9 情技応第 888 号

高度情報化支援ソフトウェア育成事業

自己反映計算に基づく Java 言語用の

開放型 Just-in-Time コンパイラ OpenJIT の研究開発

総合試験仕様書

平成 11 年 1 月

富士通株式会社

目次

第 1 章

概要

1.1 目的

デジタル技術が普遍性を持つ今日，従来の計算技術は急速に陳腐化し，新たな計算環境に適した汎用性のある技術を我が国が研究開発することが大いに求められている．例えば，広域ネットワーク，マルチメディア環境，NC，並びに電子商取引などの新しいアプリケーションにおいては，可搬性の高いプログラムが要求されており，各国ともその技術開発に凌ぎを削っている．特に，インターネット及びイントラネットを中心とした互換性が要求される環境，あるいは組み込み機器のように計算資源が限定されている環境においては，Java 言語に代表される機器間で高い可搬性を有する言語が重要視されてきている．

Java 言語では，バイトコードのコンパクトでかつ可搬性のあるプログラムの中間形式から，必要な部分を実行時にネイティブコードにコンパイルし，実行速度を向上させる Just-In-Time (JIT) コンパイラが開発されているが，技術フレームワークの欠如，JIT 自身の可搬性の欠如，最適化技術の未発達を含む問題が指摘されている．たとえば，「最適化」は通常速度の最適化であり，限られたメモリやその他の計算資源のもとで，最大の効率を得るという，組込み型のアプリケーションで必要な “Resource-Efficient”(資源効率の高い) 計算の最適化はなされない．さらに，今後様々な計算環境へ適合するため，(1) 個々のアプリケーション及び計算環境に特化した最適化と，(2) 計算環境とアプリケーションに応じたコンパイルコードの拡張が必要になってくるが，従来型の JIT は (1) は汎用性のある最適化しか行わず，また，(2) に対しては，言語の拡張や新規の機能に対応してコード生成の手法を変えるようなカスタム化は不可能でり，Java の利便性と性能に関して大いに妨げとなっている．

本開発では従来型のコンパイラ技術とは異なる，自己反映計算 (リフレクション) の理論に基づいた “Open Compiler” (開放型コンパイラ) 技術をベースとして，アプリケーションや計算環境に特化した言語の機能拡張と最適化が行える JIT コンパイラのテクノロジ “OpenJIT” を研究開発する．開発のターゲットは実用性や広範な適用性を考慮して Java 言語とするが，技術的には他の同種のプログラム言語にも適用可能である．本研究開発により，我が国がこの分野でリーダーシップをとり，我が国が得意とする組み込み機器，マルチメディア機器，並列科学技術計算などにおいて次世代の基盤技術を持つことを目標とする．

結合試験の目的は，上記の目標に基づき開発した OpenJIT コンパイラシステムが構造仕様書で記述した仕様を満たしていることを，そのそれぞれのサブプログラムを結合して試験を行うことで確認することとする．

1.2 試験対象

OpenJIT コンパイラシステムの全体図を図 1.1に示す.

本システムは大きく OpenJIT フロントエンドシステムと OpenJIT バックエンドシステムの二つに分けられる. OpenJIT フロントエンドシステムでは, Java のバイトコードを入力とし, 高レベルな最適化を施して再びバイトコードを出力する. OpenJIT バックエンドシステムでは, OpenJIT フロントエンドシステムによって得られたバイトコードに対して, より細かいレベルでの最適化を行いネイティブコードを出力する.

図 1.2, 図 1.3に OpenJIT フロントエンドシステムと OpenJIT バックエンドシステムを構成する機能の一覧を示す. ただし, OpenJIT バックエンドシステム内の OpenJIT SPARC プロセッサコード出力モジュール, OpenJIT ランタイムモジュールは契約の対象外である.

これらは更に次のに示す機能より構成されている.

- OpenJIT フロントエンドシステム
  - OpenJIT コンパイラ基盤機能
  - OpenJIT バイトコードディスコンパイラ機能
  - OpenJIT クラスファイルアノテーション解析機能
  - OpenJIT 最適化機能
  - OpenJIT フローグラフ構築機能
  - OpenJIT フローグラフ解析機能
  - OpenJIT プログラム変換機能
- OpenJIT バックエンドシステム
  - OpenJIT ネイティブコード変換機
  - OpenJIT 中間コード変換機能
  - OpenJIT RTL 変換機能
  - OpenJIT Peephole 最適化機能

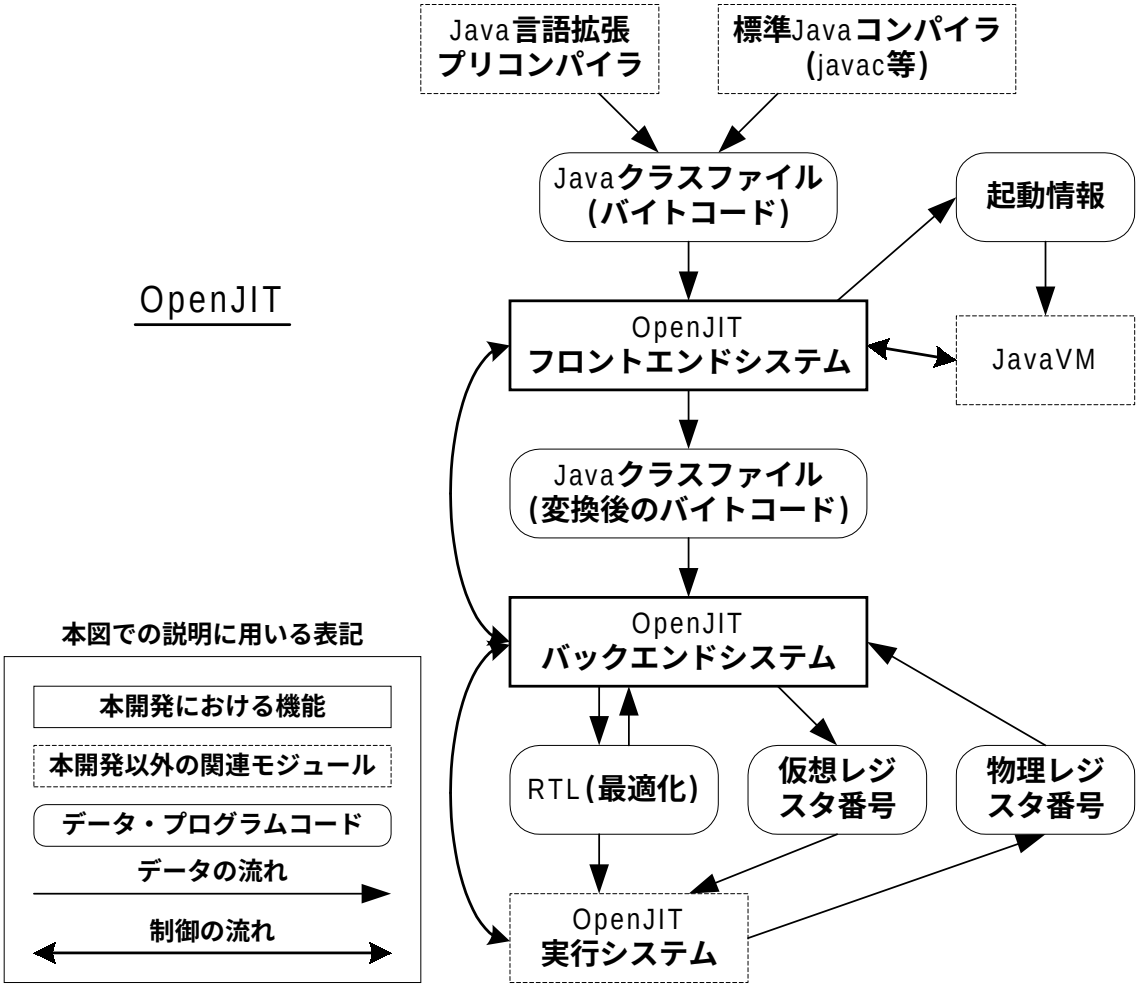

図 1.1: OpenJIT コンパイラシステム

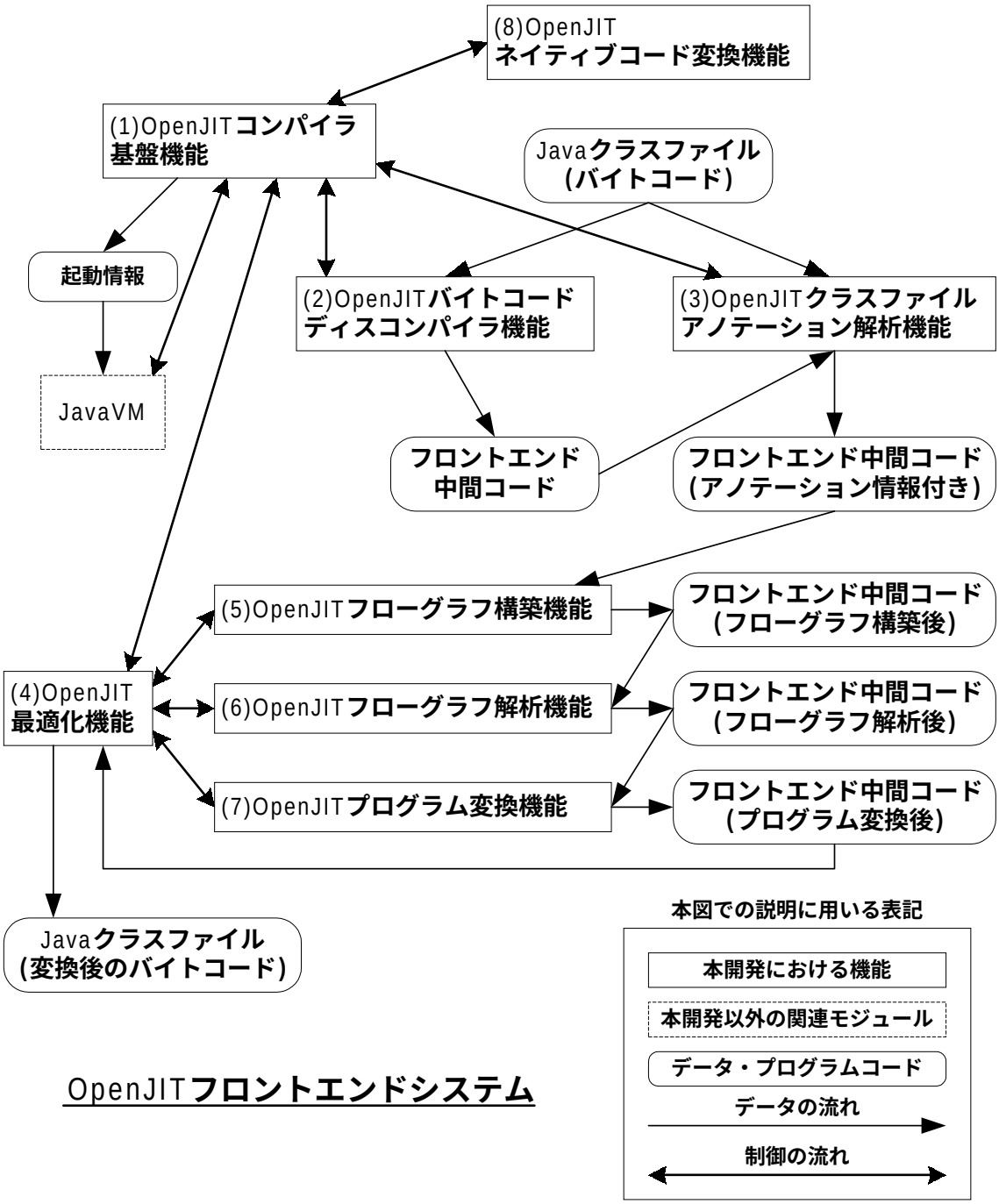

図 1.2: OpenJIT フロントエンドシステム

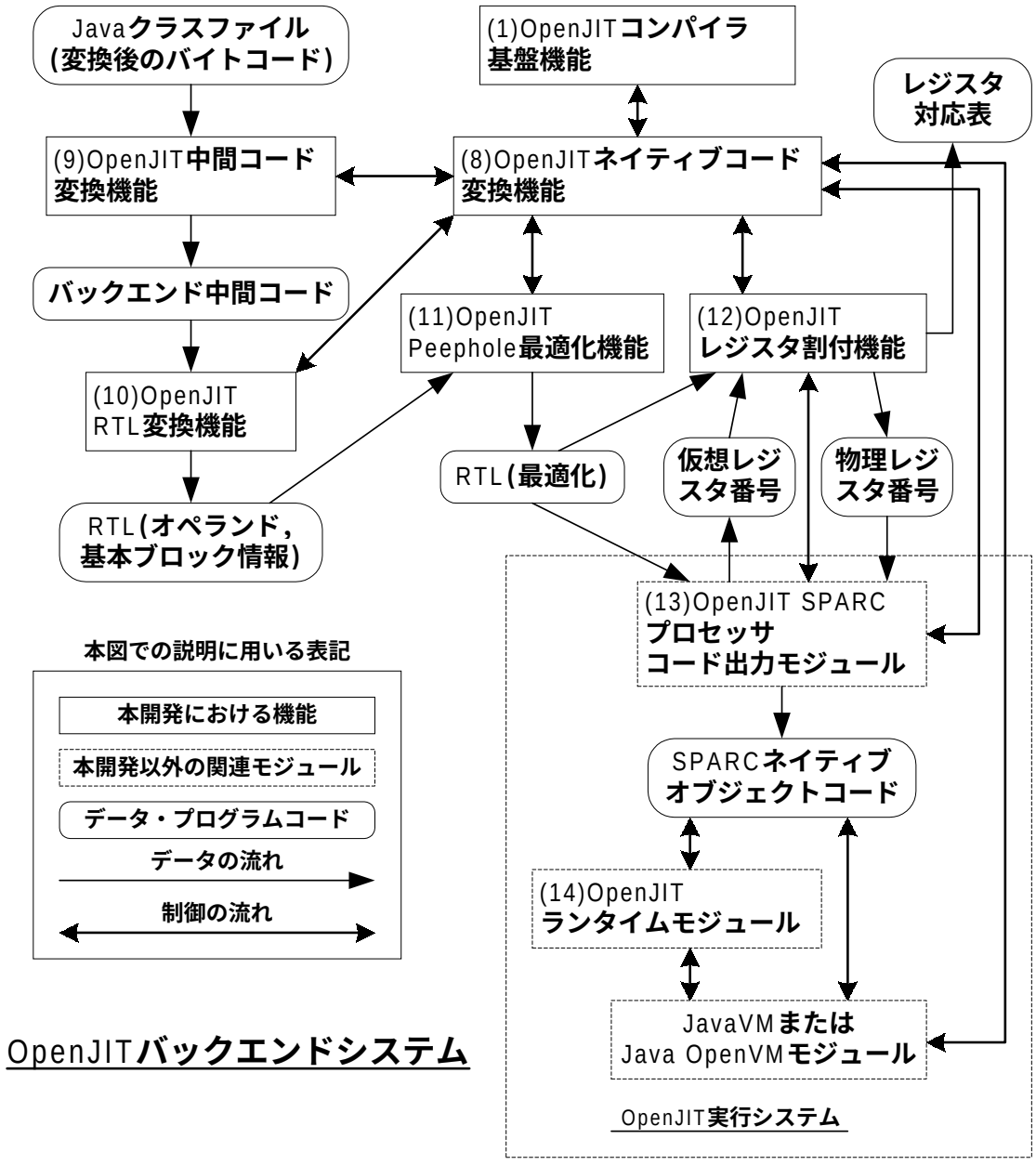

図 1.3: OpenJIT バックエンドシステム

– OpenJIT レジスタ割付機能

以下では，これらの機能概要を示す.

