プログラミング言語処理系論 (5)
Design and Implementation of Programming Language Processors

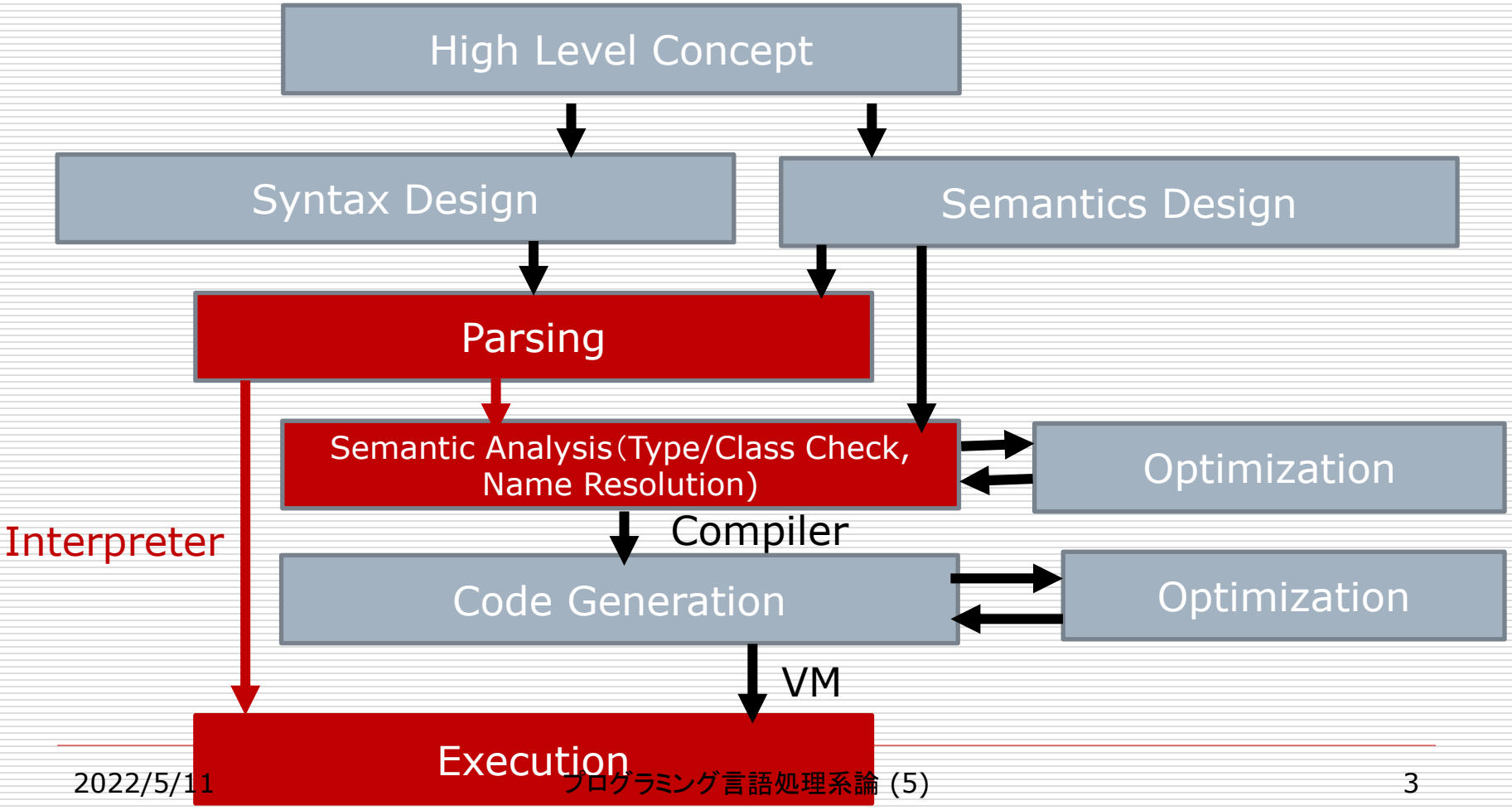
佐藤周行

（情報基盤センター/電気系専攻融合情報学コース）

今日やること

- Abstract Syntax Tree構築後 – 意味解析
  - Symbolの解析
    - 名前解決
    - Cpythonのコードでの実現
  - Source Code Traversal
- 型と型チェック
  - 型システム
  - TypeScript

