* a is empty as a* will consume everything).

Backtracking

Let's take the following grammar :

- A ::= B y z
- B ::= v | x y | x
- input: "xyz"

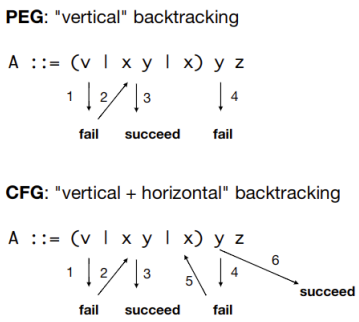

Figure 5.1: PEG vs CFG backtracking

This is not the real working way, be it does illustrate the principle. Indeed doing that would lead to exponential time complexity. CFG has horizontal backtracking thanks to is possibility to backtrack on x after trying y.

Ambiguity

It seems that CFG are always better because of what we have seen before. However there is a flip-side : ambiguity. By construction PEG cannot suffer from ambiguity.

As a remainder :

- Prefix capture (The rule B will consume all y)
  - A ::= B y z
  - B ::= x y | z
- Ambiguity (B can be xy or x, C can be z or yz):
  - A ::= B C
  - B ::= x y | x
  - C ::= z | yz

Ambiguity make the AST creation harder and also has impact on the parser performances.

Performances

The best parsing algorithm for CFG are $\mathcal{O}(n^3)$, deterministic parts of the grammar run in $\mathcal{O}(n)$ (most of useful grammar are deterministic).

For PEGs, the regular algorithm is exponential (in theory). Still, it is almost impossible to write an $\mathcal{O}(x^n)$ exponential parser. However, one often use operation is very inefficient : infix.

An example can be this one :

- S ::= P '+' S | P '-' S | P

- P ::= N '*' P | N '/' P | N
- N ::= [0-9]+

(Note that PEG does not allow left recursion by definition.) Let's assume we have a parseN function, if we try to parse "42", parse N will be called 9 times! 3 times for P, 3 times for S, 3x3=9. In general : $\mathcal{O}((P+1)^L)$ times with L : precedence levels (2 here), P : operators at each level (3 here).

A solution for that can be found, for example in Autumn we can write :

```
rule P = left_expression()
                .operand(N)
                .operand('*')
                .operand('/'); //Same for S however, operand is P in S, for precedence
```

This rewrite the grammar as :

- S ::= P ('+' S | '-' S)*
- P ::= N ('*' P | '/' P)*
- N ::= [0-9]+

P and N will be only called once! Performances are good thanks to that! However we can note that the parse tree won't be nice (not a problem with Autumn as we build an AST explicitly (we give a function to create the nodes)). If we were not using Autumn, we should create the parse tree like that and then, rewrite it

Packrat Parsers The single parser rule make memoization very easy! Packrat Parser are PEG parser with memoization. However, some practical experiments have been done in Java shown it is slower. Unless maybe if your language is very slow, or unless your infix expressions are improperly implemented.

Expressivness

- Some PEGs definition cannot be defined with CFG (A ::= $a^n b^n c^n$ for any same n we want any a b c)
- Some CFGs definition cannot be defined with PEGs (A ::= a A a | b A b | a | b | ϵ)
- Traditional PEG can't use left recursion (use repetition or Autumn to solve that, as they are the only two big case where we need left recursion)

PEG : Lookahead operators

- &<expression>, succeeds if the expression succeeds but does not consume any input
- !<expression>, succeeds if the expression fails, does not consume any input
- In theory, &X == !!X (not true in Autumn)

PEG misc

PEGs are often use in scannerless parsing (without lexer), ordered and lookahead are very useful at lexical level. PEG allow us to define "reserved works", hence lexing can still be advantageous.

PEGs are easy to extends thanks to recursive descent (with new combinators).

5.5 Summary

PEG :

- Intentional language = sentences recognized
- Similar to handwritten top down recursive descent parsers
- Desugared to CFG + lookahead
- Single parse rule
- Suffer from prefix capture
- Vertical backtracking
- Deterministic
- Ordered
- Potentially exponential, in practice largely linear (careful with infix)

CFG :

- Extensional language = set of sentences obtained by derivation
- Suffer from ambiguity
- Vertical and horizontal backtracking
- Non deterministic
- Unordered
- $\mathcal{O}(n^3)$ but often linear in practice

In the end, both formalisms are good enough, the real difference is tooling, what is available in the language we want to use, ease of use, features, performances, etc.

Chapter 6

Chomsky's Hierarchy

6.1 The hierarchy

Chomsky which is a famous linguist also known for his political engagement has defined a hierarchy of grammar :

- Type 3: Regular grammars (regular expressions, single non terminal at the end)
  - $P : N \to \Sigma * N$
- Type 2: Context-Free grammars (non terminal string of symbols)
  - $P : N \to (\Sigma \cup N)*$
- Type 1: Context-Sensitive grammars (alpha and beta represent the context)
  - $P : \alpha N \beta \to \alpha \gamma \beta$ where $(\alpha, \beta, \gamma \in (\Sigma \cup N)*)$
- Type 0: Unrestricted grammars
  - $P : (\Sigma \cup N)+ \to (\Sigma \cup N)*$

6.2 Regular vs CFGs

In regular we have a final non-terminal makes regular languages capable of expression repetition and optionality. Regular cannot express nesting.

6.3 Type 0 and 1

Let's be honnest, they are pretty much useless. Still, the principle behind context sensitivity is useful! Indeed, is func((T) * x) a multiplication or a cast? However CSGs are bad for that (not like Autumn).

6.4 Automaton mapping

Definition 6.4.1 (Automaton)**.** Formalisation of an abstract machine

A nice thing with the hierarchy is that it can be mapped to automaton :

- Type 3 (Regular grammars) : Deterministic Finite Automaton

- Type 2 (Context-Free grammar) : Nondeterministic (can have multiple transition) Pushdown (use a stack, a transition can push on it or look at it to take a decision) Automaton
- Type 1 (Context-Sensitive grammar) : Linear Bounded Automaton (Finite-tape Turing Machine)
- Type 0 (Unrestricted grammar) : Turing machine

Chapter 7

Lexing with regular expression

7.1 Importance of Lexing

7.1.1 Whitespace Handling

This chapter will show why lexing is important in addition to parsing. A simple example is whitespace (WS) handling. If we look at the grammar we have defined before (N, P, S) the expression " 1 + 1 " is not accepted because of whitespace. In order to have WS accepted we can use the following rule : $S ::= S' +' WS * P | S' -' WS * P | P$ with WS being space, \t or \n but it is not nice and can quickly become a mess. A nicer approach would be to define WS as before and to define new rules for operators like $PLUS ::=' +' WS*$.

7.1.2 CFG limitation

As we have seen before CFG does not allow reserved work like if.

7.1.3 Performances

If we don't use lexing, we need to work on each character, using a lexer we can build a token tree which can be 20 smaller. This is especially true if there are fixed overheads (combinators). It is typically developed as a loop with switch statement. It is simple enough that we can write it by hand or even better generate it (regular expressions)!

7.2 Lexers with RE

The idea is that we want to match a prefix of the (remainder of) input that matches the RE. There are two types of RE flavors :

- Standard (equivalent to regular grammars) : can be matched in $\mathcal{O}(n)$ (n=input size)
- Language specific (Perl-Compatible Regular Expression) : often in $\mathcal{O}(n)$ but some features can make it exponential (even without using them). E.g : backreferencing (capture something we have match and try to match it again).

Some experiment on this subject have been made which show the time needed by both techniques to match some rules. As we can see on the next figure the difference is huge. Note that, the y axes is not the same, with PCRE we're speaking in seconds while in the other approach we are on μs which is way smaller!

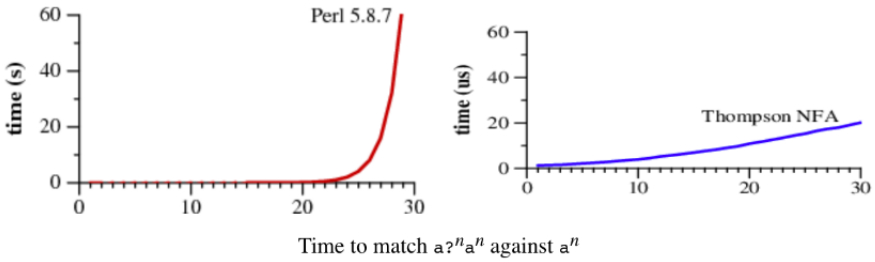

Figure 7.1: Regular vs PCRE

However we can ask ourself if this experiment is really realistic. As we have seen, it *can* be exponential in some case but are these case something that can really happen? This question is not so easy to answer if we make some research (Pr. did) we can maybe find one grammar example that is both useful and exponential.

Let's now image that matching a RE is done in $\mathcal{O}(n)$, what is the complexity of lexing? It is $\mathcal{O}(n^2)$! Why ? Because in the worst case we would need to scan to the end of the input (n) for each token. An example would be : $(a|a*x)$ with a^n as input. This is a contrived example which is not a problem in practice! This is also a nice example of theory vs practice.

7.3 Building the DFA

We have seen before that regular grammars can be recognized by a Deterministic Finite Automaton. First let's look at the DFA for rule $Tokens ::= [0-9]+||[a-z]|"if"$. The resulting automaton is this one :

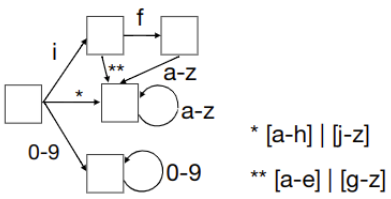

Figure 7.2: DFA example

In order to build the DFA, we first need to translate the regular expression to a NFA (Nondeterministic Finite Automaton). Then we have two options :

1.
   - Transform the NFA into a DFA
   - Minimize the DFA
2.
   - Simulate DFA with a NFA
   - Exactly the same thing, but lazily

7.4 NFA

First of all let's see an example of NFA, the left part is showing the NFA whereas the right part is the equivalent DFA. As we can see, the difference is that in NFA we can have multiple transition that use the same letter.

