特別研究報告

題目

Java バイト コード における
データ依存解析法の提案とその実装

指導教官
井上克郎 教授

報告者
誉田 謙二

平成 12 年 2 月 23 日

大阪大学 基礎工学部 情報科学科

平成11年度 特別研究報告

Javaバイトコードにおける
データ依存解析法の提案とその実装

誉田 謙二

内容梗概

Javaバイトコードとは，Javaソースコードをコンパイルしたときに生成されるバイナリファイルであるクラスファイルを構成するコードであり，仮想マシン(Java Virtual Machine)への入力となる形式である．スタック操作，変数のアクセス等々の命令セットであり，CPUの命令セットとは異なった命令セットであり，CPUのアーキテクチャに依存していない．

最近では他の言語からJavaバイトコードへのコンパイラや，Javaバイトコードから他の言語に変換する逆コンパイラ，バイトコードからネイティブコードに変換するコンパイラも存在し，様々な言語の中間言語的な存在としてJavaバイトコードを考えることができる．つまり，Java以外の言語も含めて，プログラムの最適化対象として，つまり最適化用中間言語としてJavaバイトコードをみなすことができる．

本研究ではJavaバイトコードにおけるデータ依存関係を定義し，それを抽出する手法を提案した．さらに提案した手法をツールに実装し，本手法を実現した．

本手法では，まずJavaバイトコードに対して制御フローを解析し，節点を各命令，辺を制御の移動とした制御フローグラフを構築する．そして解析に必要なフレームをあらかじめ用意し，制御フローグラフの辺をたどりながらそのフレームをより新しいものへと更新し，データ依存関係を抽出していく．そのようにして到達可能な全経路に対して起こり得るデータ依存関係を解析・抽出し，それが収束するまで解析を続ける，という方法をとった．

主な用語

Javaバイトコード
Java Virtual Machine
データ依存関係
最適化

目次

1 まえがき **5**

2 Java Virtual Machine **6**

2.1 JavaVM 6

2.1.1 クラスのロード 6

2.1.2 メモリ構造 8

2.1.3 メソッド実行時のフレームの動作 12

2.2 Java バイトコード 15

2.2.1 クラスファイルの構造 15

2.2.2 Jasmin 形式 18

3 Java バイトコードにおける依存関係 **21**

3.1 Java バイトコードにおけるデータ依存とは 21

3.2 データ依存の例 22

3.2.1 分岐のないプログラム 23

3.2.2 分岐のあるプログラム 24

4 データ依存関係解析アルゴリズム **28**

4.1 アルゴリズムの概要 28

4.2 メインアルゴリズム 29

4.3 アルゴリズム Analysis 31

4.4 アルゴリズム NoBranch 32

4.5 アルゴリズム Branch 33

5 実装したツールの内容 **36**

5.1 外部仕様 36

5.2 内部仕様 37

5.3 インターフェイス 37

6 解析アルゴリズムの正当性 **41**

7 考察・今後の課題 **42**

謝辞 **44**

参考文献 **45**

付録 **46**

A　命令ごとの def 情報フレームの更新 **47**

図目次

1 JavaVM メモリ構成 7
2 calc() 実行時のスレッド A の JavaStack 8
3 play() を再帰的に実行した時のスレッド A の JavaStack 9
4 再帰的に呼ばれた play() の実行が終了した時のスレッド A の JavaStack . . . 9
5 JavaStack とオペランドスタックの関係 11
6 メソッド play(2) 実行時のフレームの状態 14
7 クラスファイルの構造 16
8 サンプルプログラム逆アセンブル結果 (Jasmin プログラム) 19
9 分岐のないプログラムにおけるデータ依存グラフの例 23
10 分岐のあるプログラムの制御フローグラフの例 25
11 分岐のあるプログラムにおけるデータ依存グラフの例 26
12 Step1 によって構築される制御フローグラフの例 28
13 Step2 によって生成される def 情報保持フレーム 29
14 def 情報フレーム 30
15 アルゴリズム Analysis の処理 31
16 アルゴリズム NoBranch の処理例 32
17 アルゴリズム Branch の処理例 (等価な解析がある場合) 34
18 アルゴリズム Branch の処理例 (等価な解析がない場合) 35
19 メソッド play() のデータ依存関係解析結果 38
20 メソッド calc() のデータ依存関係解析結果 38
21 メソッド loop() のデータ依存関係解析結果 39
22 メソッド loop() の制御フローグラフ 40

1　まえがき

最近のソフトウェア開発に用いられるプログラム言語において，手続き型言語だけでなく，オブジェクト指向言語の利用が高まってきた．そのようにオブジェクト指向言語が注目される中，Java という言語がクローズアップされ始めた．マルチプラットホームであることを利点に，様々なソフトウェアの開発に Java が用いられるようになってきた．そのため，オブジェクト指向言語の中でも，Java に対する解析の必要性は高まってきていると言える．

Java で書かれたプログラムは，一般にクラスファイルというバイナリファイルにコンパイルされ，それが VM 上で動作する．そのクラスファイルを構成するコードが Java バイトコードである．VM 上で動作するため，機種に依存せず，マルチプラットホームといわれるわけである．

最近ではそのような VM の利点を生かして，他の言語で書かれたプログラムを Java バイトコードに変換するコンパイラもや，逆に Java バイトコードから様々な言語のソースコードに変換する逆コンパイラも存在する．つまり Java バイトコードが様々な言語のパイプライン的な役割を果たしていると考えることができる．これを利用して Java バイトコードを様々な言語の最適化用中間言語ととらえ，その解析は非常に有効であると考えられる．

一方，プログラマにとってプログラムの依存関係を理解することはそのプログラムを理解し，デバッグをする上で必要不可欠なことと言える．つまり，ソフトウェアの開発において，プログラムの依存関係を解析することは，ソフトウェアの開発効率の向上が見込まれるだけでなく，また保守に至るまで非常に重要なことと言える．そのため，様々なプログラミング言語に対する依存関係の解析が行われてきた．

しかし，今までの研究は高級言語レベルでの解析が主となっており，Java バイトコードのようなアセンブラレベルで流通した言語に対しての解析はほとんど行なわれておらず，またスタックマシンに対する解析もほとんどなされていない．よって，JavaVM というスタックマシン上で動作する Java バイトコードに対する依存関係の解析は必要であると考えられる．

本研究では Java バイトコードでの最適化に向けて，Java バイトコードにおけるデータ依存関係を定義し，その依存関係の抽出手法について述べる．また，提案した手法をツールに実装し，その手法の正当性を確認する．

2　Java Virtual Machine

2.1　JavaVM

JavaVM とは，一言でいうと，仮想の CPU をシミュレートするマシンアーキテクチャである．この仮想 CPU が解釈・実行するマシン語命令コードがクラスファイルの Java バイトコードである．つまり，Java バイトコードは直接ハードウェアの CPU によって実行されるわけではなく，仮想の CPU をシミュレートするソフトウェアプログラム JavaVM が Java バイトコードを解釈・実行することになる．

JavaVM は JavaSoft が配布するソフトウェアにも含まれているのはもちろんだが，例えば NetscapeNavigator のようなブラウザに組み込まれたものも存在する．ブラウザは受け取った HTML 文書に Java アプレットが含まれていれば JavaVM を起動し，そのアプレットに含まれる Java バイトコードを解釈・実行する．一方，JavaSoft が配布している JDK (Java Development Kit) は，入力としてクラスファイルを与えると，クラスファイルの Java バイトコードを解釈・実行する．他にも JavaVM が組み込まれたソフトウェアやハードウェアは存在するが，いずれも基本的にこれと同じ仕組みで動作している．

JavaVM がどのようなメモリ構成になっているかを図 1 に示す．

2.1.1　クラスのロード

上述のように，JavaVM はクラスファイルを読み込んでそれを実行する．JavaVM がクラスファイルをロードする方式は大きく分けて 2 種類ある．1 つは，JavaVM に元から組み込まれているクラスのローディング機構 (システムクラスローダ) を使ってクラスをロードする方式であり，もう 1 つはユーザ定義のクラスローダ (ユーザクラスローダ) を使う方式である．図 1 からも分かるように，システムクラスローダは 1 つの JavaVM 中に 1 つしか存在しないが，ユーザクラスローダは 1 つの JavaVM 中に複数存在することが出来る．

システムクラスローダは，java.lang.ClassLoader オブジェクトを使わずにクラスをロードする．JDK に含まれる JavaVM ではシステムクラスローダは環境変数 CLASSPATH で指定された場所からクラスを読み込む．CLASSPATH 以外の場所からクラスを読み込むにはユーザクラスローダを使わなければならない．例えば，NetscapeNavigator などの Web ブラウザはネットワークからクラス (アプレット) をロードするため，Web ブラウザを実装するにはユーザクラスローダを用いなければならない．また，外部ファイルやデータベースからクラスファイルをロードするときにもユーザクラスローダを用いる．

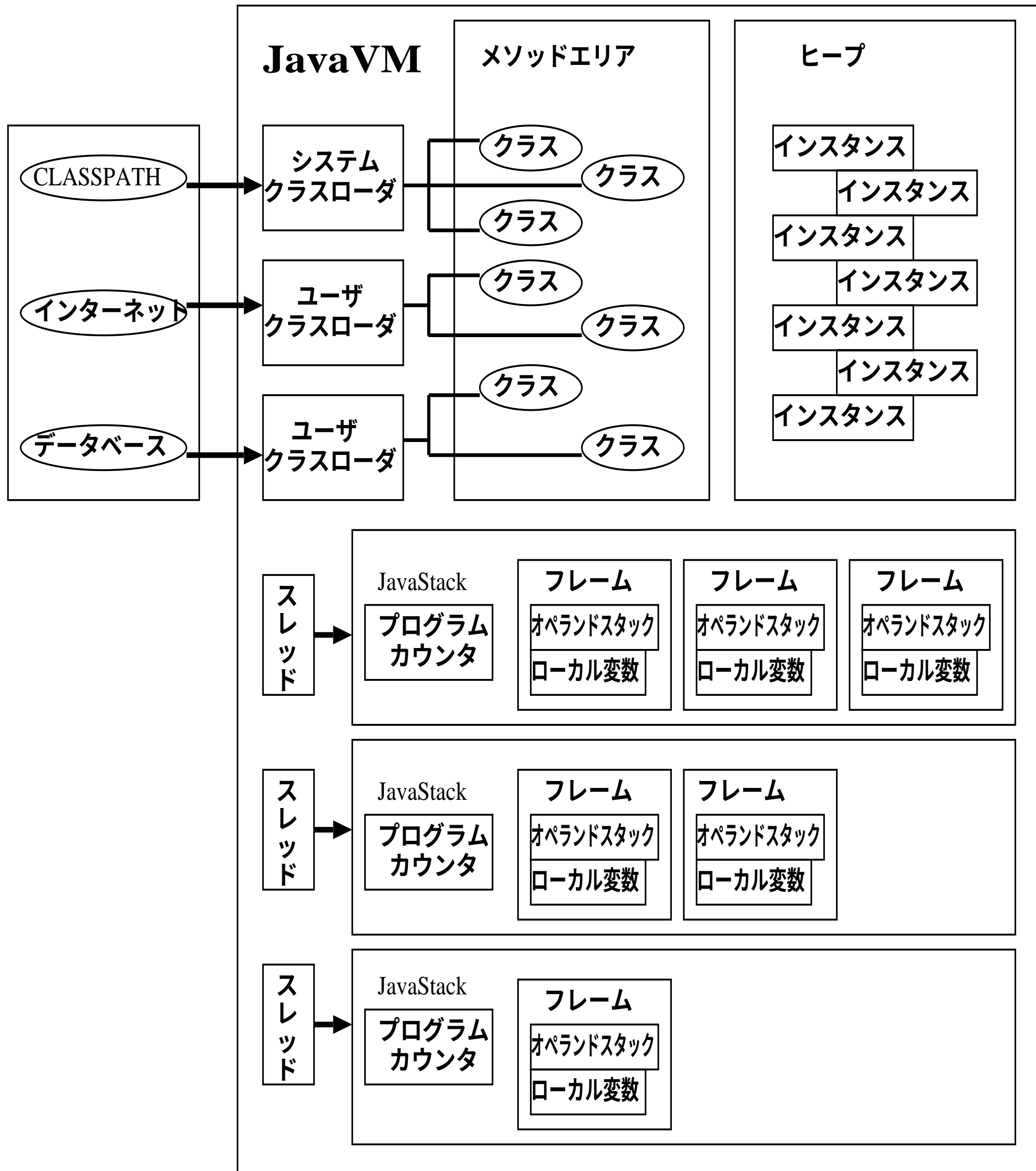


図 1: JavaVM メモリ構成

2.1.2 メモリ構造

メソッドエリア・ヒープ

JavaVM はクラスファイルを読み込み，クラスをメモリに展開する．その置き場所をメソッドエリアという．そして処理を進める中で，そのクラスのインスタンスを生成すると，そのインスタンスをヒープと呼ばれる領域に置く．メソッドエリアとヒープの様子は図 1 に示す通りである．

