プログラミング
パラダイム

Minero Aoki

なぜこんな講義があるのか

- すぐには役に立たない内容を学ぶため
- すぐに役立つ知識はすぐ役に立たなくなる
- 5日のうち1日くらいはツラいめに合うのもいいのではないかという親心

まえふり

プログラミング言語は
たくさんある

- このインターンで使ったものだけでも……

- Ruby（ウェブアプリ）

- Swift（iOSアプリ）

- Java（Androidアプリ）

- Python（機械学習）

プログラミングパラダイムもたくさんある

- 手続き型
- 関数型
- オブジェクト指向型
- 論理型

パラダイムは混在している

- Rubyは純粋オブジェクト指向だけど…
  - ブロック構文は関数型っぽくない？
  - 手続き型でもある
- OCamlは誰しも認める関数型言語だけど…
  - OはOOPのOだ

プログラミング言語の抽象・具象階層

複数の具象の共通部分、外せない部分が抽象。普通は具象から抽象の順で理解していくほうがわかりやすい。つまりパラダイムを理解するには言語を理解せねばならず、言語を理解するには言語処理系を理解しなければいけない

課題

JavaScriptコンパイラを実装せよ

……というのは

さすがに時間的に無理なので、

課題

JavaScriptコンパイラの
コードジェネレーターを実装せよ

今日実装する機能

- 整数リテラル（1, 2, 3, ...）
- 加算（1 + 2）
- グローバル変数の参照と代入（gvar = 77; gvar）
- ローカル変数の参照と代入（var lvar = 77; lvar）
- 関数呼び出し（f(77)）
- 非常に簡単な最適化

JavaScriptを厳密に知るなら

“ECMAScript 2015 Language Specification”

http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

※半分冗談だけど半分は本気です

余談

言語を理解するとは どういうことか？

- 言語を実装したら理解したことになると思うか？
- 「最小限の実装」はその言語の「本質」なのか？
- 「本質」は定義されているか？
- プログラミング言語の「本質」は、文法・意味・処理系のどこにあるのか？
- 安易に本質とか真髄とか言わないほうがよい

余談

「なぜ」は曖昧である

- 一見深淵で本質っぽい「なぜ〜なのか?」という問いは答えが複数あり、無意義であることが多い。問いは具体的にせよ
  - e.g. アリストテレスの四原因説
- 「なぜCookpadは動くのか？」
  - 質料因：AWSのコンピューターが動いているからだ
  - 形相因：毎日の料理をもっと楽しくしたいからだ
  - 作用因：プログラマーが作ったからだ
  - 目的因：レシピを投稿・検索できるようにするためだ

余談

「～とは何か」も曖昧である

- 一見深淵で本質っぽい「～とは何か?」という問いもまともな答えがない愚問。

- 「オブジェクトとは何か？」→答えられない

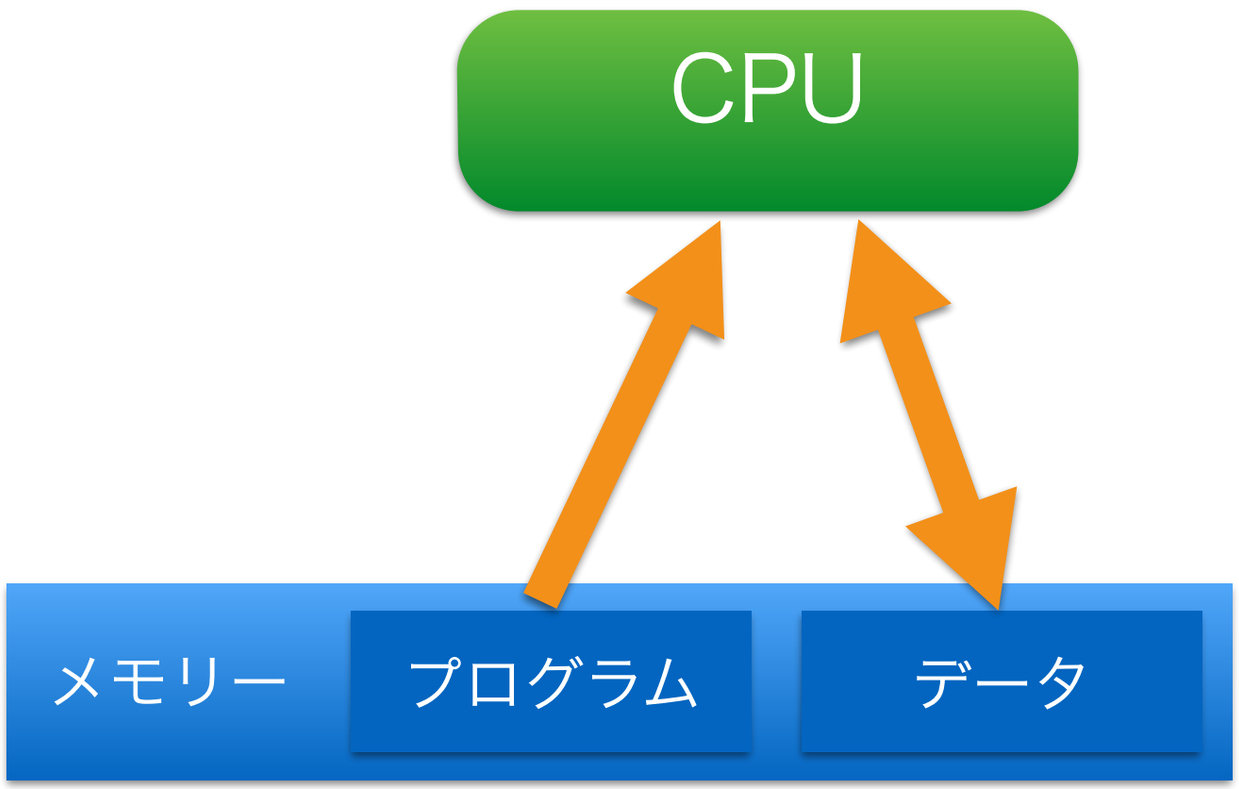
- 「関数とは何か？」→答えられない

- 「机とは何か？」→意外とうまく答えられない

- 初めて接する分野ではこの手の疑問を発してしまうことが多いが、考えても無駄な問いは慎重に自制すべきである

コンパイラの基礎知識

コンピューターが
プログラムを実行する仕組み



- コンピューターはCPUとメモリーを持つ
- CPUが直接実行できるのは機械語だけ
- メモリー上に機械語のプログラムを配置し、CPUはその指示にしたがってメモリーの値を変更する