Figure 7.3: NFA example

If we try to match a sequence in a NFA, we have two ways :

- Keeping a single pointer that rollback at the beginning if the match failed
- Keeping multiple pointer that go in parallel.

In DFA the complexity is $\mathcal{O}(n)$ well, in fact it's more $\mathcal{O}(nm^2)$ (at least in theory) where m = number of states and represent the possibility where every state is connected to each other. In practice, NFA use repeated work and more complex data structures. That, plus the fact that DFA are pure $\mathcal{O}(n)$ make them more usable in practice.

7.4.1 From regex to NFA

Here is the list of regex when applied with NFA :

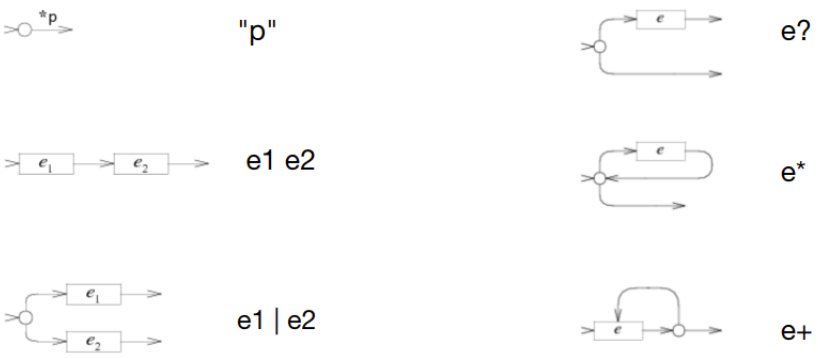

Figure 7.4: Regex to NFA

Note that e_1, e_2 are complete automaton that are plugged. Also if we don't have starting point (like in e_1e_2) it means that the starting state of e_1 become the starting state of the whole sequence.

Empty transitions are very useful in order to keep the number of transition manageable. Without them, every state would need to be link together (so, for 2 choice of 3 letters, 12 transitions, 9 with empty transitions.)

7.5 Powerset construction

Powerset construction is a technique that allow us to transform a NFA to DFA. It is quiet simple, we take concurrent equivalent states and merge them into a single one in the DFA. Figure 7.3 show the

7.6 Minimization of DFA

When we "translate" a NFA to a DFA we may end up with an increasing size! e.g a NFA of size n, may lead to an DFA of size 2^n. We cannot do anything for that expect to minimize a little the DFA.

These two algorithms are classical ones in compilers classes.

7.7 DFA simulation

It use the Thompson's algorithm :

- Run the NFA by maintaining a list of states
- Build the DFA by treating the list of states as a single DFA state.
- Need to be able to lookup a DFA state from a list of NFA state.

This approach is typically slower as we do the work while lexing. However, the main advantage is that we only build DFA node that we need!

Chapter 8

Pumping lemma

8.1 Recap

Definition 8.1.1 (Grammar)**.** Grammar = (N, Σ, P, s)

- N : Non-terminals
- Σ: Alphabet (terminals)
- P: Production rules
- Starting Symbol (S $\in$ N)
- $P : N \to \Sigma * N?$

Matchable by a DFA!

Regular languages are capable of expression repetition and optionality thanks to the optional final non-terminal. However, it can't do nesting.

We can ask ourself : how to know if a language is regular ? And how to prove (just build the grammar, is that match it works)/disprove (pumping lemma) it?

8.2 Central Recursion

Let's take an example of non-regular language: $A ::=' ('A')'|\epsilon$. This is called central recursion. Note that language with central recursion are never regular. That's why C(or Java) does not have nested comments

A way to check if a language is regular is to check if it is matchable by a DFA (that has only one piece of state, the current one). If we want to match the central recursion, we need an extra counter for the depth. It exists only one way to implement that : using non-terminal in the middle. The problem is : this is not allowed in regular languages. We can proof it using the pumping lemma.

8.3 The Pumping Lemma

Definition 8.3.1 (Pumping Lemma)**.** If L is a regular language:

- Then it exists an integer $p \geq 1$ specific to L
- Such that every sentence s in L of $length \geq p$ can be written $s = xyz$ satisfying the following conditions :
  - $|y| \geq 1$
  - $|xy| \leq p$
  - $\forall n \geq 0, xy^n z \in L$

Every long enough ($p \geq 1$ and specific to L) string in the language can be used to generate longer string through a repeating part (y).

Note that there is also a corollary : every long enough string can be decomposed into a prefix, a suffix and a repeating part.

Thanks to the pumping lemma, we can proof that $L = ({}^k)^k$ is not regular (using contraction). Indeed, as we can show that x and y have to be made only of '(' and that y cannot be empty by definition, if we try to repeat y, the parenthesis are unbalanced! Which is a contradiction.

The pumping lemma can show us that any finite language is regular! Even a really simple one like $L = a|b|c$. We just need to take a big enough p! (Like $p = 2$ in this case).

8.3.1 Why do we need $|xy| \leq p$?

If we omit it, we could prove that $L = ({}^k a+)^k$ to be regular which is obviously not the case!. We need central recursion to be bounded because of DFA state limit. The constraint ensures we're able to build a maximum depth counter-example.

8.3.2 Be care

Note the way the pumping lemma is written, it is an implication. That mean $A \implies B \neq B \implies A$. Being able to pump sentences does not make a language regular. However is it a necessary condition in order to be regular.

8.3.3 In CFGs

The pumping lemma also work in CFGs! The difference is instead of splitting a sentence in 3, we split it in 5 $s = vwxyz \implies vwwxyyz \in L$ the parenthesis language works if we pick w=(and y=) note that a language like $a^k b^k c^k$ does not work. In both case, the pumping lemma can prove it!

8.3.4 In everything else

We just have to decompose in a pattern :

- X be a set of sets (e.g regular languages)
- An infinite set (language) of objects (sentences) in X can be obtained if long object repeat some sub-elements
- We can use the pumping lemma to disprove the belonging of a language to X

Chapter 9

LL and LR algorithms

This chapter is interested in how to parse CFGs grammar. We will see two algorithms : LL and LR. Note that, they are not generic algorithms for CFGs parsing.

9.1 Historical perspective

- Chomsky's Hierarchy : mid-50s
- LL & LR : formalized in mid/late 60
- Processors at this time where not very efficient
- $\mathcal{O}(n^3)$ was laughable, way to slow

Taking that into account, the choice to make efficient implementation (no backtracking / fancy data structure) that can only parse a part of CFGs was made. It is very practical, indeed we can show that programming language typically can be parse in $\mathcal{O}(n)$.

9.2 LL

Definition 9.2.1 (LL)**.** LL = Left-to-right Leftmost derivation. It always expand the left non-terminal first.

It can parse a subset of CFGs :

- Choices: use k tokens of lookahead to decide (never backtrack)
- Nowadays we consider it as a worse PEG
- If the grammar is accepted by LL, it guarantees $\mathcal{O}(n)$, can be implemented based on table-based lookup

9.2.1 Properties

- Top down recursive descent algorithm as PEG (but only one choice alternative taken, depends on lookahead lookup)
- No left recursion allowed
- Like PEG : unambiguous

- No language hiding (
$$A ::= a * a$$
is not a valid language in LL)
- Less expressive than CFG or PEG (even than LR)

Definition 9.2.2 (LL Grammar)**.** Grammar for which we can generate an LL parser (no FIRST/-FIRST or FIRST/FOLLOW conflicts)

Definition 9.2.3 (LL Language)**.** A language that has a LL grammar (may have multiple grammars including non-LL ones.)

LL Conflicts

FIRST/FIRST conflit

- Choices starting with the same k tokens
- $A ::= ab|ac(k = 1)$

FIRST/FOLLOW conflit

- Choice can start or (if it is nullable) be followed by the same k tokens ($S ::= XY; X ::= \epsilon|a; Y ::= a|b$)

We can avoid FIRST/FIRST with left-factoring (factor out the common part at the start of two choice alternatives)

- InClassDecl ::= FieldDecl | MethodDecl
  FieldDecl ::= Modifier* Type Identifier ('=' Expression)? ';'
  MethodDecl ::= Modifier* Type Identifier '(' ParameterList ')' Body
  →
  InClassDecl ::= Modifier* Type Identifier (FieldDeclSuffix | MethDeclSuffix)
  FieldDeclSuffix ::= ('=' Expression)? ';'
  MethDeclSuffix ::= '(' ParameterList ')' Body

Figure 9.1: Left-factoring example

In the previous figure, the three first line are not LL as the two last starts with the same prefix. The 3 last lines are LL because the suffix has been extracted, so the both use the same rule with a suffix choice instead of two rules.

Note that it is not great for plain syntax tree, also makes the AST a bit more difficult to build. However it is also useful for PEG performance (we avoid parsing the same prefix many times).

9.2.2 LL(1) vs Regular Expression

As both of them use a single symbol we could ask why are they not the same.

- LL can handle central recursion
- Regular grammars are parser with $\mathcal{O}(1)$ space (current state)
- LL(1) implemented by top-down recursive descent uses $\mathcal{O}(n)$ space: the function call stack (can be use to recurse)

9.3 LR

LR algorithm is the most difficult parsing algorithm that we'll see. Still, it is not the most difficult one (hello GLR). The goal of LR is to parse every deterministic grammar. Every grammar can be parsed in $\mathcal{O}(n)$ without backtracking (we cannot explore different alternatives). Hence, it is not the same as LL(1) as LL(1) decides eagerly and ignores the context.

Definition 9.3.1 (LR parsing)**.** LR = Left-to-right Rightmost derivation. It always expand the right non-terminal first.