図 1 ではメソッドエリアとヒープが別々に存在するが，実装によってはメソッドエリアがヒープの中に割り当てられることもある．

また，ユーザクラスローダは java.lang.ClassLoader のインスタンスであるので，図 1 ではヒープに存在しないようだが，ヒープ領域に存在することになる．システムクラスローダは JavaVM に直接組み込まれているため，ヒープとは別のところに存在する．

JavaStack・フレーム

メソッドエリアに展開したクラスを解釈・実行すると，1 つのスレッドが生成される．スレッドが 1 つ生成されると，そのスレッドに対応する JavaStack が 1 つ生成される．スレッドと JavaStack の対応は常に 1 対 1 で，JavaStack はその名の通りスタック構造をしており，このスタックにはフレームと呼ばれるメソッドの作業用メモリが積まれる．スレッド毎に JavaStack が 1 つ用意され，そのスレッドでメソッドが 1 つ呼び出されるたびに JavaStack に呼び出されたメソッドに対応するフレームが 1 つ積まれることになる．メソッドが終了すると終了したメソッドに対応するフレームは取り除かれる．

ある Java のスレッド A が calc() というメソッドを実行したとすると，このメソッドの作業用メモリとしてフレーム X が作られ，それがスレッド A の JavaStack に積まれる．

図 2: calc() 実行時のスレッド A の JavaStack

ここで calc() の中で play() というメソッドが呼び出されたとすると，play() 用のフレーム Y が作られ，play() に割り当てられ，フレーム Y がスレッド A の JavaStack に積まれる．

ここで更に play() が再帰的に自分自身を呼び出すと，また再帰的に呼ばれた play() に新しいフレーム Z が割り当てられ，フレーム Y の上に積まれる．このように，フレームはメソッド毎に割り当てられるのではなく，メソッドの呼出し毎に割り当てられる．

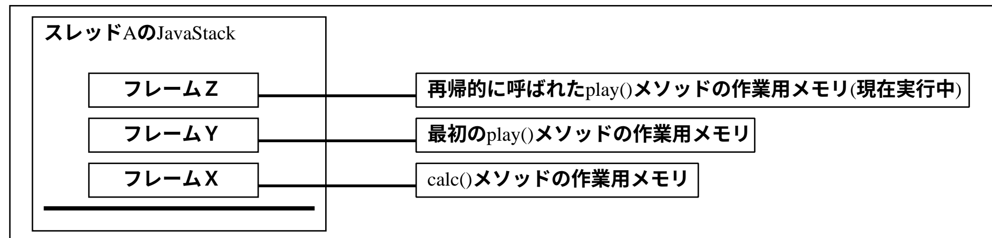


図 3: play() を再帰的に実行した時のスレッド A の JavaStack

2 回目の play() が終了すると，フレーム Z は破棄され，スレッド A の JavaStack は図 4 のようになる．つまり，フレームはメソッドの実行開始時に生成され，メソッドの実行終了と共に破棄される．スレッドが実際実行しているのは常に JavaStack のスタックトップにあるフレームのメソッドだけであり，そのメソッドをカレントメソッドという．ある JavaStack に積まれたフレームが全て破棄されるとそのスレッドが終了することになる．

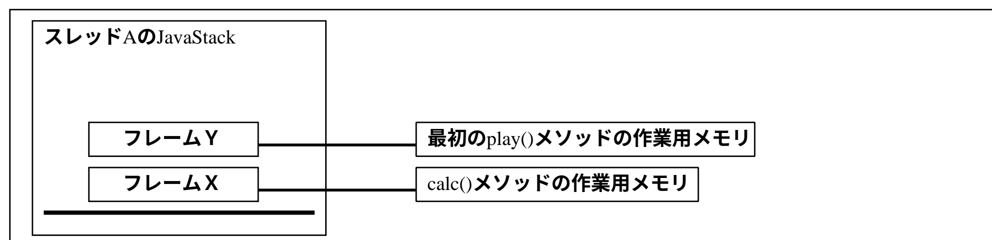


図 4: 再帰的に呼ばれた play() の実行が終了した時のスレッド A の JavaStack

付け加えておくと，JavaStack をヒープ内に割り当てる実装も許されている．

以下，フレームを構成するローカル変数，オペランド スタックについて，順に説明する．

ローカル変数

JavaVM でのローカル変数は，Java 言語でのローカル変数とは異なるものである．Java 言語でのローカル変数とは，メソッド内で定義された変数のことである．例えば次のような Java のメソッドを考える．

```
public int calc(int a,int b, int c){
  int x = a + 3;
  int y = 1;
  for (int i = 0; i < 10; i++){
    int tmp = i + x;
    y = y + tmp;
  }
  return y;
}
```

このようなメソッドであれば，x，y，i，tmp がローカル変数となる．つまりメソッド内部で定義された変数がローカル変数である．このメソッドにおいて a，b，c はメソッド引数といい，Java 言語ではローカル変数とは区別される．しかし，ローカル変数，メソッド引数ともに，そのメソッドの実行の中で一時的に使用する値の格納場所であり，メソッド実行中のみ存在し，そのメソッドの実行が終了すると破棄される．

JavaVM でのローカル変数も，メソッド実行中に一時的に使用する値の格納場所であるので，メソッド実行中のみ存在し，メソッド実行終了とともに破棄されるのは同じである．しかし，JavaVM でのローカル変数は Java 言語でのローカル変数，メソッド引数を両方含むものになる．Java 言語のローカル変数，メソッド引数がコンパイルの際に JavaVM のローカル変数に割り当てられる．

Java 言語での変数には自由に名前がつけられるが，JavaVM でのローカル変数には名前ではなく 0 から始まる通し番号がふられ，あたかも配列であるかのようにアクセスされる．しかし，必ずしもその配列のサイズは Java 言語のローカル変数とメソッド引数の個数を足した数にはならず，それはコンパイラに依存することになる．

オペランド スタック

オペランド スタックとは，その名の通り，オペランドを積み上げておくスタックの

ことであり，Java バイトコードのインストラクションはこのオペランド スタックを介してデータをやりとりする．2.1.3 節において実際メソッドを実行するときのフレームの状態の変化を説明する．

オペランド スタックと JavaStack はいずれもスタックであるが，JavaStack にはオペランド スタックとローカル変数から構成されるフレームが積まれ，オペランド スタックにはオペランドが積まれるので，全く別のものである．その関係は下図 5 に示した通りである．

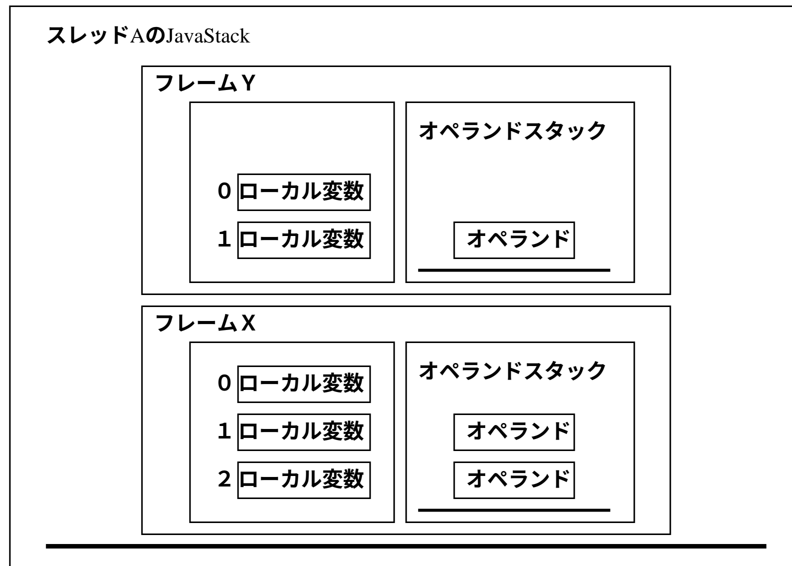


図 5: JavaStack とオペランド スタックの関係

2.1.3 メソッド実行時のフレームの動作

JavaVM がクラスを読み込み，メソッドを実行している時，その処理は JavaStack の最も上に積まれているフレーム内で行われている．実際メソッドを実行している時のフレーム内の動作を順に追っていく．

```
public int play(int a){
  int b = a + 3;
  return b
}
```

例えばこのようなメソッドをコンパイルし，生成されたクラスファイルを逆アセンブルし，jasmin 形式 (2.2.2 節参照) で表示すると，以下のようになる．

```
.method public static play(I)I
  .limit stack 2
  .limit locals 2
.line 3
  iload_0
  iconst_3
  iadd
  istore_1
.line 4
  iload_1
  ireturn
.end method
```

文献 [2] において説明されているが，フレーム内の様子を追跡するために，ここで出てくる Java バイトコードのインストラクションのフレーム内での動作について簡単に説明する．

- iload_<n>
  n 番目のローカル変数の値をオペランドスタックに積む

- iconst_<n>
  整数値 n をオペランドスタックに積む

- iadd
  オペランドスタックに積まれた値を 2 つ取り出し，それらどうしを加算した結果をオ

ペランドスタックに積む

- istore_<n>
  オペランドスタックに積まれた値を 1 つ取り出し，n 番目のローカル変数に格納する

- ireturn
  オペランドスタックに積まれた値を 1 つ取り出し，それを戻り値とし，メソッドを終了する

これらを元に，簡単にこのメソッドの実行の際のフレーム内の様子を順に説明する．

このメソッドではソースのメソッド引数 a，ローカル変数 b が対応するフレームのローカル変数 0 と 1 にそれぞれ対応する．するとフレームの追跡は比較的容易になる．

1. まず iload_0 で 0 番目のローカル変数の値をオペランドスタックに積み，
2. iconst_3 で定数 3 をオペランドスタックに積み，
3. iadd でそれら 2 つの値の加算を行い，
4. istore_1 でその加算結果を 1 番目のローカル変数に格納し，
5. iload_1 で 1 番目のローカル変数の値をオペランドスタックに積みなおし，
6. ireturn でその値を戻り値としてメソッド呼出し元へ戻る．

このように 1 つのメソッドでのほとんどのデータの受渡しをオペランドスタックを介して行い，1 つのフレーム内で 1 つのメソッドの処理を完了し，メソッドの実行が終了するとともにそのフレームは破棄されることになる (図 6).