おすすめ書籍

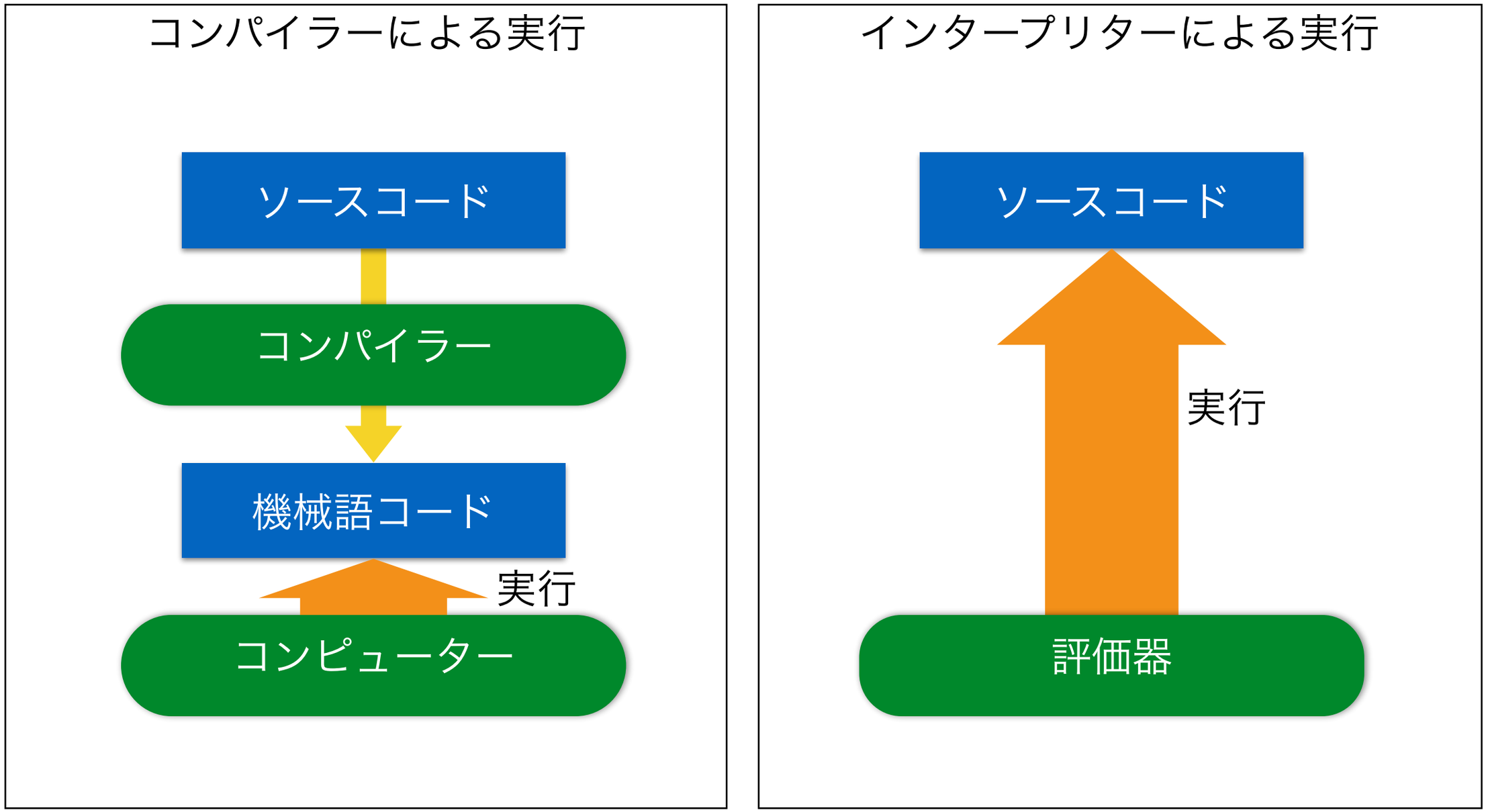
- 『はじめて読むPentium』

  http://www.amazon.co.jp/dp/4756144667/

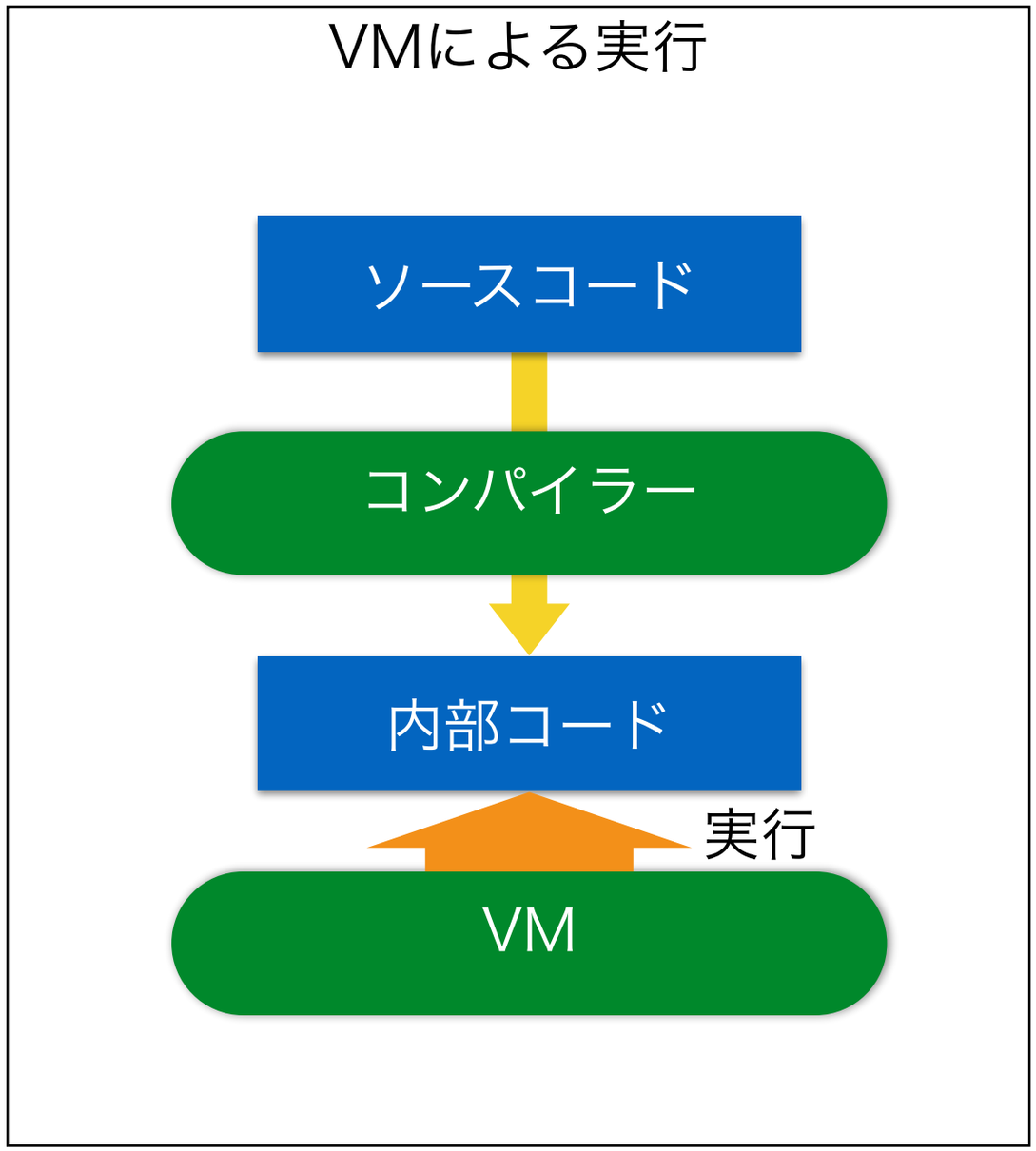
- 『はじめて読む486』

- http://www.amazon.co.jp/dp/B00OCF5YUA

コンパイラーとインタプリター

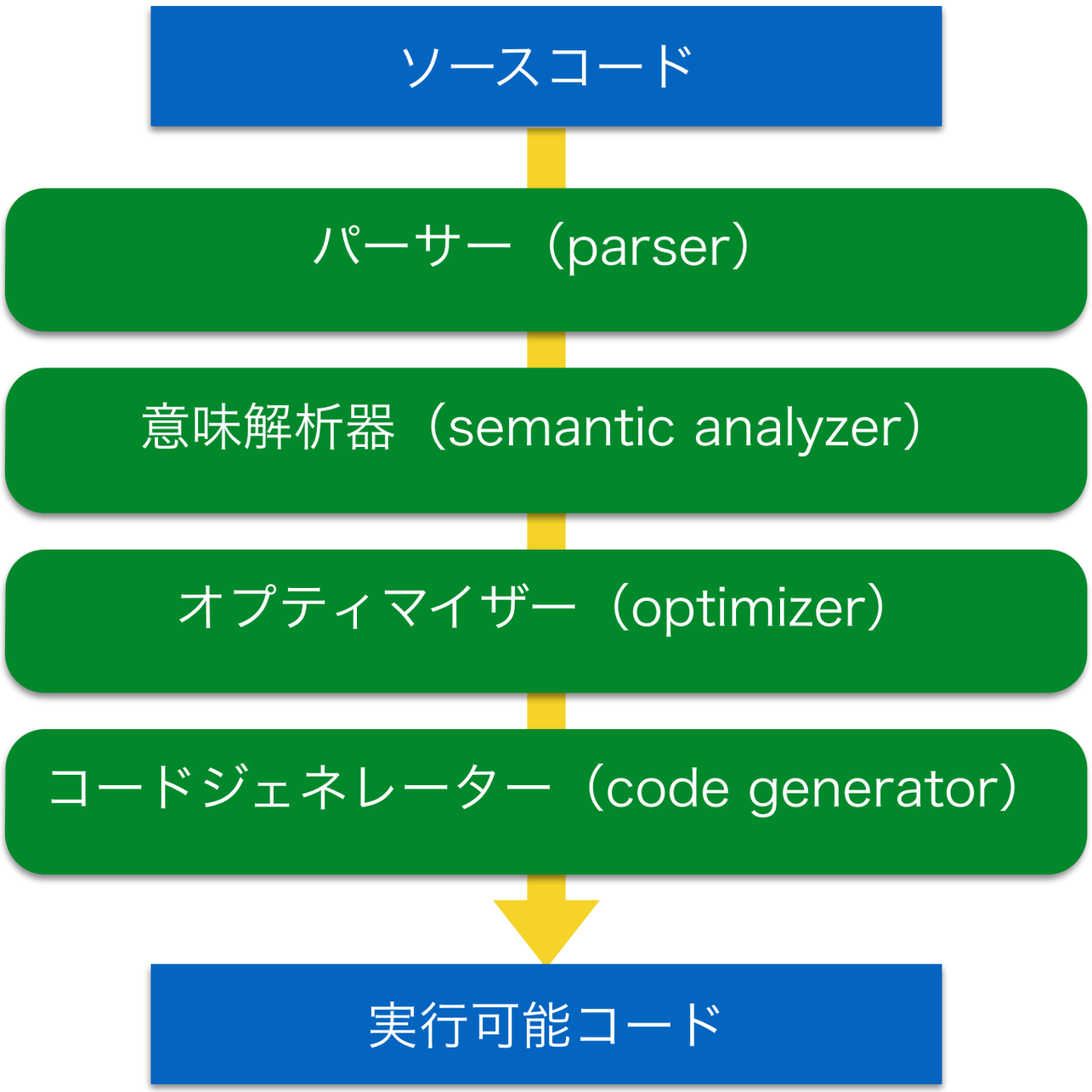


バーチャルマシン



- ソフトウェアで構築したコンピューターがバーチャルマシン（VM: Virtual Machine）
- Java VMが有名だが、最近のインタープリターはほぼこれ
- Java VMのコードはバイトコードと呼ばれる