1.2.1 OpenJIT フロントエンドシステム

OpenJIT フロントエンドシステムでは，基本的に与えられたバイトコードから，最適化および拡張を施したバイトコードへの変換を行う．クラスファイルに内在する Java のバイトコードを入力とし，高レベルな最適化およびプログラム変換を施して，再びバイトコードを出力する．

まず，OpenJIT バイトコードディスコンパイラ機能は，与えられたバイトコード列をフロー解析して，逆変換することにより， AST を得る．この際には，与えられたバイトコード列から，元のソースプログラムから生成されるコントロールグラフのリカバリを行う技術を開発する．

同時に，OpenJIT クラスファイルアノテーション機能により，このクラスファイルのアトリビュート領域に何らかのアノテーションが付記されていたときに，その情報を得る．たとえば，バイトコードへコンパイルしたときの高レベルな解析情報が，クラスファイルに付記されていることが考えられる．特に重要なのは，クラスファイル自身では得ることが難しいグローバルな解析情報であり，具体的には各コールサイトにおけるディスパッチ可能なクラスが挙げられる．この情報は， AST 上の付加情報として用いられる．

次に，得られた AST に対し， OpenJIT 最適化機能によって，最適化が施される．最適化に必要な情報は， OpenJIT フローグラフ構築機能， OpenJIT フローグラフ解析機能により抽出される．最適化時のプログラム変換は， OpenJIT プログラム変換機能が司って実施され，変換後のバイトコードがバックエンドシステムに出力される．

(1) OpenJIT コンパイラ基盤機能

OpenJIT コンパイラ基盤機能は，OpenJIT 全体の基本動作を司る．

Sun の JDK においては，Java Native Code API(Application Programmer's Interface) というコンパイラに対するインタフェースが用意されている．この API は JVM のインタプリタにネイティブコード生成を組み込むために用意されたものである．今回開発する OpenJIT コンパイラでは，この API に基づくことにより JDK に準拠の VM に OpenJIT コンパイラを組み込むことができる．この JIT コンパイラは JVM から必要なときに読み込まれ動作する．

(2) OpenJIT バイトコードディスコンパイラ機能

OpenJIT バイトコードディスコンパイラ機能は，Java のクラスファイルのそれぞれのバイトコードを，いわゆる discompiler 技術により，バイトコードレベルからコントロールフローグラフ，AST(抽象構文木) を含む抽象化レベルのプログラム表現を復元し，以後の OpenJIT の各モジュールの操作の対象とするような処理を行なう．

(3) OpenJIT クラスファイルアノテーション解析機能

OpenJIT クラスファイルアノテーション解析機能は，アノテーション情報を解析し，OpenJIT ディスコンパイラ機能が生成したプログラムグラフ (AST) に対して，コンパイル時に適切な拡張された OpenJIT のメタクラスを起動できるようにする．

(4) OpenJIT 最適化機能

OpenJIT 最適化機能は，OpenJIT フローグラフ構築機能，OpenJIT フローグラフ解析機能，および OpenJIT プログラム変換機能を用い，プログラム最適化を行なう．OpenJIT コンパイラには，標準的なコンパイラの最適化を含む最適化ライブラリ構築のためのサポートが準備される．

(5) OpenJIT フローグラフ構築機能

OpenJIT フローグラフ構築機能は，AST およびコントロールフローグラフを受け取り，対応するデータ依存グラフ，コントロール依存グラフ，を含むフローグラフを

出力する．また，クラスファイル間のクラス階層情報情報を得られる場合は，クラスファイルの関係を読み込み，オブジェクト指向解析用のクラス階層グラフも出力する．

(6) OpenJIT フローグラフ解析機能

ここでは，OpenJIT フローグラフ構築機能で構築されたプログラム表現のグラフに対し，グラフ上の解析を行なう．基本的には，一般的なグラフのデータフロー問題として定式化され，トップダウンおよびボトムアップの解析のベースとなる汎用的なアルゴリズムをサポートする．具体的には，グラフ上のデータフロー問題，マージ，不動点検出，などの一連のアルゴリズムがメソッド群として用意される．

(7) OpenJIT プログラム変換機能

OpenJIT プログラム変換機能では，OpenJIT フローグラフ解析機能の結果やユーザのコンパイラのカスタマイゼーションに従って，プログラム変換を行なう．プログラム変換のためには，AST の書き換え規則がユーザによって定義され，AST 上のパターンマッチが行われ，適用された規則に従ってプログラムの書き換えが行われる．書換え規則自身，全て Java のオブジェクトとして定義され，ユーザはあらかじめ書換え規則を定義して，OpenJIT プログラム変換機能に登録しておく．

1.2.2 OpenJIT バックエンドシステム

OpenJIT フロントエンドシステムによって最適化されたバイトコード列に対し，OpenJIT バックエンドシステムは以下の技術を用いて，さらなる最適化処理を行い，ネイティブコードを出力する．

OpenJIT ネイティブコード変換機能はバックエンド系処理全体の抽象フレームワークであり，OpenJIT バックエンドシステムの各機能のインタフェースを定義する．このインタフェースに沿って具体的なプロセッサに応じたクラスでモジュールを記述することにより，様々なプロセッサに対応することが可能となる．

OpenJIT 中間コード変換機能によって，バイトコード列からスタックオペランドを使った中間言語へと変換を行なう．バイトコードの命令を解析して分類することにより，単純な命令列に展開を行う．

得られた命令列に対し，OpenJIT RTL 変換機能は，このスタックオペランドを使った中間言語からレジスタを使った中間言語 (RTL) へ変換する．バイトコードの制御の流れを解析し，命令列を基本ブロックに分割する．バイトコードの各命令の実行時のスタックの深さを計算することで，スタックオペランドから無限個数あると仮定した仮想的なレジスタオペランドに変換する．

次に，OpenJIT Peephole 最適化機能によって，RTL の命令列の中から冗長な命令を取り除く最適化を行ない，最適化された RTL が出力され，最後に OpenJIT SPARC プロセッサコード出力モジュールにより，SPARC プロセッサのネイティブコードが出力される．OpenJIT SPARC プロセッサモジュールは，ネイティブコード生成時のレジスタ割り付けのために OpenJIT レジスタ割付機能を利用する．出力されたネイティブコードは，JavaVM によって呼び出され実行されるが，その際に OpenJIT ランタイムモジュールを補助的に呼び出す．

ただし，OpenJIT SPARC プロセッサモジュール，および OpenJIT ランタイムモジュールは，今回の開発とは別途開発が行われるため，試験の対象外である．

(1) OpenJIT ネイティブコード変換機能

Java のバイトコードからネイティブコードを出力するための抽象フレームワークである．実際は，このクラスを具体的なプロセッサに応じたクラスで特化することによって，実際のコード出力機能を定義する．それぞれのバイトコードと，プログラムの各種グラフ，およびフロントエンドシステムのプログラム解析・変換の結果を用いて，ネイティブコードへの変換を行なう．

(2) OpenJIT 中間コード変換機能

フロントエンドシステムの出力であるバイトコードを入力とする．バイトコードの各命令をグループに分別し，より単純な中間言語に変換を行なう．メソッド呼び出しのバイトコード命令について，メソッドの引数の数や型の解析を行い，中間言語列に展開する．この中間言語のオペランドはスタックで与えられる．また，Java 特有な命令列パターンを検出し，単純な中間言語に置き換える最適化を含めて行う．

(3) OpenJIT RTL 変換機能

OpenJIT 中間コード変換機能の生成結果を入力とし，スタックオペランドを使った中間言語からレジスタを使った中間言語，RTL(Register Transfer Language) に変換を行なう．中間言語列を基本ブロックに分割し，実行の制御の流れを解析することにより，スタックマシンコードからオペランドレジスタコードへの変換を行なう．OpenJIT では，無限資源のレジスタがあるとみなして RTL への変換を行なう．また，オペランドのうち型が未解決のものの型を決定する．

(4) OpenJIT Peephole 最適化機能

OpenJIT RTL 変換機能の生成した RTL を入力として，RTL に対して Peephole 最適化を施す．Peephole 最適化としては，通常行われる redundant load/store elimination を行う．また，Java 固有の Peephole 最適化も行なわれる．Java に特有な配列のインデックスの境界チェックを取り除く最適化も行なう．このモジュールは冗長な命令を取り除いて最適化された RTL を出力する．

(5) OpenJIT レジスタ割付機能

ネイティブコード生成の際，実際のプロセッサレジスタへの割付を行なう．レジスタ割付アルゴリズムを適用し，実際のプロセッサレジスタに対して割付を行なう．物理レジスタの数が足りない場合は，一時レジスタを割り付け，スピル / フィルコードを生成する．

第 2 章

試験方針

各サブシステムが機能仕様書で記述した仕様を満たしていることを確認するため，そのそれぞれの小機能について総合的に試験を行う．そのため，第 4 章で列挙する試験項目を設定し，試験を行う．

試験項目設定の方針としては，各試験で試験される小機能を明らかにするとともに，なるべく各機能を独立して調べられるような試験を行った上で，多くの機能を結合して試験するものとする．

第 3 章

試験環境

3.1 OpenJIT フロントエンドシステム

本システムを構成するサブシステムの試験では，以下のような構成のハードウェア・ソフトウェアを用いた.

() 内は本システムの動作に必要な条件である.

(1) ハードウェア構成

- プロセッサ: Sun Ultra60 <UltraSPARC-II 300MHz 2 基搭載 >
  (SPARC version 8 以降のプロセッサを搭載した Sun Workstation)
- メモリ: 256MB
  (256MB 以上)
- ハードディスク容量: 4GB
  (4GB 以上)

(2) ソフトウェア構成

- オペレーティングシステム: Sun Solaris 2.6 (Sun Solaris 2.5.1 以降)
- Java 実行環境: Sun JDK1.1.6 (Sun JDK 1.1.4 以降)

(3)　他システムとの関連（インタフェース）

Java 仮想マシン (JavaVM) には外部の JIT コンパイラを組み込むインターフェース・API が準備されている．具体的には JIT は各クラスに対するメソッドディスパッチ部位を書き換え，直接メソッド本体ではなく，JIT コンパイラが起動されるようにしておく．メソッド呼び出しによって起動された JIT コンパイラは，メソッドを構成するバイトコードをその場でコンパイルし，ネイティブコードを得て，ヒープ領域に格納する．メソッドディスパッチ部位をさらに書き換え，以後の起動では直接ネイティブコードが起動されるようにする．

このインターフェース・API は，Sun Microsystems により定められた Java JIT Interface の仕様に基づいている．

3.2 OpenJIT バックエンドシステム

本システムを構成するサブシステムの試験では，以下のような構成のハードウェア・ソフトウェアを用いた．

() 内は本システムの動作に必要な条件である．

(1) ハードウェア構成

- プロセッサ: Sun Ultra60 <UltraSPARC-II 300MHz 2 基搭載 >
  (SPARC version 8 以降のプロセッサを搭載した Sun Workstation)
- メモリ: 256MB
  (256MB 以上)
- ハードディスク容量: 4GB
  (4GB 以上)

(2) ソフトウェア構成

- オペレーティングシステム: Sun Solaris 2.6 (Sun Solaris 2.5.1 以降)
- Java 実行環境: Sun JDK1.1.6 (Sun JDK 1.1.4 以降)

(3) 他システムとの関連（インタフェース）

Java 仮想マシン (JavaVM) には外部の JIT コンパイラを組み込むインターフェース・API が準備されている．具体的には JIT は各クラスに対するメソッドディスパッチ部位を書き換え，直接メソッド本体ではなく，JIT コンパイラが起動されるようにしておく．メソッド呼び出しによって起動された JIT コンパイラは，メソッドを構成するバイトコードをその場でコンパイルし，ネイティブコードを得て，ヒープ領域に格納する．メソッドディスパッチ部位をさらに書き換え，以後の起動では直接ネイティブコードが起動されるようにする．

このインターフェース・API は，Sun Microsystems により定められた Java JIT Interface の仕様に基づいている．

第 4 章

試験項目

4.1 OpenJIT フロントエンドシステム

試験項目	OpenJIT コンパイラ基盤機能を構成する小機能項目	OpenJIT バイトコードディスコンパイラ機能を構成する小機能項目	OpenJIT クラスファイルアノテーション解析機能を構成する小機能項目	OpenJIT 最適化機能を構成する小機能項目	OpenJIT フローグラフ構築機能を構成する小機能項目	OpenJIT フローグラフ解析機能を構成する小機能項目	OpenJIT プログラム変換機能を構成する小機能項目	説明
OpenJIT コンパイラ起動試験	○	○	○	○	○	○	○	JDK のインタフェースによって OpenJIT が起動できることを確認する.
OpenJIT コンパイラ基盤機能動作試験 (1)	○							OpenJIT コンパイラ基盤機能を構成する小機能項目が他の各機能を呼び出せることを確認する.

(次のページへ続く)

(前のページからの続き)

試験項目	OpenJIT コンパイラ基盤機能を構成する小機能項目	OpenJIT バイトコードディスコンパイラ機能を構成する小機能項目	OpenJIT クラスファイルアノテーション解析機能を構成する小機能項目	OpenJIT 最適化機能を構成する小機能項目	OpenJIT フローグラフ構築機能を構成する小機能項目	OpenJIT フローグラフ解析機能を構成する小機能項目	OpenJIT プログラム変換機能を構成する小機能項目	説明
OpenJIT コンパイラ基盤機能動作試験 (2)	○							OpenJIT コンパイラ基盤機能を構成する小機能項目が他の各機能を呼び出せることを確認する.
OpenJIT バイトコードディスコンパイラ機能動作試験		○						OpenJIT バイトコードディスコンパイラ機能を構成する小機能項目の動作を確認する.
OpenJIT クラスファイルアノテーション解析機能動作試験			○					OpenJIT クラスファイルアノテーション解析機能を構成する小機能項目の動作を確認する.
OpenJIT 最適化機能動作試験				○				OpenJIT 最適化機能を構成する小機能項目の動作を確認する.
OpenJIT フローグラフ構築機能動作試験					○			OpenJIT フローグラフ構築機能を構成する小機能項目の動作を確認する.
OpenJIT フローグラフ解析機能動作試験						○		OpenJIT フローグラフ解析機能を構成する小機能項目の動作を確認する.

(次のページへ続く)

(前のページからの続き)

試験項目	OpenJIT コンパイラ基盤機能を構成する小機能項目	OpenJIT バイトコードディスコンパイラ機能を構成する小機能項目	OpenJIT クラスファイルアノテーション解析機能を構成する小機能項目	OpenJIT 最適化機能を構成する小機能項目	OpenJIT フローグラフ構築機能を構成する小機能項目	OpenJIT フローグラフ解析機能を構成する小機能項目	OpenJIT プログラム変換機能を構成する小機能項目	説明
OpenJIT プログラム変換機能動作試験							○	OpenJIT プログラム変換機能を構成する小機能項目の動作を確認する.

以下に各中機能を構成する小機能項目を挙げる.

- OpenJIT コンパイラ基盤機能を構成する小機能項目
  - OpenJIT 初期化部
  - OpenJIT コンパイラフロントエンド制御部
  - OpenJIT JNI API 登録部
- OpenJIT バイトコードディスコンパイラ機能を構成する小機能項目
  - バイトコード解析部
  - コントロールグラフ出力部
  - AST 出力部
- OpenJIT クラスファイルアノテーション解析機能を構成する小機能項目
  - アノテーション解析部
  - アノテーション登録部
  - メタクラス制御部
- OpenJIT 最適化機能を構成する小機能項目
  - 最適化制御部
  - バイトコード出力部
- OpenJIT フローグラフ構築機能を構成する小機能項目
  - AST 等入力部
  - データフローグラフ構築部
  - コントロール依存グラフ構築部
  - クラス階層解析部
- OpenJIT フローグラフ解析機能を構成する小機能項目

  - データフロー関数登録部
  - フローグラフ解析部
  - 不動点検出部
  - クラス階層解析部

- OpenJIT プログラム変換機能を構成する小機能項目
  - AST パターンマッチ部
  - AST 変換部

4.2 OpenJIT バックエンドシステム

試験項目	OpenJIT ネイティブコード変換機能を構成する小機能項目	OpenJIT 中間コード変換機能を構成する小機能項目	OpenJIT RTL 変換機能を構成する小機能項目	OpenJIT Peephole 最適化機能を構成する小機能項目	OpenJIT レジスタ割付機能を構成する小機能項目	説明
ネイティブコード変換試験	○					与えられたバイトコードに対し，ネイティブコードが生成されることを確認
メソッド情報受け渡し試験	○					メソッドに関する JDK の内部構造を Java のデータ構造に変換できているか確認
バイトコード読み出し試験	○					JDK の内部データであるバイトコードが Java で読めるかどうか確認
バックエンド中間コード変換試験		○				バイトコードからバックエンド中間コードに変換できるか確認
命令パターンマッチング試験		○				最適化したバックエンド中間コードに変換できるか確認
RTL 変換試験			○			バックエンド中間コードから RTL に変換できるか確認
Peephole 最適化試験				○		RTL の最適化ができるか確認
整数レジスタ割付試験					○	整数レジスタ割り付け機能を確認
浮動小数レジスタ割付試験					○	浮動小数レジスタ割り付け機能を確認

(次のページへ続く)

(前のページからの続き)

試験項目	OpenJIT ネイティブコード変換機能を構成する小機能項目	OpenJIT 中間コード変換機能を構成する小機能項目	OpenJIT RTL 変換機能を構成する小機能項目	OpenJIT Peephole 最適化機能を構成する小機能項目	OpenJIT レジスタ割付機能を構成する小機能項目	説明
javac 動作試験	○	○	○	○	○	OpenJIT システム全体の動作確認

以下に各中機能を構成する小機能項目を挙げる.

- OpenJIT ネイティブコード変換機能を構成する小機能項目
  - ネイティブコード変換
  - メソッド情報
  - バイトコードアクセス
  - 生成コードメモリ管理
- OpenJIT 中間コード変換機能を構成する小機能項目
  - 中間言語変換
  - メソッド引数展開
  - 命令パターンマッチング
- OpenJIT RTL 変換機能を構成する小機能項目
  - 基本ブロック分割
  - コントロールフロー解析
- OpenJIT Peephole 最適化機能を構成する小機能項目
  - データフロー解析
  - 各種 Peephole 最適化
- OpenJIT レジスタ割付機能を構成する小機能項目
  - 仮想レジスタ管理
  - 物理レジスタ管理
  - レジスタ割付

第 5 章

試験方法

5.1 OpenJIT フロントエンドシステム

4.1 節の各試験項目に関する試験方法を次ページ以降に示す.