Essentially done by DPDA:

- Stack on which symbols can be pushed (terminal and non-terminals)
- Performs a table lookup based on the stack and some tokens of lookahead(LL(1)/LL(k))

9.3.1 Shift and reduce

The algorithm use a table that maps the stack to one of the two actions :

1. shift : the next terminal onto the stack
2. reduce : items at the top of the stack to a non-terminal.

If we take our JSON grammar, an example would be :

1. remaining input { "x": 1} Stack : []
2. Shift x4 times
3. Reduce to PAIR Stack : [{, "x", :, 1]
4. Reduce to TPAIRS(from ϵ) Stack : [{, PAIR]
5. Reduce to PAIRS Stack : [{, PAIR, TPAIRS]
6. Reduce to PAIRS? Stack : [{, PAIRS]
7. Shift Stack : [{, PAIRS? }]
8. Reduce to OBJECT Stack : [OBJECT] -> VALUE

9.3.2 LR Conflicts

REDUCE/REDUCE conflicts

- Rare case, it means the same set of symbols can be reduced to the same non-terminal, in the same context (same input prefix)
- Same portion of input could be matched to different non terminals (ambiguity)
- Non-trivial example can also appear using optional rules.

SHIFT/REDUCE conflicts A famous problem for this is

```
1 if (a) if (b) s1(); else s2();
```

The else can be interpreted as the false condition of the outer or of the inner if. Depending on that, the sequence of shift/reduce will not be the same. That's why parsing tool let us define instruction for that (common default behavior is to prioritize shift over reduce).

9.3.3 Ambiguity

Definition 9.3.2 (Ambiguity)**.** Ambiguity appear when there are multiple way to parse the same input :

- Different (combination of) rules can match the same input
- Simulating different derivations
- Generating different plain parse trees

It is undecidable for CFGs. Note that deterministic grammars are unambiguous. Determinism is decidable (non-determinism = conflict). Thus, ambiguity $\implies$ Non-determinism, by modus tollens Determinism $\implies$ Unambiguity.

However, it exists unambiguous, non-deterministic grammars (palindrome language).

In the end, what is not LR ?

- Ambiguous grammars
- Corner-case speculative scenarios like above (rare case)

9.3.4 LR building table

This is the most difficult part of LR, but it is not needed to be able to use LR and its variants.

9.3.5 Variants

- LR(k): use more lookahead
- LALR: lose some context given by the prefix (smaller tables than LR)
- SLR: lose all context given by prefix (even smaller tables)
- IELR: True LR, tables as small as LALR
- GLR: can parse every CFG in $\mathcal{O}(n^3)$, deterministic ones in $\mathcal{O}(n)$

See some examples of LALR, SLR and LR(k) in the slides.

GLR

GLR is used when we need to speculate/backtrack we fork the LR stack. It uses graph-structured stacks (GSS). It shares as much of the stack as possible with the possibility to merge stacks down the line.

It is a non-deterministic pushdown automaton.

9.4 Other variants for CFG

Some other parsers can parse every CFG grammars.

- Early (as seen in Computational Linguistic course)
  - $\mathcal{O}(n^3)$
  - Popular for Linguistic (lots of ambiguity)
  - Simplest general parsing algorithm

- ANTLR/ALL(*)
  - $\mathcal{O}(n^4)$
  - Very popular java parsing tool
  - LL(k) + backtracking + caching via automata
- GLL
  - $\mathcal{O}(n^3)$
  - LL + GSS

Chapter 10

Semantic Analysis

With semantic analysis we begin the next step of our pipeline. We begin the semantic analysis with an AST.

10.1 Introduction

Semantic analysis allow us to check everything that :

- Can be check statically (without running the program) in reasonable time (no real symbolic execution (instead of real input, we use variable. The problem is when we encounter if branching because it multiply the amount of path (exponential) this phenomenon is called combinatorial explosion))
- That we didn't check in the parser (we should check as little as possible in the parser)

10.1.1 Check after parsing

We can take the example of Java modifiers if we would like to check that in the parser we would need many rules, for each modifiers, the methods without them, etc. Thus, it would only be for non-abstract classes as the other (like interface) have other constraints. Besides, it does not prevent thing like "protected static" which is forbidden in Java.

A first better way is to use Autumn parsing combinator, but still we should check that with the semantic analysis. An example of error for the following : public private void test()[] would be :

- Parser : "unexpected token 'private'"
- Semantic: "two visibility modifiers for method"

No need to say that the second one is much better. Also, incorrect ASTs can be use in IDEs (syntax highlighting, etc.)

10.1.2 Main Concerns

Type checking

Definition 10.1.1 (Type checking)**.** Check type constraints : "int x = "String";" but also type inference : "var x = "string" + 42. If the language is dynamically typed it is done during runtime and libraries.

Name Binding

Definition 10.1.2 (Name binding)**.** Check where name are defined. e.g : "int x = y + 3;" What is y? Inter-dependency : "var x = a.b.c" we need to know the type of c, that need the type of b, that need the type of a.

These two first principle cannot be done separate.

Note that Name binding also consist of lexical scoping (what is visible or not for a field/inside a method/block etc.). Dynamic scoping can also be done, however is it opposed with lexical scoping. Still it have this advantages! Emacs use this kind of scoping, this allow Emacs to edit Ruby and Python file and handle tabulation in both case.

Flow checks

Flow checks is a complex think, let's take a Java example :

```
1 int test(int x) {
2     if (x == 3) return 3;
3 }
4 int test2(int x) {
5     while(true) return 1337;
6 }
```

The first one is an invalid Java program (suppose to return an int but only return it if x==3) and flow checks spot that. The second one is valid because Java detects that the while loop will always be entered.

More...

Languages like Whiley / Dafny / Rust that statically does thinks.

10.2 The Hindley-Milner Type System

10.2.1 Introduction

Type system for polymorphic lambda calculus (System F). It is used in practice as it is the base of Haskell and ML type system. Also it can be extended in various ways.

10.2.2 Lambda calculus

Lambda calculus

Toy language that is written this way : $e ::= x|(\lambda x.e)|(ee)$ variable/Abstraction/application

Simply-Typed Lambda calculus

$e ::= x|(\lambda x : \tau.e)|(ee)|c$ where τ us the parameter type and c a constant. Note that, we need a set of base types e.g B = {a, b}. We do need constant c, because unlike classical lambda calculus we do not have the Church encoding (define number as function).

Polymorphic lambda calculus

The idea is to type the original lambda calculus without the type annotations
$e ::= x|(\lambda x.e)|(ee)|letx = e$ in e

10.2.3 Type system

Definition 10.2.1 (Type system)**.** A formal system(formalism) that determines :

- if any expression in the language is well-typed
- type of any such expression
- $e ::= x|(\lambda x : \tau.e)|(ee)|c$
- ill-typed: $((\lambda x : a.x)c1)$ where c1 has type b

We would like the type system to be :

- **Decidability** : can make a decision
- **Soundness** : everything we can prove is true
- **Completeness** : we can prove everything that is true

However the Gödel's incompleteness theorem says that "No sound system of axioms describing natural number arithmetic can be complete". Still, if we look in practice, Java does not have any of the previous properties...

For λ

- Γ = context, a set of type bindings (assignments, assumptions)
- $\vdash$ = type judgement
- σ, τ denote types

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash x{:}\sigma}\ (1) \qquad \frac{c \text{ is a constant of type } T}{\Gamma \vdash c{:}T}\ (2)$$

$$\frac{\Gamma, x{:}\sigma \vdash e{:}\tau}{\Gamma \vdash (\lambda x{:}\sigma.\ e){:}(\sigma \to \tau)}\ (3) \qquad \frac{\Gamma \vdash e_1{:}\sigma \to \tau \quad \Gamma \vdash e_2{:}\sigma}{\Gamma \vdash e_1\ e_2{:}\tau}\ (4) \qquad \texttt{((λ x:a . x) c)}$$

Figure 10.1: Type system for λ

10.2.4 Towards Hindley-Milner

The key idea is : function can have many types (polymorphism). Why is it important? Because we really need polymorphism in order to be able to write rule like $((\lambda id.((foo(id1))(id's')))(\lambda x.x))$ without polymorphism, the identity function that we use should be declared as integer or char with polymorphism it can return both!

Generalization

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_D x : \sigma} \quad [\textbf{Var}]$$

$$\frac{\Gamma \vdash_D e_0 : \tau \to \tau' \qquad \Gamma \vdash_D e_1 : \tau}{\Gamma \vdash_D e_0\ e_1 : \tau'} \quad [\textbf{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash_D e : \tau'}{\Gamma \vdash_D \lambda x\ .\ e : \tau \to \tau'} \quad [\textbf{Abs}]$$

$$\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash_D e : \forall \alpha\ .\ \sigma} \quad [\textbf{Gen}]$$

```
⊢ (λ x . x) : α → α          (Abs)
⊢ (λ x . x) : ∀ α . α → α    (Gen)
```

Which are variables are bound (unfree)?

- ∀-quantified variables
- variables appearing in function types
  - ... that never appear on their own

Figure 10.2: Generalization

Still, it this step we cannot apply the function as we still have the polymorphic type and not a specialized type.

Intantiation

Allow us to specify a type from polymorphic type!

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_D x : \sigma} \quad [\textbf{Var}]$$

$$\frac{\Gamma \vdash_D e_0 : \tau \to \tau' \qquad \Gamma \vdash_D e_1 : \tau}{\Gamma \vdash_D e_0\ e_1 : \tau'} \quad [\textbf{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash_D e : \tau'}{\Gamma \vdash_D \lambda x\ .\ e : \tau \to \tau'} \quad [\textbf{Abs}]$$

$$\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash_D e : \forall \alpha\ .\ \sigma} \quad [\textbf{Gen}]$$

$$\frac{\Gamma \vdash_D e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash_D e : \sigma} \quad [\textbf{Inst}]$$

```
⊢ (λ x . x) : α → α          (Abs)
⊢ (λ x . x) : ∀ α . α → α    (Gen)
⊢ (λ x . x) : int → int      (Inst)
```

Why do we need polymorphism?

```
((λid . (ζfoo (id 1)) (id 's'))
 (λx . x))
```

Figure 10.3: Instantiation

Let polymorphism

The let rule allow us to go to a monomorphic type!