コンパイラの一般的な構成



フェイズ1: 構文解析

フェイズ2: 意味解析

フェイズ3: 最適化

フェイズ4: コード生成

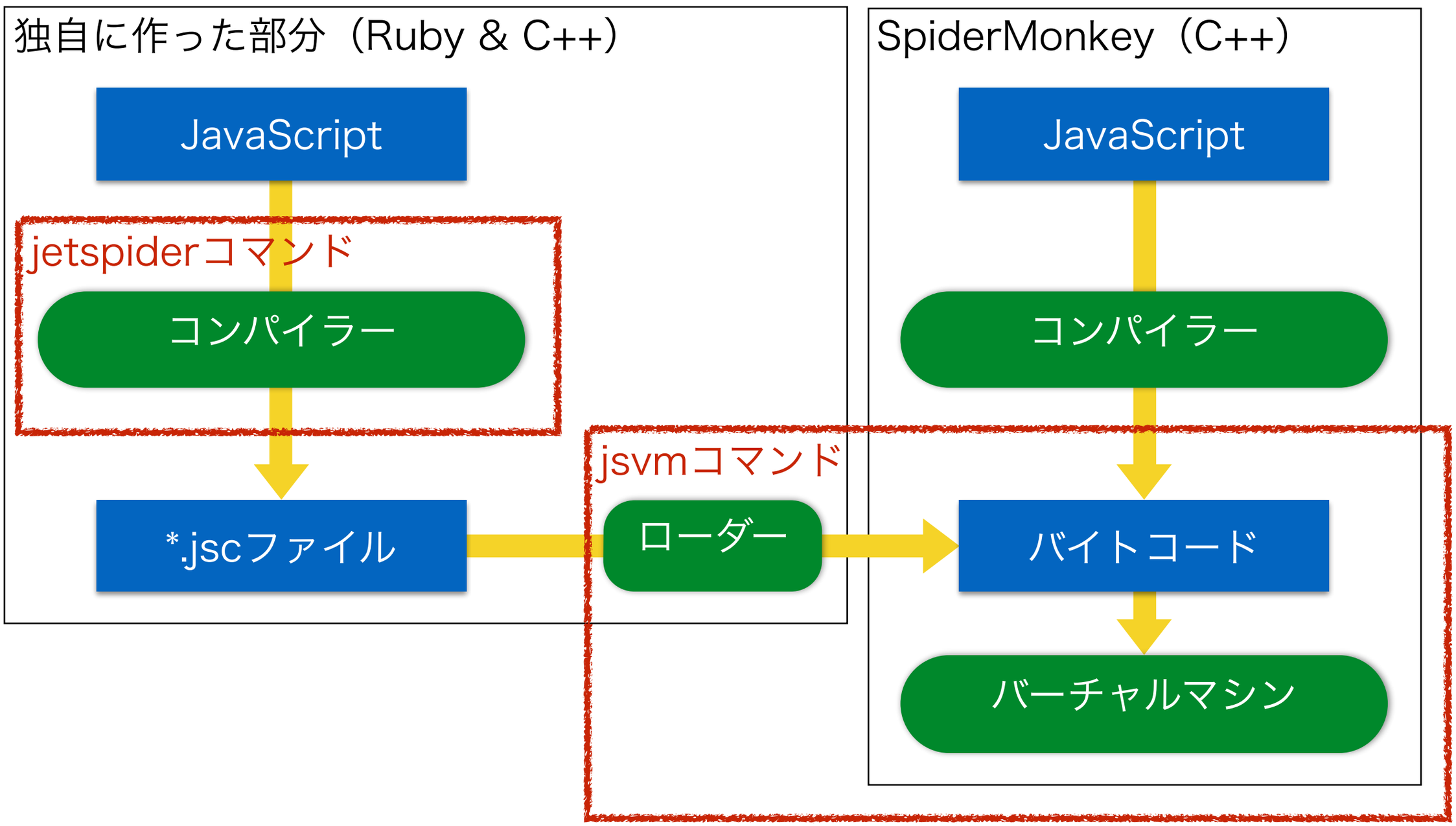
JetSpider

- JetSpiderは今回のインターン専用に開発したJavaScriptコンパイラ

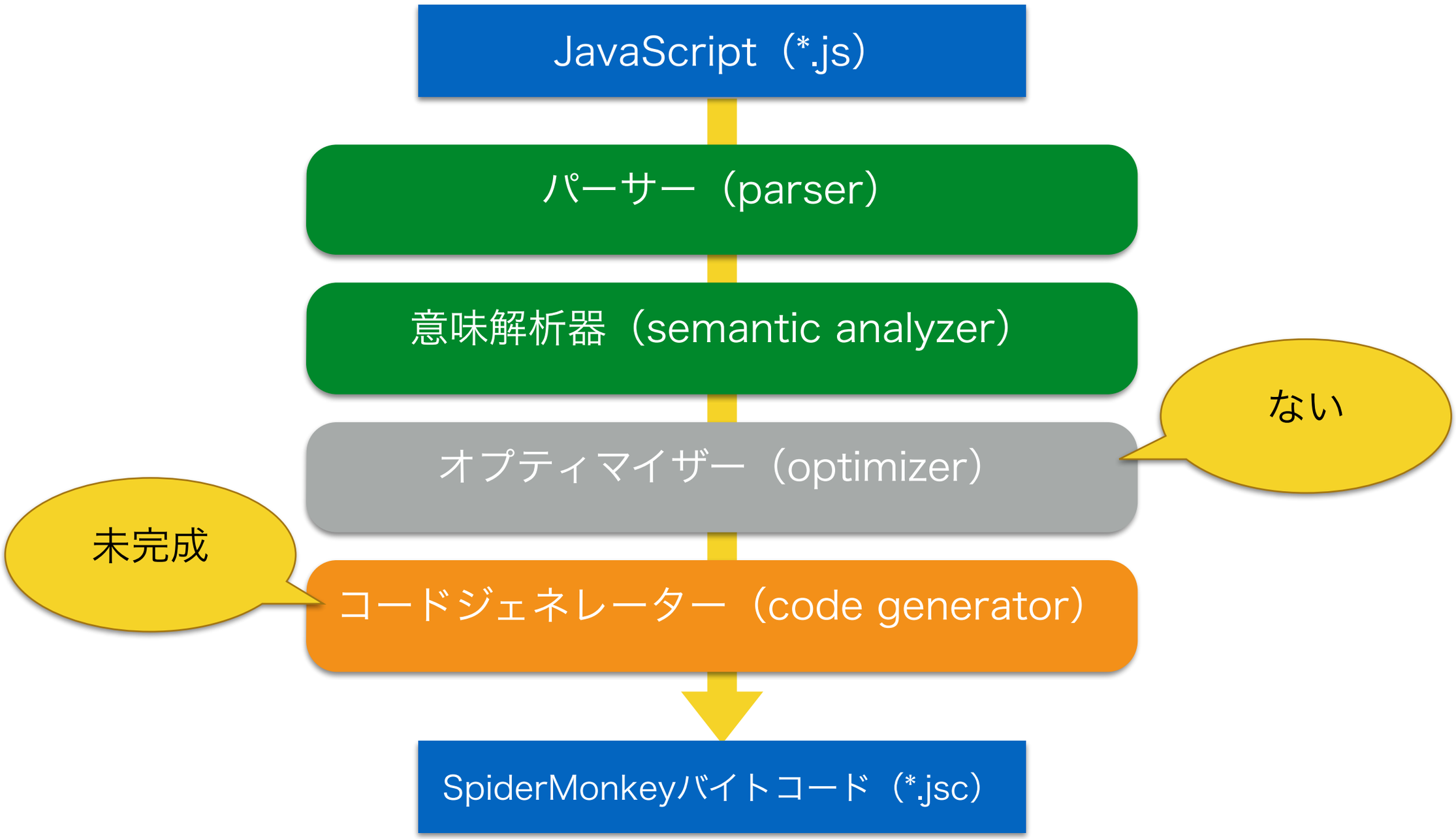
- FirefoxのJavaScript実装であるSpiderMonkey VM用のバイトコードを生成する

- コンパイラのjetspiderコマンドと、VMのjsvmコマンドから成る

JetSpiderのアーキテクチャ



JetSpiderの実装状況



JetSpiderの入手

- 以下のレポジトリをfork、cloneしてください
  - https://github.com/aamine/jetspider-course
  - clone先にcdしてbundle install

オプションは--helpに聞け

```
% ./bin/jetspider --help
jetspider [options] {JS_FILE | -e SOURCE}
        --dump-tokens                Dumps lexical tokens and quit.
        --dump-ast                   Dumps AST and quit.
        --dump-semantic              Dumps semantic tree and quit.
        --dump-object                Dumps object file and quit.
    -e, --source=TEXT                Give source code from option.
        --help                       Prints this message and quit.
```

```
% ./bin/jsvm --help
Usage: ./bin/jsvm [options] SOURCE
Options:
    -p, --print        Executes source and print the last result.
    -t, --trace        Executes source with tracing.
    -d, --disassemble  Disassembles source.
```

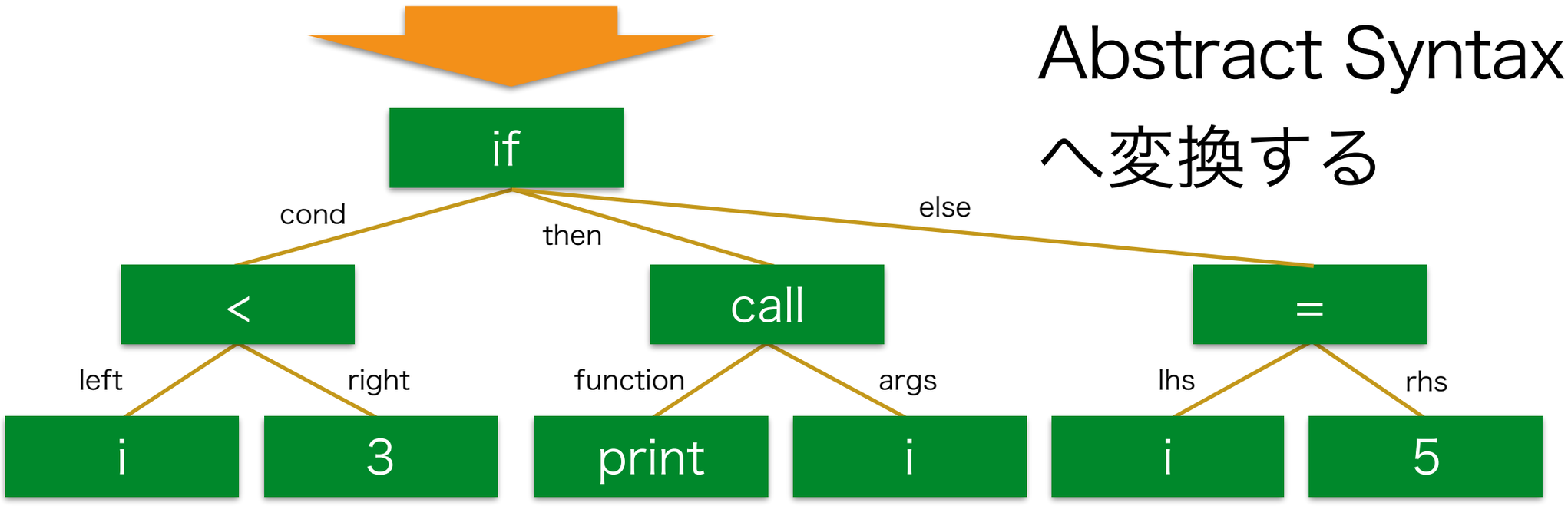
JetSpiderのソースツリー

- bin/
  - jetspider　　コンパイラー
  - jsvm　　VM
- lib/jetspider/
  - compiler.rb　　コンパイラドライバー
  - parser.rb　　パーサー
  - resolver.rb　　意味解析器
  - code_generator.rb　　コードジェネレーター　　ここを書く
  - assembler.rb　　バイトコードDSL
  - object_file.rb　　JSCファイル