パースとパース以降の作業



Parse → ASTの構築とその後

- Parseが終了したら、対応するAST（Abstract Syntax Tree）が生成される
  - ASTは何を表現しているか？
  - シンボルの「名前」の管理
    - 同じ名前をもつ2つのシンボルが同じかどうか？
    - シンボルは、どの範囲（文面での範囲、実行における時間的な範囲）で有効か？
    - これらの問題の「解決」（resolution）
  - この情報をどこまで保持するか？
    - 実行時まで？

たとえば

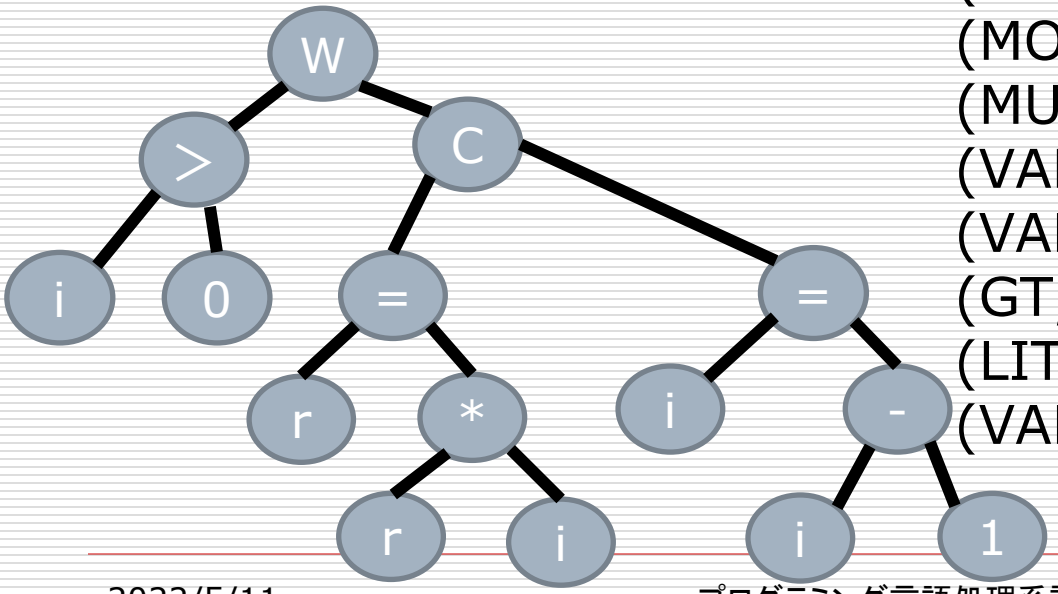
```
def fact(n) {
   if (n==0) 1
   else n*fact(n-1)
};
n=3;
```

Parse → ASTの構築とその後（続き）

- シンボルテーブルの管理
  - 大域変数の管理
  - 局所変数の管理
  - 動的な言語では、実行時にも使用
  - 静的にすべての名前解決をする処理系ではリンク時まで
- コードセグメントの管理
  - 関数コードの保存
- データセグメントの管理
  - 変数にデータをバインドする
  - では、一般的には、何を用意するとプログラムの実行に十分なのか？
  - 「実行環境」（次回以降のテーマ）

- 制御構造も木構造で表現

```
while (i > 0) {
    r := r * i;
    i := i – 1;
}
```




```
(WHILE, 2, 11)
(COMP, 6, 10)
(MOV, i, 9)
(SUB, 7, 8)
(LIT, 1)
(VAR, i)
(MOV, r, 5)
(MUL, 3, 4)
(VAR, i)
(VAR, r)
(GT, 0, 1)
(LIT, 0)
(VAR, i)
```