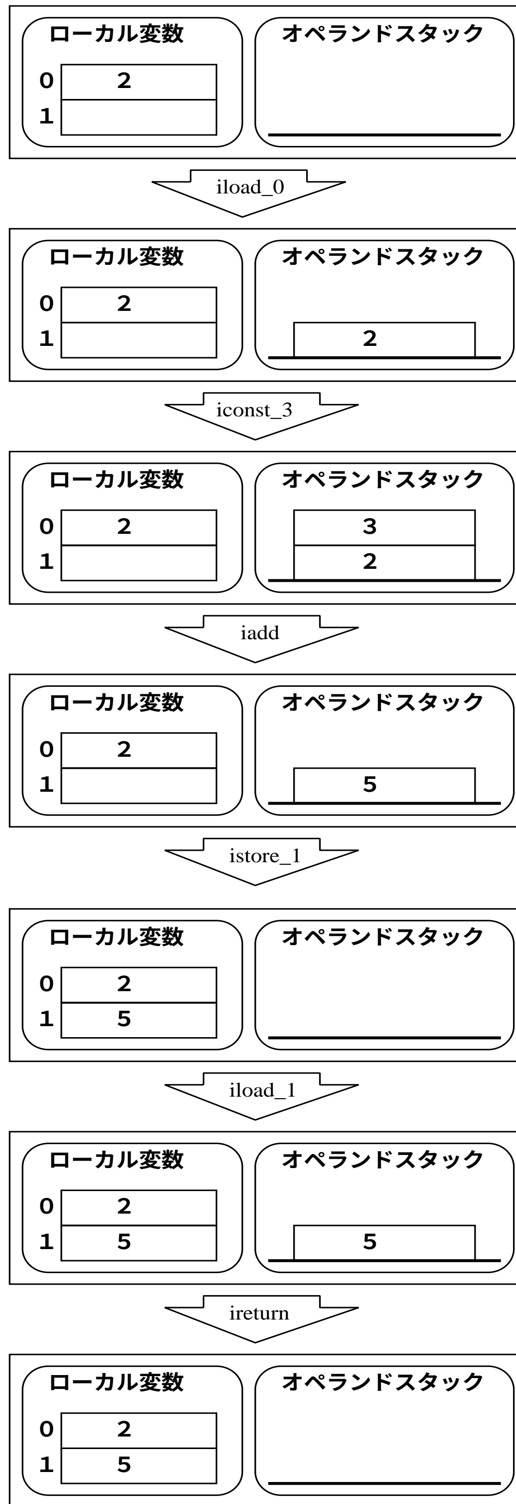


図 6: メソッド play(2) 実行時のフレームの状態

2.2 Java バイトコード

JavaVM が読み込み，解釈・実行するファイルをクラスファイルといい，1 つのクラスファイルは必ず 1 つのクラスから構成されている．つまり，Java コンパイラは Java プログラムに記述されたクラスの数だけのクラスファイルを生成することになる．

クラスファイルはバイトコードというコードにより構成されており，ファイル自体はバイナリファイルであるため，そのままではテキストのように人間が容易に解釈できる形式で表示することができない．そのため，人間が解釈しやすいように，つまりは可読なテキスト形式への変換が必要となる．

そこで，バイナリファイル (実行形式ファイル) のマシン語を逆アセンブルし，アセンブラプログラムに変換することを考える．マシン語とアセンブラプログラムの構造はほぼ 1 対 1 で対応しているため，バイナリファイルのマシン語の解析をする代わりにこのアセンブラプログラムに対して解析を行う．この手法はクラスファイルの解析時にはしばしば用いられており，その理由はアセンブラプログラムがマシン語に比べて格段に読みやすく，しかもマシン語に極めて近いレベルでの解析が可能であるからである．

JavaVM が読み込むマシン語は厳密に定義されているが，それを逆アセンブルしたアセンブラプログラムについては，正規の言語として定義されている言語の形式は存在しない．つまり，クラスファイルを逆アセンブルした結果について，表示する形式が何通りも存在する．そのどの形式のアセンブラプログラムに対して解析を行うかは全くの任意であるが，今回はその中で Jasmin 形式 (2.2.2 節) にしたがったアセンブラプログラム (Jasmin プログラム) に対して解析を行う．

2.2.1 クラスファイルの構造

Java プログラムから生成されるクラスファイルは，コンパイラによって様々であるが，その構造は図 7 に示すようになっており，全てのクラスファイルはこのフォーマットに従うことになっている．全てのコンパイラが同じバイトコード列を持つクラスファイルを生成するわけではなく，本論文で紹介するクラスファイルは全て JDK1.1.8 に含まれる Java コンパイラによって生成されたクラスファイルである．

クラスファイルは図 7 で見るように，C 言語でいう，様々なメンバを持つ構造体のような構造をしている．そこで，クラスファイル全体の説明として，その各メンバの説明を順に進めていく．

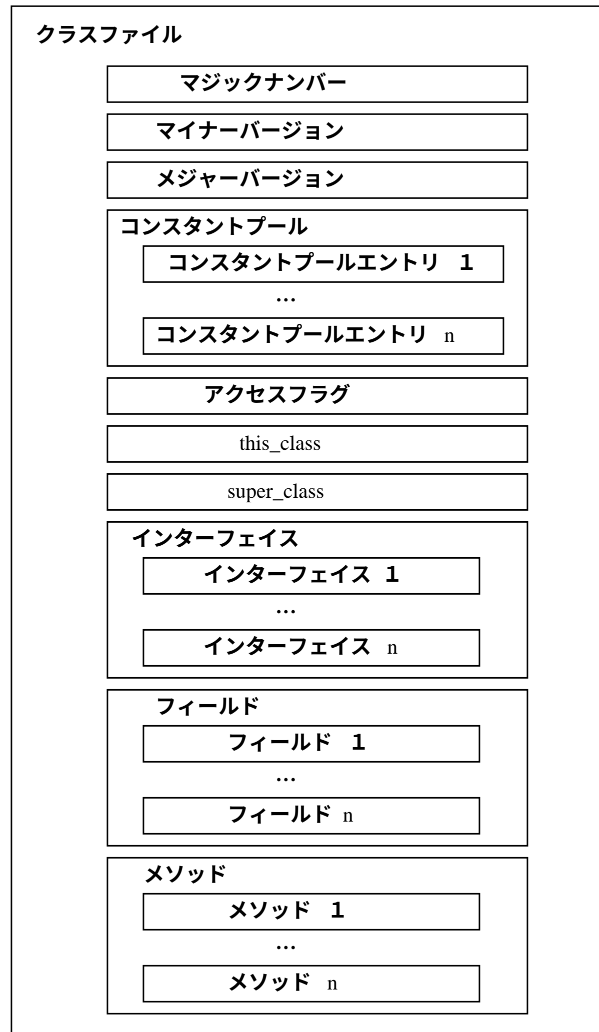


図 7: クラスファイルの構造

マジックナンバー

これはこのファイルがクラスファイルであることを認識するためのもので，ファイルの先頭がマジックナンバーで始まっていないと JavaVM はそのファイルをクラスファイルではないと判断し，以降の読み込みを行わないことになる．

メジャーバージョン・マイナーバージョン

JavaVM はクラスファイルを読み込む際，この値が自分のサポートするバージョンの範囲内にあるかを判定する．範囲内になければ読み込み・実行を行わないことになる．

コンスタントプール

プログラムに含まれる定数を格納する領域で，例えば出力すべき文字列や，整数定数が格納されている．また，それだけでなく，インストラクションのオペランドをあらわす文字列 (例えば java.io.PrintStream クラスであれば"java/io/PrintStream") などもここに格納されている．

アクセスフラグ

Java のクラスやインターフェイスには他のクラスなどからの独立性を高めるため，他のクラスなどからの参照などを制限する機能がある．そのためのフラグであり，他クラスからのアクセスの際にはこのフラグがチェックされる．

this_class・super_class

そのクラス自身と，そのスーパークラスのクラス参照のコンスタントプールのエントリが格納されている．ここに格納されている番号のコンスタントプールのエントリをたどるとこのクラス，スーパークラスの名前が分かる．

インターフェイス

そのクラスがインプリメントしているインターフェイスへの参照のコンスタントプールへのエントリが格納されており，ここに格納されている番号のコンスタントプールのエントリをたどるとこのクラスのインプリメントしているインターフェイスが分かる．

フィールド

Java では，クラス内で定義された変数を様々なメソッドで操作することにより処理が行われる．メソッド内だけで参照できるローカル変数ではなく，他のメソッドなどからも参照できる変数に関する情報が格納されている．

メソッド

実際にデータを操作するメソッドに関する記述がここになされる．各メソッドの処理内容だけでなく，そのメソッドの戻り値や属性などに関する情報もここに含まれる．

2.2.2 Jasmin 形式

JavaVM に対応するマシン語のフォーマットの 1 つとして，Jasmin 形式という形式がある．Jasmin 形式に関する詳細な文法などに関しては文献 [1] を参照するとして，ここでは Java サンプルプログラムをコンパイルし，それを逆アセンブルした Jasmin プログラムに対して説明をする．

Jasmin 形式では，2.2.1 節で説明したクラスファイルに含まれる情報が読みやすく変換され，表示される．例えば次のような Java プログラムに対しての Jasmin プログラムを考える．

```
public class sample{
   public static int i;
   public static void calc (int a){
      int b=a * i + 3;
      System.out.println(b);
   }
   static void main (String args[]){
      i = 13;
      calc(4);
   }
}
```

Jasmin プログラムではコンスタントプールのエントリの内容は逆アセンブルの時点でメソッド内や，スーパークラス名などに全て展開されコンスタントプールは Jasmin プログラムには現れず，そのため主に，メソッドの内容がかなり解析しやすくなる．上の Java プログラムをコンパイルし，生成されたクラスファイルを逆アセンブルすると，図 8 のような Jasmin プログラムが得られる．

クラスファイルを Jasmin 形式に逆アセンブルするとクラスに関する情報がこのように非常に見やすくなる．図 8 より，このクラスは sample クラスであり，このクラスのスーパークラスが java.lang.Object クラスであり，変数 i がフィールドに存在し，3 つのメソッドが定義されていることが分かる．

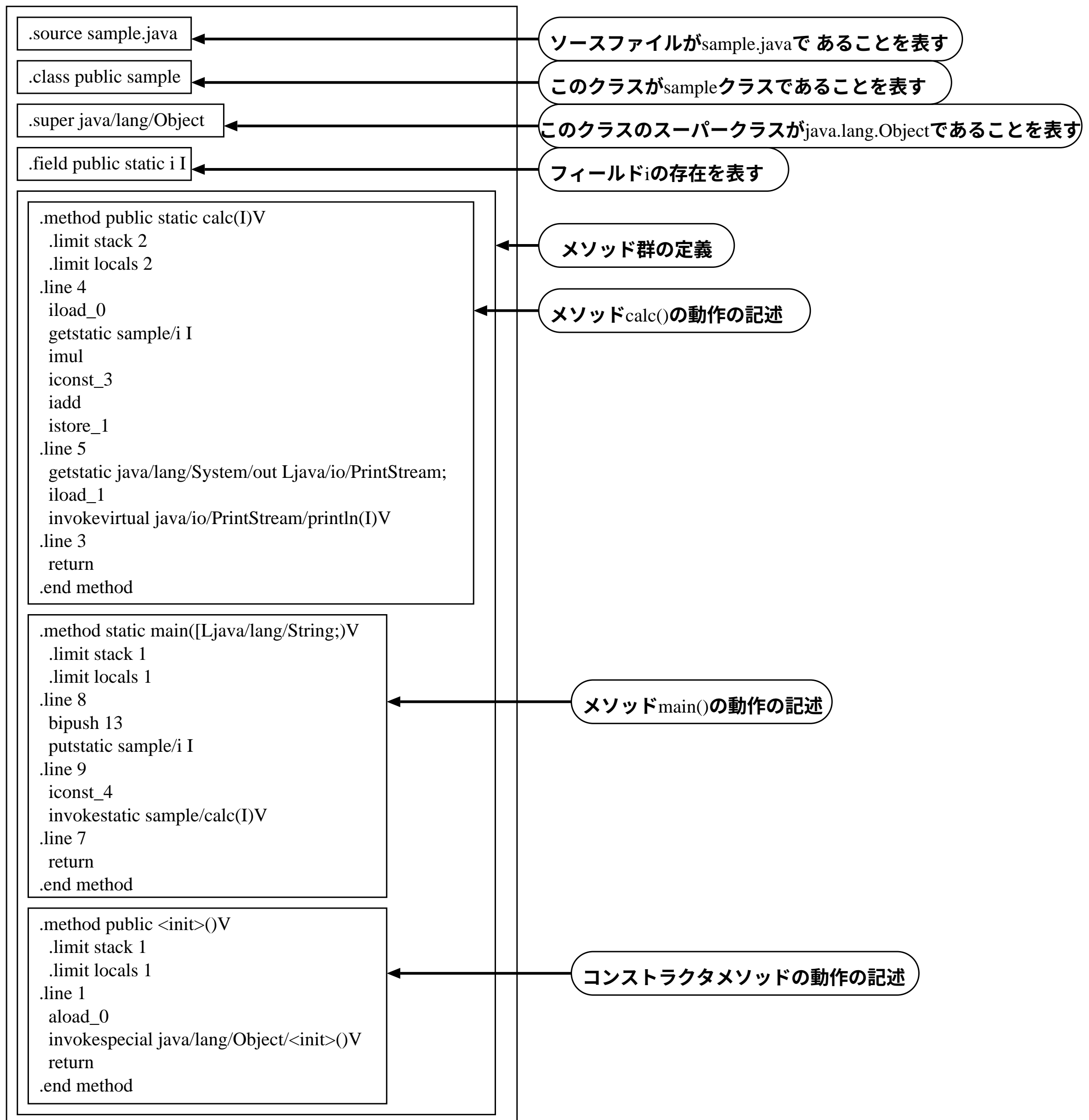


図 8: サンプルプログラム逆アセンブル結果 (Jasmin プログラム)