コンパイルの過程

フェイズ1: 構文解析

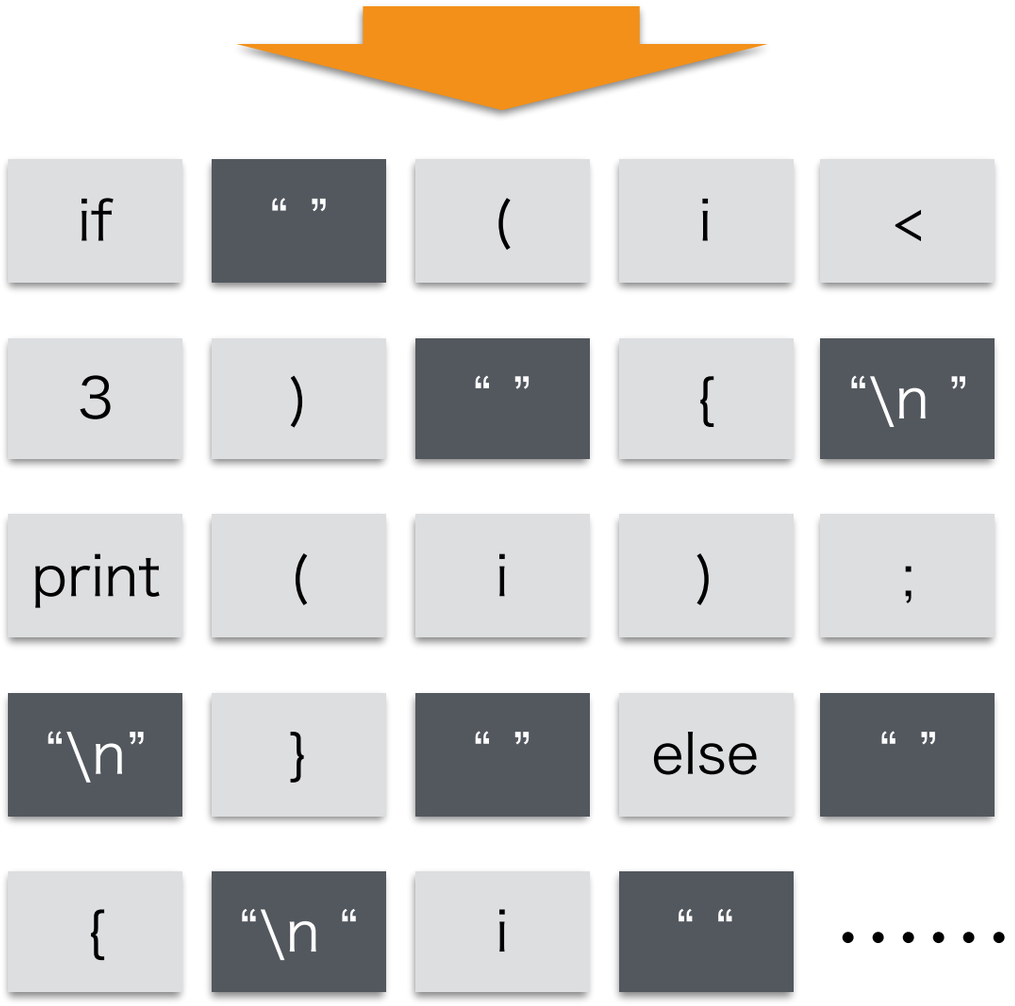
```
if (i < 3) {
    print(i);
} else {
    i = 5;
}
```



- ソースコード（テキスト）の構文をチェックすると同時に、扱いやすいようツリー構造（AST, Abstract Syntax Tree）へ変換する

字句解析

“if (i < 3) {\n print(i);\n} else {\n i = 5;\n}”



- 構文解析ではまずソースコード（文字の列）をトークン（単語）の列に分割する
- コメントや空白はこの段階で消してしまう

（ことが多い。JSやRubyには暗黙の文終端があるため改行の情報は残す必要がある。JavaDocやSQLヒントなどコメントが必要な場合もある）

[op] トークン列を見る

jetspider --dump-tokens FILE.js

▼ソースコード（exp/arith.js）

```
function f(x) {
    return 1 + x * 3;
}
```

```
% ./bin/jetspider exp/arith.js --dump-tokens
FUNCTION function
S         " "
IDENT     f
(         (
IDENT     x
)         )
S         " "
{         {
S         "\n    "
RETURN    return
S         " "
NUMBER    1
S         " "
+         +
S         " "
IDENT     x
S         " "
*         *
S         " "
NUMBER    3
;         ;
S         "\n"
}         }
S         "\n"
```

狭義の構文解析

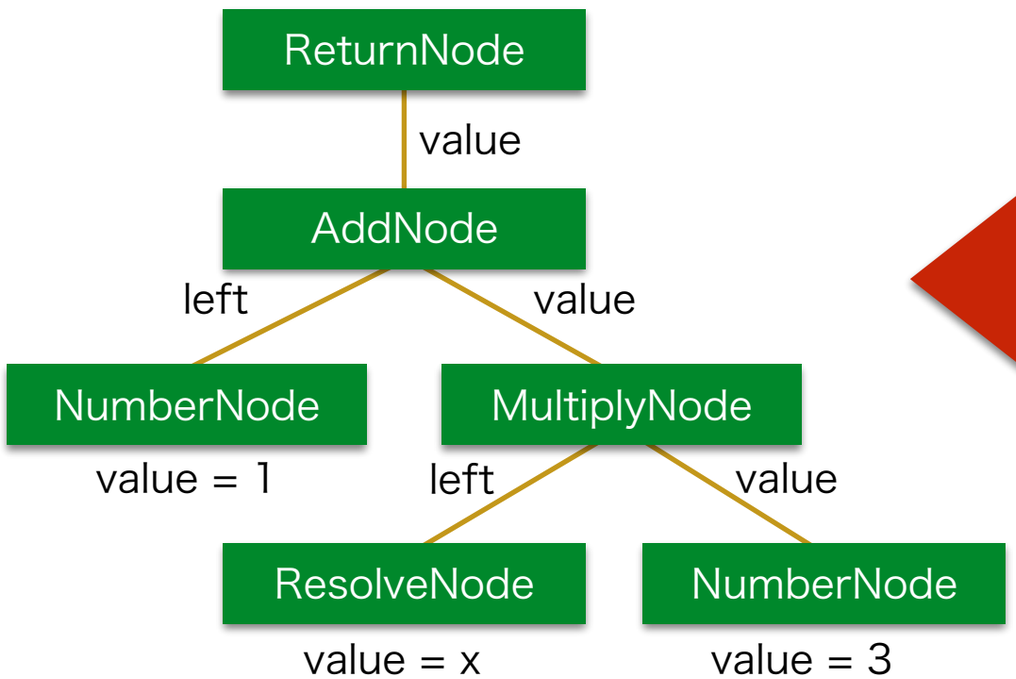
- トークン列のパターンから、より大きな構造を発見していく
- 発見したらツリーにする

[op] ASTを見る

jetspider --dump-ast FILE.js

▼ソースコード（exp/arith.js）

```
function f(x) {
    return 1 + x * 3;
}
```




```
% ./bin/jetspider --dump-ast exp/arith.js
type: SourceElementsNode
value:
- type: FunctionDeclNode
  value: f
  arguments:
  - type: ParameterNode
    value: x
  function_body:
    type: FunctionBodyNode
    value:
      type: SourceElementsNode
      value:
      - type: ReturnNode
        value:
          type: AddNode
          left:
            type: NumberNode
            value: 1
          value:
            type: MultiplyNode
            left:
              type: ResolveNode
              value: x
            value:
              type: NumberNode
              value: 3
```