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash_D x : \sigma} \quad [\textbf{Var}]$$

$$\frac{\Gamma \vdash_D e_0 : \tau \to \tau' \qquad \Gamma \vdash_D e_1 : \tau}{\Gamma \vdash_D e_0\ e_1 : \tau'} \quad [\textbf{App}]$$

$$\frac{\Gamma,\ x : \tau \vdash_D e : \tau'}{\Gamma \vdash_D \lambda\, x\ .\ e : \tau \to \tau'} \quad [\textbf{Abs}]$$

$$\frac{\Gamma \vdash_D e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash_D e : \forall\, \alpha\ .\ \sigma} \quad [\textbf{Gen}]$$

$$\frac{\Gamma \vdash_D e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash_D e : \sigma} \quad [\textbf{Inst}]$$

$$\frac{\Gamma \vdash_D e_0 : \sigma \qquad \Gamma,\ x : \sigma \vdash_D e_1 : \tau}{\Gamma \vdash_D \textbf{let}\ x = e_0\ \textbf{in}\ e_1 : \tau} \quad [\textbf{Let}]$$

Figure 10.4: Let polymorphism

We just need to rewrite it like : $letid = (\lambda x.x) in ((foo(id1))(id's'))$

10.2.5 Type system vs Typing algorithm

Thanks to inference rules we have the semantics of the type system. Rules allow us to prove statements about types, as the system is sound the created statements are true. However, the way to prove statements is not given (algorithm is needed for that).

10.3 Using Uranium

Uranium is a library written in Java. It is inspired by formal grammars, attribute grammars and reactive programming. The goal of Uranium is the specification of dependencies (using rules) e.g : "the type of a function call depends on the type signature of the function being called, and on the type of each of the arguments". Thus it is still explicit thanks to code computation and verification of attributes.

10.3.1 Attribute grammars

The goal of attributes grammars is to calculate attributes on AST nodes. Many types:

- Synthesized attributes : computed from children node attributes
- Inherited attributes : computed from parent node attributes
- Combination of both
- Other have been proposed in practice

It is called this way because it was originally designed within the grammar (which is a bad idea).

10.3.2 Uranium: Basics

Uranium computes attributes (AST Node, String) pair using rules:

- One or more dependency attributes
- One or more exported attributes
- Code that can read the dependency values, write exported values
- Automatically called when all dependency values are available
- All of this is done by the reactor

Check SemanticsAnalysis.java to have an example.

Chapter 11

Tree-Walk Interpreter

Like we have done in the semantic analysis we will traverse a tree in order to interpret our document. Indeed as seen in figure 2.1 we do not have a "compiler" step in the pipeline. However we can replace the two last box by an interpreter (or a tree-walk interpreter) or bytecode generation and a virtual machine (bytecode interpreter). Many design are possibles! Different interpreter works differently, the tree-walk interpreter walks a tree that match more closely the source than the structure of sources. Bytecode interpreter runs over instructions, so we have to transform our tree in a linear way.

Tree-walk interpreter/AST interpreters are slow because they do a lot of megamorphic calls but also because of their nature, with tree walks interpreter we can reuse stuff we already have (AST and semantics analyses) without the need to generate other stuff. Still, making this choice make the interpreter slow. If we want to speed up the process we can generate many data structure and write a bytecode interpreter. In this case, there is no particular trade off, either choose simplicity or speed.

Sometimes speed is not to worry (like in DSL) indeed, for code that won't run often does not need to be very fast! It also provide developers time (most precious resource!) as simpler code is easiest to maintain. Easiest to change the tree code than the code generation if we want to change it. It is also useful to test implementation (form of redundancy, so can be used to test more complicated implementation).

11.1 Implementation

See slides for examples. One simple implementation is the visitor pattern! Another way is dynamic lookup this is implemented in the utility package. It allow us to register different methods as we have done in semantic analysis. Still, the main difference with semantic is the automatic recursion. Indeed, in the semantic part we have define *PRE_VISIT* and *POST_VISIT* and the semantic analysis does all the job by itself, visiting parent and children without explicit control. In the interpreter it is not the case because we ***want*** to control how and when the expression is evaluated.

11.1.1 Visitor pattern vs Dynamic lookup

- Visitor pattern is much faster (only one virtual call)
- Dynamic lookup is slower (need to look at the Class object) but it is good enough to have a prototype. It is also more extensible as anyone can extend it with handlers for new subclasses (visitor pattern would have needed to edit the visitor interface not always possible!).

11.2 Statements

We when add statements we simply use what our base language (Java in the course) use to evaluate our own node (like *System.out.println* to print something). The function always need to return Object as they are still part of the interpreter so we have to cast value when we visit and either return a value or null of not needed.

11.3 Node Hierarchy

Everything in our language implement Node, it is fine but we have to be careful (indeed, we could use a *print* in the condition of a *if* node). In order to avoid that we can use abstract *ExpressionNode* and *StatementNode* (has it have been done in the project). These node would still implement the global Node but subnode would be create in order to add restriction in our language.

11.4 Scope

Here is a simple idea of dealing with scope (not the best way to do it!) As shown in slide 17, we add classes to get the variables from the reactor of the semantic analysis. Using a map, we can just set a new variable in it and get it with a get function.

Still, this (writing in scope) have a major issue:

- Recursive function will call multiple instantiations of the same scope. (The value of variable will be override!)
- Reusing the semantic analysis scope won't allow us to reuse the result of semantic in different instantiations of the program we would always need to redo the all compilation pipeline.

A better way of dealing with scope is to use a "Scope storage" using a stack, every time a new scope is created (code block, function, etc.) we add a new scope to the stack (done in the project). Storages are nested and calling a function reset the nesting (when we call a function, we will only get its scope, not the scope of the function that called it.).

When we want to lookup:

1. Get the scope of the node (computed during semantic analysis)
2. Traverse nested storage (inner to outer) in order to find the storage that match the scope
3. Read/Write the variable.

Thanks to function call resetting the nesting we will never find the wrong scope for a variable.

Still, there are some pitfall with this methods:

- Global scope : as function call reset the scope/storage we have to code global variables/functions explicitly! (If we find nothing in the created scope, look at the global one, or compare directly the scope get with the global one)
- Closure (nested function that capture outer variables, (environnement, function) pair) can be read-only (Java way, mark closure variable during semantic analysis then create and store the environnement each time the closure is instantiated) or read/write (need a wrapper that allow read/write on the variable)

```
1 void foo(){
2     int x = 42;
3     Runnable bar = () -> {System.out.println(x); x=0;};
4     bar.run(); bar.run(); x=43; bar.run();
5 }
```

This little example is interesting (even if it does not compile in Java but let's assume it does). With the read/write method, the first call will write 42, then the variable will be assign to 0, the second call will print 0 while the third one will print 43. We have a "bidirectional connection". It we would have capture the value of x instead, the second and third call would have only print 0.

11.5 Tips

When designing a language we have to think about many things:

- How arrays are represented?
- Does our language have null?
- Polymorphism (multiple value type possible) are they Object[] and sometimes optimizes (need to be immutable in that case!)
- For return/break/continue : use exception for control flow!
- Be care if the language is dynamically typed (type checks in the interpreters!)
- Function calls : create the storage for its scope the visit the node for its body.

Chapter 12

JVM Bytecode

This chapter will talk about the "Code Generation" part of the figure 2.4. Optimization will come latter as we want first to have something that runs, than to optimize it. In the end optimization is just a way to generate fast machine code, so these to concepts are tangled together. Besides, in the JVM pipeline, optimization does not matter as much. Going from semantic analysis to bytecode generation which is executed in the virtual machine, it's then the job of the VM (JIT in case of Java) to optimize the code.

12.1 Code Generation

As the course name suggests "Languages and Translators" compilers are a little bit like translators, they take the original source code and translate it into:

- Machine code (Intel x86, LLVM, etc.)
  - Need of manual optimization
  - Very large instruction set (x86 > 1000)
  - It is the lowest level but also more work(but the fastest!)
- Byte code (higher level and smaller JVM 200)
- Source code (e.g Java poet translate our language in an other one) (Simplest but more limited, slowest (as we do duplicate work, all the pipeline is run twice))

12.1.1 Why JVM?

- JVM byte code have a good effort/result trade off
- Built-in in garbage collector (not good if you want your own)
- Built-in JIT compiler
  - No need for bytecode optimization
  - Per-CPU instruction selection (always select the best instruction for the CPU that runs the code!)
- Built-in polymorphism
- Ecosystem (call to and from Java)
- Can map line to numbers to instructions
- Portable (does not need to have different generation for Unix/Mac/Windows, etc)!
- Can be package with JVM, jlink, jvm, GraalVM, etc. in order to allow the client to not install many libraries.

12.2 Bytecode

See slide 6 in order to have an example. The bytecode file starts with some metadata: name of the class, java version, ACC_PUBLIC (method is public), ACC_SUPER is historical than we have constant pool (#21 and #2) which tell the classes but also the number of methods, attributes, interface, etc.

12.2.1 Constant pool

Definition 12.2.1 (Constant pool)**.** Place where constant data are stored. It is a mapping from key to value. It use Strings, including:

- string literals used in programs
- (full) class names, fields and methods names
- methods and fields (type) descriptors

but also big integer and floating-point number and dynamically computer constant (for optimization).

i.e: #1 = Methodref #2.#3 which point to 2 and 3 #3 NameAndType which point to <init> and :()V (parameters that returns void). See other examples on slide 7. See other examples on slides 8-9.