Symbol table

- 「名前」を格納する領域は？
- 名前空間
- スコープ
  - 局所変数、大域変数
- 何を格納する場所を用意するのが良いのか？（ヒープの設計）
- まずは変数管理のためにオブジェクトのテーブルを作る

```
typedef struct _vardat {
  int kind;
  int name;
  void *val;
  int param;
  int paramlen;
  int res_t;
} t_vardat;

t_vardat vars[];
```

CPythonでは…

- symtable.c でやっていること
  - ASTをトラバースして、
  - 変数の出現の観察
  - どの変数と変数の参照が同一のものかの同定
  - コード生成のときに本質的な情報として利用

- Pythonの不思議な変数利用規約
  - 未定義の変数への参照は、大域変数
  - 一旦代入されると局所変数へ
- 実行例を観察する

symtable.cを見る

- ASTをトラバースして解析
  - 後述するが、この準同型ライクな解析はAbstract Interpretaionの基本
- 変数の解析
  - 「環境」ごとにDictionaryを準備
  - その中で、大域変数、定義されていない大域変数、局所変数を同定 → スコープ

中心となる関数

```
#define SET_SCOPE(DICT, NAME, I) { ¥
    PyObject *o = PyLong_FromLong(I); ¥
    if (!o) ¥
        return 0; ¥
    if (PyDict_SetItem((DICT), (NAME), o) < 0) { ¥
        Py_DECREF(o); ¥
        return 0; ¥
    } ¥
    Py_DECREF(o); ¥
}
```

- analyze_name()
  - 与えられたスコープを設定した上で解析
- analyze_block()
  - Scopeが変更され得る
    - Scopesを…
    - localsを…
  - Scopeの調整をしてanalyze_nameの呼び出し

- symbol_update()
- ここまでがベースとなる処理

- 次からのsymtable_visit_*()
  - Source code traversalの典型
  - （問題 5）Pythonのsymtable_visit_*()を読み、その内部を以下の観点から説明せよ
    1. ASTの構造に従ったtraversalの仕方
    2. 各端点における処理の準同型に従った処理
    3. 全体としてなされる処理の記述

関数の導入

- 関数の導入
  - コードセグメント（関数本体）
  - フレーム
    - フレーム＝関数呼び出しを実行するための環境
    - 大域変数、局所変数
    - スコープの管理
    - 引数の渡し方（binding/association）
      - コンパイル言語なら、パラメタと実引数の対応をきちんととることが前提
- 関数のことは次回まとめてやります

オブジェクトの導入

- Classの導入
  - Class 階層の導入
    - Extends keyword
  - Access の許認可の問題
    - Private/Protected/Public
- Type/Class Check
  ← Semantic Analysis の最大の問題の一つ
  - Static/Dynamic
  - Subclass
  - Polymorphic Types

複数パスの表現

- ソースファイル中での表現

```
main(int ac, char **av)
{
  if (ac >= 2){
    if (!strcmp(*++av, "-d"))
      debug = 1;
  }

  code = yyparse();
  if (!code) exit(1);
  nextPass(entry);
…
}
```