フェイズ2: 意味解析

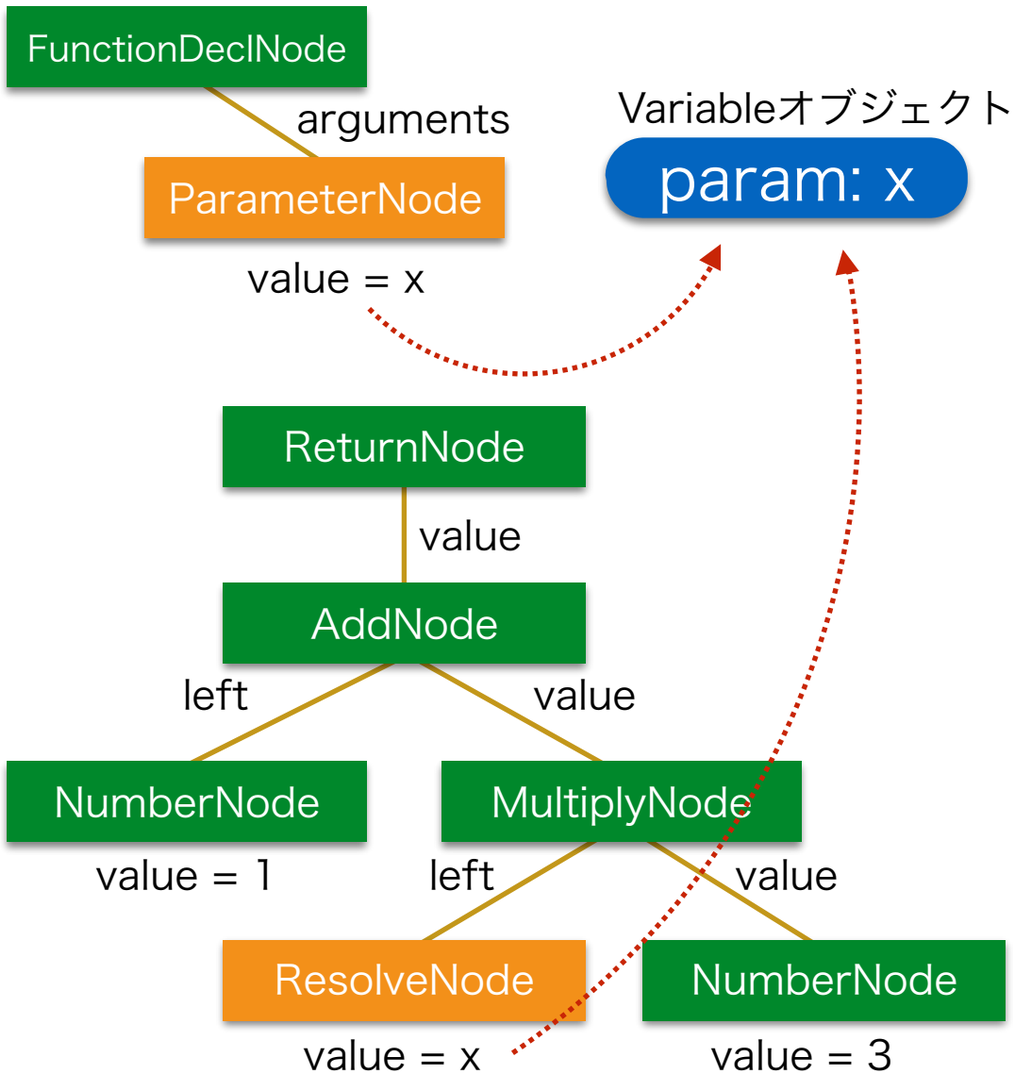
- 変数参照と変数宣言を結びつけたり、型チェックをしたりする

- ASTに上記の情報を付加したデータ構造（中間コード）を出力する。ASTをそのまま使う場合もあれば、バイトコードに近いコードを使うこともある

- JetSpiderでは参照情報付きのASTを出力する

[op] 中間コードを見る

jetspider --dump-semantic FILE.js

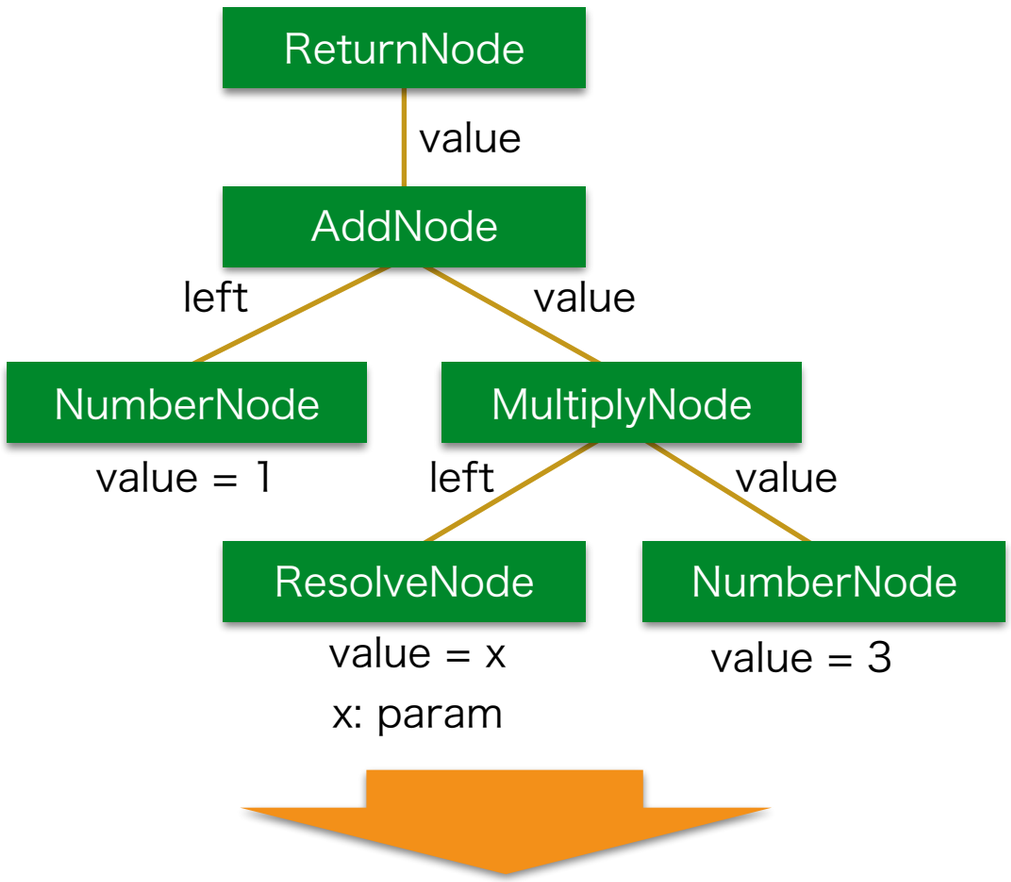



```
% ./bin/jetspider --dump-sem exp/arith.js
- type: FunctionDeclNode
  value: f
  arguments:
  - type: ParameterNode
    value: x
  function_body:
    type: FunctionBodyNode
    value:
      type: SourceElementsNode
      value:
      - type: ReturnNode
        value:
          type: AddNode
          left:
            type: NumberNode
            value: 1
          value:
            type: MultiplyNode
            left:
              type: ResolveNode
              value: x
              ref: "[function f]:param:x"
            value:
              type: NumberNode
              value: 3
- type: SourceElementsNode
  value:
  -
```

フェイズ3: 最適化

- **プログラムの意味を変えずに**実行速度を速くしたり、使用メモリ量を減らすなどの改善を行う
- 伝統的には意味解析とコード生成の間に位置付けられるが、コンパイルのあらゆる段階で行うことができる
  - 最近はリンク時最適化というものもある e.g. LLVM
- JetSpiderでは最適化を（まだ）行っていない

フェイズ4: コード生成

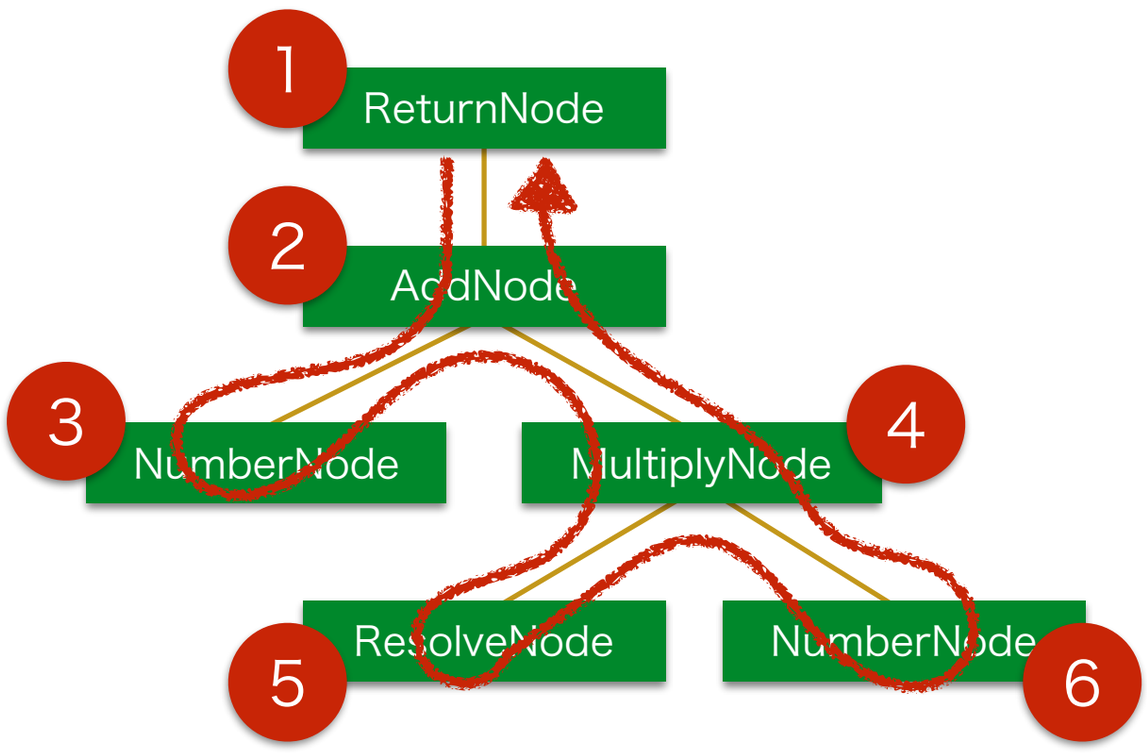



```
dd 01           int8 1
54 00 00        getarg 0
dd 03           int8 3
1d              mul
1b              add
05              return
```

- 意味解析済みのデータを元に、機械語やVMコードを生成する
- 実際はコードだけではなく、シンボルテーブル（変数表）やリンク情報も含むオブジェクトファイルを生成する

Visitorパターンで ASTをトラバースする

トラバース（traverse）



- traverse: ツリーのノードを順番にたどっていく操作

- 手続き型言語なら再帰呼び出し、OOPではVisitorパターンを使うのが定石

Visitorパターン

```
class NantokaVisitor
  def visit_ReturnNode(node)
    visit node.value      ← return文の式をトラバース
  end
  def visit_AddNode(node)
    visit node.left       ← 左側の枝をトラバース
    visit node.value      ← 右側の枝をトラバース
  end
  def visit_NumberNode(node)
    リーフなのでもうたどる枝はない
  end
end
```

visitメソッドの実装

- 他の言語だと大変だがRubyなら__send__で一発

```
def visit(node)
  __send__(“visit_#{node.class}”)
end
```