12.3 JVM Bytecode Instructions

Now we will see the 200 JVM Bytecode instructions.

12.3.1 Execution Model/Data

Different data storage are used in the JVM bytecode:

- Locals (parameters, local variables)
- Stack (no register)
- Object with fields
- Static fields

Methods:

- Static methods (always know the code that will be executed)
- Virtual methods (need to look at which class and thus, to which implementation to call)

Note that, the JVM deals with 32 bits values however we know that double and long are 64 bits each: both occupy two slots (stack of size 2). Shorts and bit are extended to 32 bits.

12.3.2 Instructions

Definition 12.3.1 (JVM Instruction)**.** Each instruction has a name and optionally some immediate byte operands referencing the constant pool (most frequent), offsets (for jumps) or immediate (small) integer values (short maximum)). They can modify the stack (push, pop, reorder, etc.) Coded as one-byte opcode, maximum 256 instructions are possible. There is still a possibility to extend them with variable length opcode (not needed actually, the opcode 256 is the opcode for variable length and the second byte would give 256 more possibilities.)

Load/Store

Definition 12.3.2 (Push/Load)**.** Push(load) variables on the stack, store value in variables. Pattern: (A/F/D/I/L)(LOAD/STORE)[_0_1_2_3].

- Address (Object), Float, Double, Integer, Long
- Numbers are optional and mean load/store from/to variable number 0, 1, ... as it it optinal we could also specify the variable index as an operand
- e.g: ALOAD (load an object from a variable), ISTORE_0 (store an integer in variable 0)
- Why do we have 0/1/2/3? Why not always the generic method? Well, 0,1,2,3 only take one byte while the generic one need one byte for the instruction and 1 for the variable index. As we want the bytecode to be as small as possible (less read/write, less cache write, etc.). 0, 1, 2, 3 where just encoded because they are the most used once and developer of the JVM had space to encode them.

There are 50 of them! Simple form: one parameter (one byte) to give the variable index.

Definition 12.3.3 (IINC)**.** Parameters: two bytes. Adds the parameter value to the integer variable. Often used (like in for loop)

Definition 12.3.4 (WIDE)**.** Modifier of LOAD/STORE/IINC to access a wider range of local variables. It uses three bytes of variable space index instead of just one. (Preserve the opcode space).

Constants

Push constants onto the stack, there are 20 instructions for that.

Definition 12.3.5 (ACONST_NULL)**.** Push null onto the stack.

Definition 12.3.6 (ICONST(_M1/_0/_1/_2/_3/_4/_5))**.** Push integer -1 / 0 / 1 / 2 / 3 / 4 / 5 onto the stack

Definition 12.3.7 (FCONST(_0/_1/_2))**.** Push float 0 / 1 / 2 onto the stack.

Definition 12.3.8 ((L/D)CONST(_0/_1))**.** Push long (or double) 0 or 1 onto the stack.

Definition 12.3.9 (LDC, LDC_W, LDC2_W)**.** LDC take a pool constant reference and load the constant. LDC_W also take a constant pool reference but take 2 bytes instead of one and LDC2_W is the same but allow us to push two 32bits value (long or double).

Definition 12.3.10 (BIPUSH/SIPUSH)**.** BIPUSH push a byte. SIPUSH push a short.

Theoretically LDC[2]_W would allow us to do everything, but once again, encode the most common operation is nice to reduce the executable size.

Type conversions

Java has implicit type conversion (1+2L will convert the integer into a long)

Definition 12.3.11 ((I/D/F/L)2(I/D/F/L))**.** No identity conversion, just pop a value and push its conversion onto the stack.

Definition 12.3.12 (I2(B/C/S))**.** This operation is the truncation of integer to a byte (null the unused bits).

Arithmetic/Binary Operations

Pop two values from the stack and push the result of the operation on the stack.

Definition 12.3.13 ((D/F/I/L)(ADD/DIV/MUL/SUB/NEG/REM))**.** Does the corresponding operation...

Definition 12.3.14 ((I/L)(AND/OR/SHL/SHR/USHR/XOR))**.** Only for integer and long we have the classical operation and shift left/right(SHL/SHR) but also unsigned shift right(USHR).

Stack Manipulation

Definition 12.3.15 (DUP)**.** DUP[2][_X1/_X2]. Duplicate a value (take a value on the top of the stack.) Adding the 2 duplicate 2 values or a double size value. As DUP does not push value onto the stack but insert above the top value, X1 means "skip a variable", so if the stack is value2, value1 (top of the stack). It will be value1, value2, value1 (top of the stack here). X2 is the same but skip two values then insert the duplicate.

Definition 12.3.16 (POP)**.** POP[2]. Remove the top value. POP2 remove either two values or a 2 bytes value. Note that the value is just removed, we cannot do anything with it.

Definition 12.3.17 (SWAP)**.** Swap the two top values of the stack. We can notice that there is no SWAP2 instruction, so how can we remove 2 bytes long value? Just use DUP_X2 then POP.

Arrays

Definition 12.3.18 ((A/F/D/I/L/C/B/S)A(LOAD/STORE))**.** As variable LOAD/STORE from array of a certain type (CHAR/BYTE/SHORT) we have these three types as in arrays they are actually stored as bytes.

LOAD take an arrayref and an index from the top of the stack remove them and push the read value onto the stack. STORE take an arrayref, an index and a value and write in into the array.

Note that BALOAD works for both boolean and bytes (in order to avoid representing boolean as integer). Why not as boolean as bits? Best guess is it save the instruction to retrieve bit while retrieve bytes, thus the best guess is that developers are okay to waste a bit of memory to avoid these new operations.

Definition 12.3.19 (NEWARRAY)**.** [A]NEWARRAY creates an array of primitive ANEWARRAY creates an array of object. Both case take a constant pool reference that holds type descriptor or a class object for the component type of the object. They also read the length of the array and then push the array onto the stack.

Definition 12.3.20 (MULTIANEWARRAY)**.** Create a multi dimensional array of objects. As parameters it take the type (constant pool reference) and dimensions. On the stack we will push as many counts as the dimensions.

The reason of the always present A is that, in Java even if we create an 2D array of integer, we will create an array with component that are pointers to integer arrays.

Note that a 2D array of integer will use ANEWARRAY (of pointers) then fill it with NEWARRAY (of integers). MULTIANEWARRAY would be used to create 3D array (using a size of 2) and the last dimension would store array of integers created with NEWARRAY.

Definition 12.3.21 (ARRAYLENGTH)**.** Pop the array from the stack and push its length onto it.

Field Access

Definition 12.3.22 (GETFIELD/PUTFIELD)**.** Take a field reference as an immediate parameter (itself consisting of a class, field name and field descriptor (type).)

GETFIELD takes an objectref from the stack and push the field value. PUTFIELD takes an objectref and a value from the stack and assign them.

Definition 12.3.23 (GETSTATIC/PUTSTATIC)**.** Same as the previous but for static fields (no objectef needed.)

Comparisons

Comparisons are not implemented like binary/arithmetic operations in order to save space. Besides, CPUs architectures like Intel x86 already have these operation with the same definition as Java. Also, they have separated integers (because much more used but very very frequent on control flow jump).

Definition 12.3.24 ((D/F)CMP(G/L))**.** D and F are for Double and Float, take two values and push the result onto the stack. Let's assume A and B have been taken from the stack: $A > B \rightarrow 1$, $A < B \rightarrow -1$, $A == B \rightarrow 0$

G and L are used to deal with NaN values if the instruction contains G 1 will be push onto the stack, if it contains L it will be -1.

Definition 12.3.25 (LCMP)**.** Exactly the same but for long.

Definition 12.3.26 (INSTANCEOF)**.** Take a class reference as an immediate parameter and an object from the stack. Then push 1 if the object is of the given class, 0 otherwise.

Definition 12.3.27 (CHECKCAST)**.** Same as INSTANCEOF but throws an exception if needed.

Jumps and Conditionals

Parameter for these instructions is always an offset in bytes (2 bytes, 4 for _W). It allow to jump backward or forward.

Definition 12.3.28 (GOTO)**.** GOTO[_W]. Unconditional jump. W is for wide that allow the code to jump very far away.

Definition 12.3.29 (IF_ACMP(EQ/NQ))**.** Two stack operands. Compare the object pointer (the *real* same object). It jumps if they are equals otherwise execute the next instruction. NE is the opposite.

Definition 12.3.30 (IF_ICMP(EQ/NE/GE/GT/LE/LT))**.** Two stack operands. Same but for integers.

Definition 12.3.31 (IF(EQ/NE/GE/GT/LE/LT))**.** Single stack operand (EQ=0, GT>0, LT<0). Jump if the operand is 0 (EQ) or other operand. It is the result of the previous CMP so we can combine them!

Definition 12.3.32 (IF)**.** IF[NON]NULL. Jump if null or not null.

Object Creation

Definition 12.3.33 (NEW)**.** Use a class reference as a parameter. It allocate memory for the object (does not call the constructor!). In order to call the constructor we have to call a special instruction which is INVOKESPECIAL <init> and the bytecode verifier checks that a constructor is called.

Method Invocation

Parameter is always the method reference (class, name, method descriptor (type)). All methods invocation instruction have a reference as immediate parameter.

Definition 12.3.34 (INVOKESTATIC)**.** Static methods, not used with "this" on the stack. We always know the code that will be executed!

Definition 12.3.35 (INVOKEVIRTUAL)**.** Classical methods calls, using "this" on the stack. It get the object class and get the method implementation from its method table (all methods a class can use also the methods from the superclass even if they are not override). The specific code is not known at the compilation time (can change at each call!)

Definition 12.3.36 (INVOKEINTERFACE)**.** Same as INVOKEVIRTUAL but for interface methods but the lookup is a bit more complex because we can only inherit from one class but we can implement many interfaces.