試験項目: OpenJIT コンパイラ起動試験	(7) 合否判定

(1) 試験目的，試験内容

JDK のインタフェースによって OpenJIT が起動できることを確認する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 JDK のインタフェースによって OpenJIT が起動できることが予想される．

確認方法 empty クラスを実行し，OpenJIT フロントエンド基盤機能が無事起動された時点で，OpenJIT.Sparc オブジェクトの toString() メソッドを呼び出すことで，オブジェクトの内容を標準出力に出力する．

(4) 試験条件

empty クラスとして，以下に定義するものを用いる．

```
class empty {
  public static void main(String args[]) {}
}
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. 環境変数 CLASSPATH, JAVA_COMPILER を設定する.
2. java empty を実行する.

(6) 試験結果

試験項目: OpenJIT コンパイラ基盤機能動作試験 (1)	(7) 合否判定

(1) 試験目的，試験内容

OpenJIT コンパイラ基盤機能を構成する小機能項目が他の各機能を呼び出せることを確認する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 OpenJIT コンパイラ基盤機能を構成する小機能項目が他の各機能を呼び出せることが予想される．

確認方法 empty クラスを実行し，OpenJIT フロントエンド基盤機能が起動された後，OpenJIT バイトコードディスコンパイラ機能，OpenJIT クラスファイルアノーテション解析機能，OpenJIT 最適化機能を実現するクラスのコンストラクタのみを起動する．各コンストラクタ内では，コンストラクタの処理の後に，起動確認のメッセージを標準出力に出力する．

(4) 試験条件

empty クラスとして，以下に定義するものを用いる．

```
class empty {
  public static void main(String args[]) {}
}
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. 環境変数 CLASSPATH, JAVA_COMPILER を設定する.
2. java empty を実行する.

(6) 試験結果

試験項目: OpenJIT コンパイラ基盤機能動作試験 (2)	(7) 合否判定

(1) 試験目的，試験内容

OpenJIT コンパイラ基盤機能を構成する小機能項目の動作を確認する．

(2) 試験データの内容

Test クラスとして，以下に定義するものを用いる．

```
public class Test {
    public static void main(String args[]) {
        System.out.println("Hello, World!");
    }
}
```

(3) 予想結果及び確認方法

予想結果 OpenJIT コンパイラのフロントエンド及びバックエンドの実行が行われる．

確認方法 検査に必要な出力を行うように改造された OpenJIT コンパイラフロントエンド及びバックエンドを用いて Test クラスを実行し，標準出力の内容を確認する．

(4) 試験条件

検査に必要な出力を行うように改造された OpenJIT コンパイラフロントエンド及びバックエンドを用いる．

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. Test.java をコンパイルする (`javac Test.java`).
2. 環境変数 JAVA_COMPILER を消去する.
3. Test クラスを実行する (`java Test`).

(6) 試験結果

試験項目: OpenJIT バイトコードディスコンパイラ機能動作試験	(7) 合否判定

(1) 試験目的，試験内容

OpenJIT バイトコードディスコンパイラ機能を構成する小機能項目の動作を確認する．

(2) 試験データの内容

Exam クラスとして，以下に定義するものを用いる．

```
import java.util.Enumeration;

public class Exam {
    public Exam(Enumeration enum) {
        System.out.println((enum != null ? "items of Enumeration"
                            : "enum is null"));
        if (enum == null)
            return;
        while (enum.hasMoreElements()) {
            System.out.println(enum.nextElement());
        }
        System.out.println("end of Enumeration");
    }
}
```

(3) 予想結果及び確認方法

予想結果 バイトコードをディスコンパイルした結果としての AST が得られる．

確認方法 確認に用いるクラス Exam を定義した上でテストドライバを起動し，標準出力結果を確認する．

(4) 試験条件

テストドライバの内容は，付録 A.1.1を参照のこと．

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. Exam.java をコンパイルする (`javac Exam.java`).
2. テストドライバを起動する (`java Test 6 Exam.class`).

(6) 試験結果

試験項目: OpenJIT クラスファイルアノテーション解析機能動作試験	(7) 合否判定

(1) 試験目的，試験内容

OpenJIT クラスファイルアノテーション解析機能を構成する小機能項目の動作を確認する.

(2) 試験データの内容

A1 型のアノテーション情報に対するメタクラスとして次の定義で与えられる A1 クラスを用いる.

```
import OpenJIT.frontend.discompiler.Metaclass;
public class A1 extends Metaclass {
    public void metaInvoke() {
        System.out.println("metaclass A1: invoked.");
    }
    public String toString() {
        return "metaclass A1";
    }
}
```

(3) 予想結果及び確認方法

予想結果 A1 型のアノテーション情報に対して適切なメタクラスが起動される.

確認方法 アノテーション解析部全体試験用テストドライバを起動し，標準出力結果を確認する.

(4) 試験条件

アノテーション解析部全体試験用テストドライバの内容については，付録 A.1.2.4参照.

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. A1 クラスをコンパイルする (`javac A1.java`).
2. テストドライバを起動する (`java TestAll A1`).

(6) 試験結果

試験項目: OpenJIT 最適化機能動作試験	(7) 合否判定

(1) 試験目的，試験内容

OpenJIT 最適化機能を構成する小機能項目の動作を確認する.

(2) 試験データの内容

特になし.

(3) 予想結果及び確認方法

予想結果 最適化制御部とバイトコード出力部が起動される.

確認方法 最適化制御部動作試験用のテストドライバである Test クラスを実行する.

(4) 試験条件

テストドライバの内容に関しては，付録 A.1.3参照.

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. Test クラスを実行する (`java Test all`).

(6) 試験結果

試験項目: OpenJIT フローグラフ構築機能動作試験	(7) 合否判定

(1) 試験目的，試験内容

OpenJIT フローグラフ構築機能を構成する小機能項目の動作を確認する．

(2) 試験データの内容

Test クラスとして，以下に定義するものを用いる．

```
public class Test {
    public static void main(String args[]) {
        System.out.println("Hello, World!");
    }
}
```

(3) 予想結果及び確認方法

予想結果 入力を受け取り，データフローグラフの構築，コントロール依存グラフの構築，クラス階層解析を行うメソッド順に呼ばれることが予想される．

確認方法 DataFlowGraph クラス， ControlDependencyGraph クラス， ClassHierarchyGraph クラスを検査に必要な出力を行うように改造した OpenJIT コンパイラフロントエンド及びバックエンドを用いて Test クラスを実行し，標準出力の内容を確認する．

(4) 試験条件

検査に必要な出力を行うように改造された DataFlowGraph クラス， ControlDependencyGraph クラス， ClassHierarchyGraph クラスを用いた OpenJIT コンパイラを用いる．

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. Test.java をコンパイルする.
2. java Test を実行する.

(6) 試験結果

試験項目: OpenJIT フローグラフ解析機能動作試験	(7) 合否判定

(1) 試験目的，試験内容

OpenJIT フローグラフ解析機能を構成する小機能項目の動作を確認する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 各種フローグラフ解析機能が呼び出され，解析が行われることが予想される．

確認方法 OpenJIT フローグラフ解析機能のクラスを検査に必要な出力を行うように改造した OpenJIT コンパイラフロントエンド及びバックエンドを用いて Test クラスを実行し，標準出力の内容を確認する．

(4) 試験条件

検査に必要な出力を行うように改造された FlowGraphAnalysis クラス，DFFunctionRegister クラス，ReachingAnalyzer クラス，AvailableAnalyzer クラス，LivenessAnalyzer クラス，FixedPointDetector クラス，ClassHierarchyAnalysis クラスを用いた OpenJIT コンパイラを用いる．

また，Test クラスとして，以下に定義するものを用いる．

```
public class Test {
  public Test() {
    int a = 1;
    int b = 0;
    int c = 3;
    a = b + 1;
    b = c + a;
    a = b + c;
} }
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. Test.java をコンパイルする.
2. java Test を実行する.

(6) 試験結果

試験項目: OpenJIT プログラム変換機能動作試験	(7) 合否判定

(1) 試験目的，試験内容

OpenJIT プログラム変換機能を構成する小機能項目の動作を確認する．

(2) 試験データの内容

変換ルールとして整数定数 3 を表す AST を整数定数 7 を表す AST に変換するルールを登録し，整数定数 3 を表す AST をマッチし，変換する．

(3) 予想結果及び確認方法

予想結果 プログラム変換が行われる．

確認方法 プログラム変換機能の動作を試験するテストドライバを実行し，標準出力の内容を確認する．

(4) 試験条件

使用するテストドライバについては，付録 A.1.4.6参照.

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. テストドライバを実行する (java Test).

(6) 試験結果

5.2 OpenJIT バックエンドシステム

4.2 節の各試験項目に関する試験方法を次ページ以降に示す.

試験項目: ネイティブコード変換試験	(7) 合否判定

(1) 試験目的，試験内容

与えられたバイトコードに対し，ネイティブコードが生成されることを確認する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 ネイティブコードがメモリ上に生成されていることが予想される．

確認方法 デバッガ (gdb) でブレークポイントを設定し，nativeTest クラスを実行する．プログラムがブレークポイントで停止した時点で，デバッガのコマンドを使ってネイティブコードが生成されていることを確認する．

(4) 試験条件

nativeTest クラスとして，以下に定義するものを用いる．

```
class nativeTest {
    static int args_size;
    public static void main(String argv[]) {
        args_size = argv.length;
    }
}
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. 環境変数 CLASSPATH, JAVA_COMPILER を設定する.
2. デバッガ (gdb) を起動する.
3. ファイル api.c の OpenJIT_compile 関数の do_execute_java_method_vararg を呼び出した後の行にブレークポイントを設定する.
4. run -Dcompile.enable=nativeTest nativeTest を実行する.
5. mb->CompiledCode から mb->CompiledCodeInfo のサイズの逆アセンブルを行う.

(6) 試験結果

試験項目: メソッド情報受け渡し試験	(7) 合否判定

(1) 試験目的，試験内容

メソッドに関するＪＤＫの内部構造を Java のデータ構造に変換できているか試験する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 メソッドに関する情報が Java で読めていることが予想される．

確認方法 付録 A.2.1のクラスを定義し，OpenJIT システムに組み込む．その後，OPENJIT_COMPILER 環境変数を変更し，empty クラスを実行する．メソッドに関する情報が標準出力に出力される．

(4) 試験条件

empty クラスとして，以下に定義するものを用いる．

```
class empty {
  public static void main(String args[]) {}
}
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. 環境変数 CLASSPATH, JAVA_COMPILER を設定する.

2. 環境変数 OPENJIT_COMPILER を OpenJIT/TestMethod に設定する.

3. java empty を実行する.

(6) 試験結果

試験項目: バイトコード読み出し試験	(7) 合否判定

(1) 試験目的，試験内容

JDK の内部データであるバイトコードが Java で読めるかどうか試験する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 バイトコードが Java で読めていることが予想される．

確認方法 付録 A.2.2のクラスを定義し， OpenJIT システムに組み込む．その後，OPENJIT_COMPILER 環境変数を変更し， empty クラスを実行する．読み出したバイトコードが標準出力に 16 進表示で出力される．

(4) 試験条件

empty クラスとして，以下に定義するものを用いる．

```
class empty {
  public static void main(String args[]) {}
}
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. 環境変数 CLASSPATH, JAVA_COMPILER を設定する.

2. 環境変数 OPENJIT_COMPILER を OpenJIT/TestBytecode に設定する.

3. java empty を実行する.

(6) 試験結果

試験項目: バックエンド中間コード変換試験	(7) 合否判定

(1) 試験目的，試験内容

バイトコードからバックエンド中間コードに変換できるか試験する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 バイトコードがバックエンド中間コードに変換されていることが予想される．

確認方法 付録 A.2.3のクラスを定義し，OpenJIT システムに組み込む．その後，OPENJIT_COMPILER 環境変数を変更し，empty クラスを実行する．バイトコードがバックエンド中間コードに変換された結果が標準出力に出力される．

(4) 試験条件

empty クラスとして，以下に定義するものを用いる．

```
class empty {
  public static void main(String args[]) {}
}
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. 環境変数 CLASSPATH, JAVA_COMPILER を設定する.

2. 環境変数 OPENJIT_COMPILER を OpenJIT/TestParse に設定する.

3. java empty を実行する.

(6) 試験結果

試験項目: 命令パターンマッチング試験	(7) 合否判定

(1) 試験目的，試験内容

特定のバイトコード列に対し，最適化したバックエンド中間コードに変換できるか試験する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 特定のバイトコード列が特定のバックエンド中間コードに変換されていることが予想される．

確認方法 付録 A.2.3のクラスを定義し，OpenJIT システムに組み込む．その後，OPENJIT_COMPILER 環境変数を変更し，PatternMatch クラスを実行する．PatternMatch クラスがバックエンド中間コードに変換された結果が標準出力に出力される．

(4) 試験条件

PatternMatch クラスとして，以下に定義するものを用いる．

```
class PatternMatch {
    public static void main(String argv[]) {
        boolean b = true;
        long lx = 0;
        long ly = 0;
        double dx = 0.0;
        double dy = 0.0;

        b = !b;
        b = lx == ly;b = lx != ly;b = lx > ly;
        b = lx < ly;b = lx >= ly;b = lx <= ly;
        b = dx == dy;b = dx != dy;b = dx > dy;
        b = dx < dy;b = dx >= dy;b = dx <= dy;
    }
}
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. 環境変数 CLASSPATH, JAVA_COMPILER を設定する.

2. 環境変数 OPENJIT_COMPILER を OpenJIT/TestParse に設定する.

3. java PatternMatch を実行する.

(6) 試験結果

試験項目: RTL 変換試験	(7) 合否判定

(1) 試験目的，試験内容

バックエンド中間コードから RTL に変換できるか試験する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 バックエンド中間コードから RTL に変換されていることが予想される．

確認方法 起動時のオプションとして -Dcompile.debug=1 を指定し，empty クラスを実行する．RTL が標準出力に出力される．

(4) 試験条件

empty クラスとして，以下に定義するものを用いる．

```
class empty {
  public static void main(String args[]) {}
}
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. 環境変数 CLASSPATH, JAVA_COMPILER を設定する.

2. java -Dcompile.debug=1 -Dcompile.enable=.compile empty を実行する.

(6) 試験結果

試験項目: Peephole 最適化試験	(7) 合否判定

(1) 試験目的，試験内容

RTL の最適化ができるか試験する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 RTL が最適化されていることが予想される．

確認方法 起動時のオプションとして -Dcompile.debug=2 を指定し， empty クラスを実行する．最適化された RTL が標準出力に出力される．

(4) 試験条件

empty クラスとして，以下に定義するものを用いる．

```
class empty {
  public static void main(String args[]) {}
}
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. 環境変数 CLASSPATH, JAVA_COMPILER を設定する.

2. java -Dcompile.debug=2 -Dcompile.enable=.compile empty を実行する.