[op] コンパイルする

jetspider FILE.js

```
% ./bin/jetspider exp/arith.js
% ls -l exp/arith.jsc
-rw-r--r--  1 minero-aoki  staff  228 Aug 25 10:59 exp/arith.jsc
```

- *.jsファイルを指定するだけでOK
- 拡張子を*.jscに変えたファイル（JSCファイル）が生成される

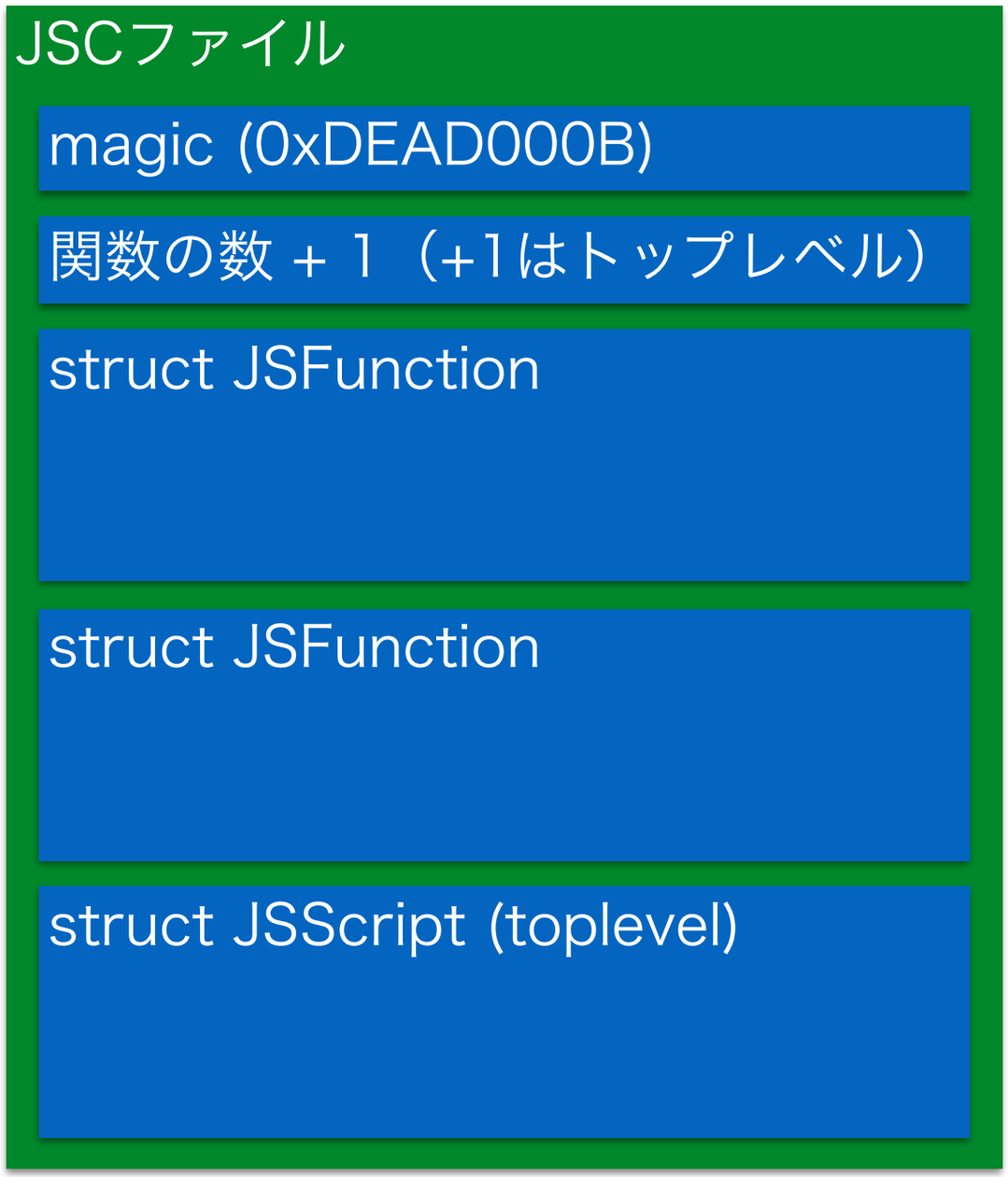
[op] JSCファイルを見る

jetspider --dump-object FILE.js

```
% ./bin/jetspider --dump-obj exp/arith.js
- n_vars: 0
  n_args: 1
  vars:
  - x
  prolog_length: 0
  js_version: 185
  n_fixed: 0
  script_bits: 32
  code:
  - dd 01           int8 1
  - 54 00 00        getarg 0
  - dd 03           int8 3
  - 1d              mul
  - 1b              add
  - 05              return
  - c5              stop
  srcnotes: ''
  filename: exp/arith.js
  lineno: 1
  n_slots: 3
  static_level: 1
  atoms: []
  blocks: []
  upvars: []
  regexps: []
  closed_args: []
  closed_vars: []
  trynotes: []
  consts: []
- n_vars: 0
  n_args: 0
以下略
```

- バイトコードだけではなく、JSCファイルの項目がすべて表示される
- 当然だけどコードジェネレーターが完成していないとエラーになってしまう
- 実装した結果を確認するために使用

JSCファイルの構造



- SpiderMonkey内では関数定義1つに対してstruct JSFunctionが1つ生成される
- JSCファイルにはソースファイルで定義されたすべての関数に対応するstruct JSFunctionが格納される
- 最後の1つは常にstruct JSScriptでメンバーが少ない

[op] JSCファイルを実行する

- 単に*.jscファイルを指定すると、そのコードを実行する

```
jsvm FILE.jsc
```

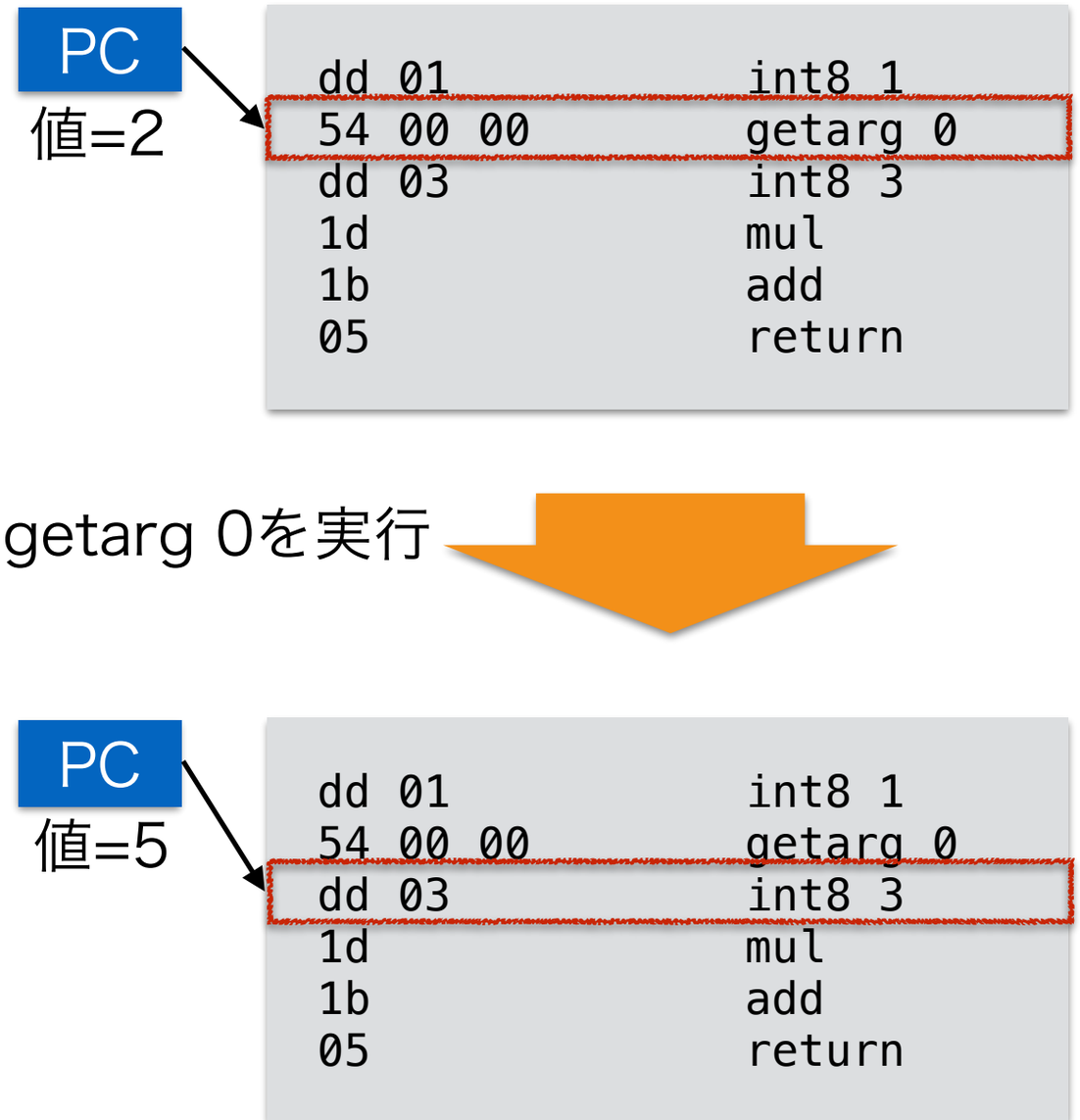
- --printオプション付きだと、実行したあと最後の式の値を表示する

```
jsvm --print FILE.jsc
```