Definition 12.3.37 (INVOKESPECIAL)**.** Static linking using "this" pointer. Also used for private methods (cannot be overriden it is always the same class), constructors (always know which class to look in) and super methods (when we do a call, the compiler know the type of the "this" object thanks to that it can find the first ancestor class that implement the method!).

Definition 12.3.38 (INVOKEDYNAMIC)**.** It is useful to implement dynamic languages. Basically it is a wormhole instruction. The first time it is called it goes to a bootstrap method that returns a CallSite (need to be implemented!). Once it is done, the instruction is patch (does not go to the bootstrap but directly to the CallSite). Note that it then can be optimized like any other function call. If we save the CallSite object we can change the target of the target method!

Allowing that is very useful for polymorphic inline caches (always use a method tailored to the types that have actually been seen (+ check if the values are supported.)).

Misc Instruction

Definition 12.3.39 (RETURN)**.** [A/F/D/I/L]RETURN returns void if no prefix otherwise returns the value.

Definition 12.3.40 (ATHROW)**.** Throw exception.

Definition 12.3.41 (MONITOR(ENTER/EXIT))**.** Enter and exit monitor (kind of mutex)

Definition 12.3.42 (NOP)**.** Does nothing:

Definition 12.3.43 (LOOKUPSWITCH/TABLESWITCH)**.** Use to optimize the switch statement!

12.4 JIT Register Allocation

Many lower lover instructions map easily to underlying machine code however most CPU use registers (register (small but very fast), L1, L2 caches or RAM (big but really slow)). Caching happens automatically but not in registers! They must be assigned manually. We cannot map the JVM stack as it would be too expensive so we have to decide what we can put in the different register (as a lot of machine code instruction must use registers). it is done by using a register allocation algorithm.

12.5 Differences with Java

Even if the JVM was made for Java the have some differences:

- Generic type does not exists in JVM (they mostly exists in the semantic part). Generic types are not reified (no object for it). Still C# have reified generics or we could create a language with generic types using the JVM.
- No checked exceptions : all exceptions act like the RunTimeException in the JVM.

These are the main two differences between JVM and Java.

12.6 Loading and Verification

Class files are loaded into the JVM when required (first time the class is being referred). The JVM use the name to locate the class in the class path. When the class file is loaded the bytecode is verified (constructor called, final variables are no rewritten, etc.). A very important verification is the Typestate: any instruction reachable through multiple paths (first instruction after an if-else, indeed we could have been by the if or the else part, if one part push a value on the stack but not the other we have a problem!) must be reached with the stack in the same state (size and types).

12.6.1 Typestate

Definition 12.6.1 (name)**.** Any instruction reachable through multiple paths (first instruction after an if-else, indeed we could have been by the if or the else part, if one part push a value on the stack but not the other we have a problem!) must be reached with the stack in the same state (size and types).

Java knows the type of every local and every slot on the stack (the machine code does not know that!).

It is not possible to iterate an array to copy it on the stack because the typestate would be different depending of the array type.

The stack can't overflow if you don't recurse.

12.7 Summary

JVM performs stack-based computation, use stacks + locals + constant pool + fields. Build in GC (cost of the memory control), built-in JIT complier! JVM Bytecode has a low level stack operations, jumps. It can transfers between stack and locals, fields, constant pool. It has highly-abstracted polymorphic method calls.

12.8 Generating Bytecode with ASM

ASM is a low level library and comprehensive to generate Bytecode, other libraries exists (BCEL, Javassists, etc.). Some interesting libraries are ByteBuddy or CGLIB which are focused on higher level patterns (dynamic proxy) it is a class that implement an interface automatically typically delegating the methods implementation while running code before or after the code (verification, log, etc.).

The advantage of ASM is that it handle the constant pool, takes care of the stack and space, manage jumps.

If we want to use ASM, we can use javap to reverse engineer to know what the Java compiler would done, read the JavaDoc, manual, Wikipedia page, etc.

12.8.1 Notations

Field descriptors

Describe the value of type. We only need to define them when we define or access a field. Here is the table of fields descriptors:

FieldType term	Type	Interpretation
B	byte	signed byte
C	char	Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L *ClassName* ;	reference	an instance of class *ClassName*
S	short	signed short
Z	boolean	true or false
[	reference	one array dimension

Figure 12.1: Fields descriptors table

Notice that there are some "traps" B is for Byte so Boolean are Z. Long is noted as J. Note that class are noted L Classname; so either we read an single letter or a semi colon and we know we have to look the the next two inputs. In order to describe an array we only need the bracket [then the type, i.e: [I or [[I dor a 2D int array. V for void type is not needed here as it is a method descriptor and we're looking at field descriptors here.

Method descriptor

Here is the syntax (<param1 field desc><param2 field desc>...)<return field desc>. Fields descriptors for the parameters, then for the return type.The only syntax here is the (), we don't need to separate them as we always know how much we need to read!

Slash-separated class name

References to classes are done like that: java/util/String (probably for historical reasons linked to how it is stored in the disk). These name are unique Let's imagine a "tricky" example:

```
1 package pkg;
2 class A {
3       static class X{...}
4       ...
5 }
6 package pkg;
7 class B extends A{...}
```

We could imagine two class name: pkg/A$X and pkg/B$X however, only the first one is valid!

Chapter 13

Hardware consideration

In this chapter we will see how to execute code fast and we will see that hardware is an important factor on this.

Nowadays, optimization is rarely done on AST. So, our figure 2.4, even if we rework our AST to be optimized. A more realistic pipeline in nowadays compiler is this one:

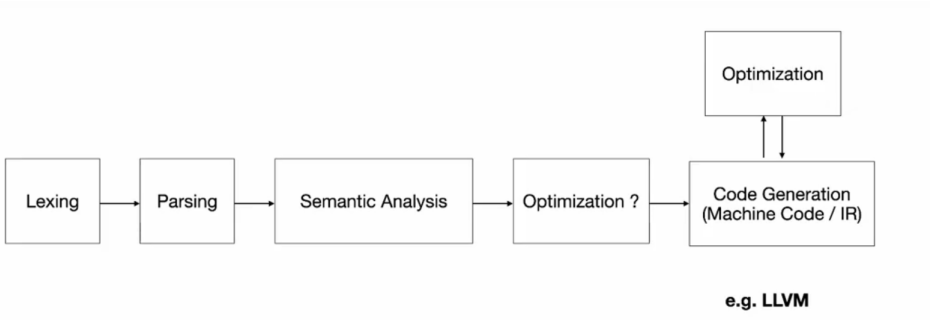

Figure 13.1: LLVM pipeline

Optimization is done on machine code(or close representation of it). It use special data structure (graph) to optimize it. Other representation exists (see slides). In cases of JIT, the optimization is run on the machine the program is executed instead of the developer compiler.

We can also profile JIT with an interpreter that create a profile that will be optimized.

13.1 What makes program slow?

The more instructions we have (exactly the number of CPU cycles) that are needed to execute the program (i.e: 4Gz clock on a CPU means 4 billions of cycles per second).

Memory access is also very slow (it is more a problem than CPU cycles). This is often call the memory wall. An "everyday" concern with that problem for almost every developer would be *quick sort* vs *merge sort*. Even if both are in $\mathcal{O}(nlog(n)$ (average case) with a worst case of $\mathcal{O}(n^2)$ in quick sort. However, quick sort is faster than merge sort, why? First of all, the worst case is rare, but also because quick sort has more "memory locality" (jumps around in memory much less than merge sort).

13.2 Memory hierarchy

The following figure shows the memory hierarchy:

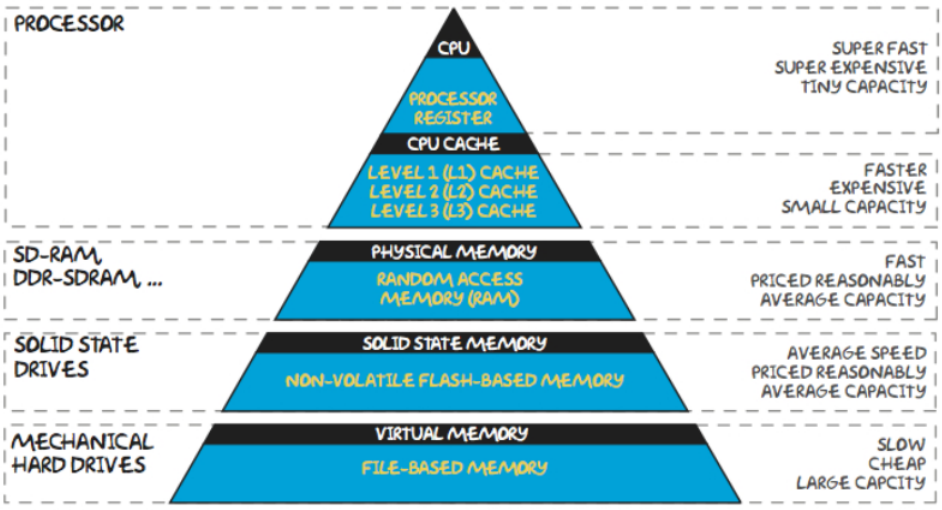

Figure 13.2: Memory Hierarchy

Definition 13.2.1 (Register)**.** Top of the hierarchy there are registers (very few of them, low capacity, very expensive, they have to be managed explicitly).

Definition 13.2.2 (Caches)**.** Just below register, they are managed automatically, the caches every access in the RAM.

Definition 13.2.3 (RAM)**.** Main memory for the program, every time we use something it will be inserted in the cache.

When we want to read something in the memory the CPU will first try to read in L1 cache, if not found, L2, then L3. Let's say something is found in L3 it will be re-write in L1 and L2 in order to speed-up future access, using the principle of "if something is needed now, there is a high chance it will be needed again very soon".