(6) 試験結果

試験項目: 整数レジスタ割付試験	(7) 合否判定

(1) 試験目的，試験内容

整数レジスタ割り付け機能を試験する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 整数レジスタの割り付けがされ，割り付けらなかったレジスタに一時レジスタが使われ，スピルコードが生成されていることが予想される．

確認方法 起動時のオプションとして -Dcompile.debug=2 を指定し， TestRegInt クラスを実行する．最適化された RTL が標準出力に出力される．

また，デバッガ (gdb) でブレークポイントを設定し， nativeTest クラスを実行する．プログラムがブレークポイントで停止した時点で，デバッガのコマンドを使って，ネイティブコードがレジスタ割付されていることを確認する．

(4) 試験条件

TestRegInt クラスとして，以下に定義するものを用いる．

```
class TestRegInt {
    public static void main(String argv[]) { test(); }
    public static void test() {
        int i1,i2,i3,i4,i5,i6,i7,i8,i9,i10;
        int i11,i12,i13,i14,i15,i16,i17,i18,i19,i20;
        int i21,i22,i23,i24,i25,i26,i27,i28,i29,i30;

        i1=1; i2=2; i3=3; i4=4; i5=5;
        i6=6; i7=7; i8=8; i9=9; i10=10;
        i11=11; i12=12; i13=13; i14=14; i15=15;
        i16=16; i17=17; i18=18; i19=19; i20=20;
        i21=20; i22=22; i23=23; i24=24; i25=25;
        i26=26; i27=27; i28=28; i29=29; i30=30;
    }
}
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. 環境変数 CLASSPATH, JAVA_COMPILER を設定する.
2. デバッガ (gdb) を起動する.
3. ファイル api.c の OpenJIT_compile 関数の do_execute_java_method_vararg を呼び出した後の行にブレークポイントを設定する.
4. run -Dcompile.debug=2 -Dcompile.enable=.test TestRegInt を実行する.
5. mb->CompiledCode から mb->CompiledCodeInfo のサイズの逆アセンブルを行う.

(6) 試験結果

試験項目: 浮動小数レジスタ割付試験	(7) 合否判定

(1) 試験目的，試験内容

浮動小数レジスタ割り付け機能を試験する．

(2) 試験データの内容

特になし．

(3) 予想結果及び確認方法

予想結果 浮動小数レジスタの割り付けがされ，割り付けらなかったレジスタに一時レジスタが使われ，スピルコードが生成されていることが予想される．

確認方法 起動時のオプションとして -Dcompile.debug=2 を指定し， TestRegInt クラスを実行する．最適化された RTL が標準出力に出力される．

また，デバッガ (gdb) でブレークポイントを設定し， nativeTest クラスを実行する．プログラムがブレークポイントで停止した時点で，デバッガのコマンドを使って，ネイティブコードがレジスタ割付されていることを確認する．

(4) 試験条件

TestRegFloat クラスとして，以下に定義するものを用いる．

```
class TestRegFloat {
    public static void main(String argv[]) { test(); }
    public static void test() {
        float i1,i2,i3,i4,i5,i6,i7,i8,i9,i10;
        float i11,i12,i13,i14,i15,i16,i17,i18,i19,i20;
        float i21,i22,i23,i24,i25,i26,i27,i28,i29,i30;

        i1=1.0F; i2=2.0F; i3=3.0F; i4=4.0F; i5=5.0F;
        i6=6.0F; i7=7.0F; i8=8.0F; i9=9.0F; i10=10.0F;
        i11=11.0F; i12=12.0F; i13=13.0F; i14=14.0F; i15=15.0F;
        i16=16.0F; i17=17.0F; i18=18.0F; i19=19.0F; i20=20.0F;
        i21=20.0F; i22=22.0F; i23=23.0F; i24=24.0F; i25=25.0F;
        i26=26.0F; i27=27.0F; i28=28.0F; i29=29.0F; i30=30.0F;
    }
}
```

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. 環境変数 CLASSPATH, JAVA_COMPILER を設定する.
2. デバッガ (gdb) を起動する.
3. ファイル api.c の OpenJIT_compile 関数の do_execute_java_method_vararg を呼び出した後の行にブレークポイントを設定する.
4. run -Dcompile.debug=2 -Dcompile.enable=.test TestRegFloat を実行する.
5. mb->CompiledCode から mb->CompiledCodeInfo のサイズの逆アセンブルを行う.

(6) 試験結果

試験項目: javac 動作試験	(7) 合否判定

(1) 試験目的，試験内容

JDK に付属の javac コマンドを実行して， OpenJIT システム全体の動作が正しいことを確認する.

(2) 試験データの内容

特になし.

(3) 予想結果及び確認方法

予想結果 javac コマンドによって生成されたクラスファイルが， OpenJIT システムを使わないときと使ったときとで同じであることが予想される.

確認方法 OpenJIT システムを使わずに javac を実行して生成されたクラスファイルと， OpenJIT システムを使って生成されたクラスファイルが diff コマンドなどを使って同じであることを確認する.

(4) 試験条件

JDK に付属の demo プログラムの SpreadSheet.java を javac の入力として使う.

(次ページへ続く)

(前ページからの続き)

(5) 試験手順

試験手順は以下の通りである.

1. /tmp/A ディレクトリを作成し，そこに SpreadSheet.java をコピーする.

2. SpreadSheet.java を javac を用いてコンパイルする.

3. /tmp/B ディレクトリを作成し，そこに SpreadSheet.java をコピーする.

4. 環境変数 CLASSPATH，LD_LIBRARY_PATH，JAVA_COMPILER を設定する.

5. SpreadSheet.java を javac を用いてコンパイルする.

6. /tmp ディレクトリで diff -sr A B を実行する.

(6) 試験結果

付録 A

試験方法補足

A.1 OpenJIT フロントエンドシステム

A.1.1 ディスコンパイル機能用テストドライバ

ディスコンパイル機能の試験で用いられるテストドライバは，次の A.1.1.1, A.1.1.2, A.1.1.3, A.1.1.4, A.1.1.5で定義されるクラス群で構成されている.

A.1.1.1 OpenJIT.frontend.discompiler.driver.Constants クラス

```
package OpenJIT.frontend.discompiler.driver;

public interface Constants {
    public static final int BYTECODE_PARSER = 0;
    public static final int CONTROL_FLOW_GRAPH = 1;
    public static final int BASICBLOCK_ANALYZER = 2;
    public static final int EXPRESSION_ANALYZER = 3;
    public static final int DOMINATOR_TREE = 4;
    public static final int STRUCTURED_CFG = 5;
    public static final int DISCOMPILED_AST = 6;
};
```

A.1.1.2 OpenJIT.frontend.discompiler.driver.StandaloneDiscompiler クラス

```
package OpenJIT.frontend.discompiler.driver;
```

```
import OpenJIT.frontend.classfile.*;
import OpenJIT.frontend.discompiler.*;
import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.PrintStream;

public class StandaloneDiscompiler extends ClassFile implements Constants {
    public StandaloneDiscompiler(InputStream is) throws IOException {
        super(is);
    }

    private String canonClassName(String name) {
        return name.replace('/', '.');
    }

    public void print(PrintStream out, int level) {
        IndentedPrintStream iout = new IndentedPrintStream(out, 4);
        iout.println(AccessFlag.toString(accessFlags));
        StringBuffer buf = new StringBuffer();
        String thisClassName = constantPool.resolveClassName(thisClass);
        buf.append("class ").append(canonClassName(thisClassName))
            .append(" extends ")
            .append(canonClassName(constantPool.resolveClassName(superClass)));
        if (interfaces.length > 0) {
            buf.append(" implements ")
                .append(canonClassName(constantPool
                    .resolveClassName(interfaces[0])));
            for (int i = 1, count = interfaces.length; i < count; i++)
                buf.append(", ")
                    .append(canonClassName(constantPool
                        .resolveClassName(interfaces[i])));
        }
        buf.append(" {");
        iout.println(buf.toString());
```

```
iout.inc();
for (int i = 0, count = fields.length; i < count; i++) {
    FieldInfo field = fields[i];
    StringBuffer sbuf = new StringBuffer();
    sbuf.append(AccessFlag.toString(field.accessFlags()))
        .append(NameAndType.toString(((ConstantUTF8)constantPool
            .itemAt(field.descriptorIndex())).bytes()))
        .append(" ")
        .append(constantPool.itemAt(field.nameIndex()).toString());
    ConstantValueAttribute attr
        = (ConstantValueAttribute)field.attributes()
        .lookup(Attributes.CONSTANTVALUE);
    if (attr != null) {
        sbuf.append(" = ");
        int index = attr.constantValueIndex();
        ConstantPoolItem constant = constantPool.itemAt(index);
        if (constant.isString()) {
            sbuf.append("\"")
                .append(constantPool.resolveString(index))
                .append("\"");
        } else
            sbuf.append(constant.toString());
    }
    sbuf.append(";");
    iout.println(sbuf.toString());
}
TestDiscompiler discompiler[] = new TestDiscompiler[methods.length];
for (int i = 0, count = methods.length; i < count; i++) {
    MethodInfo method = methods[i];
    StringBuffer sbuf = new StringBuffer();
    sbuf.append(AccessFlag.toString(method.accessFlags()));
    String name = constantPool.itemAt(method.nameIndex()).toString();
    ConstantUTF8 descriptor
        = (ConstantUTF8)constantPool.itemAt(method.descriptorIndex());
    try {
        NameAndType sig
            = new NameAndType(null, name, descriptor.bytes());
```

```
        sbuf.append(sig.toString());
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("cannot parse: " + descriptor);
        throw e;
    }
    if (method.attributes().lookup(Attributes.CODE) != null) {
        sbuf.append(" {");
        iout.println(sbuf.toString());
        iout.inc();
        StandaloneMethodInformation methodInfo
            = new StandaloneMethodInformation(method);
        discompiler[i] = new TestDiscompiler(methodInfo);
        switch (level) {
        case BYTECODE_PARSER:
            discompiler[i].printBytecode(iout);
            break;
        case CONTROL_FLOW_GRAPH:
            discompiler[i].printCFG(iout);
            break;
        case BASICBLOCK_ANALYZER:
            discompiler[i].printBBACFG(iout);
            break;
        case EXPRESSION_ANALYZER:
            discompiler[i].printEACFG(iout);
            break;
        case DOMINATOR_TREE:
            discompiler[i].printDT(iout);
            break;
        case STRUCTURED_CFG:
            discompiler[i].printSCFG(iout);
            break;
        case DISCOMPILED_AST:
            discompiler[i].printAST(iout);
            break;
    }
        iout.dec();
        iout.println("}");
```

```
            } else {
                sbuf.append(";");
                iout.println(sbuf.toString());
            }
        }
        iout.dec();
        iout.println("}");
    }
}
```

A.1.1.3 OpenJIT.frontend.discompiler.driver.StandaloneMethodInformation クラス

```
package OpenJIT.frontend.discompiler.driver;

import OpenJIT.Constants;
import OpenJIT.ExceptionHandler;
import OpenJIT.frontend.classfile.*;
import OpenJIT.frontend.discompiler.*;
import OpenJIT.frontend.util.IntKeyHashtable;

public class StandaloneMethodInformation implements MethodInformation, Constants {
    private MethodInfo method;
    private CodeAttribute code;
    private ConstantPool constantPool;
    private byte[] bytecode;

    public StandaloneMethodInformation(MethodInfo method) {
        this.method = method;
        this.constantPool = method.classFile().constantPool();
        code = (CodeAttribute)method.attributes().lookup(Attributes.CODE);
        bytecode = code.code();
    }

    /**
     * Returns the number of local variables.
     */
```

```
public int nlocals() {
    return code.maxLocals();
}

/**
 * Returns whether the method is static one.
 */
public boolean isStatic() {
    return (method.accessFlags() & ACC_STATIC) != 0;
}

private String thisClassName;
/**
 * Returns a String of the class name of the discompiled method belongs to.
 */
public synchronized String thisClassName() {
    if (thisClassName != null)
        return thisClassName;
    thisClassName = method.classFile().thisClassName();
    return thisClassName;
}

/**
 * Returns the name of the class indeced by index into ConstantPool.
 */
public String className(int index) {
    return constantPool.resolveString(index);
}

/**
 * Returns the width of the field/method indeced by given index into
 * ConstantPool.
 */
public int fieldWidth(int index) {
    NameAndType sig = nameAndType(index);
    switch (sig.type()[0]) {
    case SIGC_LONG:
```

```
    case SIGC_DOUBLE:
        return 2;
    default:
        return 1;
    }
}

/**
 * Returns the name and type information of method/field reference
 * indeced by index into ConstantPool.
 */
private IntKeyHashtable nameAndTypes = new IntKeyHashtable();
public NameAndType nameAndType(int index) {
    NameAndType result = (NameAndType)nameAndTypes.get(index);
    if (result != null)
        return result;
    String className = constantPool.resolveClassName(index);
    String name = constantPool.resolveMemberName(index);
    byte descriptor[] = constantPool.resolveMemberDescriptor(index);
    result = new NameAndType(className, name, descriptor);
    nameAndTypes.put(index, result);
    return result;
}

/**
 * Returns the kind of ConstantPool entry indeced by given index.
 */
public int kindOfConstant(int index) {
    return constantPool.itemAt(index).tag();
}

/**
 * Returns an int value of ConstantPool indeced by given index.
 */
public int constantInt(int index) {
    return constantPool.resolveInt(index);
}
```

```
/**
 * Returns an float value of ConstantPool indeced by given index.
 */
public float constantFloat(int index) {
    return constantPool.resolveFloat(index);
}

/**
 * Returns an String value of ConstantPool indeced by given index.
 */
public String constantString(int index) {
    return constantPool.resolveString(index);
}

/**
 * Returns an double value of ConstantPool indeced by given index.
 */
public double constantDouble(int index) {
    return constantPool.resolveDouble(index);
}

/**
 * Returns an long value of ConstantPool indeced by given index.
 */
public long constantLong(int index) {
    return constantPool.resolveLong(index);
}

/**
 * Returns the array of ExceptionHandler.
 */
public ExceptionHandler[] exceptionHandler() {
    return code.exceptionTable();
}

/**
```

```
 * Returns the length of bytecode[].
 */
public int bytecodeLength() {
    return bytecode.length;
}

/**
 * Returns (signed)byte-value at pc of bytecode[].
 */
public int byteAt(int pc) {
    return bytecode[pc];
}

/**
 * Returns (unsigned)byte-value at pc of bytecode[].
 */
public int unsignedByteAt(int pc) {
    return bytecode[pc] & 0xff;
}

/**
 * Returns (signed)short-value at pc of bytecode[].
 */
public int shortAt(int pc) {
    return (short)((bytecode[pc] << 8) + unsignedByteAt(pc + 1));
}

/**
 * Returns (unsigned)short-value at pc of bytecode[].
 */
public int unsignedShortAt(int pc) {
    return ((unsignedByteAt(pc) << 8) + unsignedByteAt(pc + 1));
}

/**
 * Returns int-value at pc of bytecode[].
 */
```

```
    public int intAt(int pc) {
        return ((shortAt(pc) << 16) + unsignedShortAt(pc + 2));
    }
}
```

A.1.1.4 OpenJIT.frontend.discompiler.driver.Test クラス

```
import OpenJIT.frontend.discompiler.driver.StandaloneDiscompiler;
import java.io.FileInputStream;
import java.io.IOException;

public class Test {
    public static void main(String args[]) {
        if (args.length > 1) {
            int level = Integer.parseInt(args[0]);
            try {
                FileInputStream fin = new FileInputStream(args[1]);
                StandaloneDiscompiler discompiler
                    = new StandaloneDiscompiler(fin);
                discompiler.print(System.out, level);
                fin.close();
            } catch (IOException e) {
                System.err.println("No such file: " + args[1]);
                System.exit(1);
            }
        } else
            usage();
    }

    static void usage() {
        System.err.println("usage: java Test level filename");
    }
}
```

A.1.1.5 OpenJIT.frontend.discompiler.driver.TestDiscompiler クラス

```
package OpenJIT.frontend.discompiler.driver;
```

```
import OpenJIT.frontend.discompiler.*;
import OpenJIT.frontend.tree.Node;
import OpenJIT.frontend.util.IndentedPrintStream;

public class TestDiscompiler extends Discompiler {
    public TestDiscompiler(MethodInformation method) {
        super(method);
    }

    public void printBytecode(IndentedPrintStream out) {
        bytecodeInfo.print(out);
    }

    public void printCFG(IndentedPrintStream out) {
        ControlFlowGraph cfg = new ControlFlowGraph(method, bytecodeInfo);
        cfg.createCFG();
        cfg.print(out);
    }

    public void printBBACFG(IndentedPrintStream out) {
        BasicBlockAnalyzer cfg = new BasicBlockAnalyzer(method, bytecodeInfo,
                                                        astFactory);
        cfg.createCFG();
        cfg.print(out);
    }

    public void printEACFG(IndentedPrintStream out) {
        structurelessCFG.print(out);
    }

    public void printDT(IndentedPrintStream out) {
        structurelessCFG.printTree(out);
    }

    public void printSCFG(IndentedPrintStream out) {
        ControlFlowAnalyzer scfg = new ControlFlowAnalyzer(structurelessCFG);
```

```
        scfg.print(out);
    }

    public void printAST(IndentedPrintStream out) {
        Node head = discompile();
        head.print(out.out);
    }
}
```

A.1.2 OpenJIT クラスファイルアノテーション解析機能

A.1.2.1 アノテーション解析部用テストドライバ

```
import OpenJIT.frontend.discompiler.*;
import java.io.*;

public class TestAnnotationAnalysis extends AnnotationAnalyzer {
    public static void main(String args[]) throws Exception {
if (args.length == 0)
    return;

Annotation annotation = new Annotation(args[0]);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
oos.writeObject(annotation);
oos.flush();
byte[] attribute = baos.toByteArray();

System.out.println(readAnnotation(attribute));
    }
}
```

A.1.2.2 アノテーション登録部用テストドライバ

```
import OpenJIT.frontend.discompiler.*;
import java.io.*;

public class TestAnnotationRegister extends AnnotationAnalyzer {
    public static void main(String args[]) throws Exception {
if (args.length == 0)
    return;
```

```
Annotation annotation = new Annotation(args[0]);

try {
    registerAnnotation(annotation);
} catch (DiscompilerError e) {
    System.out.println("registerAnnotation: fail");
    throw e;
}
System.out.println("registerAnnotation: success");
    }
}
```

A.1.2.3 メタクラス制御部用テストドライバ

```
import OpenJIT.frontend.discompiler.*;
import java.io.*;

public class TestMetaclass extends AnnotationAnalyzer {
    public static void main(String args[]) throws Exception {
if (args.length == 0)
    return;

Annotation annotation = new Annotation(args[0]);

System.out.println(addMetaobject(annotation));
    }
}
```

A.1.2.4 アノテーション解析部全体試験用テストドライバ

```
import OpenJIT.frontend.discompiler.*;
```

```
import java.io.*;

public class TestAll extends AnnotationAnalyzer {
    public static void main(String args[]) throws Exception {
if (args.length == 0)
    return;

Annotation annotation = new Annotation(args[0]);
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(baos);
oos.writeObject(annotation);
oos.flush();
byte[] attribute = baos.toByteArray();