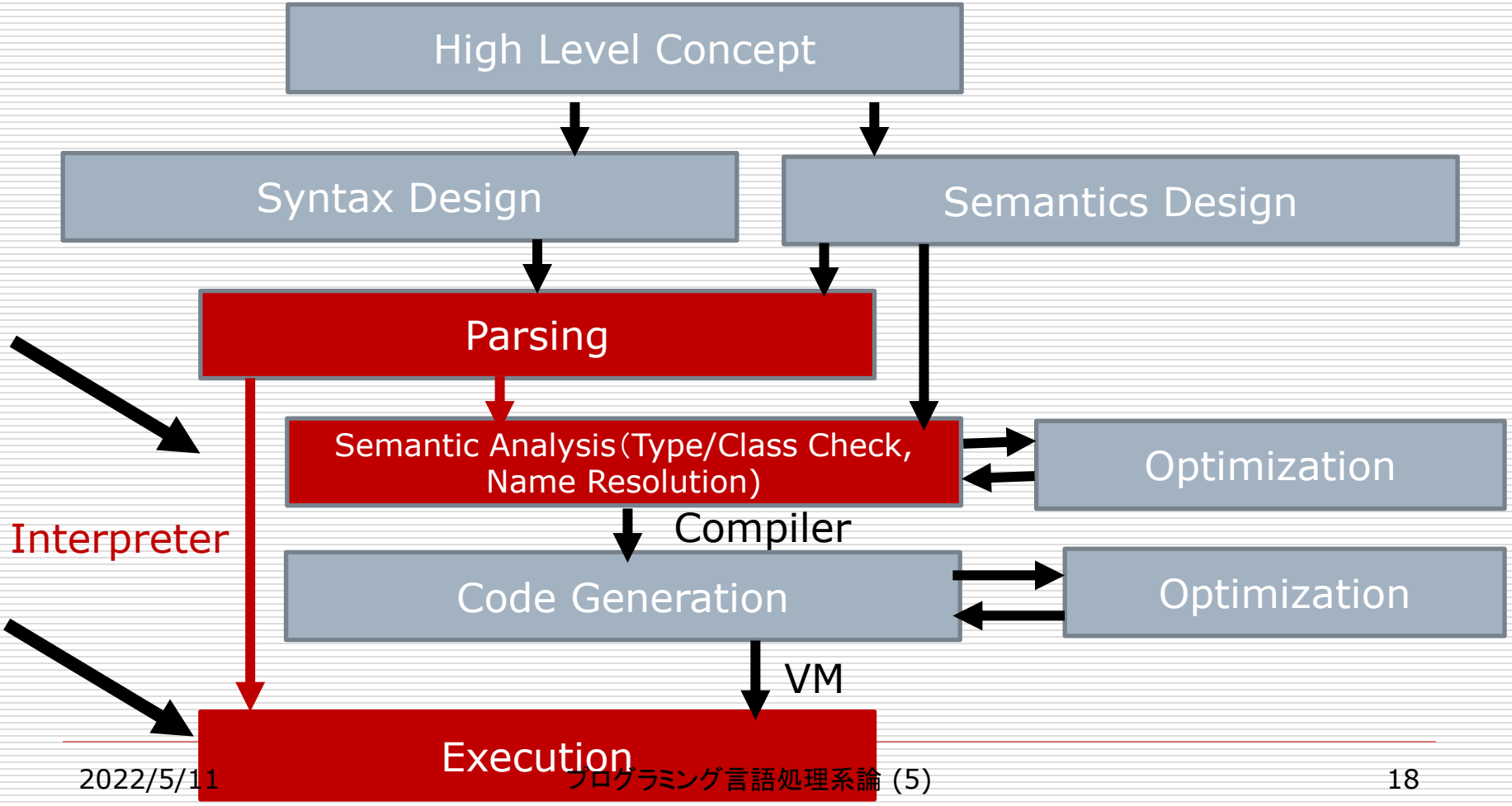
```
Node *entry;

Top: …  {Entry = …};
/* 最初か最後に、後々の
   処理のエントリーポイン
   トを設定しておく */
```

パスの考え方

- 近年の考え方では、パースまでと、パース以後の最適化をわけています。
- パースまででできることは高が知れていると認識が一般的になりました。
  - 最適化ルーチン（６月後半くらい）
- インタラクティブな言語では、パースまでの処理と、それ以後の処理（最適化等）のバランスが問題になっています。

Parse以降の作業


High Level Concept
Syntax Design
Semantics Design
Parsing
Semantic Analysis（Type/Class Check, Name Resolution)
Optimization
Interpreter
Compiler
Code Generation
Optimization
VM
Execution

Semantic Analysis

- symtable.cの中を見てきた
- Typecheckを説明する前に…

- source code traversal
  - 準同型の考え方の導入

プログラムの実行

- Source Tree Traversalによる実行
  - Perl5はこの方式を取っています。
  - 直感的でわかりやすい
- 簡単なものについてeval関数を書くことは素直に書ける。
- pから生成されるコードを[p]と書くことにすると
  - [WHILE e st] = **while** [e] [st]
  - [x = e] = [x] **=** [e]
  - [f(e1,e2, …, en)] = [f]**(**[e1]**,**[e2]**,**…**,**[en]**)**
  - …
- ちょっとまて