- 関数呼び出しを実装するまでは値を表示できないので、このオプションで結果を確認する

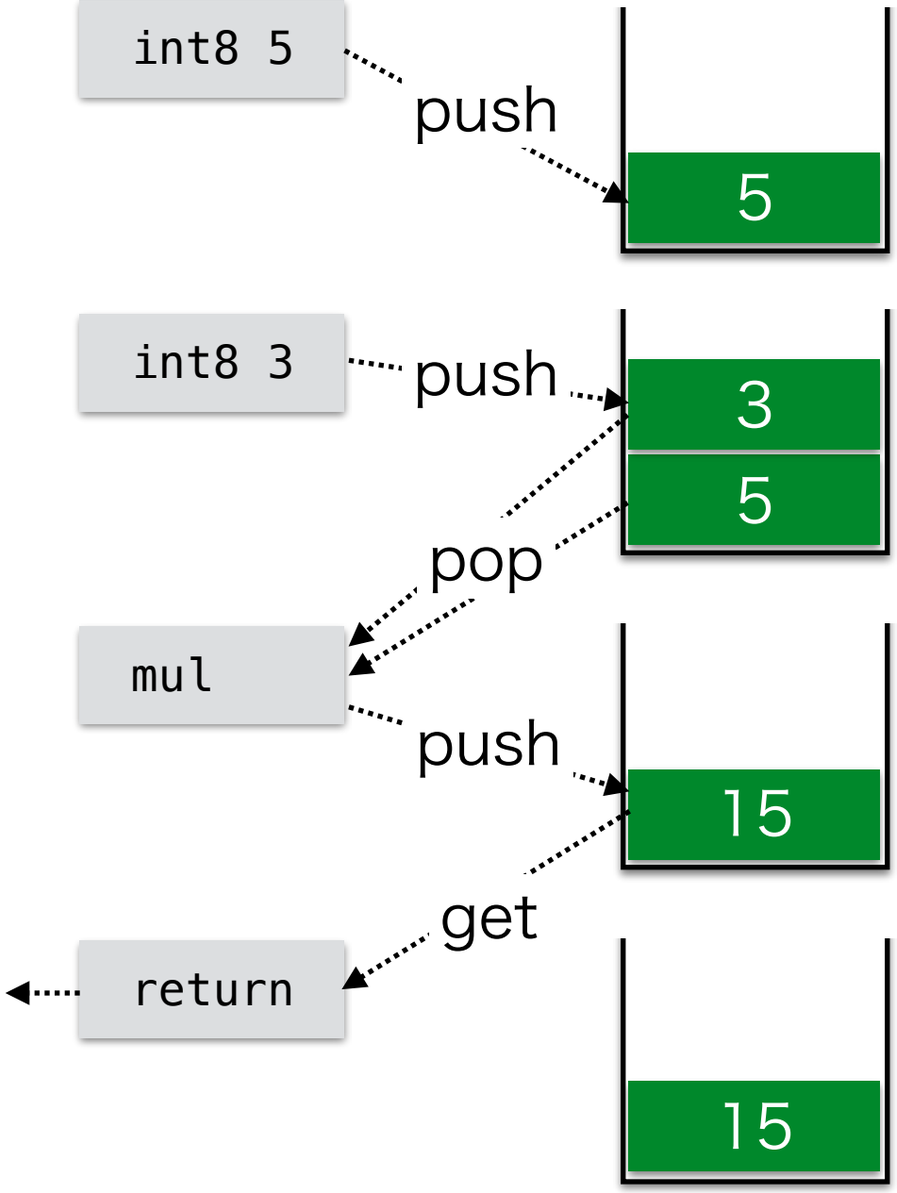
VMによる実行過程

SpiderMonkey VM 実行の仕組み（1）PC



- 1命令は1〜バイトからなる（可変長）。例えばかけ算をするmul命令は0x1Dの1バイト
- PC (Program Counter)レジスターは現在実行中の（これから実行する）バイトコード位置を指す
- PCが指す命令を実行してPCを次に進める、がVMの基本の動き

SpiderMonkey VM
実行の仕組み（2）スタック



- SpiderMonkeyのVMはスタックマシン
- 各命令は、VMスタックの先頭に値を積む（push）、取り出して計算する（pop）、参照して計算する、のいずれかの処理を行う（組み合わせもアリ）
- c.f. レジスタマシン

[op] VMの動作を見る

jsvm --trace FILE.jsc

```
% ./bin/jsvm --trace exp/lvar.js
   1: 00000: 00    nop
  stack:
   6: 00001: f1 00 00    callglobal "f"
  stack: function f() {\n    var a = 1, b;\n
p(a);\n} undefined
   6: 00004: 3a 00 00    call 0
  inputs: f, f @ 2
  output: (void 0) @ 0
  stack:
   2: 00000: 3f    one
  output: 1 @ 1
  stack: 1
   2: 00001: 57 00 00    setlocal 0
  inputs: 1 @ 1
  output: (void 0) @ 0
  stack:
   2: 00005: 56 00 01    getlocal 1
  output: b @ 1
  stack: undefined
   2: 00008: 51    pop
  inputs: b @ 1
  stack:
   3: 00009: d9 00 00    callgname "p"
....
```

- --traceオプションを付けると以下の内容が出力できる
- 実行しようとしているバイトコード
- pushとpop
- 変化後のVMスタック

JetSpiderでバイトコードを生成するには

- Assemblerオブジェクトに定義されている、バイトコードと同名のメソッドを呼べばよい
- CodeGeneratorクラス内では@asmにAssemblerオブジェクトが入っている
- 例
  - @asm.pop
  - @asm.getarg 2

課題1

- 整数定数（NumberNode）を実装せよ
  - 符号付き8ビット整数をpushする”int8”命令を使う
  - ［追加課題］符号付き32ビット整数まで扱えるようにせよ
  - ［追加課題］文字列リテラルを実装せよ

課題の実装ステップ

1. 実装すべきJavaScriptのコードを確認する
2. jetspider --dump-ast, -semでASTを見る
3. jsvm --disassembleで出力すべきバイトコードを見る
4. CodeGeneratorのメソッドを実装する
5. jetspider --dump-objで出力結果を確認する
6. 最初のJavaScriptのコードを動かして確認する

「正解」を見る

- SpiderMonkeyによるコンパイル結果を見ることができる。

- トップレベルを逆アセンブル

```
jsvm --disassemble FILE.js
```

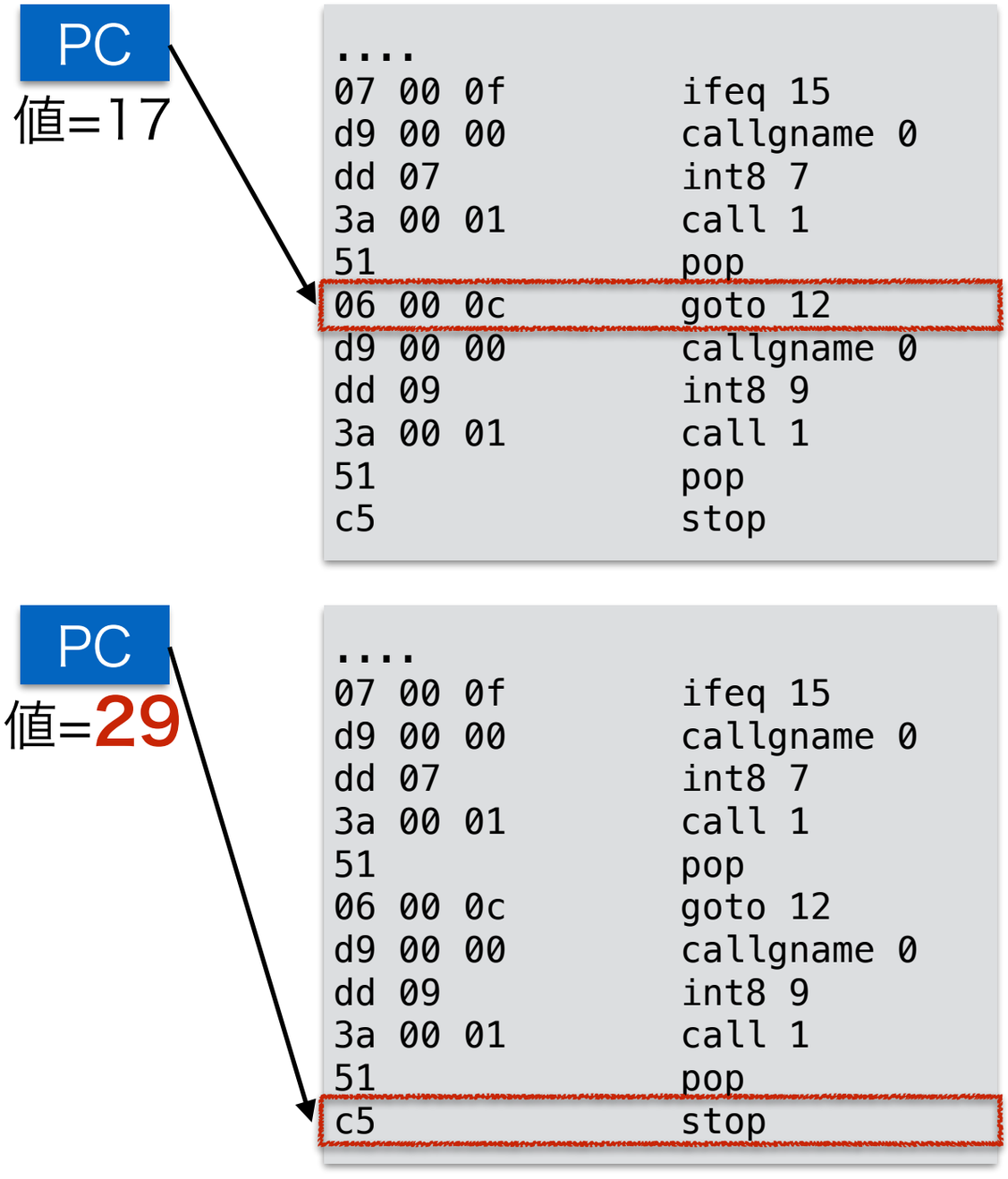
- 特定の関数を逆アセンブル

```
jsvm --disassemble FILE.js FUNCTION
```

課題2

- “+“演算子（AddNode）を実装せよ
  - “-”演算子（SubtractNode）の実装が参考になる
- [追加課題] 単項の”+”, “-“演算子を実装せよ
- [追加課題] 前置・後置の”++”演算子を実装せよ