Note that, concerning memory, the bigger it is, the slower it is but also, the faster it is, the more extensive it becomes. Still, it is not the only issue, in order to be fast, registers must be near the CPU, however, the CPU has a limited amount of available space and the most of it is taken by instructions (and gates). In the end, registers are not only a financial issue, but also a performance one.

The last two layers, could be merge into one as "mechanical drives", here there is a distinction between SSD and HDD.

As many process are run in parallel, a process does not have direct access to the memory, but to a virtual memory address space which will access the "real" memory. If there is not enough space on the main memory (RAM) the virtual memory address could be map to the the swap. Not that the memory of a program can only exists either in RAM or in swap, but not in both at the same time. See speed comparison on slide 8. Even RAM is 200 times slower than L1 will SSD is 3 000 000 times slower than L1.

Note that, "branch mispredict" is the estimation of time loose when we have to branch (if statement).

Of course, these times changes depending on the CPU and the technology evolution.

Slide 9 shows the CPU cycle needed for different operation. As we can see, memory write is one cycle. Indeed, we can gives the order to write, than do other things. Still, there is a potential issue with that, if we need to read from where we just wanted to write, we have to wait for it to write. Compilers are sometimes able to optimize that but it is not always the case. We can also see that floating point maths is slower than integer maths, mul/div slower than add/sub, etc.

13.3 Cache

Each cache is composed of cache lines (slots). If we take a recent Intel Core i7 for example, it has a L1 cache of 32KB / 64B Lines, L2: 256KB / 64B Lines (associativity: 8). What does it mean? Whenever RAM is accessed a 64B-aligned chunk of memory is pulled into the caches (so, 64B-aligned just mean that we load a multiple of 64 bytes in the cache).

If we want to know the cache line index (in order to do cache lookup) the formula is the following: $(address\%cachesize)/64$ (let's say address is 129, cache size is 32Kb truncating with 64 we get 1, which is the second line of the cache).

Note that L1 does not contains the 32Kb most recently accessed memory because of cache conflicts (address 1 and 129 would have the same cache line of 1). That's where the associativity appears, an associativity of 8 means that up to 8 conflicts are allowed, there are 8 version of each cache line, in the end, it means the the total memory has only a cache size of 4kb. Not that all "sub caches" are checked in parallel.

Some other caches exists TLB (Translation lookaside buffer, maps the virtual addresses to the real memory address) and instruction cache that contains the CPU instructions.

13.4 Instructions

As we have seen before, instructions does not have the the same cost. As it is CPU dependant, two CPUs can have different cost for the same instruction.

Details of CPU affect performance:

- Memory caches
- Instruction pipelining

13.4.1 Instruction pipelining

Every instruction is executed in multiple stages: fetch(fetch the instruction), decode(understanding the instruction structure and convert into to the real CPU instruction (micro-code)), execute (execute the micro-code), writeback(write in register if needed).

The idea is that, when we execute an instruction, we can decode the next one and fetch the second one, doing that will allow to keep the pipeline as busy as possible if it fails to do that it is called "pipeline stall". Note, that sometimes, we are forced to have that stall, like in branch mispredict (in a if statement, the CPU will try to predict the branch speculatively, if it is wrong, it have to cancel the actual instruction and wait to fetch the next one. So, in case of mispredict, we'll get some states that does nothing, waiting for the "real" next instruction). An other way of pipeline stall is data dependency, it occurs when an instruction depend on the result of an other instruction. Compilers try to optimize (reorder instructions) that but it is not always possible.

More advanced pipelining is Superscalar processors exploit ILP (Instruction Level Parallelism) which runs multiple pipeline in simultaneously on the same CPU core. It is used for independent instructions from the same program. These pipelines must be filled explicitly with VLIW (Very Long Instruction Word).

Another variant is SMT (Simultaneous Multi-Threading), the most know is the Intel's variant called "Hyper-Threading". It filled the pipeline with instructions from another thread or process in order to be sure to keep the pipeline busy. There is still a pitfall for that, as the two thread will access the same cache (as running on the same core), it can be slower than running the two threads separately because the second thread can pollute the cache for the first thread and thus, degrade is performances.

There is also SIMD (Single Instruction Multiple Data) it is not really a pipeline architecture but a capacity of processors to exploit data-level parallelism. It use SIMD instruction that allow to run an instruction on an array of data. It is very useful for image, sound processing, number

crunching, machine learning, etc. is SIMD always the solution? No! First it only on data on which we want to use the same operation but also, data needs to be packed together (contiguous in memory). Thus, even without SIMD it is already pretty fast (as the would be cache in the same time) SIMD is just even faster.

13.5 How can programmer improve performance?

We should be very careful on algorithm and data structure that we use. We should consider both average and worst case complexities, but also consider the quantity and pattern of memory access.

We should also give as much hints as possible to the compiler (as it is not always possible for the compiler to infer it) some example could be the "final" modifier of Java or storing an intermediate value instead of recomputing it.

Compilers are *often* smart Truffle is sometimes used by final, even if it's not supposed to. This is part of the compiler optimization budget. Optimization is important but we don't want the compiler to loose all its time doing that. Specify "final" allow it to not loosing time as it trust the developer.

In the end, compiler improve cache utilization (cache locality) but also improve processor utilization (avoid pipeline stalls) and remove obstacles to optimization!

Chapter 14

Optimization

In the previous chapter we have look at how the compiler can improve the performance. Cache utilization:

- cache locality (access things stored together at the same time)
- Register allocation smartly (as we manage them!).
- Avoid bloating the code cache (do not emmit a lot of instruction even if could theoretically it could made the code faster in term of cycle because they will pollute the code cache that will itself cause cache miss which cost a lot of time!)
- Avoid cacbe collision by disaligning memory
- Escape analysis: don't allocate heap memory that will be deallocated before leaving the function

CPU utilization (avoid pipeline stalls)

- Instruction scheduling (move code to fill the pipeline (be care with dependencies))
- Partial evaluation (constant folding & propagation (do things in compiler time instead of runtime))
- Dead code elimination (remove code that is unreachable can improve cache and avoid loosing time in branching)
- Common subexpression elimination (avoid recompute the same thing twice)
- Auto-vectorization (automatically emit SMID instruction, data-level parallelism SIMD)
- JIT: select the good instructions for the actual CPU

A third one is remove obstacles to optimizations: inlining (as optimization is done in function and not cross function, inlining allow more optimization). We can also "cheat" is to create two versions of the code that are easy to optimize. Create specialized version associated with inlining allow very nice optimization with respect of condition/previous functions, etc. Note that in we would like to avoid jumps and loop as much as possible (jump is costly!). The compiler really likes "basic blocks" (sequence of instructions with a single entry point and a single exit point, no jumps in it.)

14.1 Modern optimization process

Very few optimization is done on the AST, instead the AST is converted into IR and then, run through different passes the same IR can be use, or we can use a different one. Note that HotSpot and GraalVM's IR is NOT Java Bytecode. Phases can be repeated $inlining \rightarrow constant - folding \rightarrow dead - codeelimination \rightarrow inlining \rightarrow ...$ optimizations enable further optimizations! Remember the example of chapter 2

14.2 Optimization Concepts

14.2.1 Static Single Assignment Form (SSA)

Definition 14.2.1 (SSA)**.** Form where variables can only be assigned once, create new variables when needed.

Almost every language are mutable (variables can be reassigned), SSA deals with that by creating new variables. It factors out work that is needed for many other optimizations (dead code elimination: if a variable is not use, skip compute/store it. It is also easier to reason in term of immutable variables.) There is an issue for code like this one:

```
1 if(cond) x = 1;
2 else x = 2;
3 print(x)
```

The SSA code would be:

```
1 if(cond) x1 = 1;
2 else x2 = 2;
3 x3 = φ(cond, x1, x2)
4 print(x3);
```

This is called a phi nodes, to reconcile branches. Note that the SSA form is not executed! It is just used to perform optimizations.

14.2.2 Data Flow and Control Flow

Data Dependence Graph (DDG) encode data dependencies ($x = y + z$ x depends of y and z). This is made possible thanks to SSA. What we what is to determine liveness of a data.

Definition 14.2.2 (Liveness)**.** Tell us if the data is used/needed in the end.

Control Flow Graph (CFG) encodes ordering relationship s between instructions it tells us what instruction has to happen before each other. It runs from the top to the bottom (except for loop) back edges for loop, splits and merges are used for condition. This graphs gives us the reachability

Definition 14.2.3 (Reachability)**.** Tell us if an instruction can be reached or not.

Both *Liveness* and *Reachability* are use to detect dead code (does not compute/store value that are never used nor instructions that are not reachable.) Note unreachability is not always the developer's fault, it can result of other optimizations. Look at the example in slide 10. We can see that the IR show the storage in the field but not in the local variable indeed, the field is part of the control flow graph while the local variable is not. Why? Because writing to a field is a side effect (other code can also modify it) while local variable can be move around.

Note that, when designing a language designing our own optimizations is not a priority!

14.3 Other optimizations

14.3.1 Partial Evaluation (PE)

Partial evaluation can refer to the Futurama projections the (take interpreter, feed it into a specializer to get a compiler). What we'll see it's the same idea in a smaller scale.

- Constant folding and propagation (remove variable by there value)
- Dead code elimination (if condition is always false, remove it a compilation time, remove never used assignment or unreachable code)

- Bypassing virutal lookups (when we now the concrete type of a variable)
- Removing runtime checks (if the array size and index are known we can remove the checks) in dynamic language we can remove typecheck if we know the type of the variable.

14.3.2 Why does compiler hate jumps and loops?

Jumps may cause cache misses which, once again is a loose of time. Conditional jumps might cause branch mispredictions (pipeline stalls). It is also harder to optimize a basic block that have multiple predecessors.

Concerning loops, the code can be executed a lot but we can only make assumption that hold in each iteration. Note that, in the other hands, compilers also love loops as programs spend a lot of time in hot loops (loops that are called often). Most of the optimizations are done in loop.