Source Tree Traversal

- ASTをそのまま実行
- evalの構造を見てみる
  - 教材のeval.c中の関数eval

- 構造を保ったまま変換することを一般的にhomomorphism （準同型）と言う
  - 式の評価については、準同型
  - assignment, function call等、少数のものについては実行側で少し準備が必要

Source tree traversalに向けてさまざまな準備

- コードの配置
  - 関数定義の実体
- データオブジェクトの配置
  - データセグメントを定義する
- 変数のハンドリング
  - symbol table + (code/data segment)
- 関数コールの設計
- ということで、教材の中身を少し見てみる

alternative

- コンパイラシステムの構築
  - 仮想マシンの設計 （ISA決めないといけない）
    - stack machine vs. register machine
  - 仮想マシン上のコンパイラ設計、コード生成
  - 仮想マシン上での実行
    - 変数の評価
      - By name (スクリプト言語では特に)
- この話は次回以降

Type Check and Class Check

- Typeとは何か？

- 基本型(primitive types)と派生型(derived type)(導出型)
  - int, float, double, bool as primitive types
  - As derived types,
    - Function type
    - Pointer type
    - Array type
    - Structure type
    - Union type

Type導入の目的

- Type導入の目的は2つ
  - プログラムが正しく動作するための情報
    - e.g. ポインタどうしの足し算をしてはいけない
  - プログラムが効率的に動作するための情報
    - プログラム内で＋が出てきたときに、int対象の＋かdouble対象の＋かがわかると、効率的なコードが生成できる
- 特に関数呼び出しの時の静的型チェックが歴史的に重要視されてきた

Type Check

- 変数、関数は型とともに宣言する
- 型に関するルールを推論規則の形で書く

e.g. a: point(int) → *a: int

f: (int, double)(int), i:int, d: double

→ f(i, d) : int

- プログラム中、推論規則を適用して、データ(式)の型を決めていく
- 矛盾がでなければ型チェック成功

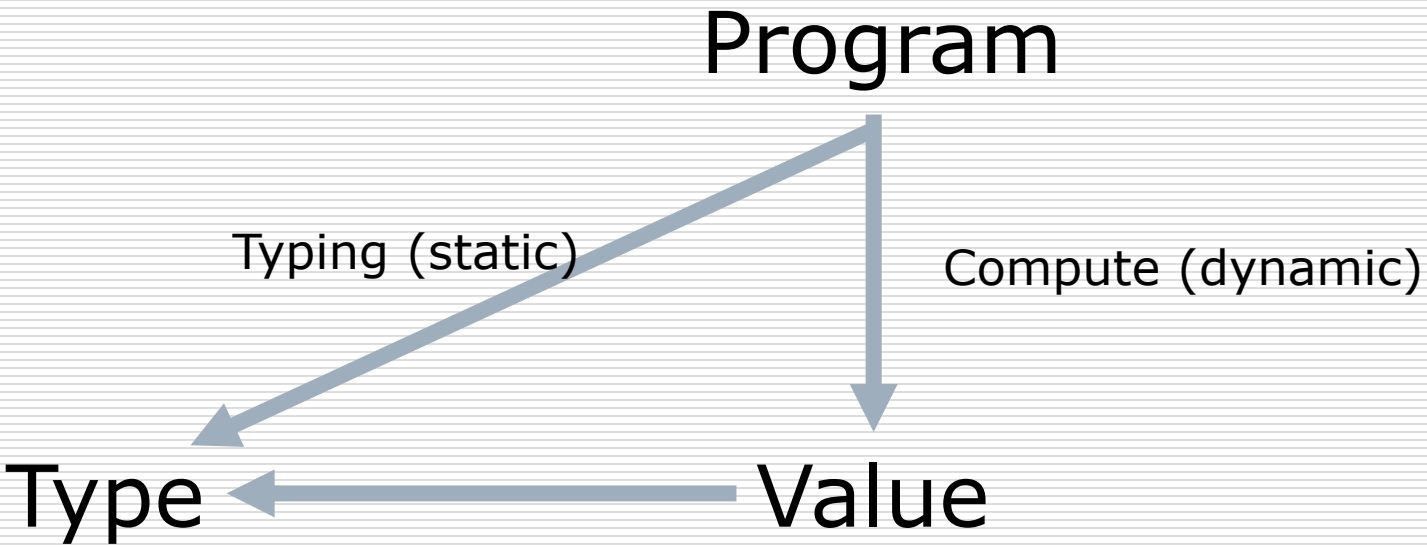
例えば

- eval([PLUS, e1, e2]) = [e1]+[e2]
  に並行させて
- tychk([PLUS, e1, e2]) =

```
if (tychk(e1) == INT &&
    tychk(e2)== INT)
    INT;
else error;
```