SpiderMonkey VM
実行の仕組み（3）無条件ジャンプ



- 無条件ジャンプ命令gotoはPCを変更する
- 「goto 12」は「PCを+12」する効果がある

SpiderMonkey VM
実行の仕組み（4）条件付きジャンプ

PC
値=5

```
....
07 00 0f          ifeq 15
d9 00 00          callgname 0
dd 07             int8 7
3a 00 01          call 1
51                pop
06 00 0c          goto 12
d9 00 00          callgname 0
dd 09             int8 9
3a 00 01          call 1
51                pop
c5                stop
```

PC
値=**20**

```
....
07 00 0f          ifeq 15
d9 00 00          callgname 0
dd 07             int8 7
3a 00 01          call 1
51                pop
06 00 0c          goto 12
d9 00 00          callgname 0
dd 09             int8 9
3a 00 01          call 1
51                pop
c5                stop
```

- 「ifeq 15」は「スタック先頭が**false**なら**PC**を+15」

たぶん、「trueならそのまま次の命令に行き、falseなら飛べ」という意図だと思うけど……。
ちなみにifeqという名前のくせに等価比較もしない。

JetSpiderでの
逆順方向へのジャンプ

- 進行方向と逆へジャンプするときは、
  @asm.locationを使う

```
loc = @asm.location
@asm.pop                    ← locはこの命令のオフセットを格納する
.....
@asm.goto loc               ← locの位置へジャンプ
```

JetSpiderでの進行方向へのジャンプ

- 進行方向へジャンプするときは、まだ生成していないコードのオフセットが必要になる
- @asm.lazy_locationで具体的な値の埋まっていないLocationオブジェクトを生成し、ジャンプしたい先でfix_locationを呼んで値を埋める

```
loc = @asm.lazy_location
@asm.goto loc                ← locへジャンプ
.....
@asm.fix_location(loc)       ← locの位置を定義
```

課題3

- if文（IfNode）を実装せよ。else節は省略できることを忘れないように注意。
- ［発展課題］無駄なジャンプをできるだけ減らせ。else節がある場合は2回、ない場合は1回が最小である
- ［発展課題］条件演算子（a?b:c）を実装せよ

課題4

- while文（WhileNode）を実装せよ

- ［追加課題］break文とcontinue文を実装せよ

代表的な実行速度最適化

- 命令強度の低減
  - int8 1 (dd 01) → one (3f)
  - x * 4 → x << 2
- 定数のコンパイル時計算（定数畳み込み）
  - print(i + 1 + 2 * 3) → print(i + 7)
- 処理のループ外への移動
  - for (var i = 1; i < 10000; i++) { print(x * 2 + i); }

    ↓

    var n = x * 2;
    for (var i = 1; i < 10000; i++) { print(**n** + i); }

プログラムの意味とは

最適化のため次のような定数畳み込みをしたとする

print(1 + 2);	→	print(3);

- プログラムの意味は変わっていないように思える
  - 変わっていないもの（≒意味）
    - 式の値（3）
    - プログラム外に見える変化（「3」と表示される）
  - 変わったもの（≠意味）
    - 計算過程
    - メモリ上の値の変更パターン

※ただし言語により「意味」は異なる

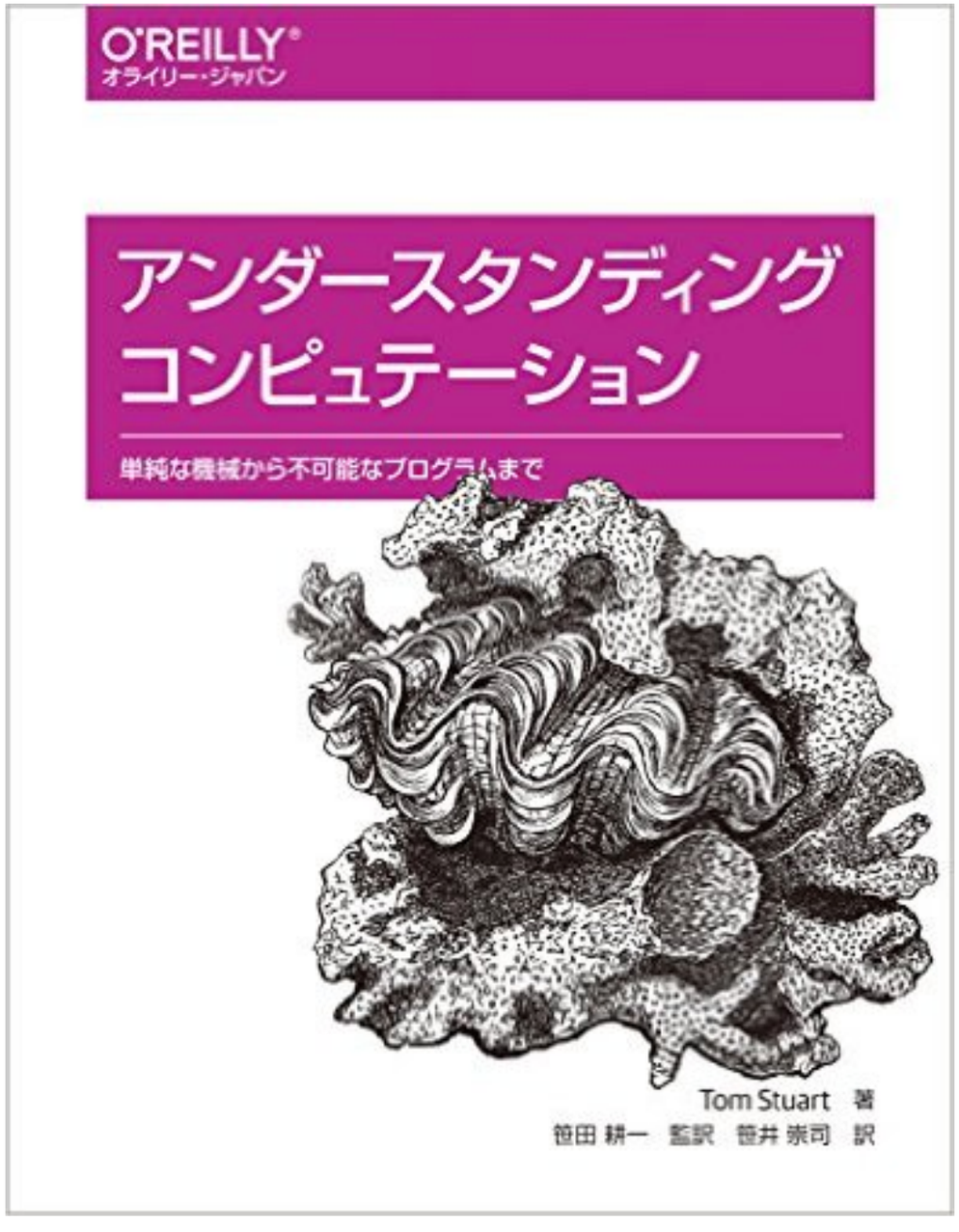
プログラムの意味を定義・表現する3つの方法

- 操作的意味論（Operational Semantics）
  - ある言語の表現を、より形式的な言語に変換することで意味を表現する。コンパイラーの定義に近い
    → c.f. チューリングマシン、ラムダ計算
- 表示的意味論（Denotational Semantics）
  - 集合や関数など数学的な概念を用いて意味を表現する。インタプリターの定義に近い。
- 公理的意味論（Axiomatics Semantics）
  - なんか論理式で表現するらしい

詳しくは本で

『アンダースタンディングコンピュテーション』

http://www.amazon.co.jp/dp/487311697X



課題5

- 整数が1のときはone命令を使うよう最適化せよ
- ［追加課題］文内での定数畳み込みを実装せよ

  ASTレベルで処理する方法と、バイトコードレベルで処理する方法の2つが考えられる。今回はAddNode、1段に限って畳み込めばよい

JetSpiderでの変数の管理

- トップレベルと関数それぞれの変数スコープをScopeクラスで管理している

- --dump-semanticで”ref”が表示されるノード（VarDeclNode, ParameterNode, ResolveNode）にはvariableというプロパティがあり、参照しているVariableオブジェクトが得られる

- グローバル・ローカルの区別やローカル変数IDなどはVariableオブジェクトから得られる

- 詳細はjetspider/ast.rb, resolver.rb参照

課題6

- グローバル変数の参照と代入（VarStatementNode、ResolveNode、OpEqualNode）を実装せよ
  - 注意！ グローバル変数の代入では左辺の評価時にbindgname命令を発行する必要がある
    - ResolveNodeのn.variableにはVariableオブジェクトが入っていて、Variableオブジェクトから変数名（name）やローカル変数ID（index）が取れる
- ［追加課題］グローバル変数は常に”$”で始まるというルールを追加して、”var”宣言なしでもローカル変数が定義されるように改造せよ（意味解析器Resolverクラスの変更が必要）

課題7

- ローカル変数の参照と代入（VarStatementNode、ResolveNode、OpEqualNode）を実装せよ
- 関数のパラメーターも別途扱う必要がある

課題8

- 関数呼び出し（FunctionCallNode）とreturn文（ReturnNode）を実装せよ
- ローカル変数やグローバル変数を参照すれば関数が得られるので、あとはcall命令で呼べばOK
- ……が、グローバル変数の場合は、変数参照のときと同じく関数を得る時点でcallgname NAMEが必要

発展課題1

- 以下のオブジェクト指向機能を実装せよ
- new式
- プロパティの読み書き
    - obj.prop形式だけでよい。obj[‘prop’]は不要（文字列リテラルを実装していないと意味ないため）
- メソッド呼び出し
    - メソッド定義はすでに定義されている関数をプロパティにセットすることで代替する（下記参照）
      function obj_m() { return 77; }
      obj.m = obj_m;

発展課題2［難］

- 無名関数を実装せよ
- ただし外部の変数は参照できなくてよい
- 実はまだ青木も実装してないので何かあるかもしれない。がんばれ。