以下，メソッドの記述について少し説明を加える．

このJasminプログラムでは3つのメソッドcalc()，main()，<init>が定義されており，それぞれの記述の構造は次のフォーマットに従っている．

```
.method <アクセスフラグの内容> <メソッド名> (<メソッド引数の型>)<メソッド戻り値>
  .limit stack <最大使用するスタック領域>
  .limit locals <最大使用するローカル変数領域>
  <命令1>
              :
              :
  <命令n>
.end method
```

JavaVMはメソッドを実行するとき，フレームというそのメソッドの計算を行う作業用領域をJavaStack上に確保する．そこで確保すべき作業用領域の大きさを確定するため，各メソッドはそのメソッドを処理するのに必要なスタック領域とローカル変数領域の大きさを明示しなければならない．もし仮に，無限に多くのスタック領域を必要とする可能性のあるクラスファイルを生成し，JavaVMに読み込ませようとすると，JavaVMはそのメソッドを実行するのに必要な作業用領域の大きさが分からないので，そのメソッドを実行できない．そのため，Java言語のコンパイラは，確保すべき作業用領域が確定するようなメソッドしか生成しないようになっており，それによってJavaVMでの動作が保証されている．

Javaバイトコードにはおよそ200の命令があり，それらの組合せによりメソッドの処理が構成される．その命令の動作一覧は文献[2]に一覧にしてまとめてあるのでそちらを参照されたい．

3 Java バイトコードにおける依存関係

Java バイトコードにおいて，メソッドの動作の記述は図 8 で見るように命令の連続で構成されており，JavaVM はその命令を 1 つ 1 つ読み込み，それぞれの命令に対応する処理を行う．これはどのプログラム言語でも同じであり，そのため，他のプログラム言語においてもデータ依存関係が定義されているように，Java バイトコードに対しても同様にデータ依存関係を定義し，節点を各命令，データ依存関係を辺としたグラフを構築することができる．

3.1 Java バイトコードにおけるデータ依存とは

一般のデータ依存

一般に，あるプログラムに含まれる命令 s_1 および s_2 について考えるとき，次の 3 つの条件を全て満たすとき，命令 s_1 から命令 s_2 へ変数 v に関してデータ依存があるという．

- 命令 s_1 で変数 v を定義している
- 命令 s_2 で変数 v を参照している
- 命令 s_1 から命令 s_2 への実行可能な経路で，その経路において命令 s_1 から命令 s_2 間に変数 v を再定義している文が存在しない，というものが存在する．

Java バイトコードの 1 つのメソッド実行時には，2.1.2 節，2.1.3 節で説明したように，データはローカル変数だけでなく，オペランドスタックを介しても行われるため，Java バイトコードにおいてデータ依存関係を解析するには，オペランドにおいてのデータのやりとりも考慮する必要があるため，この定義をそのまま当てはめることはできない．

そこで，Java バイトコードにおけるデータ依存関係には新たに定義を与える．

Java バイトコードにおけるデータ依存

Java バイトコードにおいて，命令 s_1 および s_2 について考えるとき，次のいずれかの条件を満たすとき，命令 s_1 から命令 s_2 へデータ依存があるという．

- 命令 s_1 から命令 s_2 へ変数 v に関してデータ依存がある．
- 命令 s_1 から命令 s_2 へオペランドスタックに関してデータ依存がある．

このように Java バイトコードにおけるデータ依存関係には 2 種類ある．ここで,「変数 v に関してデータ依存がある」,「オペランドスタックに関してデータ依存がある」に与えられる定義は次に示す通りである．

変数に関してのデータ依存

Java バイトコードにおいて，命令 s_1 および s_2 について考えるとき，次の 3 つの条件すべてを満たすとき，命令 s_1 から命令 s_2 へ変数 v に関してデータ依存があるという．

- 命令 s_1 で変数 v に値を格納している，かつ
- 命令 s_2 で変数 v に値の値を参照している，かつ
- 命令 s_1 から命令 s_2 への実行可能な経路で，その経路において命令 s_1 から命令 s_2 間に変数 v に値を格納している命令が存在しない，というものが存在する．

オペランド スタックに関してのデータ依存

Java バイトコードにおいて，命令 s_1 および s_2 について考えるとき，次の 3 つの条件すべてを満たすとき，命令 s_1 から命令 s_2 へオペランド スタックに関してデータ依存があるという．

- 命令 s_1 でオペランド スタックにデータを積んでいる，かつ
- 命令 s_2 でオペランド スタックからデータを取り出している，かつ
- 命令 s_1 から命令 s_2 への実行可能な経路で，その経路において命令 s_1 から命令 s_2 間に s_1 が積んだデータを取り出している命令が存在しない，というものが存在する．

3.2 データ依存の例

3.1 節で定義したデータ依存関係を考えると，節点と有向辺を持つグラフを考えることができる．有向グラフにおいて，その節点はプログラムに含まれる各命令を表し，その有向辺は 2 つの節点間のデータ依存関係を表す (データ依存辺と呼ぶ)．

以下，考えるデータ依存は，同一メソッド内のみと限定し，例えば他のクラスのフィールドを介してのデータ依存については考慮しない．また，そういうプログラムは扱わないものとする．つまり，他のメソッドの呼び出しがなく，同一メソッド内のローカル変数と，オペランド スタックのみを用いての処理に関してのデータ依存について例を挙げて説明する．また，各命令ごとの処理内容は文献 [2] に記載されているので，そちらを参照されたい．

3.2.1 分岐のないプログラム

2.1.3 節で紹介したメソッド play(2) の実行時のフレームの様子は図 6 の通りになると説明したが，これをもとにいて 3.1 節で定義したデータ依存関係を解析すると，次の図 9 のようなデータ依存関係を示すグラフを得ることができる．

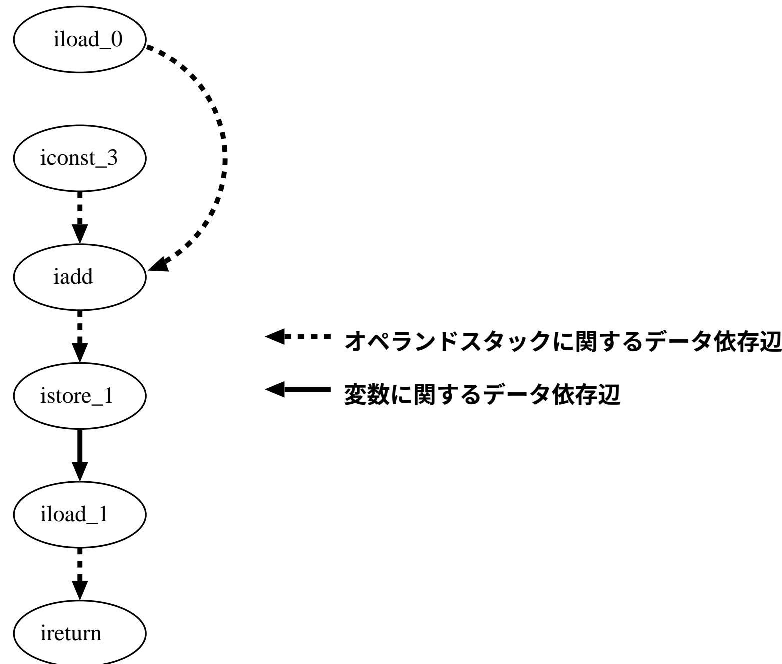


図 9: 分岐のないプログラムにおけるデータ依存グラフの例

このメソッドにおいて，変数に関するデータ依存辺は次の 1 つである．

- istore_1 でローカル変数 1 に格納したデータを iload_1 でオペランド スタックに積む

また，オペランド スタックに関するデータ依存辺は次の 2 つである．

- iload_0 と iconst_3 でオペランド スタックに積んだデータを iadd により取り出す (と同時に加算結果をオペランド スタックに積む)
- iadd でオペランド スタックに積んだデータを istore_1 により取り出す (と同時にローカル変数 1 に格納する)
- iload_1 でオペランド スタックに積んだデータを ireturn により取り出す

分岐のないプログラムでは考え得る経路が 1 通りしかなく，先頭の命令から順に実行が行われるので，解析すべき経路が 1 通りしかなく，データ依存関係もそれほど困難ではない．このように図 9 より 3.1 節で定義したデータ依存関係が存在することがわかる．

3.2.2 分岐のあるプログラム

分岐があるプログラムでは，それに含まれる分岐命令により，様々に制御が移動する可能性があるため，考え得る経路が 1 通りではなくなり，プログラムの制御は単純に先頭から順に移動するわけではなくなる．そのため，その中に存在するデータ依存関係も複雑になる．

例えば，次のような Java のメソッドをコンパイルし，逆アセンブルすると，その下のような Jasmin プログラムが得られるが，この Java バイトコードについては図 10 のように A，B の 2 通りの経路が存在する．

```
public static int calc (int a, int b){
  if (a==b) a=3;
  else a=4;
  return a;
}
```

```
.method public static calc(II)I
  .limit stack 2
  .limit locals 2
.line 3
  iload_0
  iload_1
  if_icmpne L10
  iconst_3
  istore_0
  goto L12
L10:
.line 4
  iconst_4
  istore_0
L12:
.line 5
  iload_0
  ireturn
.end method
```