Annotation test = readAnnotation(attribute);
System.out.print("after analysis: ");
System.out.println(test);

try {
    registerAnnotation(test);
} catch (DiscompilerError e) {
    System.out.println("registerAnnotation: fail");
    throw e;
}
System.out.println("registerAnnotation: success");

System.out.print("after metaobject added: ");
System.out.println(addMetaobject(annotation));

annotation.metaobject.metaInvoke();
    }
```

```
}
```

A.1.3 最適化機能用テストドライバ

最適化制御部動作試験に用いられるテストドライバは，次のように定義される．

```
import OpenJIT.frontend.discompiler.ControlFlowGraph;
import OpenJIT.frontend.flowgraph.Optimizer;
import OpenJIT.frontend.tree.Node;

public class Test extends Optimizer {
    int debuglevel;

    public Test() {
        super(null);
        debuglevel = 0;
    }

    public byte[] optimize(byte bytecode[], Node ast, ControlFlowGraph cfg) {
        if (debuglevel != 2)
            System.out.println("optimize: ok");
        if (debuglevel != 1)
            generateBytecode();
        return null;
    }

    public byte[] generateBytecode() {
        System.out.println("generateBytecode: ok");
        return null;
    }

    public static void main(String args[]) {
        Test test = new Test();
        if (args.length != 1)
```

```
            return;
        if (args[0].equals("optimize"))
            test.debuglevel = 1;
        else if (args[0].equals("gen"))
            test.debuglevel = 2;
        else if (args[0].equals("all"))
            test.debuglevel = 3;
        test.optimize(null, null, null);
    }
}
```

A.1.4 プログラム変換機能用テストドライバ

A.1.4.1 AST 変換ルール登録部動作試験

AST 変換ルール登録部の試験で用いられるテストドライバは，次のように定義される．

```
import OpenJIT.frontend.tree.*;
import OpenJIT.frontend.flowgraph.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

public class Test extends ASTTransformer {
    public Test() {
        super(null);
    }

    public static void main(String args[]) {
        Test test = new Test();
        test.test();
    }

    public void test() {
        Expression from = new IntExpression(0, 3);
        Expression to = new IntExpression(0, 7);
        registerRule(from, to);
        Enumeration keys = dst.keys();
        Enumeration elements = dst.elements();
        while (keys.hasMoreElements()) {
            System.out.print(keys.nextElement());
            System.out.print(" -> ");
            System.out.println(elements.nextElement());
```

```
        }
    }
}
```

A.1.4.2 AST パターンマッチ部動作試験 (1)

AST パターンマッチ部動作試験 (1) で用いられるテストドライバは，次のように定義される.

```
import OpenJIT.frontend.tree.*;
import OpenJIT.frontend.flowgraph.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

public class Test extends ASTTransformer {
    public Test() {
        super(null);
    }

    public static void main(String args[]) {
        Test test = new Test();
        test.test();
    }

    public void test() {
        Expression from = new IntExpression(0, 3);
        Expression to = new IntExpression(0, 7);
        registerRule(from, to);
        Node result = match(from);
        if (result != null) {
            System.out.print(from);
```

```
            System.out.print(" matches with ");
            System.out.println(result);
        }
    }
}
```

A.1.4.3 AST パターンマッチ部動作試験 (2)

AST パターンマッチ部動作試験 (2) で用いられるテストドライバは，次のように定義される.

```
import OpenJIT.frontend.tree.*;
import OpenJIT.frontend.flowgraph.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

public class Test extends ASTTransformer {
    public Test() {
        super(null);
    }

    public static void main(String args[]) {
        Test test = new Test();
        test.test();
    }

    public void test() {
        Expression from = new IntExpression(0, 3);
        Expression to = new IntExpression(0, 7);
        registerRule(from, to);
        Node result = match(to);
```

```
        if (result == null) {
            System.out.print(to);
            System.out.println(" doesn't match with any rules.");
        }
    }
}
```

A.1.4.4 AST 変換部動作試験 (1)

AST 変換部動作試験 (1) で用いられるテストドライバは，次のように定義される．

```
import OpenJIT.frontend.tree.*;
import OpenJIT.frontend.flowgraph.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

public class Test extends ASTTransformer {
    public Test() {
        super(null);
    }

    public static void main(String args[]) {
        Test test = new Test();
        test.test();
    }

    public void test() {
        Expression from = new IntExpression(0, 3);
        Expression to = new IntExpression(0, 7);
        registerRule(from, to);
        Node result = transform(from);
```

```
        System.out.print(from);
        System.out.print(" is transformed to ");
        System.out.println(result);
    }
}
```

A.1.4.5 AST 変換部動作試験 (2)

AST 変換部動作試験 (2) で用いられるテストドライバは，次のように定義される．

```
import OpenJIT.frontend.tree.*;
import OpenJIT.frontend.flowgraph.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

public class Test extends ASTTransformer {
    public Test() {
        super(null);
    }

    public static void main(String args[]) {
        Test test = new Test();
        test.test();
    }

    public void test() {
        Expression from = new IntExpression(0, 3);
        Expression to = new IntExpression(0, 7);
        registerRule(from, to);
        Node result = transform(to);
```

```
        System.out.print(to);
        System.out.print(" is transformed to ");
        System.out.println(result);
    }
}
```

A.1.4.6 OpenJIT プログラム変換機能動作試験

OpenJIT プログラム変換機能動作試験で用いられるテストドライバは，次のように定義される．

```
import OpenJIT.frontend.tree.*;
import OpenJIT.frontend.flowgraph.*;
import java.util.Vector;
import java.util.Hashtable;
import java.util.Enumeration;

public class Test extends ASTTransformer {
    public Test() {
        super(null);
    }

    public static void main(String args[]) {
        Test test = new Test();
        test.test();
    }

    public void test() {
        Expression from = new IntExpression(0, 3);
        Expression to = new IntExpression(0, 7);
        registerRule(from, to);
```

```
        System.out.print("registering: ");
        System.out.print(from);
        System.out.print(" -> ");
        System.out.println(to);

        Enumeration keys = dst.keys();
        Enumeration elements = dst.elements();
        while (keys.hasMoreElements()) {
            System.out.print("registered rule: ");
            System.out.print(keys.nextElement());
            System.out.print(" -> ");
            System.out.println(elements.nextElement());
        }

        Node result = match(from);
        if (result != null) {
            System.out.print(from);
            System.out.print(" matches with ");
            System.out.println(result);
            result = transform(from);
            System.out.print(from);
            System.out.print(" is transformed to ");
            System.out.println(result);
        }
    }
}
```

A.1.5 OpenJIT フロントエンド用テストドライバで使用されているその他のクラス

OpenJIT フロントエンド用テストドライバでは，クラスファイルを読み込むために次のように定義される OpenJIT.frontend.classfile パッケージを使用している．

A.1.5.1 public class AccessFlag implements Constants

```
/*
 * $Id: AccessFlag.java,v 1.1 1998/12/20 18:45:21 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.Constants;

public class AccessFlag implements Constants {
    public static final boolean isPublic(int flags) {
        return (flags & ACC_PUBLIC) != 0;
    }
    public static final boolean isPrivate(int flags) {
        return (flags & ACC_PRIVATE) != 0;
    }
    public static final boolean isProtected(int flags) {
        return (flags & ACC_PROTECTED) != 0;
    }
    public static final boolean isStatic(int flags) {
        return (flags & ACC_STATIC) != 0;
    }
    public static final boolean isFinal(int flags) {
        return (flags & ACC_FINAL) != 0;
    }
```

```
public static final boolean isSuper(int flags) {
    return (flags & ACC_SUPER) != 0;
}
public static final boolean isSynchronized(int flags) {
    return (flags & ACC_SYNCHRONIZED) != 0;
}
public static final boolean isVolatile(int flags) {
    return (flags & ACC_VOLATILE) != 0;
}
public static final boolean isTransient(int flags) {
    return (flags & ACC_TRANSIENT) != 0;
}
public static final boolean isNative(int flags) {
    return (flags & ACC_NATIVE) != 0;
}
public static final boolean isInterface(int flags) {
    return (flags & ACC_INTERFACE) != 0;
}
public static final boolean isAbstract(int flags) {
    return (flags & ACC_ABSTRACT) != 0;
}

public static String toString(int flags) {
    StringBuffer buf = new StringBuffer();
    if (isPublic(flags))
        buf.append("public ");
    if (isPrivate(flags))
        buf.append("private ");
    if (isProtected(flags))
        buf.append("protected ");
    if (isStatic(flags))
```

```
            buf.append("static ");
        if (isFinal(flags))
            buf.append("final ");
        if (isSynchronized(flags))
            buf.append("synchronized ");
        if (isVolatile(flags))
            buf.append("volatile ");
        if (isTransient(flags))
            buf.append("transient ");
        if (isNative(flags))
            buf.append("native ");
        if (isAbstract(flags))
            buf.append("abstract ");
        return buf.toString();
    }
}
```

A.1.5.2 public abstract class AttributeInfo

```
/*
 * $Id: AttributeInfo.java,v 1.6 1998/12/15 14:40:34 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

public abstract class AttributeInfo {
    protected ClassFile classFile;
    protected int attributeNameIndex;
    protected int attributeLength;
```

```
protected String attributeName;

public AttributeInfo(int nameIndex) {
    attributeNameIndex = nameIndex;
}

public AttributeInfo(int nameIndex, ClassFileInputStream stream,
                     ClassFile cF) throws IOException {
    classFile = cF;
    attributeNameIndex = nameIndex;
    attributeLength = stream.readU4();
    attributeName = classFile.constantPool.resolveString(nameIndex);
}

public void write(ClassFileOutputStream stream) throws IOException {
    stream.writeU2(attributeNameIndex);
    stream.writeU4(attributeLength);
}

public String toString() {
    return "attribute_name_index = "
        + Integer.toString(attributeNameIndex)
        + ", attribute_length = " + Integer.toString(attributeLength);
}

public String attributeName() {
    return attributeName;
}

abstract public void print(IndentedPrintStream out);
```

```
}
```

A.1.5.3 public class Attributes

```
/*
 * $Id: Attributes.java,v 1.8 1998/12/20 18:45:22 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>Attributes</code> represent a
 * set of </code>AttributeInfo</code> objects.  It appears in
 * </code>ClassFile</code> for 'SourceFile',
 * in </code>FieldInfo</code> for 'ConstantValue', in </code>MethodInfo</code>
 * for 'Code', 'Exceptions' and in </code>CodeAttribute</code> for
 * debug informations.
 * <p>
 * Each of thier have very similar style and complex, and users may
 * plan to design thier subclasses, for this reason, it uses Factory
 * Method pattern.
 */
public class Attributes {
    public static final String SOURCEFILE = "SourceFile";
    public static final String CONSTANTVALUE = "ConstantValue";
    public static final String CODE = "Code";
    public static final String EXCEPTIONS = "Exceptions";
    public static final String LINENUMBERTABLE = "LineNumberTable";
    public static final String LOCALVARIABLETABLE = "LocalVariableTable";
```

```
/**
 * It holds a set of </code>AttributeInfo</code>.
 */
protected AttributeInfo attributes[];

/**
 * Constructor.  It reads bytestream from </code>stream</code> and
 * initialize itself.  To determine what kind of attributes to be
 * read, it references a ClassFile.constantPool.
 */
public Attributes(ClassFileInputStream stream, ClassFile cF)
    throws IOException {
    int count = stream.readU2();
    attributes = new AttributeInfo[count];
    for (int i = 0; i < count; i++) {
        int nameIndex = stream.readU2();
        String attributeName = cF.constantPool.resolveString(nameIndex);
        if (attributeName.equals(SOURCEFILE))
            attributes[i] = sourceFile(nameIndex, stream, cF);
        else if (attributeName.equals(CONSTANTVALUE))
            attributes[i] = constantValue(nameIndex, stream, cF);
        else if (attributeName.equals(CODE))
            attributes[i] = code(nameIndex, stream, cF);
        else if (attributeName.equals(EXCEPTIONS))
            attributes[i] = exceptions(nameIndex, stream, cF);
        else if (attributeName.equals(LINENUMBERTABLE))
            attributes[i] = lineNumberTable(nameIndex, stream, cF);
        else if (attributeName.equals(LOCALVARIABLETABLE))
            attributes[i] = localVariableTable(nameIndex, stream, cF);
        else
```

```
            attributes[i] = unknown(attributeName, nameIndex, stream, cF);
        }
    }

    /**
     * These are factory method to construct various attribute object
     * called by constructor.  Subclasses can override these to
     * change the implementation of each attribute object.
     */
    public AttributeInfo sourceFile(int nameIndex, ClassFileInputStream stream,
                                    ClassFile cF) throws IOException {
        return new SourceFileAttribute(nameIndex, stream, cF);
    }

    public AttributeInfo constantValue(int nameIndex,
                                       ClassFileInputStream stream,
                                       ClassFile cF) throws IOException {
        return new ConstantValueAttribute(nameIndex, stream, cF);
    }

    public AttributeInfo code(int nameIndex, ClassFileInputStream stream,
                              ClassFile cF) throws IOException {
        return new CodeAttribute(nameIndex, stream, cF);
    }

    public AttributeInfo exceptions(int nameIndex, ClassFileInputStream stream,
                                    ClassFile cF) throws IOException {
        return new ExceptionsAttribute(nameIndex, stream, cF);
    }

    public AttributeInfo lineNumberTable(int nameIndex,
```

```
                                        ClassFileInputStream stream,
                                        ClassFile cF)
    throws IOException {
    return new LineNumberTableAttribute(nameIndex, stream, cF);
}

public AttributeInfo localVariableTable(int nameIndex,
                                           ClassFileInputStream stream,
                                           ClassFile cF)
    throws IOException {
    return new LocalVariableTableAttribute(nameIndex, stream, cF);
}

/**
 * This is treated specially because subclass may override it to
 * handle some AttributeInfo about what I don't know but she know.
 */
public AttributeInfo unknown(String attributeName, int nameIndex,
                             ClassFileInputStream stream,
                             ClassFile cF) throws IOException {
    return new GenericAttribute(nameIndex, stream, cF);
}

/**
 * Write this into </code>stream</code> with the style of Java classfile.
 */
public void write(ClassFileOutputStream stream) throws IOException {
    int count = attributes.length;
    stream.writeU2(count);
    for (int i = 0; i < count; i++)
        attributes[i].write(stream);
```

```
    }

    /**
     * Pretty printer of ClassFile object.
     */
    public void print(IndentedPrintStream out) {
        int count = attributes.length;
        out.println("u2 attributes_count = " + Integer.toString(count) + ";");
        out.println("attribute_info attributes[] = {");
        out.inc();
        for (int i = 0; i < count; i++)
            attributes[i].print(out);
        out.dec();
        out.println("}");
    }

    /**
     * Returns an AttributeInfo named </code>attributeName</code>.
     */
    public AttributeInfo lookup(String attributeName) {
        for (int i = 0, count = attributes.length; i < count; i++) {
            AttributeInfo attribute = attributes[i];
            if (attribute.attributeName.equals(attributeName))
                return attribute;
        }
        return null;
    }
}
```

A.1.5.4 public class ClassFile implements RuntimeConstants

```
/*
```

```
 * $Id: ClassFile.java,v 1.11 1998/12/28 15:30:01 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.java.RuntimeConstants;
import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintStream;

/**
 * Each instances of the class </code>ClassFile</code> represent a
 * classfile which represent a Java class definition.
 * <p>
 * Its constructor invokes makeInner() method to construct inner
 * structures of ClassFile object for customization about the representation
 * of its inner structure.  If the user decide to modify some of inner
 * structures, the only work should do is to make its subclasses and modify
 * the </code>new*</code> -- factory methods -- to construct appropriate
 * objects.
 * <p>
 * It can be used to not only represent newer created (by a program)
 * ClassFile, but also read from </code>*.class</code> file.
 */
public class ClassFile implements RuntimeConstants {
    /**
     * Each of these are just represent the value in </code>*.class</code>.
     */
    protected int magic;
```

```
protected int minorVersion;
protected int majorVersion;
protected int accessFlags;
protected int thisClass;
protected int superClass;
protected int interfaces[];

/**
 * </code>constantPool</code> represent a ConstantPool structure,
 * it is just an array in classfile but ConstantPool class gives
 * more abstract operations like new value appendings.
 */
protected ConstantPool constantPool;

/**
 * Each elements of </code>fields[]</code> represent a field
 * contained in this java class.
 */
protected FieldInfo fields[];

/**
 * Each elements of </code>methods[]</code> represent a method
 * contained in this java class.
 */
protected MethodInfo methods[];

/**
 * </code>attributes</code> represent some additional informations
 * like file name of the source code.
 */
protected Attributes attributes;
```

```
/**
 * Factory methods for inner complex structures.
 */
protected ConstantPool newConstantPool(ClassFileInputStream stream)
    throws IOException {
    return new ConstantPool(stream);
}
protected Attributes newAttributes(ClassFileInputStream stream,
                                   ClassFile cF) throws IOException {
    return new Attributes(stream, cF);
}