14.3.3 Loop optimizations

Loop Invariant Code Motion (LICM)

- The idea is to extract code from the loop (hoisting) (like an foo=true assignment, that will be removed from every iteration)
- Move code between iterations (global value numbering) (if x[i] and x[i+1] are used in the same iteration move x[i+1] to the next iteration to avoid new memory read)

Loop unrolling

Fully unroll statically bounded loops (a for loop that call 3 times a functions will be unroll as three call to the function).

Peel off We could also peel off the first iteration it will helps us to make assumption for the other iteration (can sometimes enable more optimizations).

Loop inversion/rotation instead of:

```
1 while(x) <body>
```

we will have:

```
1 if(x){
2     do{
3         <body>
4     } while(x);
5 }
```

It seems more complicated but it avoid jumps as the test is at the end of the loop!

Loop fission and fusion Improve locality by splitting/merging identically-bounded loops (see slide 15 for example).

Loop unswitching instead of having a if else inside the loop, we will transform by if/else with a for loop in each statement. It makes sense as the loops takes more time to be executed than conditions.

Identifying induction variables the compiler try to identify induction variables, in the example on slide 15 the compiler is able to transform the MUL operation by a SUM because it understand how it works, of course, it this toy example a smart compiler would have figure the loop is useless and completely drop it.

14.4 Other

14.4.1 Common sub-expression elimination

Avoid computing the same value twice. Note that it is only possible if the computation does not introduce side effects (writing to memory, System call, etc.), or its opposite rematerialization (to avoid read from memory, which can be slower).

Tail-call optimization

It converts tail recursive function to loops, it removes the stackoverflow risk and function call overhead.

Escape analysis

If an allocation does not outlive the function, allocate it on the stack instead of on the heap (allocation is expensive and touch new memory, which is bad for the cache).

Alias analysis

This is for language with pointers where we should ask ourself if two pointers can refer to (or overlap) the same memory region. If they can, we should be careful especially, we should wait for a write to finish before making an other one!

Peephole

Pattern based optimizations / strength reduction, it replace the use of an operator by the use of a simpler operator (for the CPU). See examples on slide 17. Note that things like push followed by a pop can happen as a result of other optimizations.

14.5 Why don't we always inline?

The first reason is the executable size that will be generated if we would inline everything. Compilation time, even if we could optimize the code again and again, nowadays computer are fast (and developers expensive) so we don't want developers to loose time during the compilation. With a JIT the compilation time is a complete performance lost (time spend compiling is time not spend in the program but also, program is running as a slow version).

However the main disadvantage is that, inlining create instructions, if we inline everything we will have a lot of instructions which will cause a lot of cache misses (which are very expensive).

We have seen that inlining is used to optimize function, so we can we do if we don't do inlining? We can compute per-function properties:

- Is the function recursive (directly or indirectly) it is useful cause it gives hints on what we should optimize. Maybe we don't want to inline recursive function as we know we won't be able to inline the all function.

- Does it stores its parameters somewhere? Useful for escape analysis, as if the function does not store its parameters, we can then escape analyze them and allocate them on the stack instead of the heap.
- Is it read-only? (does not write to any memory location) helps with common sub-expression elimination.

14.6 Summary

How can we improve compiler performances?

- Improve cache utilization (cache locality (access things stored together at the same time), cache occupation (make sure we don't put too many things in the cache)).
- Improve CPU utilization (if instructions and pipeline stalls)
- Remove obstacles to optimizations (inline, basic blocks as big as possibles)

Chapter 15

Profile guided optimization

The idea is to get profile information during the compilation (or in the interpreter) of the program, create a profile then use this profile to compile efficiently the program. We can see the pipeline using profile here:

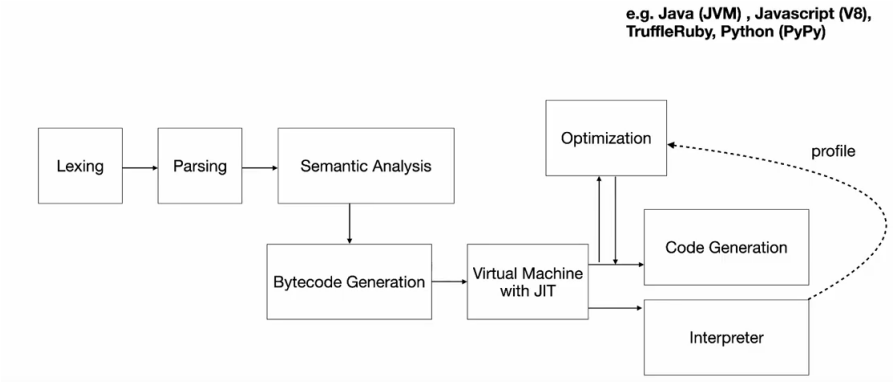

Figure 15.1: Pipeline with profiling

We can also do ahead of time profile optimization as this pipeline shows:

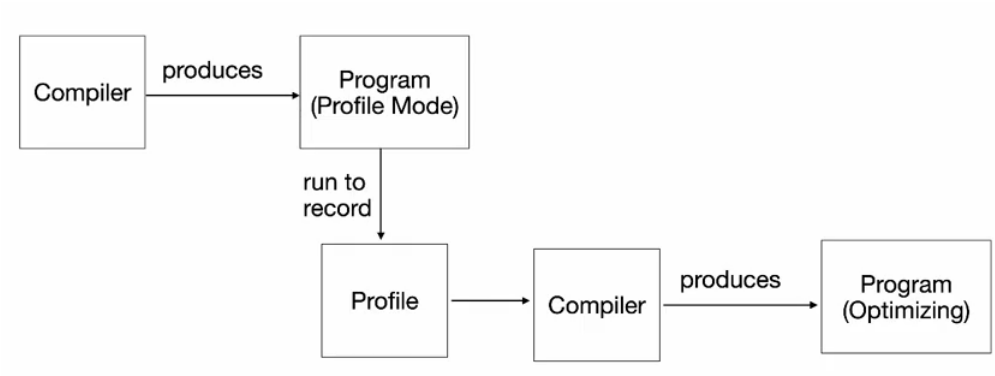

Figure 15.2: Ahead of time PGO

The profile helps determine what is common/uncommon and which specialized version to optimize.

15.1 What's profiled?

The number of calls to each method and the number of loop iteration indeed, if a loop does millions of iteration we want to optimize it as soon as possible. This can lead to an interesting point where a function called once but with millions of iterations can be replacement from it's slow version to the fast one during its execution (called unstacked replacement or osr).

Concrete type of Object are also profile (particularity efficient on OOP) it helps to determine the actual type of virtual calls.

We can also profile the probability of each branch in a if. Values can be also cached thanks to profile, but in order to do that we must be sure that the value are stables. Which mean that they won't change, won't change often or small different values will be used. Sometimes, we start recording, then give up because to values becomes to broad.

15.2 Using the profile

15.2.1 Call and loop counts

They are used to determine and prioritize which function to compile (if a function is not called enough, we won't expend resources trying to compile it.) On the other hand, if we must compile many function, we want to prioritize the most used one.

15.2.2 Concrete types

Very useful with virtual methods, let's take Java as an example, any non-static, non-final function is called polymorphic. A polymorphic can be monomorphic, bimorphic, ..., megamorphic. We can then use *monomorphisation* to replace all non megamorphic call by its concrete type!

Monomorphic calls

- Add a check that the test is indeed the type we've always seen so far (not necessary if only 1 class if we can't update at runtime. Note that Java allow loading classes at runtime.)
- Call the call target directly.

Doing this gives us two advantages:

1. Avoid a potentially expensive lookup
2. As we know the exact code that will be called, we can inline it.

Bimorphic calls

Replace the lookup by type checks on the known types

```
1 method()
```

becomes:

```
1 if(type1) method1();
2 else if(type2) method2();
3 else deopt();
```

deopt() allow to de-optimize in case new classes are introduce during runtime.

Megamorphic calls

The more classes we get, the less advantages we have cause we will start to pollute the cache. In case of megamorphic we will just give up inlining.

15.3 Dynamically typed languages

If we try to compile these languages we will ends up with a lot of code (need to check for every type possibility, see slide 8) which is bad for performances:

- More code
- So more cache pollution
- So more cache misses
- Besides, we cannot inline must otherwise the cache would end up killed by the amount of code.
- The control flow will be huge (does not allow nice and optimizable basic blocks)
- It would also be very hard to make any assumptions as there is many ways to reach some code.

15.3.1 Specialization

It's the solution to the problem we just saw. It's like loop unswitching but used in every operation. It moves conditions to the top and basic blocks in the middle. Instead of checking type of operation for every block, we do it only once! Using it with profiles allow to not compiles branches that have been never seen in practice.

We can even go further and write our own specialization (like in Truffle). Thanks to that, we can gives the compiler higher level hints about optimization that we, programmers, know and for which it is unaware. See example slide 10. On the example of Truffle on slide 12, we can see that, thanks to profiling we avoid a lot of specialization compilation as the program see, it's always integer specialization on the left and string specialization on the right. If we use numbers, specialization will be compiled out, but if we use variable, we'll just have guards on the top that does typechecks. If a guard fails, we need to de-optimize and update the profile, then we can recompile with the new profile! Another example is given on slide 14.

For now we have only seen specialization for type, but actually we can do it on anything we can compute! Especially for conditions, where the goal will be to create fast path for simple cases and compile complex one only if we really need them (remember that adding compiled code is bad as it bloat cache). These kind of specialization can be use for empty dictionaries/arrays as the logic is much more simpler in that cases. String encoding or field structures can also be cached.

We can also cache values that are expensive to compute. Often these are values that are computed during runtime (method lookup) and not user-defined. In Truffle these caches are per-specialization. Note the specialization condition may depend on the cached value.