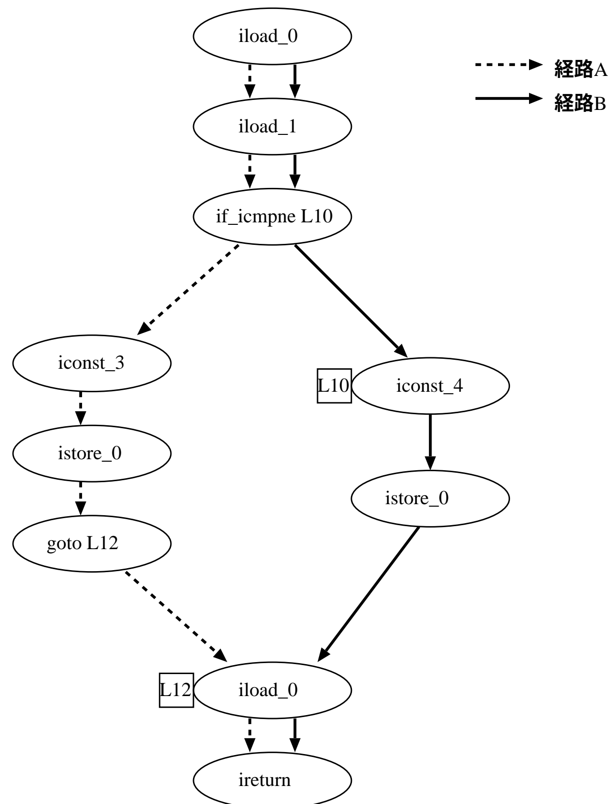


図 10: 分岐のあるプログラムの制御フローグラフの例

このようなプログラムにおいては，複数の経路が存在するため，すべての経路に対してデータ依存関係を調べ，それら和集合が全体としてのデータ依存関係となる．

つまり，

$V_1 \equiv$ 経路 A で通る命令節点の集合

$E_1 \equiv$ 経路 A で発生するデータ依存辺

$V_2 \equiv$ 経路 B で通る命令節点の集合

$E_2 \equiv$ 経路 B で発生するデータ依存辺

としたとき，経路 A については G_1=(V_1,E_1) というデータ依存関係を表すグラフができ，経路 B については G_2=(V_2,E_2) というデータ依存関係を表すグラフができる．

このとき，このメソッド全体のデータ依存関係を表すグラフは G=($V_1 \cup V_2$,$E_1 \cup E_2$) で与えられる．

Java バイトコードにおけるデータ依存を表すグラフ

一般に，複数の経路 1〜n が存在するメソッドにおけるデータ依存関係を表すグラフ G は次の式で与えられる

$$G=(\bigcup_{i=1}^{n} V_i, \bigcup_{i=1}^{n} E_i)$$

ただし，

$V_i \equiv$ 経路 i で通る命令節点の集合

$E_i \equiv$ 経路 i で発生するデータ依存辺

これをメソッド calc() に対して考えると，次の図 11 のようなデータ依存関係を表すグラフができる．

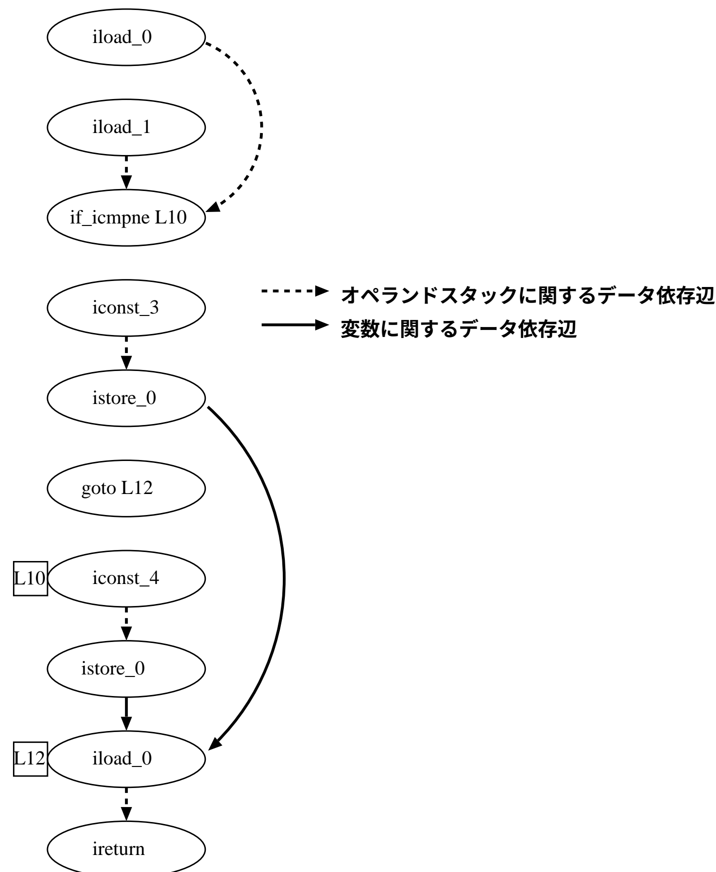


図 11: 分岐のあるプログラムにおけるデータ依存グラフの例

ここで，経路 A，経路 B それぞれにおけるデータ依存辺は次のようになっている．

- 経路 A におけるデータ依存関係
  - 変数に関するデータ依存
    - istore_0 でローカル変数 0 に格納したデータを iload_0 でオペランド スタックに積む
  - オペランド スタックに関するデータ依存
    - iload_0 と iload_1 でオペランド スタックに積んだデータを if_icmpne により取り出す
    - iconst_3 でオペランド スタックに積んだデータを istore_0 により取り出す (と同時にローカル変数 0 に格納する)
    - iload_0 でオペランド スタックに積んだデータを ireturn により取り出す
- 経路 B におけるデータ依存関係
  - 変数に関するデータ依存
    - istore_0 でローカル変数 0 に格納したデータを iload_0 でオペランド スタックに積む
  - オペランド スタックに関するデータ依存
    - iload_0 と iload_1 でオペランド スタックに積んだデータを if_icmpne により取り出す
    - iconst_4 でオペランド スタックに積んだデータを istore_0 により取り出す (と同時にローカル変数 0 に格納する)
    - iload_0 でオペランド スタックに積んだデータを ireturn により取り出す

これらの和集合が図 11 に表れていることがわかる．

4 データ依存関係解析アルゴリズム

4.1 アルゴリズムの概要

アルゴリズム全体は，基本的に次のようなアルゴリズムに基づいて解析を進める．

Step1 メソッド全体を走査し，起こり得る経路をすべて計算し，各命令を節点，起こり得る制御の移動を辺とする制御フローグラフを構築する．

Step2 そのメソッドの処理に必要な数のローカル変数領域，オペランドスタック領域を用意する (各ローカル変数やオペランドスタックが定義された命令を保持するために用いる，以下これを def 情報フレームと呼ぶ)．

Step3 メソッドの先頭命令から，Step1 で構築したグラフの辺に沿って，Step2 で用意した def 情報フレームを用いてデータ依存解析を進める．

このとき，Step1 で構築されるグラフは例えば次のようなグラフである．

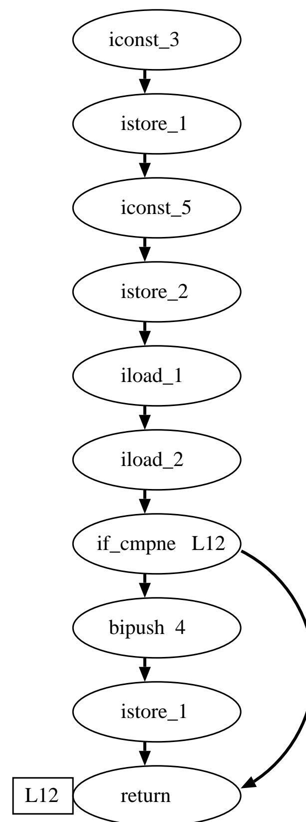


図 12: Step1 によって構築される制御フローグラフの例

この各節点は次のような情報を保持するものとする．

- 命令名
- その命令に付いているパラメータ
- 次に実行する可能性のある命令への辺 (これを Step1 で引くことになる)
- データ依存関係を表す辺 (これがデータ依存辺となる)
- その命令処理開始時のローカル変数・オペランドスタックの def 情報フレームの履歴 (これは条件付分岐命令のみに保持させる)

また，Step2 で用意される各変ローカル変数やオペランドスタックが定義された命令を保持する def 情報フレームは次のようなものであり，初期設定としてすべてが未定義となっている．

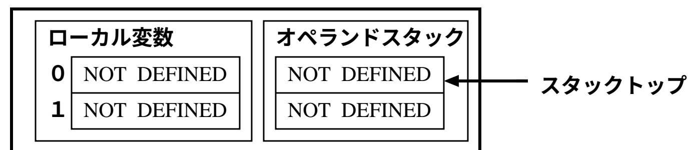


図 13: Step2 によって生成される def 情報保持フレーム

このフレームを保持し，メソッドが実行されるのと同様の経路を通って，オペランドスタックとローカル変数の定義・参照される状況を追跡することによりデータ依存関係を解析していくことになる．

プログラム全体を入力として 4.2 節で説明するアルゴリズムを適用すると，求めたいデータ依存関係を表すグラフを得ることができる．

4.2 メインアルゴリズム

入力 Jasmin 形式で表示されたバイトコードの 1 つのメソッド

出力 データ依存辺が引かれたグラフ

Step1 メソッドを走査し，命令間の制御移動を考え，1 つの命令を 1 つの節点，起こり得る制御の移動を辺とした制御フローグラフを構築する．

Step2 そのメソッドを処理するのに必要な大きさの def 情報フレームを確保する．

Step3 Step2 で確保した def 情報フレーム内を全て未定義で初期化する．

Step4 プログラム開始節点と Step3 で初期化した def 情報フレームを入力として，アルゴリズム Analysis(4.3 節参照) によりデータ依存関係を解析する．

Step1 において，ある命令が処理された後に，次に制御が移動する可能性のある命令を知るためには，各命令の処理内容 (文献 [2]) を参照すればよい．

また，Step1 で制御フローグラフを構築すると，各節点の命令名・パラメータ・次に実行する命令への辺は確定され，データ依存辺は 1 つも引かれず，def 情報フレームの履歴はなし，という状態のグラフができあがることになる．

Step2 で確保する def 情報フレームに保持される情報は，各ローカル変数などが定義された命令であり，未定義の部分は"NOT DEFINED" が保持されるものとする．(下図 14 参照).

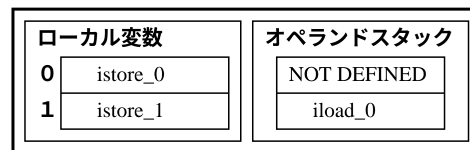


図 14: def 情報フレーム

4.3 **アルゴリズム** Analysis

入力 命令節点，def 情報フレーム

出力 入力命令節点からの経路におけるデータ依存辺

処理 入力命令節点から起こり得る全ての経路に対してデータ依存解析を行い，必要なデータ依存辺を引く

Step1 if (入力命令節点が条件付分岐命令でないなら)
入力として与えられた命令節点と def 情報フレームを入力としてアルゴリズム NoBranch (4.4 節参照) によりその命令以下の経路について解析を行う．

Step2 else if (入力命令節点が条件付分岐命令なら)
入力として与えられた命令節点と def 情報フレームを入力としてアルゴリズム Branch (4.5 節参照) によりその命令以下の経路について解析を行う．

このアルゴリズム内では実質データ依存辺を引く，という動作はせず，条件分岐命令と，そうでない命令に分けることが処理である．実際データ依存辺を引くのはこのアルゴリズムから呼び出される，NoBranch および Branch である．

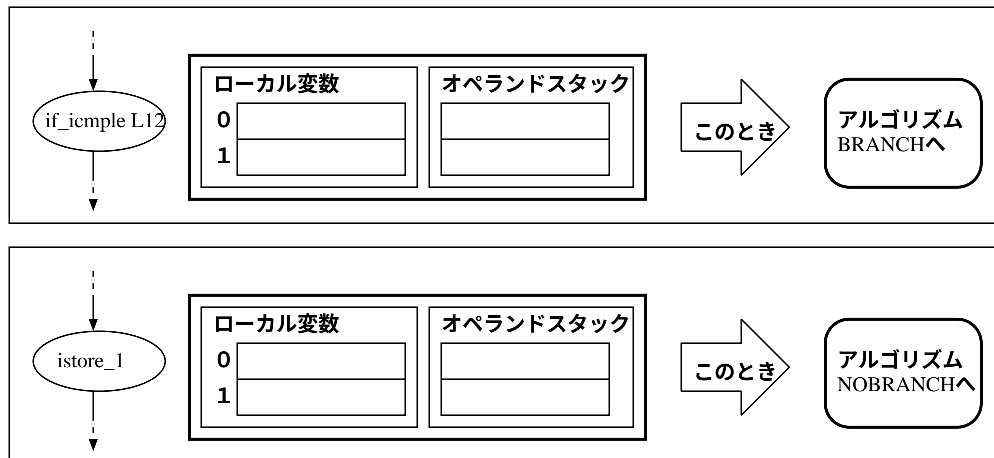


図 15: アルゴリズム Analysis の処理

4.4 アルゴリズム NoBranch

入力 命令節点 (条件付分岐命令以外)，def 情報フレーム

出力 入力命令節点以降の経路におけるデータ依存辺

処理 入力命令節点から起こり得る全ての経路に対してデータ依存解析を行い，必要なデータ依存辺を引く

Step1 入力命令節点の命令に対応する処理を行い，必要ならばデータ依存辺を引き，入力 def 情報フレームを最新の情報に更新する．

Step2 if (次に処理する命令節点がないなら) 何もせず終了

Step3 else if (次に処理する命令節点があるなら)
次に処理する命令節点と，更新された def 情報フレームを入力としてアルゴリズム Analysis (4.3 節参照) によりその命令以下の経路について解析を行う．