/**
 * Factory methods of FieldInfo/MethodInfo initialized from classfile.
 */
protected FieldInfo newFieldInfo(ClassFileInputStream stream)
    throws IOException {
    return new FieldInfo(stream, this);
}
protected MethodInfo newMethodInfo(ClassFileInputStream stream)
    throws IOException {

    return new MethodInfo(stream, this);
}

/**
 * Constructor.  To keep some chances of modification of its
 * inner structure, it uses factory methods for more complex
 * structures instantiation.
```

```
 */
public ClassFile(InputStream is) throws IOException {
    ClassFileInputStream stream = new ClassFileInputStream(is);
    magic = stream.readU4();
    if (magic != JAVA_MAGIC)
        throw new UnknownFileException("magic not match");
    minorVersion = stream.readU2();
    if (minorVersion != JAVA_MINOR_VERSION)
        throw new UnknownFileException("minor version not match");
    majorVersion = stream.readU2();
    if (majorVersion != JAVA_VERSION)
        throw new UnknownFileException("major version not match");
    constantPool = newConstantPool(stream);
    accessFlags = stream.readU2();
    thisClass = stream.readU2();
    superClass = stream.readU2();
    int count = stream.readU2();
    interfaces = new int[count];
    for (int i = 0; i < count; i++)
        interfaces[i] = (short)stream.readU2();
    count = stream.readU2();
    fields = new FieldInfo[count];
    for (int i = 0; i < count; i++)
        fields[i] = newFieldInfo(stream);
    count = stream.readU2();
    methods = new MethodInfo[count];
    for (int i = 0; i < count; i++)
        methods[i] = newMethodInfo(stream);
    attributes = newAttributes(stream, this);
}
```

```
/**
 * Write this into </code>stream</code> with the style of Java classfile.
 */
public void write(OutputStream os) throws IOException {
    ClassFileOutputStream stream = new ClassFileOutputStream(os);
    stream.writeU4(magic);
    stream.writeU2(minorVersion);
    stream.writeU2(majorVersion);
    constantPool.write(stream);
    stream.writeU2(accessFlags);
    stream.writeU2(thisClass);
    stream.writeU2(superClass);
    int count = interfaces.length;
    stream.writeU2(count);
    for (int i = 0; i < count; i++)
        stream.writeU2(interfaces[i]);
    count = fields.length;
    stream.writeU2(count);
    for (int i = 0; i < count; i++)
        fields[i].write(stream);
    count = methods.length;
    stream.writeU2(count);
    for (int i = 0; i < count; i++)
        methods[i]. write(stream);
    attributes.write(stream);
}

/**
 * Pretty printer of ClassFile object.
 */
public void print(PrintStream out) {
```

```
IndentedPrintStream iout = new IndentedPrintStream(out, 4);
iout.println("ClassFile {");
iout.inc();
iout.println("u4 magic = 0x" + Integer.toHexString(magic));
iout.println("u2 minor_version = " + Integer.toString(minorVersion));
iout.println("u2 major_version = " + Integer.toString(majorVersion));
constantPool.print(iout);
iout.println("u2 access_flags = 0x"
             + Integer.toHexString(accessFlags));
iout.println("u2 this_class = " + Integer.toString(thisClass));
iout.println("u2 super_class = " + Integer.toString(superClass));
int count = interfaces.length;
iout.println("u2 interfaces_count = " + Integer.toString(count));
if (count > 0) {
    StringBuffer buf = new StringBuffer();
    buf.append("{ " + Integer.toString(interfaces[0]));
    for (int i = 1; i < count; i++)
        buf.append(", " + Integer.toString(interfaces[i]));
    buf.append(" }");
    iout.println("u2 interfaces[] = " + buf.toString());
} else
    iout.println("u2 interfaces[]");
count = fields.length;
iout.println("u2 fields_count = " + Integer.toString(count));
iout.inc();
for (int i = 0; i < count; i++)
    fields[i].print(iout);
iout.dec();
count = methods.length;
iout.println("u2 methods_count = " + Integer.toString(count));
iout.inc();
```

```
        for (int i = 0; i < count; i++)
            methods[i].print(iout);
        iout.dec();
        attributes.print(iout);
        iout.println("}");
    }

    /**
     * Accessors.
     */
    public ConstantPool constantPool() {
        return constantPool;
    }

    public String thisClassName() {
        return constantPool.resolveClassName(thisClass);
    }

    public String superClassName() {
        return constantPool.resolveClassName(superClass);
    }
}
```

A.1.5.5 public class ClassFileInputStream

```
/*
 * $Id: ClassFileInputStream.java,v 1.4 1999/01/02 07:57:39 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import java.io.InputStream;
```

```
import java.io.IOException;

public class ClassFileInputStream {
    private InputStream stream;

    public ClassFileInputStream(InputStream stream) {
        this.stream = stream;
    }

    public int readU1() throws IOException {
        int d = stream.read();
        if (d == -1)
            throw new IOException("End of file");
        return d;
    }

    public int readU2() throws IOException {
        return (readU1() << 8) | readU1();
    }

    public int readU4() throws IOException {
        return (readU2() << 16) | readU2();
    }

    public long readU8() throws IOException {

        long high = readU4();
        long low = readU4();
        long result = (high << 32) | low;
        return result;
    }
```

```
    public float readFloat() throws IOException {
        return Float.intBitsToFloat(readU4());
    }

    public double readDouble() throws IOException {
        return Double.longBitsToDouble(readU8());
    }

    public byte[] readBytes(int length) throws IOException {
        byte result[] = new byte[length];
        for (int i = 0; i < length; i++)
            result[i] = (byte)(readU1() & 0xff);
        return result;
    }
}
```

A.1.5.6 public class ClassFileOutputStream

```
/*
 * $Id: ClassFileOutputStream.java,v 1.2 1998/11/24 02:15:01 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import java.io.OutputStream;
import java.io.IOException;

public class ClassFileOutputStream {
    private OutputStream stream;

    public ClassFileOutputStream(OutputStream stream) {
```

```
    this.stream = stream;
}

public void writeU1(int d) throws IOException {
    stream.write(d);
}

public void writeU2(int d) throws IOException {
    writeU1((d >> 8) & 0xff);
    writeU1(d & 0xff);
}

public void writeU4(int d) throws IOException {
    writeU2((d >> 16) & 0xffff);
    writeU2(d & 0xffff);
}

public void writeU8(long d) throws IOException {
    writeU4((int)((d >> 32) & 0xffffffff));
    writeU4((int)(d & 0xffffffff));
}

public void writeFloat(float d) throws IOException {
    writeU4(Float.floatToIntBits(d));
}

public void writeDouble(double d) throws IOException {
    writeU8(Double.doubleToLongBits(d));
}

public void writeBytes(byte bytes[]) throws IOException {
```

```
        int length = bytes.length;
        for (int i = 0; i < length; i++)
            writeU1(bytes[i]);
    }
}
```

A.1.5.7 public class CodeAttribute extends AttributeInfo

```
/*
 * $Id: CodeAttribute.java,v 1.7 1998/12/15 14:40:35 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.ExceptionHandler;
import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>CodeAttribute</code> represent the
 * body of the method contining it.
 */
public class CodeAttribute extends AttributeInfo {
    protected int maxStack;
    protected int maxLocals;
    protected byte code[];
    protected ExceptionHandler exceptionTable[];
    protected Attributes attributes;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
```

```
public CodeAttribute(int nameIndex, ClassFileInputStream stream,
                     ClassFile cF) throws IOException {
    super(nameIndex, stream, cF);
    maxStack = stream.readU2();
    maxLocals = stream.readU2();
    code = stream.readBytes(stream.readU4());
    int count = stream.readU2();
    exceptionTable = new ExceptionHandler[count];
    for (int i = 0; i < count; i++) {
        ExceptionHandler handler = new ExceptionHandler();
        handler.startPC = stream.readU2();
        handler.endPC = stream.readU2();
        handler.handlerPC = stream.readU2();
        handler.catchType = stream.readU2();
        exceptionTable[i] = handler;
    }
    attributes = cF.newAttributes(stream, cF);
}

/**
 * Write this into </code>stream</code> with the style of Java classfile.
 */
public void write(ClassFileOutputStream stream) throws IOException {
    super.write(stream);
    stream.writeU2(maxStack);
    stream.writeU2(maxLocals);
    int length = code.length;
    stream.writeU4(length);
    for (int i = 0; i < length; i++)
        stream.writeU1(code[i]);
    length = exceptionTable.length;
```

```
    stream.writeU2(length);
    for (int i = 0; i < length; i++) {
        ExceptionHandler handler = exceptionTable[i];
        stream.writeU2(handler.startPC);
        stream.writeU2(handler.endPC);
        stream.writeU2(handler.handlerPC);
        stream.writeU2(handler.catchType);
    }
    attributes.write(stream);
}

/**
 * Return simple description of this object in a String.
 */
public String toString() {
    return "Code_attribute { " + super.toString()
        + ", max_stack = " + Integer.toString(maxStack)
        + ", max_locals = " + Integer.toString(maxLocals)
        + ", code_length = " + Integer.toString(code.length)
        + ", code[], exception_table_length = "
        + Integer.toString(exceptionTable.length)
        + attributes.toString()
        + " }";
}

/**
 * Pretty printer.
 */
public void print(IndentedPrintStream out) {
    out.println("Code_attribute {");
    out.inc();
```

```
        out.println("u2 attribute_name_index = "
                    + Integer.toString(attributeNameIndex) + ";");
        out.println("u4 attribute_length = "
                    + Integer.toString(attributeLength) + ";");
        out.println("u2 max_stack = " + Integer.toString(maxStack) + ";");
        out.println("u2 max_locals = " + Integer.toString(maxLocals) + ";");
        out.println("u4 code_length = 0x"
                    + Integer.toHexString(code.length) + ";");
        out.println("u1 code[];");
        int count = exceptionTable.length;
        out.println("u2 exception_table_length = "
                    + Integer.toString(count) + ";");
        out.println("exception_table[] = {");
        out.inc();
        for (int i = 0; i < count; i++) {
            ExceptionHandler handler = exceptionTable[i];
            out.println("{ " + Integer.toString(handler.startPC)
                        + ", " + Integer.toString(handler.endPC)
                        + ", " + Integer.toString(handler.handlerPC)
                        + ", " + classFile.constantPool
                                    .resolveString(handler.catchType)
                        + "}");
        }
        out.dec();
        out.println("}");
        attributes.print(out);
        out.dec();
        out.println("}");
    }
    /**
     * Accessors.
```

```
     */
    public int maxLocals() {
        return maxLocals;
    }
    public int maxStack() {
        return maxStack;
    }
    public byte[] code() {
        return code;
    }
    public int codeLength() {
        return code.length;
    }
    public ExceptionHandler[] exceptionTable() {
        return exceptionTable;
    }
}
```

A.1.5.8 public class ConstantClass extends ConstantPoolItem

```
/*
 * $Id: ConstantClass.java,v 1.6 1998/12/15 14:40:35 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ConstantClass</code> represent a
 * CONSTANT_Class_info structure of which exists in classfile.
```

```
 */
public class ConstantClass extends ConstantPoolItem {
    protected int nameIndex;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public ConstantClass(ClassFileInputStream stream) throws IOException {
        super(CONSTANT_CLASS);
        nameIndex = stream.readU2();
    }

    /**
     * Constructor.  This is used to construct appropriate instance with
     * </code>index</code> into ConstantPool containing Constant_UTF8_info
     * which represent the classname.
     */
    public ConstantClass(int index) {
        super(CONSTANT_CLASS);
        nameIndex = index;
    }

    /**
     * Returns hashcode calculated from its nameIndex.
     */
    public int hashCode() {
        return nameIndex;
    }

    /**
     * Returns whether the contents are same as given </code>obj</code>.
```

```
 */
public boolean equals(Object obj) {
    if (!(obj instanceof ConstantClass))
        return false;
    return nameIndex == ((ConstantClass)obj).nameIndex;
}

int nameIndex() {
    return nameIndex;
}

/**
 * Write this into </code>stream</code> with the style of Java classfile.
 */
public void write(ClassFileOutputStream stream) throws IOException {
    stream.writeU1(tag);
    stream.writeU2(nameIndex);
}

/**
 * Return simple description of this object in a String.
 */
public String toString() {
    return "class name #" + Integer.toString(nameIndex);
}

/**
 * Pretty printer.
 */
public void print(IndentedPrintStream out) {
    out.println("CONSTANT_Class {");
```

```
        out.inc();
        out.println("u1 tag = " + Integer.toString(tag) + ";");
        out.println("u2 name_index = " + Integer.toString(nameIndex) + ";");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.9 public class ConstantDouble extends ConstantPoolItem

```
/*
 * $Id: ConstantDouble.java,v 1.5 1998/11/28 11:13:28 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ConstantDouble</code> represent a
 * CONSTANT_Double_info structure of which exists in classfile.
 */
public class ConstantDouble extends ConstantPoolItem {
    protected double value;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public ConstantDouble(ClassFileInputStream stream) throws IOException {
        super(CONSTANT_DOUBLE);
        value = stream.readDouble();
```

```
}

/**
 * Constructor.  This is used to construct appropriate instance from
 * </code>Double</code> object.
 */
public ConstantDouble(Double value) {
    super(CONSTANT_DOUBLE);
    this.value = value.doubleValue();
}

/**
 * Returns its content as </code>Double</code> object.
 */
public Double toDouble() {
    return new Double(value);
}

/**
 * Write this into </code>stream</code> with the style of Java classfile.
 */
public void write(ClassFileOutputStream stream) throws IOException {
    stream.writeU1(tag);
    stream.writeDouble(value);
}

/**
 * Return simple description of this object in a String.
 */
public String toString() {
    return Double.toString(value);
```

```
    }

    /**
     * Pretty printer.
     */
    public void print(IndentedPrintStream out) {
        out.println("CONSTANT_Double {");
        out.inc();
        out.println("u1 tag = " + Integer.toString(tag) + ";");
        out.println("u8 bytes = " + Double.toString(value) + ";");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.10 public class ConstantFieldref extends ConstantRef

```
/*
 * $Id: ConstantFieldref.java,v 1.5 1998/11/28 11:13:28 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ConstantFieldref</code> represent a
 * CONSTANT_Fieldref_info structure of which exists in classfile.
 */
public class ConstantFieldref extends ConstantRef {
    /**
```

```
 * Constructor.  This is used to construct from parts of classfile.
 */
public ConstantFieldref(ClassFileInputStream stream) throws IOException {
    super(CONSTANT_FIELD, stream);
}

/**
 * Constructor.  This is used to construct appropriate instance with
 * </code>classIndex</code> and </code>nameAndTypeIndex</code> into
 * ConstantPool.
 */
public ConstantFieldref(int classIndex, int nameAndTypeIndex) {
    super(CONSTANT_FIELD, classIndex, nameAndTypeIndex);
}

/**
 * Returns whether the contents are same as given </code>obj</code>.
 */
public boolean equals(Object obj) {
    if (!(obj instanceof ConstantFieldref))
        return false;
    ConstantFieldref item = (ConstantFieldref)obj;
    return (classIndex == item.classIndex
            && nameAndTypeIndex == item.nameAndTypeIndex);
}

/**
 * Return simple description of this object in a String.
 */
public String toString() {
    return "Fieldref class#" + Integer.toString(classIndex) + " sig#"
```

```
            + Integer.toString(nameAndTypeIndex);
    }

    /**
     * Pretty printer.
     */
    public void print(IndentedPrintStream out) {
        out.println("CONSTANT_Fieldref {");
        out.inc();
        out.println("u1 tag = " + Integer.toString(tag) + ";");
        out.println("u2 class_index = " + Integer.toString(classIndex) + ";");
        out.println("u2 name_and_type_index = "
                    + Integer.toString(nameAndTypeIndex) + ";");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.11 public class ConstantFloat extends ConstantPoolItem

```
/*
 * $Id: ConstantFloat.java,v 1.5 1998/11/28 11:13:28 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ConstantFloat</code> represent a
 * CONSTANT_Float_info structure of which exists in classfile.
```

```
 */
public class ConstantFloat extends ConstantPoolItem {
    protected float value;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public ConstantFloat(ClassFileInputStream stream) throws IOException {
        super(CONSTANT_FLOAT);
        value = stream.readFloat();
    }

    /**
     * Constructor.  This is used to construct appropriate instance from
     * </code>Float</code> object.
     */
    public ConstantFloat(Float value) {
        super(CONSTANT_FLOAT);
        this.value = value.floatValue();
    }

    /**
     * Returns its content as </code>Float</code> object.
     */
    public Float toFloat() {
        return new Float(value);
    }

    /**
     * Write this into </code>stream</code> with the style of Java classfile.
     */
```

```
    public void write(ClassFileOutputStream stream) throws IOException {
        stream.writeU1(tag);
        stream.writeFloat(value);
    }

    /**
     * Return simple description of this object in a String.
     */
    public String toString() {
        return Float.toString(value);
    }

    /**
     * Pretty printer.
     */
    public void print(IndentedPrintStream out) {
        out.println("CONSTANT_Float {");
        out.inc();
        out.println("u1 tag = " + Integer.toString(tag) + ";");
        out.println("u4 bytes = " + Float.toString(value) + ";");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.12 public class ConstantInteger extends ConstantPoolItem