図式として

- 下の図式はCommuteするか？



すすんだ話題

- 式が、複数の型を持つと考えられることがある

  e.g. (lambda (x) x)
- Polymorphic Typesにより、複数の型を持つことを表現することができる
- 現在、実用になっている（C++, Java, 多くの関数型言語）polymorphic type system
  - ad hoc polymorphism
  - predicative polymorphism

Polymorphicな場合

- 一番一般的な型(クラス)を最初に割り当て
- 情報が増えるごとに具体化
- Milnerのアルゴリズム for predicative p-types
  Robin Milner. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 17(3):348–375, 1978.
  (世間の評判:本人、何やっていたか、当時絶対わかっていなかった→この論文は読んでも無駄です)
- 後々、unificationを援用したすっきりしたアルゴリズムにまとめられた

Objectとクラス

- クラスはオブジェクト指向での型に相当する
- Structure type + access qualifier
- サブクラスを許すものが多い
  - Extends
  - サブクラスは、型の階層に自明でない構造を持ち込んだ重要な概念
- 抽象クラスを許すものがある
  - abstract
  - implements
  - methodの実体を後から定義する

Access Qualifier

- Private
- Protected
- Public

- なんだかんだ言って、オブジェクトを導入しようとすると、GCその他、実行環境のサポートが大変なんだ…
  - 「この言語にオブジェクトを導入しました」というブログの記事を簡単に信じてはいけません

C++では

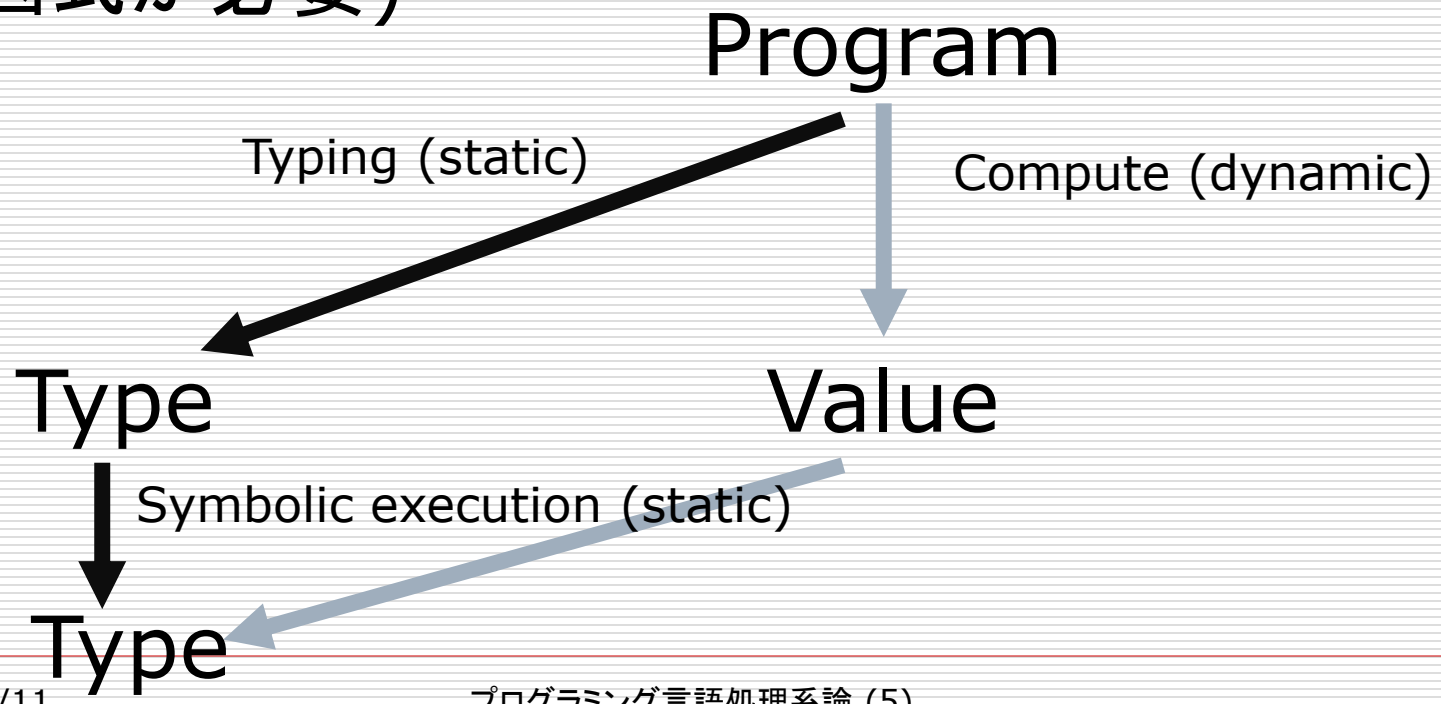
- Signature
  - Classの持つメンバーの集合
- Pseudo Signature
  - Signatureが特定のメンバーを持つかどうか
  - これは1990年代の泥臭い成果があって…
    - SML#
  - C++の人は多分これに気づいていない