条件分岐でない命令に対してはこのアルゴリズムによりデータ依存辺を引く．Step1 における命令ごとの def 情報フレームに対する処理は付録 A に添付したので，そちらを参照されたい．def 情報フレームに対する処理を行う際に，必要なデータ依存辺を引く．

再帰的にアルゴリズム Analysis が適用されるが，このアルゴリズム NoBranch においては Step2 により，次に処理する命令節点が存在しない場合，つまりある経路の終了点において，その経路に関しての解析を終了する．

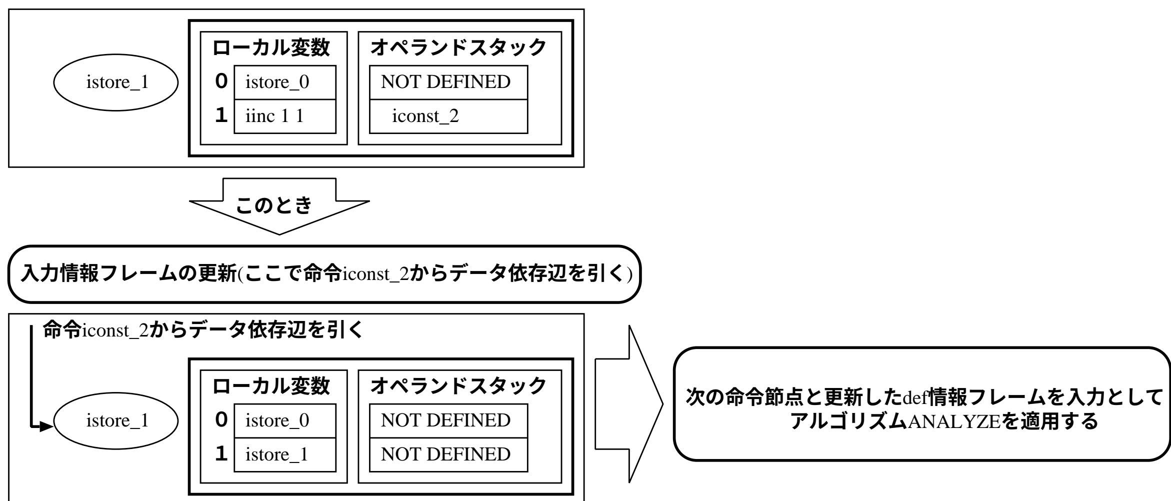


図 16: アルゴリズム NoBranch の処理例

4.5 アルゴリズム Branch

入力 命令節点 (条件付分岐命令)，def 情報フレーム

出力 入力命令節点以降の経路におけるデータ依存辺

処理 入力命令節点から起こり得る全ての経路に対してデータ依存解析を行い，必要なデータ依存辺を引く

Step1 if (入力命令節点に付加されている def 情報フレームの履歴 (4.1 節参照) の中
に入力 def 情報フレームと等価なものが含まれているなら)
何もしないで終了

Step2 else if (入力命令節点に付加されている def 情報フレームの履歴 (4.1 節参照) の中
に入力 def 情報フレームと等価なものが含まれていないなら)

- 入力 def 情報フレームを入力命令節点の def 情報フレームの履歴に追加する．
- 入力命令節点の命令に対応する処理を行い，必要ならばデータ依存辺を引き，入力 def 情報フレームを最新の情報に更新する．
- for (次に処理する可能性のある全ての命令節点について)
  更新した def 情報フレームを複製し，その複製と，制御が移動する命令節点を入力としてアルゴリズム Analysis (4.3 節参照) によりその命令以下の経路について解析を行う．

条件分岐命令に対してはこのアルゴリズムによりデータ依存辺を引く．こちらも Step2 における各命令ごとの def 情報フレームに対する処理は付録 A に添付したので，そちらを参照されたい．def 情報フレームに対する処理を行う際に，必要なデータ依存辺を引く．

条件分岐命令には次に実行される可能性のある命令が複数存在する．それら全てに対して経路が存在するので，Step2 において制御が移動する可能性のある命令節点について処理をすることにより，全ての経路に対する処理を実現する．

しかし，全ての経路に対して処理を行うとはいえ，ループ構造になっていると，永久にそのループを解析し続け，全体としての解析が終了しなくなる．そこで，4.1 節で説明した，命令節点に付加した def 情報フレームの履歴を用いる．この履歴の中に入力 def 情報フレームと等価なものが含まれていれば，その状態で以前その命令節点以降の経路を解析したことを意味するので，それ以降の解析は行わない (Step1)．含まれていなければその状態で以前その命令節点以降の経路を解析したことがないことを意味するので，それ以降に考え得る全ての経路について解析を進めることになる (Step2)．

再帰的にアルゴリズム Analysis が適用されるが，このアルゴリズム Branch においては Step1 により，def 情報フレーム履歴に入力 def 情報フレームが含まれている場合，つまり以前に等価な解析が行われた場合は，それ以上の解析で新たにデータ依存辺が引かれることはないので，その経路に関しての解析を終了する．

つまり，条件分岐付分岐命令では，以前に，以下に示す「等価な」解析がない限り解析を進める，ということになる．

「等価な解析」

ここで,「等価な解析」とは、次の 2 つの項目がともに満たされた分岐以降の解析をいう

1. 処理中の条件付分岐命令について、以前にその命令を読み込んだことがある
2. 処理開始時の def 情報フレームが以前の処理開始時の def 情報フレームと全く同じである (未定義なものは未定義、定義されているものはその命令が同じである)

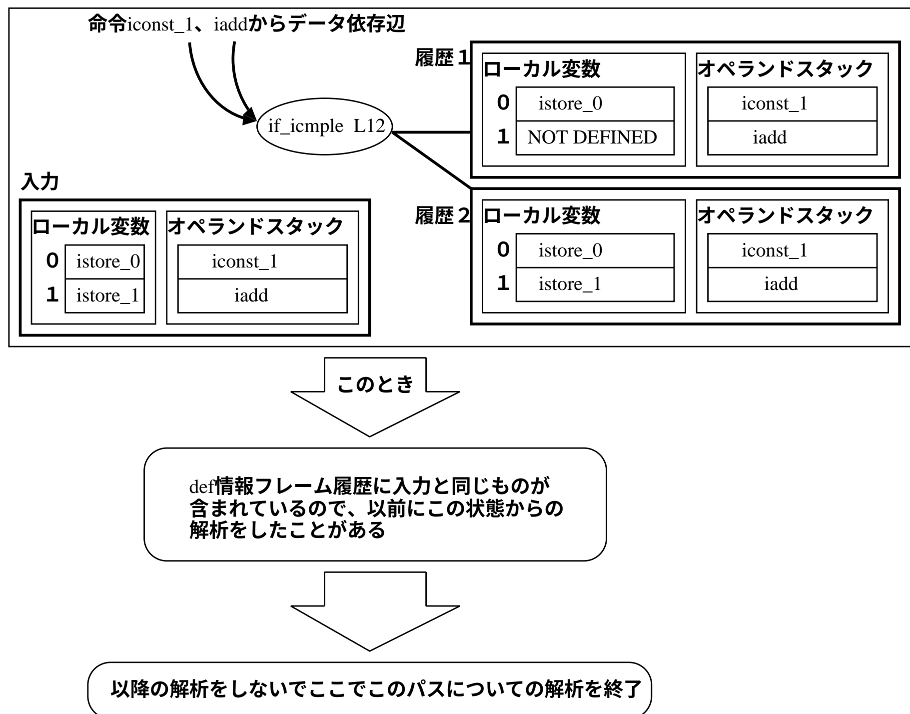
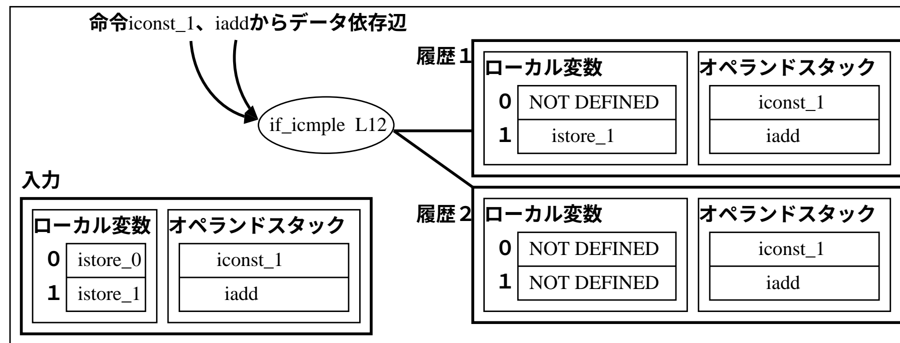
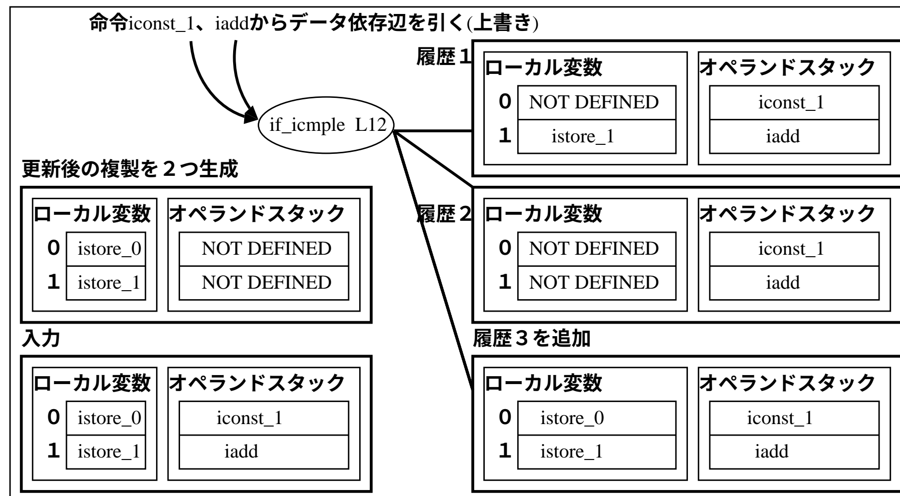


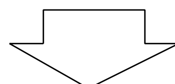
図 17: アルゴリズム Branch の処理例 (等価な解析がある場合)



def情報フレーム履歴に入力と同じものが含まれていない → 以前この解析をしたことがない

命令節点のdef情報フレーム履歴に入力情報フレームを追加
命令iconst_1、iaddにデータ依存辺を引く(上書きとなる、ここでdef情報フレームを更新する)
更新したdef情報フレームの複製を２つ作成





if_icmpleから制御が移動する先は２つある

次の命令節点と複製したdef情報フレームを入力として
アルゴリズムANALYZEを適用する

L12の次の命令節点と複製したdef情報フレームを入力として
アルゴリズムANALYZEを適用する

図 18: アルゴリズム Branch の処理例 (等価な解析がない場合)

5　実装したツールの内容

本研究では，4 節で説明したアルゴリズムを提案するだけでなく，提案したアルゴリズムの正当性を確認するため，ツールとして実装しした．以下に作成したツールの説明をする．