```
/*
 * $Id: ConstantInteger.java,v 1.5 1998/11/28 11:13:28 maruyama Exp $
 */

package OpenJIT.frontend.classfile;
```

```
import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ConstantInteger</code> represent a
 * CONSTANT_Integer_info structure of which exists in classfile.
 */
public class ConstantInteger extends ConstantPoolItem {
    protected int value;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public ConstantInteger(ClassFileInputStream stream) throws IOException {
        super(CONSTANT_INTEGER);
        value = stream.readU4();
    }

    /**
     * Constructor.  This is used to construct appropriate instance from
     * </code>Integer</code> object.
     */
    public ConstantInteger(Integer integer) {
        super(CONSTANT_INTEGER);
        value = integer.intValue();
    }

    /**
     * Returns its content as Integer object.
     */
```

```
    public Integer toInteger() {
        return new Integer(value);
    }

    /**
     * Write this into </code>stream</code> with the style of Java classfile.
     */
    public void write(ClassFileOutputStream stream) throws IOException {
        stream.writeU1(tag);
        stream.writeU4(value);
    }

    /**
     * Return simple description of this object in a String.
     */
    public String toString() {
        return Integer.toString(value);
    }

    /**
     * Pretty printer.
     */
    public void print(IndentedPrintStream out) {
        out.println("CONSTANT_Integer {");
        out.inc();
        out.println("u1 tag = " + Integer.toString(tag) + ";");
        out.println("u4 bytes = " + Integer.toString(value) + ";");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.13 public class ConstantInterfaceMethodref extends ConstantRef

```
/*
 * $Id: ConstantInterfaceMethodref.java,v 1.6 1998/11/28 11:13:28 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ConstantInterfaceMethodref</code>
 * represent a CONSTANT_InterfaceMethodref_info structure of which
 * exists in classfile.
 */
public class ConstantInterfaceMethodref extends ConstantRef {
    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public ConstantInterfaceMethodref(ClassFileInputStream stream)
        throws IOException {
        super(CONSTANT_INTERFACEMETHOD, stream);
    }

    /**
     * Constructor.  This is used to construct appropriate instance with
     * </code>classIndex</code> and </code>nameAndTypeIndex</code> into
     * ConstantPool.
     */
    public ConstantInterfaceMethodref(int classIndex, int nameAndTypeIndex) {
        super(CONSTANT_INTERFACEMETHOD, classIndex, nameAndTypeIndex);
```

```
}

/**
 * Returns whether the contents are same as given </code>obj</code>.
 */
public boolean equals(Object obj) {
    if (!(obj instanceof ConstantInterfaceMethodref))
        return false;
    ConstantInterfaceMethodref item = (ConstantInterfaceMethodref)obj;
    return (classIndex == item.classIndex
            && nameAndTypeIndex == item.nameAndTypeIndex);
}

/**
 * Return simple description of this object in a String.
 */
public String toString() {
    return "InterfaceMethodref class#" + Integer.toString(classIndex)
        + " sig#" + Integer.toString(nameAndTypeIndex);
}

/**
 * Pretty printer.
 */
public void print(IndentedPrintStream out) {
    out.println("CONSTANT_InterfaceMethodref {");
    out.inc();
    out.println("u1 tag = " + Integer.toString(tag) + ";");
    out.println("u2 class_index = " + Integer.toString(classIndex) + ";");
    out.println("u2 name_and_type_index = "
                + Integer.toString(nameAndTypeIndex) + ";");
```

```
        out.dec();
        out.println("}");
    }
}
```

A.1.5.14 public class ConstantLong extends ConstantPoolItem

```
/*
 * $Id: ConstantLong.java,v 1.5 1998/11/28 11:13:29 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ConstantLong</code> represent a
 * CONSTANT_Long_info structure of which exists in classfile.
 */
public class ConstantLong extends ConstantPoolItem {
    protected long value;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public ConstantLong(ClassFileInputStream stream) throws IOException {
        super(CONSTANT_LONG);
        value = stream.readU8();
    }

    /**
```

```
 * Constructor.  This is used to construct appropriate instance from
 * </code>Long</code> object.
 */
public ConstantLong(Long value) {
    super(CONSTANT_LONG);
    this.value = value.longValue();
}

/**
 * Returns its content as </code>Long</code> object.
 */
public Long toLong() {
    return new Long(value);
}

/**
 * Write this into </code>stream</code> with the style of Java classfile.
 */
public void write(ClassFileOutputStream stream) throws IOException {
    stream.writeU1(tag);
    stream.writeU8(value);
}

/**
 * Return simple description of this object in a String.
 */
public String toString() {
    return Long.toString(value);
}

/**
```

```
     * Pretty printer.
     */
    public void print(IndentedPrintStream out) {
        out.println("CONSTANT_Long {");
        out.inc();
        out.println("u1 tag = " + Integer.toString(tag) + ";");
        out.println("u8 bytes = " + Long.toString(value) + ";");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.15 public class ConstantMethodref extends ConstantRef

```
/*
 * $Id: ConstantMethodref.java,v 1.6 1998/11/28 11:13:29 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ConstantMethodref</code> represent a
 * CONSTANT_Methodref_info structure of which exists in classfile.
 */
public class ConstantMethodref extends ConstantRef {
    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public ConstantMethodref(ClassFileInputStream stream) throws IOException {
```

```
    super(CONSTANT_METHOD, stream);
}

/**
 * Constructor.  This is used to construct appropriate instance with
 * </code>classIndex</code> and </code>nameAndTypeIndex</code> into
 * ConstantPool.
 */
public ConstantMethodref(int classIndex, int nameAndTypeIndex) {
    super(CONSTANT_METHOD, classIndex, nameAndTypeIndex);
}

/**
 * Returns whether the contents are same as given </code>obj</code>.
 */
public boolean equals(Object obj) {
    if (!(obj instanceof ConstantMethodref))
        return false;
    ConstantMethodref item = (ConstantMethodref)obj;
    return (classIndex == item.classIndex
            && nameAndTypeIndex == item.nameAndTypeIndex);
}

/**
 * Return simple description of this object in a String.
 */
public String toString() {
    return "Methodref class#" + Integer.toString(classIndex) + " sig#"
        + Integer.toString(nameAndTypeIndex);
}
```

```
    /**
     * Pretty printer.
     */
    public void print(IndentedPrintStream out) {
        out.println("CONSTANT_Methodref {");
        out.inc();
        out.println("u1 tag = " + Integer.toString(tag) + ";");
        out.println("u2 class_index = " + Integer.toString(classIndex) + ";");
        out.println("u2 name_and_type_index = "
                    + Integer.toString(nameAndTypeIndex) + ";");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.16 public class ConstantNameAndType extends ConstantPoolItem

```
/*
 * $Id: ConstantNameAndType.java,v 1.5 1998/11/28 11:13:29 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ConstantNameAndType</code> represent
 * a CONSTANT_NameAndType_info structure of which exists in classfile.
 */
public class ConstantNameAndType extends ConstantPoolItem {
    protected int nameIndex;
```

```
protected int descriptorIndex;

/**
 * Constructor.  This is used to construct from parts of classfile.
 */
public ConstantNameAndType(ClassFileInputStream stream)
    throws IOException {
    super(CONSTANT_NAMEANDTYPE);
    nameIndex = stream.readU2();
    descriptorIndex = stream.readU2();
}

/**
 * Constructor.  This is used to construct appropriate instance with
 * </code>nameIndex</code> and </code>typeIndex</code> into
 * ConstantPool.
 */
public ConstantNameAndType(int nameIndex, int typeIndex) {
    super(CONSTANT_NAMEANDTYPE);
    this.nameIndex = nameIndex;
    descriptorIndex = typeIndex;
}

/**
 * Returns its hashcode calculated from nameIndex and descriptorIndex.
 */
public int hashCode() {
    return (nameIndex & 0xff) | ((descriptorIndex & 0xff) << 8)
        | ((nameIndex & 0xff00) << 8) | ((descriptorIndex & 0xff00) << 16);
}
```

```
/**
 * Returns whether the contents are same as given </code>obj</code>.
 */
public boolean equals(Object obj) {
    if (!(obj instanceof ConstantNameAndType))
        return false;
    ConstantNameAndType item = (ConstantNameAndType)obj;
    return (nameIndex == item.nameIndex
            && descriptorIndex == item.descriptorIndex);
}

/**
 * Write this into </code>stream</code> with the style of Java classfile.
 */
public void write(ClassFileOutputStream stream) throws IOException {
    stream.writeU1(tag);
    stream.writeU2(nameIndex);
    stream.writeU2(descriptorIndex);
}

/**
 * Return simple description of this object in a String.
 */
public String toString() {
    return "NameAndType name#" + Integer.toString(nameIndex)
        + " descriptor#" + Integer.toString(descriptorIndex);
}

/**
 * Pretty printer.
 */
```

```
    public void print(IndentedPrintStream out) {
        out.println("CONSTANT_NameAndType {");
        out.inc();
        out.println("u1 tag = " + Integer.toString(tag) + ";");
        out.println("u2 name_index = " + Integer.toString(nameIndex) + ";");
        out.println("u2 descriptor_index = "
                    + Integer.toString(descriptorIndex) + ";");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.17 public class ConstantPool implements RuntimeConstants

```
/*
 * $Id: ConstantPool.java,v 1.8 1998/12/15 14:40:35 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.java.RuntimeConstants;
import OpenJIT.frontend.util.IndentedPrintStream;
import OpenJIT.frontend.util.LookupHashtable;
import java.io.IOException;
import java.util.Vector;
import java.util.Hashtable;

/**
 * Each instances of the class </code>ConstantPool</code> represent a
 * set of </code>ConstantPoolItem</code> objects.  Each subclass of
 * ConstantPoolItem is used to represent some constant value for
 * classfile itself and bytecode.
```

```
 */
public class ConstantPool implements RuntimeConstants {
    /**
     * Factory methods for various kind of </code>ConstantPoolItem</code>
     */
    protected ConstantPoolItem readUTF8(ClassFileInputStream stream)
        throws IOException {
        return new ConstantUTF8(stream);
    }
    protected ConstantPoolItem makeUTF8(String value) {
        return new ConstantUTF8(value);
    }

    protected ConstantPoolItem readUnicode(ClassFileInputStream stream)
        throws IOException {
        throw new UnknownFileException("CONSTANT_Unicode appears");
    }
    protected ConstantPoolItem makeUnicode(String value) {
        throw new UnknownFileException("CONSTANT_Unicode appears");
    }

    protected ConstantPoolItem readInteger(ClassFileInputStream stream)
        throws IOException {
        return new ConstantInteger(stream);
    }
    protected ConstantPoolItem makeInteger(Integer value) {
        return new ConstantInteger(value);
    }

    protected ConstantPoolItem readFloat(ClassFileInputStream stream)
        throws IOException {
```

```
        return new ConstantFloat(stream);
    }
    protected ConstantPoolItem makeFloat(Float value) {
        return new ConstantFloat(value);
    }

    protected ConstantPoolItem readLong(ClassFileInputStream stream)
        throws IOException {
        return new ConstantLong(stream);
    }
    protected ConstantPoolItem makeLong(Long value) {
        return new ConstantLong(value);
    }

    protected ConstantPoolItem readDouble(ClassFileInputStream stream)
        throws IOException {
        return new ConstantDouble(stream);
    }
    protected ConstantPoolItem makeDouble(Double value) {
        return new ConstantDouble(value);
    }

    protected ConstantPoolItem readClass(ClassFileInputStream stream)
        throws IOException {
        return new ConstantClass(stream);
    }
    protected ConstantPoolItem makeClass(int nameIndex) {
        return new ConstantClass(nameIndex);
    }

    protected ConstantPoolItem readString(ClassFileInputStream stream)
```

```
    throws IOException {
    return new ConstantString(stream);
}
protected ConstantPoolItem makeString(int stringIndex) {
    return new ConstantString(stringIndex);
}

protected ConstantPoolItem readField(ClassFileInputStream stream)
    throws IOException {
    return new ConstantFieldref(stream);
}
protected ConstantPoolItem makeField(int classIndex, int descIndex) {
    return new ConstantFieldref(classIndex, descIndex);
}

protected ConstantPoolItem readMethod(ClassFileInputStream stream)
    throws IOException {
    return new ConstantMethodref(stream);
}
protected ConstantPoolItem makeMethod(int classIndex, int descIndex) {
    return new ConstantMethodref(classIndex, descIndex);
}

protected ConstantPoolItem readInterface(ClassFileInputStream stream)
    throws IOException {
    return new ConstantInterfaceMethodref(stream);
}
protected ConstantPoolItem makeInterface(int classIndex, int descIndex) {
    return new ConstantInterfaceMethodref(classIndex, descIndex);
}
```

```
protected ConstantPoolItem readNameAndType(ClassFileInputStream stream)
    throws IOException {
    return new ConstantNameAndType(stream);
}
protected ConstantPoolItem makeNameAndType(int nameIndex, int descIndex) {
    return new ConstantNameAndType(nameIndex, descIndex);
}

/**
 * </code>constantItems</code> contains a set of ConstantPoolItems
 * in variable length array.
 */
protected Vector constantItems;

/**
 * These are for guarantee that each ConstantPoolItem are the unique
 * object in one ConstantPool.
 */
private LookupHashtable utf8s = new LookupHashtable() {
    protected Object resolve(Object key) {
        ConstantPoolItem item = makeUTF8((String)key);
        int index;
        synchronized (constantItems) {
            index = constantItems.size();
            constantItems.addElement(item);
        }
        return new Integer(index);
    }
};

private LookupHashtable integers = new LookupHashtable() {
```

```
    protected Object resolve(Object key) {
        ConstantPoolItem item = makeInteger((Integer)key);
        int index;
        synchronized (constantItems) {
            index = constantItems.size();
            constantItems.addElement(item);
        }
        return new Integer(index);
    }
};

private LookupHashtable floats = new LookupHashtable() {
    protected Object resolve(Object key) {
        ConstantPoolItem item = makeFloat((Float)key);
        int index;
        synchronized (constantItems) {
            index = constantItems.size();
            constantItems.addElement(item);
        }
        return new Integer(index);
    }
};

private LookupHashtable longs = new LookupHashtable() {
    protected Object resolve(Object key) {
        ConstantPoolItem item = makeLong((Long)key);
        int index;
        synchronized (constantItems) {
            index = constantItems.size();
            constantItems.addElement(item);
            constantItems.addElement(null);
```

```
        }
        return new Integer(index);
    }
};

private LookupHashtable doubles = new LookupHashtable() {
    protected Object resolve(Object key) {
        ConstantPoolItem item = makeDouble((Double)key);
        int index;
        synchronized (constantItems) {
            index = constantItems.size();
            constantItems.addElement(item);
            constantItems.addElement(null);
        }
        return new Integer(index);
    }
};

private LookupHashtable classes = new LookupHashtable() {
    protected Object resolve(Object key) {
        ConstantPoolItem item = (ConstantClass)key;
        int index;
        synchronized (constantItems) {
            index = constantItems.size();
            constantItems.addElement(key);
        }
        return new Integer(index);
    }
};

private LookupHashtable strings = new LookupHashtable() {
```

```
    protected Object resolve(Object key) {
        ConstantPoolItem item = (ConstantString)key;
        int index;
        synchronized (constantItems) {
            index = constantItems.size();
            constantItems.addElement(key);
        }
        return new Integer(index);
    }
};

private LookupHashtable fieldrefs = new LookupHashtable() {
    protected Object resolve(Object key) {
        ConstantPoolItem item = (ConstantFieldref)key;
        int index;
        synchronized (constantItems) {
            index = constantItems.size();
            constantItems.addElement(key);
        }
        return new Integer(index);
    }
};

private LookupHashtable methodrefs = new LookupHashtable() {
    protected Object resolve(Object key) {
        ConstantPoolItem item = (ConstantMethodref)key;
        int index;
        synchronized (constantItems) {
            index = constantItems.size();
            constantItems.addElement(key);
        }
```

```
        return new Integer(index);
    }
};

private LookupHashtable interfacerefs = new LookupHashtable() {
    protected Object resolve(Object key) {
        ConstantPoolItem item = (ConstantInterfaceMethodref)key;
        int index;
        synchronized (constantItems) {
            index = constantItems.size();
            constantItems.addElement(key);
        }
        return new Integer(index);
    }
};

private LookupHashtable nameAndTypes = new LookupHashtable() {
    protected Object resolve(Object key) {
        ConstantPoolItem item = (ConstantNameAndType)key;
        int index;
        synchronized (constantItems) {
            index = constantItems.size();
            constantItems.addElement(key);
        }
        return new Integer(index);
    }
};

/**
 * Constructor.  This reads bytestream from </code>stream</code> and
 * initialize constantItems.
```