抽象実行 (Abstract Interpretation)

- Type checkのためのコードは、evalの簡略版として書くことができる
  - 最初に変数宣言を強制するところでは、あらかじめ与えられた型との整合性をチェック
  - そうでなければ、一番「一般的な型」を想定してtype check 開始
- CSでは、プログラム実行（動的）の前にプログラムの性質を解析することが重要なテーマの一つになっていた
  - Static Analysis
- Type Checkは、Static Analysisの重要な分野の一つ（だった）
- 実用的にはsymbolic executionとして採用されている
- （注意） 抽象化との関係でのAbstract Interpretationは少し頭を使う（Static Analysisについての教科書を読みましょう）

Symbolic Executionで拡張する

- 下の図式はCommuteするか？(後述しますが、条件型やRefinementを導入するとこの図式が必要)



一方、Cpythonでは

- 型チェックは動的に行われる
  - Modules/mathmodules.c
  - string → double, long → double, …
- クラスのチェック
  - このチェックも基本的に動的
  - Objects/abstract.c

CPythonのmathmodule.c

- ASSIGN_DOUBLE()を見てみる
  - PyFloat_CheckExact()
    - これが動的型チェック
    - PyFloat_As
  - PyFloat_AS_DOUBLE()
    (Include/floatobject.h)
  - PyFloat_AsDouble()
    (Objects/floatobject.c)
  - その他 Py*_From*()

スクリプト言語での動的型チェックの要請

- 柔軟な表現を求める
  - 1+2.0 くらいは書けるようにしてよ
  - “1.0” + 3.0 = 4.0 くらい書けるようにしてよ
    - v = “1.0”; w = 3.0; print(v+w)
- 動的にしか決まらない型は、動的にしかチェックできない
- PythonのNumberなどは計算時の型計算を許すのは、見てきた通り

で、動的型チェックでよいのか？という問題

- 動的型チェックは、バグの温床

- 特に、プロジェクトとして大きなプログラムを開発するときのインタフェースがないと困る
- プログラムを実行しないと「想定しない挙動」が明らかにならないというのは明らかに困る
- 「静的型チェック」のスクリプト言語への「再」導入
  - TypeScript, RUST, …

TypeScript

- JavaScriptへ、source to source translationを行う（実質的にアノテーションを外すだけではない）
- JavaScriptは、動的に型付けされる言語
- Typescriptは、プログラムの文面にアノテーションの形で「型」を書く
  - TypeScriptのプログラムはそのままJavaScriptのプログラム
  - JavaScriptのプログラムでTypeScriptの型付けが不可能のものがある