5.1　外部仕様

今回作成したデータ依存関係解析ツールの書式は以下の通りである．

jdep　FILENAME

このツールは入力として与えられた Jasmin 形式で記述された Java バイトコードにおけるデータ依存関係を解析し，その依存関係を表すグラフを表示するものである．解析の結果得られる依存関係を表すグラフが標準出力に表示される．

指定できるオプションはなく，ファイルが存在しない，または指定されたファイルがオープンできない場合は，エラーを出力し，処理を終了する．指定するファイルに特別な拡張子は必要なく，またその補完も行わない．

また，入力となるファイルは Jasmin 形式で記述された Java バイトコードで，.method から始まり，.end method で終る，すなわち 1 つのメソッドに相当する Java バイトコードである．

このツールが解析できるのは同一メソッド内のローカル変数およびオペランドスタックに関するデータ依存関係のみであり，あるクラスのフィールドに関するデータ依存や，複数のメソッドに渡るデータ依存については，正確にデータ依存関係を抽出することはできない．

各命令を節点としたグラフの表示方法として，各節点の通し番号，ラベル名 (ついていれば)，命令名，パラメータ (あれば) を表示した後，その節点からデータ依存辺が引かれる命令節点の番号が表示される．

例えば，ともにラベルがついていない番号 30 の命令 istore_2 から，番号 47 の命令 iload_2 に対してデータ依存辺が引かれるときには，次のように表示される．

```
30    istore_2    47
      :
47    iload_2
```

このように依存辺がその命令の番号を表示することによって表される．ある命令節点から複数のデータ依存辺が引かれる場合には命令の番号が並べられることになる．

作成したツールの実際の動作は 5.3 節で説明する．

5.2 内部仕様

本ツールの実装に用いた言語は C 言語 (文献 [6] 参照) であり，内部での解析様子は次のようなものである．

解析は 4.1 節で示したように，大きく 3 段階に分けることができる．

1 段階目では，まず入力ファイルをトークンに区切り，各命令を節点，制御移動を辺とするフローグラフを構築する．具体的には，一旦全ての命令節点をリスト状に並べ，各節点について起こり得る制御の移動を判定し，適切な命令節点に対して経路が存在することが示す辺を引いていく．データ依存関係を解析するのにはここで構築された制御フローグラフの辺をたどることになる．

2 段階目ではそのメソッドを処理するのに必要な領域を確保する．Java バイトコードのメソッドでは，そのメソッドを処理するのに必要なローカル変数，スタックの作業領域の大きさがあらかじめ分かるため，その領域分に対応するメモリ領域をフレームとして確保する．そして，解析を開始する準備として，作成したフレーム内を全て未定義で初期化する．メソッド開始時には全ての変数が未定義の状態であるので，この作業が必要となる．

そして 3 段階目では 1 段階目で構築したグラフと 2 段階目で作成したフレームを利用し，アルゴリズム Analysis(4.3 節)，アルゴリズム NoBranch(4.4 節)，アルゴリズム Branch (4.5 節) を適用してデータ依存関係を抽出する．具体的には，2 段階目で作成したフレームを更新しながら，1 段階目で構築した制御フローグラフの辺をたどることにより，起こり得る全ての経路に対して解析を行い，命令節点間にデータ依存辺を追加していく．そして，新しくデータ依存辺が追加されない，つまり解析が収束したところで解析を終了する．

5.3 インターフェイス

試作したツールを実際に動作さたときの入力と出力の様子を以下に示す．

入力としては，分岐がないプログラムと，分岐のあるプログラムそれぞれを与えた．プログラムの解析終了性を確認するため，分岐のあるプログラムではループの存在しないプログラムと，ループの存在するプログラムの両方を与え，適切なデータ依存関係が抽出されるかを調べた．

動作時の出力としては，5.1 でも説明したが，以下に示すように，各節点についているラベル名 (なければ出力されない)，その命令にふられた命令番号，その命令の動作を知るのに最低限必要なパラメータ，そこから依存辺が引かれる先の命令番号が表示されることにより，データ依存関係を表すグラフが表示される．これを見ることによりデータ依存関係を知ることができる．また，ある命令節点から複数の命令節点に向かって辺が引かれる場合には，辺の始点となる節点の横に複数の命令番号が列挙される．

- 分岐のないプログラムを入力として与えた場合 (3.2.1 節の例)

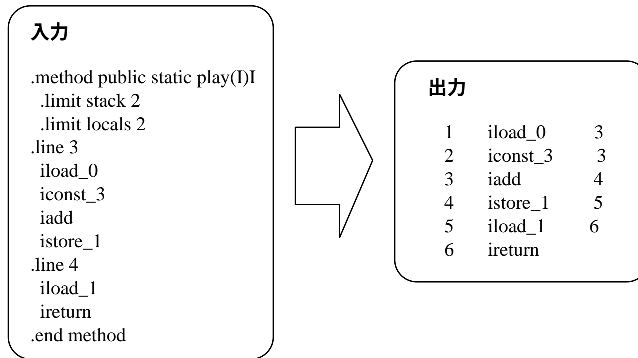


図 19: メソッド play() のデータ依存関係解析結果

この出力と 3.2.1 節の図 9 を比較すると，ループのないメソッド play() について，適切なデータ依存関係が抽出できているといえる．

- ループのない分岐プログラムを入力として与えた場合 (3.2.2 節の例)

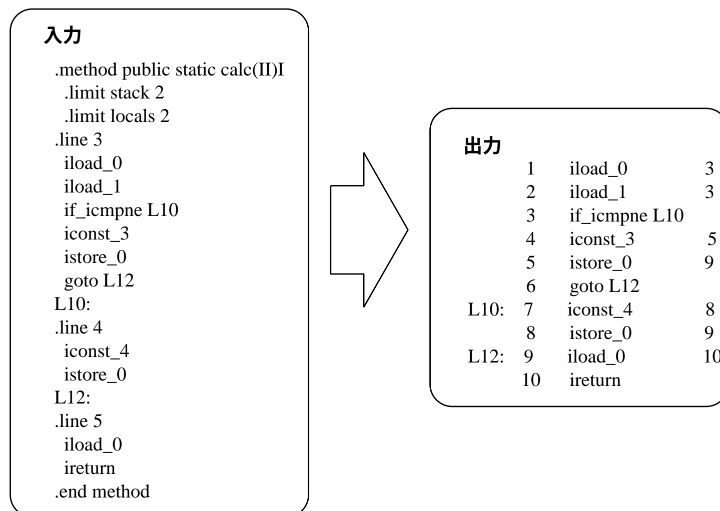


図 20: メソッド calc() のデータ依存関係解析結果

この出力と 3.2.2 節の図 11 を比較すると，ループのないメソッド calc() について，適切なデータ依存関係が抽出できているといえる．

- ループのある分岐プログラムを入力として与えた場合

　次のようなループのある Java プログラムをコンパイルしてできたクラスファイルを逆アセンブルしたものを入力として与えた．

```
public class sample{
   public static int loop(int a){
      int b=0;
      while (a!=b) b=b+1;
      return b;
   }
}
```

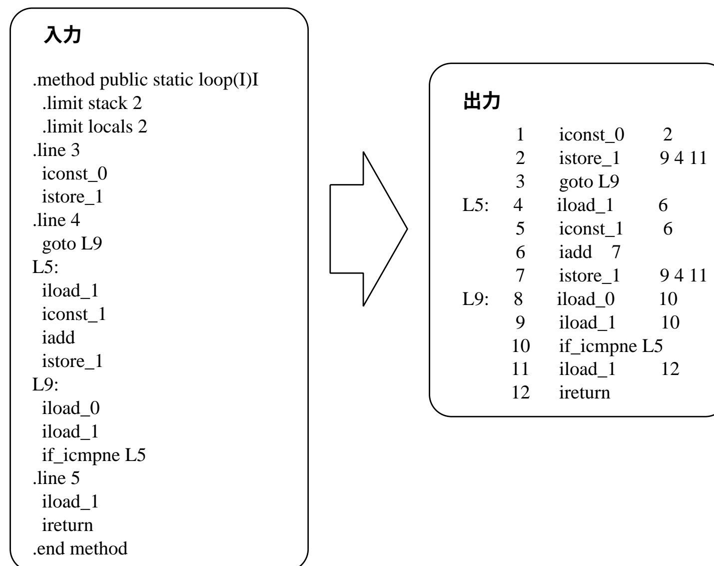


図 21: メソッド loop() のデータ依存関係解析結果

　この Java バイトコードのメソッド loop() では，番号 8～10 が条件判定部，番号 4～7 がループ内の処理内容に相当する (図 22)．

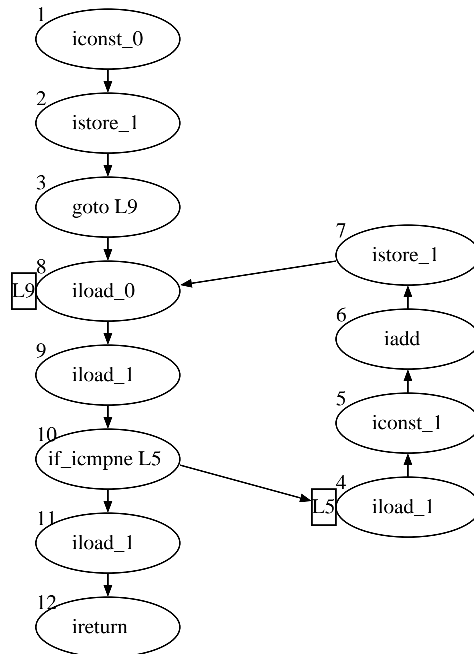


図 22: メソッド loop() の制御フローグラフ

条件判定部の番号 9 の命令 iload_1，ループ内の番号 4 の命令 iload_1，最後の部分の番号 11 の命令 iload_1 が参照するデータは，いずれも初期設定の番号 2 の命令 istore_1 またはループ内の番号 7 の命令 istore_1 いずれかで定義されたデータであり，そのデータ依存関係も含め，適切なデータ依存関係が抽出されているのが分かる．

6 解析アルゴリズムの正当性

同一メソッド内におけるデータ依存関係は，そのプログラムの分岐の有無に関わらず，実際の動作が行われる経路に沿って定義・参照を順に追っていくことにより全て抽出できることは明白である．

5.3 節に示したツールの出力と図 9，図 11，3.1 節，3.2 節で定義した Java バイトコードにおけるデータ依存関係を比較すると，求めるべきデータ依存関係は全て適切に抽出されており，本アルゴリズムにを実装したツールにより同一メソッド内に存在するデータ依存関係は抽出できると考えられる．

複雑に制御が移動するプログラムについても行うべき処理は同じであり，起こりうる経路を制御の移動と同様に走査しながら解析すればデータ依存関係が抽出できる．ツールの出力により正確にデータ依存関係が全て抽出できているかを確認するのは非常に困難であるが，今回提案したアルゴリズムは，各命令の次に実行され得る命令全てについて再帰的にアルゴリズムを適用し，解析を行うので，プログラム中に存在するデータ依存関係は全て抽出できていると考えられる．

また，データ依存関係の解析を終了すべき状態は，次の 2 つの状態である．

1. 解析を進める制御の移動先がそれ以上ない状態 (ある経路の終了時)
2. それ以上解析を進めても新しくデータ依存辺が引かれない状態

本アルゴリズムではアルゴリズム NoBranch(4.4 節) において，経路の終了点における解析の終了を保証している (1 の状態での終了を保証)．

また，アルゴリズム NoBranch(4.5 節) において，等価な解析を開始するときに以降の経路について解析をしないで終了する，という処理を行っている．同じ状態で同じ命令以降の経路について解析を進めても，新しくデータ依存辺が引かれることがないのは明らかであるので，これにより 2 の状態での終了を保証している．