クラスの自明でない定義

```
schuko@DESKTOP-K1HRBIP:~$ cat classeg.d.ts
declare class Position {
    private x;
    private y;
    constructor(x: number, y: number);
    diag(this: Position): number;
}
declare var x: Position;
schuko@DESKTOP-K1HRBIP:~$ cat classeg.js
var Position = /** @class */ (function () {
    function Position(x, y) {
        this.x = x;
        this.y = y;
    }
    Position.prototype.diag = function () {
        return this.x - this.y;
    };
    return Position;
}());
var x = new Position(1, 2);
x.diag();
```

型システム

- 定義の整理
- 型:
  - 式term e と型term Tに関して 関係 e:T
- 型推論: ei:Ti (i≧-1) からe:Tを導出する

$$\frac{e0:T0 \ \ldots \ en:Tn}{e:T}$$

- 型の意味:
  - （集合論的）e:Tに対して〚e〛∈ 〚T〛
  - (圏論的) e:Tに対して〚e 〛: 1→ 〚T〛
- Syntaxと意味の区別をしましょう
- この授業では、意味の話を展開しません
  - 〚T〛が集合(圏のオブジェクト)として成立するかどうかが議論になる…

- 準同型：e:Tとする。計算 e → e′ に対して e′:T （ 意味的には ⟦e⟧ ＝ ⟦e′⟧ ）
- e:TとなるTがなければ、 ⟦e⟧ は定められない

- 型チェック：e:Tが与えられたときに、それが型推論の結果として導出できるかを確認（チェック）すること

型チェックと言っても

- 型推論の構成と対になる
  - 以下のようなプログラムを書くわけにはいかない。
    a:number = 1:number
    c:number = a:number+2.0:number
- 型推論は、容易？
  - 型構成子に対応する分解をすればよいだけなら…
  - 型階層（subtype, supertype）の反映
  - 型パラメタの処理
  - TypeScriptには全部あります

TypeScriptの型システム

- 基本型
  "string" | "number" | "bigint" |
  "boolean" | "symbol" | "undefined" |
  "object" | "function“
- any, unknown, undefined, null
- literal
  - 式term(の一部)を取り込む
    var g: 7 = 3+4;

- union, intersection
  type TwoD = {x:number, y:number}
  type ThreeD = {x:number, y:number, z:number}
  type TwoOrThreeD = TwoD|ThreeD
  type TwoAndThreeD = TwoD & Three
- 典型例
  type nodetype = ’a’ | ’ol’ | ’ul’
  type strOrnum = string | number

- typeof, instanceof
  - これも、式termと型termの混合
  - 昔はこれ(を含むもの)をreflectionと言っていた
- 条件型、型ガード
  type ISString<T> = T extends string? true: false
  - これは、型termへの計算の導入

関数型

- JavaScript functionに対してcall signatureを対応させる

  function sum(a:number, b:number) { return a+b; } : (a:number, b:number) => number

- 引数の形状
  - Optional (?), Rest (…)

- Polymorphism as generics function

```
filter<T>(arr:T[],
          f:(item:T)=>boolean):T[]
{ let result = []
  for (let i=0; i < arr:length; i++) {
        …
}
```

Class

- Classは、メンバーとメソッドの集合
- abstract class, implementsによる抽象化具体化
- extendsによる型階層

型階層

表 1: TypeScript の型階層

				any			
↓	↓	↓	↓	↓	↓	↓	↓
string	number	bigint	boolean	symbol	undefined	object	function
↓	↓			↓		↓	
string enum	number enum			unique symbol		array	
						↓	
						tuple	

表 2: クラスの階層と array, function 内の階層

S	S	`array<S>`	S1	S2	`S1 => S2`
↓	↓	↓	↑	↓	↓
T (such that T extends S)	T	`array<T>`	T1	T2	`T1 => T2`

表 3: literal の階層

string	number	···
↓	↓	↓
string literal	number literal	

- 型階層と型推論

$$\frac{e:T \quad T < T'}{e:T'}$$

- Refinement
  - Discriminated union 型に対し、条件文（if, switch）によって型の条件を追加していく方法