命令節点は有限であるので，その節点間に引かれる辺の数も有限である．すなわち 2 の状態での解析終了を保証している限り．無限に解析を続け，アルゴリズムが終了しないということはあり得ない．つまりこのアルゴリズムは必ず終了するといえる．

7　考察・今後の課題

本論文では，Java バイトコードにおいて，データ依存関係を定義し，同一メソッド内において，それを解析するアルゴリズムを提案し，提案したアルゴリズムを実際のツールとして実装した．結果として，今回定義したデータ依存関係が提案したアルゴリズムにより抽出できることが分かった．よってこのアルゴリズムを用いることで，Java バイトコードに対してデータ依存関係を表すグラフを構築することが可能となることが分かった．

しかし，その反面，今回作成したツールは JavaVM でいう単一フレーム内のみの動作を追跡し，データ依存関係を解析したにすぎない．実際 Java 言語を用いてプログラミングをすると，フィールドの操作はもちろんのこと，メソッド間の呼出しなども頻繁に行われ，複数のメソッドに対してのデータ依存関係を解析する必要が出てくる．これをツールに実現するには，呼び出されるメソッドを含むクラスのクラスファイルを逆アセンブルした結果を全て保持する必要がある．今回作成したツールではそのような機能は考えず，最低限の解析を実現することを目標にしたため，１つのメソッド内における解析のみの機能を備えたものとなった．アルゴリズムとしては，他メソッドへ制御が移動する際に，その移動先のメソッドに対して制御フローグラフの辺を引くことになるので解析は可能であると考えられるが，詳細についてはアルゴリズムの強化が必要である．

また，今回は静的な解析のみ，つまり実際にプログラムを実行させ，その実行系列に対して解析を行う動的な解析ではなく，プログラム本体を解析の対象とし，起こり得る全ての経路についての解析の和を全体の解析結果とする手法を採用したため，実際の動作を考慮すると，不必要なデータ依存関係が抽出されている可能性がある．それを解消するため，動的な解析への移行も今後の課題として挙げることができる．

実際 Java バイトコードに対して人間が手を加えてプログラミングを行うということは今現在，ほとんどあり得ないので，どこで定義されたデータをどこで参照しているかが視覚的に分かっても，それをもとに人間がデバッグする，というようなことは少ないと考えられる．しかし，上で述べたような複数クラス・メソッド間に渡る解析が可能となると，Java バイトコード上でのプログラム依存グラフ (PDG) の構築が可能となり，Java バイトコード上でプログラムスライス (文献 [8]) などを考えることができ，Java バイトコードの解析が容易となり，ソースプログラムの最適化などを考える際に，Java バイトコード上での最適化を利用する，など，実用的なシステムに発展すると思われる．

また，解析効率であるが，全ての経路を順次調べるのではなく，複数の経路を一度に解析できれば効率化が望めると考えられる．例えば，図 10 において，経路 A，経路 B の順に解析を進めるのではなく，命令 ireturn の直前の命令 iload_0 まで経路 A の解析を進め，そこで経路 A に関する解析を待機させ，経路 B の解析を経路 A と合流するまで進め，それ以降

の解析を両方の経路について行う，などの工夫があげられる．つまり，各分岐に対する合流節点を定義し，それぞれの分岐以降の解析をその合流節点まで進め，全ての分岐からの解析が合流し次第，それ以降の解析を進める，というものである．今回ツールに入力として与える程の大きさのプログラムなら大きな差は現れないが，実用化をするにあたり，解析にかかる時間は短縮されるべきだと考えられる．

謝辞

本研究において，種々の御指導を頂きました大阪大学基礎工学部情報科学科井上克郎教授に深く感謝いたします.

本研究の全過程を通して，御指導，御助言して頂きました楠本真二助教授に心から感謝いたします.

本研究を通して，逐次適切な頂御助言を頂きました松下誠助手に心から感謝いたします.

最後に，その他様々な御指導，御助言を頂いた大阪大学 基礎工学部 情報科学科 井上研究室の皆様に深く感謝いたします.

参考文献

[1] J.Meyer，T.Dowing: “Java Virtual Machine”，O’Reilly & Associates，Inc (1997).

[2] J.Meyer，T.Dowing 共著，鷲見 豊訳・編著:”JAVA バーチャルマシン”，オライリー・ジャパン，オーム社 (1997).

[3] J.Meyer，T.Dowing 共著，安藤 進訳:”JAVA 実践プログラミング”，オライリー・ジャパン，オーム社 (1996).

[4] A.V.Aho，R.Sethi，J.D.Ullman:”Compilers Principles,Techniques, and Tools”，Addison-Wesley Publishing Company(1986).

[5] A.V. エイホ，R. セシィ，J.D. ウルマン共著，原田 賢一訳:” コンパイラ II 原理・技法・ツール”，サイエンス社 (1997).

[6] B.W. カーニハン，D.M. リッチー共著，石田 晴久訳:” プログラミング言語 C 第 2 版”，共立出版 (1996).

[7] M.Weiser:”Program slicing”, IEEE Transactions on Software Engineering，10(4):352-357, (1984).

[8] D.Binkley，K.B.Gallagher:“Program Slicing”，Advances in Computers 43: 1-50, (1996).

[9] F.Tip:”A survey of program slicing techniques”，Journal of Programming Languages, 3(3):121-189，(1995).

付録

付録として以下を添付した.

A. Java バイトコードの各命令ごとの def 情報フレームの更新

A　命令ごとの def 情報フレームの更新

- aaload
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  3. 2 で取り出したデータがあるローカル変数と同一のオブジェクトを指していたらそのローカル変数を定義した命令からデータ依存辺を引く.
  4. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.
  5. 3 で，同一のオブジェクトを指すローカル変数があれば，4 で積んだデータがそのローカル変数と同じオブジェクトを指すということを記憶しておく.

- aastore，bastore，castore，fastore，iastore，sastore
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  3. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  4. 3 で取り出したデータがあるローカル変数と同一のオブジェクトを指していたらそのローカル変数がこの命令で定義されたことを記憶しておく.

- aconst_null，bipush，fconst_<n>，iconst_<n>，ldc，ldc_w，new，sipush
  1. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- aload，aload_<n>
  1. 指定されたローカル変数をスタックに積み，この命令で定義されたことを記憶しておき，そのローカル変数を定義した命令から依存辺を引く.
  2. 1 でスタックに積まれたデータがそのローカル変数と同じオブジェクトを指すということを記憶しておく.

- anewarray，arraylength，f2i，fneg，i2b，i2c，i2f，i2s，ineg，instanceof，newarray
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- areturn，athrow，freturn，ireturn，pop，monitorenter，monitorexit，ifeq，ifge，ifle，ifgt，iflt，ifne，ifnonnull，ifnull，lookupswitch，tableswitch
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.

- astore，fstore，istore，astore_<n>，fstore_<n>，istore_<n>
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. 指定されたローカル変数がこの命令で定義されたことを記憶しておく.

- baload，caload，faload，iaload，saload
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  3. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- nop，return，ret，wide，breakpoint，impdep1，impdep2，checkcast，goto
  1. 何もしない

- d2f，d2i，l2f，l2i
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない).
  3. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- d2l，dneg，l2d，lneg
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない).
  3. スタックにデータを 1 つ積む (2 ワードで 1 つなのでここでは何もしない).
  4. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- dadd，ddiv，dmul，drem，dsub，ladd，land，ldiv，lmul，lor，lrem，lsub，lxor
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない).
  3. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  4. スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない).
  5. スタックにデータを 1 つ積む (2 ワードで 1 つなのでここでは何もしない).
  6. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- daload，laload
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  3. スタックにデータを 1 つ積む (2 ワードで 1 つなのでここでは何もしない).
  4. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- dastore，lastore
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない).
  3. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  4. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  5. 4 で取り出したデータがあるローカル変数と同一のオブジェクトを指していたらそのローカル変数がこの命令で定義されたことを記憶しておく.

- dcmpg，dcmpl，lcmp
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．
  2. スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない)．
  3. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．
  4. スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない)．
  5. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく．

- dconst_<n>，lconst_<n>，ldc2_w
  1. スタックにデータを 1 つ積む (2 ワードで 1 つなのでここでは何もしない)．
  2. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく．

- dload，lload，dload_<n>，lload_<n>
  1. スタックにデータを 1 つ積む (2 ワードで 1 つなのでここでは何もしない)．
  2. 指定されたローカル変数をスタックに積み，この命令で定義されたことを記憶しておき，そのローカル変数を定義した命令から依存辺を引く．

- dreturn，lreturn
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．
  2. スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない)．

- dstore，lstore，dstore_<n>，lstore_<n>
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．
  2. スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない)．
  3. 指定されたローカル変数がこの命令で定義されたことを記憶しておく．

- dup
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．
  2. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく．
  3. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく．

- dup2
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．
  2. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．
  3. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく．
  4. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく．
  5. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく．
  6. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく．

- dup2_x1
    1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
    2. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
    3. スタックからデータを 1 つ取り出す.
    4. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.
    5. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.
    6. 3 で取り出したデータをそのままスタックに積む (定義情報を変化させない).
    7. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.
    8. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- dup2_x2
    1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
    2. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
    3. スタックからデータを 1 つ取り出す.
    4. スタックからデータを 1 つ取り出す.
    5. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.
    6. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.
    7. 4 で取り出したデータをそのままスタックに積む (定義情報を変化させない).
    8. 3 で取り出したデータをそのままスタックに積む (定義情報を変化させない).
    9. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.
    10. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- dup_x1
    1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
    2. スタックからデータを 1 つ取り出す.
    3. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.
    4. 2 で取り出したデータをそのままスタックに積む (定義情報を変化させない).
    5. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- dup_x2
    1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
    2. スタックからデータを 1 つ取り出す.
    3. スタックからデータを 1 つ取り出す.
    4. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.
    5. 3 で取り出したデータをそのままスタックに積む (定義情報を変化させない).
    6. 2 で取り出したデータをそのままスタックに積む (定義情報を変化させない).
    7. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- f2d，f2l，i2d，i2l
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.
  3. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- fadd，fcmpg，fcmpl，fdiv，fmul，frem，fsub，iadd，iand，idiv，imul，ior，irem，ishl，ishr，isub，iushr，ixor
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  3. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- fload，iload，fload_<n>，iload_<n>
  1. 指定されたローカル変数をスタックに積み，この命令で定義されたことを記憶しておき，そのローカル変数を定義した命令から依存辺を引く.

- iinc
  1. 指定されたローカル変数を定義した命令から依存辺を引き，それがこの命令で定義されたことを記憶しておく.

- lshl，lshr，lushr
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  3. スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない).
  4. スタックにデータを 1 つ積む (2 ワードで 1 つなのでここでは何もしない).
  5. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- multianewarray
  1. 指定された次元の数だけスタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく.

- pop2，if_acmp，if_acmp
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.
  2. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く.

- getstatic
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．
  2. フィールドの型が double または long ならば，スタックにデータを 1 つ積む (2 ワードで 1 つなのでここでは何もしない)．
  3. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく．

- getstatic
  1. フィールドの型が double または long ならば，スタックにデータを 1 つ積む (2 ワードで 1 つなのでここでは何もしない)．
  2. スタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく．

- invokeinterface，invokespecial，invokestatic，invokevirtual
  1. 指定された引数のだけスタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．
  2. インターフェイスの戻り値の型が void なら何もしない．
  3. インターフェイスの戻り値の型が double または long ならば，スタックにデータを 1 つ積む (2 ワードで 1 つなのでここでは何もしない)．
  4. インターフェイスの戻り値の型が void でないならスタックにデータを 1 つ積み，この命令で定義されたことを記憶しておく．

- putfield
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．
  2. フィールドの型が double または long ならば，スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない)．
  3. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．

- putstatic
  1. スタックからデータを 1 つ取り出し，それを定義した命令から依存辺を引く．
  2. フィールドの型が double または long ならば，スタックからデータを 1 つ取り出す (2 ワードで 1 つなのでここでは何もしない)．