```
 */
public ConstantPool(ClassFileInputStream stream) throws IOException {
    constantItems = new Vector();
    constantItems.addElement(null);
    int count = stream.readU2();
    for (int i = 1; i < count; i++) {
        int tag = stream.readU1();
        ConstantPoolItem item;
        switch (tag) {
        case CONSTANT_UTF8:
            item = readUTF8(stream);
            utf8s.put(item.toString(), new Integer(i));
            constantItems.addElement(item);
            break;
            /*
             * case CONSTANT_UNICODE:
             *     item = readUnicode(stream);
             *     unicodes.put(item, new Integer(i));
             *     constantItems.addElement(item);
             *     break;
             */
        case CONSTANT_INTEGER:
            item = readInteger(stream);
            integers.put(((ConstantInteger)item).toInteger(),
                         new Integer(i));
            constantItems.addElement(item);
            break;
        case CONSTANT_FLOAT:
            item = readFloat(stream);
            floats.put(((ConstantFloat)item).toFloat(), new Integer(i));
            constantItems.addElement(item);
```

```
            break;
        case CONSTANT_LONG:
            item = readLong(stream);
            longs.put(((ConstantLong)item).toLong(), new Integer(i));
            constantItems.addElement(item);
            constantItems.addElement(null);
            i++;
            break;
        case CONSTANT_DOUBLE:
            item = readDouble(stream);
            doubles.put(((ConstantDouble)item).toDouble(), new Integer(i));
            constantItems.addElement(item);
            constantItems.addElement(null);
            i++;
            break;
        case CONSTANT_CLASS:
            item = readClass(stream);
            classes.put(item, new Integer(i));
            constantItems.addElement(item);
            break;
        case CONSTANT_STRING:
            item = readString(stream);
            strings.put(item, new Integer(i));
            constantItems.addElement(item);
            break;
        case CONSTANT_FIELD:
            item = readField(stream);
            fieldrefs.put(item, new Integer(i));
            constantItems.addElement(item);
            break;
        case CONSTANT_METHOD:
```

```
                item = readMethod(stream);
                methodrefs.put(item, new Integer(i));
                constantItems.addElement(item);
                break;
            case CONSTANT_INTERFACEMETHOD:
                item = readInterface(stream);
                interfacerefs.put(item, new Integer(i));
                constantItems.addElement(item);
                break;
            case CONSTANT_NAMEANDTYPE:
                item = readNameAndType(stream);
                nameAndTypes.put(item, new Integer(i));
                constantItems.addElement(item);
                break;
            default:
                throw new UnknownFileException("Unknown tag ("
                                               + Integer.toString(tag)
                                               + ") appears");
            }
        }
    }

    /**
     * Write this into </code>stream</code> with the style of Java classfile.
     */
    public void write(ClassFileOutputStream stream) throws IOException {
        int count = constantItems.size();
        stream.writeU2(count);
        for (int i = 1; i < count; i++) {
            ConstantPoolItem d = (ConstantPoolItem)constantItems.elementAt(i);
            if (d != null)
```

```
            d.write(stream);
    }
}

/**
 * Accessor.
 */
public ConstantPoolItem itemAt(int index) {
    return (ConstantPoolItem)constantItems.elementAt(index);
}

/**
 * Pretty printer.
 */
public void print(IndentedPrintStream out) {
    int count = constantItems.size();
    out.println("u2 constant_pool_count = "
                + Integer.toString(count) + ";");
    out.println("cp_info constant_pool[] = {");
    out.inc();
    for (int i = 1; i < count; i++) {
        ConstantPoolItem item
            = (ConstantPoolItem)constantItems.elementAt(i);
        out.println("// [" + Integer.toString(i) + "]");
        if (item == null) {
            out.println("null");
        } else
            item.print(out);
    }
    out.dec();
    out.println("}");
```

```
}

/**
 * Returns a String object appropriate for given </code>index</code>.
 */
public String resolveString(int index) {
    ConstantPoolItem item
        = (ConstantPoolItem)constantItems.elementAt(index);
    if (item.isString())
        return resolveString(((ConstantString)item).stringIndex);
    if (item.isClass())
        return resolveString(((ConstantClass)item).nameIndex);
    return item.toString();
}

/**
 * Returns the class name for given </code>index</code> as String object.
 */
public String resolveClassName(int index) {
    ConstantPoolItem item
        = (ConstantPoolItem)constantItems.elementAt(index);
    if (item.isClass())
        return resolveString(((ConstantClass)item).nameIndex);
    if (item.isField() || item.isMethod() || item.isInterface())
        return resolveString(((ConstantRef)item).classIndex);
    throw new ResolveError("Invalid use of resolveClassName()");
}

/**
 * Returns the name of member for given </code>index</code>
 * as String object .
```

```
 */
public String resolveMemberName(int index) {
    ConstantPoolItem item
        = (ConstantPoolItem)constantItems.elementAt(index);
    if (item.isField() || item.isMethod() || item.isInterface()) {
        item = (ConstantPoolItem)constantItems
            .elementAt(((ConstantRef)item).nameAndTypeIndex);
        return resolveString(((ConstantNameAndType)item).nameIndex);
    }
    throw new ResolveError("Invalid use of resolveClassName()");
}

/**
 * Returns the descriptor of member for given </code>index</code>
 * as String object.
 */
public byte[] resolveMemberDescriptor(int index) {
    ConstantPoolItem item
        = (ConstantPoolItem)constantItems.elementAt(index);
    if (item.isField() || item.isMethod() || item.isInterface()) {
        item = (ConstantPoolItem)constantItems
            .elementAt(((ConstantRef)item).nameAndTypeIndex);
        return ((ConstantUTF8)
                constantItems.elementAt(((ConstantNameAndType)item)
                                        .descriptorIndex)).bytes;
    }
    throw new ResolveError("Invalid use of resolveClassName()");
}

/**
 * Returns an appropriate int value for given </code>index</code>
```

```
 * to CONSTANT_Integer.
 */
public int resolveInt(int index) {
    ConstantPoolItem item
        = (ConstantPoolItem)constantItems.elementAt(index);
    if (item.isInteger()) {
        return ((ConstantInteger)item).value;
    }
    throw new ResolveError("Invalid use of resolveClassName()");
}

/**
 * Returns an appropriate float value for given </code>index</code>
 * to CONSTANT_Float.
 */
public float resolveFloat(int index) {
    ConstantPoolItem item
        = (ConstantPoolItem)constantItems.elementAt(index);
    if (item.isFloat()) {
        return ((ConstantFloat)item).value;
    }
    throw new ResolveError("Invalid use of resolveClassName()");
}

/**
 * Returns an appropriate double value for given </code>index</code>
 * to CONSTANT_Double.
 */
public double resolveDouble(int index) {
    ConstantPoolItem item
        = (ConstantPoolItem)constantItems.elementAt(index);
```

```
    if (item.isDouble()) {
        return ((ConstantDouble)item).value;
    }
    throw new ResolveError("Invalid use of resolveClassName()");
}

/**
 * Returns an appropriate long value for given </code>index</code>
 * to CONSTANT_Long.
 */
public long resolveLong(int index) {
    ConstantPoolItem item
        = (ConstantPoolItem)constantItems.elementAt(index);
    if (item.isLong()) {
        return ((ConstantLong)item).value;
    }
    throw new ResolveError("Invalid use of resolveClassName()");
}

/**
 * Returns ConstantPoolItem's index into appropriate item for
 * given constant.
 */
public int lookupUTF8(String value) {
    return ((Integer)utf8s.lookup(value)).intValue();
}

public int lookupInt(int value) {
    return ((Integer)integers.lookup(new Integer(value))).intValue();
}
```

```
public int lookupFloat(float value) {
    return ((Integer)floats.lookup(new Float(value))).intValue();
}

public int lookupLong(long value) {
    return ((Integer)longs.lookup(new Long(value))).intValue();
}

public int lookupDouble(double value) {
    return ((Integer)doubles.lookup(new Double(value))).intValue();
}

public int lookupClass(String name) {
    return ((Integer)classes.lookup(makeClass(lookupUTF8(name))))
        .intValue();
}

public int lookupString(String value) {
    return ((Integer)strings.lookup(makeString(lookupUTF8(value))))
        .intValue();
}

public int lookupNameAndType(String name, String type) {
    return ((Integer)nameAndTypes
            .lookup(makeNameAndType(lookupUTF8(name), lookupUTF8(type))))
        .intValue();
}

public int lookupFieldref(String className, String name, String type) {
    return ((Integer)fieldrefs
            .lookup(makeField(lookupClass(className),
```

```
                                lookupNameAndType(name, type)))).intValue();
    }

    public int lookupMethodref(String className, String name, String type) {
        return ((Integer)methodrefs
                .lookup(makeMethod(lookupClass(className),
                                   lookupNameAndType(name, type)))).intValue();
    }

    public int lookupInterfaceMethodref(String className, String name,
                                        String type) {
        return ((Integer)interfacerefs
                .lookup(makeInterface(lookupClass(className),
                                      lookupNameAndType(name, type))))
            .intValue();
    }
}
```

A.1.5.18 public abstract class ConstantPoolItem implements RuntimeConstants

```
/*
 * $Id: ConstantPoolItem.java,v 1.6 1998/12/15 14:40:35 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.java.RuntimeConstants;
import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
```

```
 * Each instances of the subclass of class </code>ConstantPoolItem</code>
 * represent a CONSTANT_*_info structure of which exists in classfile.
 */
public abstract class ConstantPoolItem implements RuntimeConstants {
    protected int tag;

    /**
     * Constructor.
     */
    protected ConstantPoolItem(int tag) {
        this.tag = tag;
    }

    /**
     * Each subclasses of </code>ConstantPoolItem</code> must be override
     * these methods.
     */
    abstract public void write(ClassFileOutputStream stream)
        throws IOException;
    abstract public String toString();
    abstract public void print(IndentedPrintStream out);

    /**
     * For decision of the kind of an subclass's object.
     */
    public final boolean isUTF8() {
        return tag == CONSTANT_UTF8;
    }
    public final boolean isUnicode() {
        return tag == CONSTANT_UNICODE;
    }
```

```
public final boolean isInteger() {
    return tag == CONSTANT_INTEGER;
}
public final boolean isFloat() {
    return tag == CONSTANT_FLOAT;
}
public final boolean isLong() {
    return tag == CONSTANT_LONG;
}
public final boolean isDouble() {
    return tag == CONSTANT_DOUBLE;
}
public final boolean isClass() {
    return tag == CONSTANT_CLASS;
}
public final boolean isString() {
    return tag == CONSTANT_STRING;
}
public final boolean isField() {
    return tag == CONSTANT_FIELD;
}
public final boolean isMethod() {
    return tag == CONSTANT_METHOD;
}
public final boolean isInterface() {
    return tag == CONSTANT_INTERFACEMETHOD;
}
public final boolean isNameAndType() {
    return tag == CONSTANT_NAMEANDTYPE;
}
public final int tag() {
```

```
        return tag;
    }
}
```

A.1.5.19 public abstract class ConstantRef extends ConstantPoolItem

```
/*
 * $Id: ConstantRef.java,v 1.3 1998/11/28 11:13:29 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import java.io.IOException;

/**
 * Each instances of the subclass of class </code>ConstantRef</code>
 * represent a CONSTANT_*ref_info structure of which exists in classfile.
 * The only differences between those subclasses are thier tags.
 */
public abstract class ConstantRef extends ConstantPoolItem {
    protected int classIndex;
    protected int nameAndTypeIndex;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    protected ConstantRef(int tag, ClassFileInputStream stream)
        throws IOException {
        super(tag);
        classIndex = stream.readU2();
        nameAndTypeIndex = stream.readU2();
    }
```

```
    /**
     * Constructor.  This is used to construct appropriate instance with
     * </code>tag</code>, </code>classIndex</code> and
     * </code>nameAndTypeIndex</code>.
     */
    protected ConstantRef(int tag, int classIndex, int nameAndTypeIndex) {
        super(tag);
        this.classIndex = classIndex;
        this.nameAndTypeIndex = nameAndTypeIndex;
    }

    /**
     * Returns its hashcode calculated from classIndex and nameAndTypeIndex.
     */
    public int hashCode() {
        return (classIndex & 0xff) | ((nameAndTypeIndex & 0xff) << 8) |
            ((classIndex & 0xff00) << 8) | ((nameAndTypeIndex & 0xff00) << 16);
    }

    /**
     * Write this into </code>stream</code> with the style of Java classfile.
     */
    public void write(ClassFileOutputStream stream) throws IOException {
        stream.writeU1(tag);
        stream.writeU2(classIndex);
        stream.writeU2(nameAndTypeIndex);
    }
}
```

A.1.5.20 public class ConstantString extends ConstantPoolItem

```
/*
 * $Id: ConstantString.java,v 1.5 1998/11/28 11:13:29 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ConstantString</code> represent a
 * CONSTANT_String_info structure of which exists in classfile.
 */
public class ConstantString extends ConstantPoolItem {
    protected int stringIndex;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public ConstantString(ClassFileInputStream stream) throws IOException {
        super(CONSTANT_STRING);
        stringIndex = stream.readU2();
    }

    /**
     * Constructor.  This is used to construct appropriate instance with
     * </code>index</code> into ConstantPool containing Constant_UTF8_info
     * which represent the contents of a </code>String</code> object.
     */
    public ConstantString(int index) {
        super(CONSTANT_STRING);
```

```
    stringIndex = index;
}

/**
 * Returns hashcode calculated from its nameIndex.
 */
public int hashCode() {
    return stringIndex;
}

/**
 * Returns whether the contents are same as given </code>obj</code>.
 */
public boolean equals(Object obj) {
    if (!(obj instanceof ConstantString))
        return false;
    return stringIndex == ((ConstantString)obj).stringIndex;
}

int stringIndex() {
    return stringIndex;
}

/**
 * Write this into </code>stream</code> with the style of Java classfile.
 */
public void write(ClassFileOutputStream stream) throws IOException {
    stream.writeU1(tag);
    stream.writeU2(stringIndex);
}
```

```
    /**
     * Return simple description of this object in a String.
     */
    public String toString() {
        return "string at #" + Integer.toString(stringIndex);
    }

    /**
     * Pretty printer.
     */
    public void print(IndentedPrintStream out) {
        out.println("CONSTANT_String {");
        out.inc();
        out.println("u1 tag = " + Integer.toString(tag) + ";");
        out.println("u2 string_index = "
                    + Integer.toString(stringIndex) + ";");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.21 public class ConstantUTF8 extends ConstantPoolItem

```
/*
 * $Id: ConstantUTF8.java,v 1.6 1998/12/20 18:45:22 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;
import java.io.UnsupportedEncodingException;
```

```
/**
 * Each instances of the class </code>ConstantUTF8</code> represent a
 * CONSTANT_UTF8_info structure of which exists in classfile.
 */
public class ConstantUTF8 extends ConstantPoolItem {
    protected byte[] bytes;
    protected String string;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public ConstantUTF8(ClassFileInputStream stream) throws IOException {
        super(CONSTANT_UTF8);
        int length = stream.readU2();
        bytes = stream.readBytes(length);
    }

    /**
     * Constructor.  This is used to construct appropriate instance from
     * </code>String</code> object.
     */
    public ConstantUTF8(String string) {
        super(CONSTANT_UTF8);
        try {
            bytes = string.getBytes("UTF8");
        } catch (UnsupportedEncodingException e) {
            throw new Error("Why UTF8 cannot use");
        }
    }
```

```
/**
 * Write this into </code>stream</code> with the style of Java classfile.
 */
public void write(ClassFileOutputStream stream) throws IOException {
    stream.writeU1(tag);
    stream.writeU2(bytes.length);
    stream.writeBytes(bytes);
}

/**
 * Return simple description of this object in a String.
 */
public synchronized String toString() {
    if (string != null)
        return string;
    try {
        string = new String(bytes, "UTF8");
    } catch (UnsupportedEncodingException e) {
        throw new Error("Why UTF8 cannot use");
    }
    return string;
}

/**
 * Pretty printer.
 */
public void print(IndentedPrintStream out) {
    out.println("CONSTANT_Utf8 {");
    out.inc();
    out.println("u1 tag = " + Integer.toString(tag) + ";");
    out.println("u2 length = "
```

```
                    + Integer.toString(bytes.length) + ";");
        out.println("u1 bytes[] = \"" + toString() + "\";");
        out.dec();
        out.println("}");
    }

    /**
     * Returns hashCode calculated from bytes.
     */
    public int hashCode() {
        return toString().hashCode();
    }

    public byte[] bytes() {
        return bytes;
    }
}
```

A.1.5.22 public class ConstantValueAttribute extends AttributeInfo

```
/*
 * $Id: ConstantValueAttribute.java,v 1.7 1998/12/20 18:45:22 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ConstantValueAttribute</code> represent
 * that field containing it is a constant field.
```

```
 */
public class ConstantValueAttribute extends AttributeInfo {
    /**
     * It holds an index into ConstantPool that represent the constant value.
     */
    protected int constantValueIndex;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public ConstantValueAttribute(int nameIndex, ClassFileInputStream stream,
                                  ClassFile cF) throws IOException {
        super(nameIndex, stream, cF);
        constantValueIndex = stream.readU2();
    }

    /**
     * Write this into </code>stream</code> with the style of Java classfile.
     */
    public void write(ClassFileOutputStream stream) throws IOException {
        super.write(stream);
        stream.writeU2(constantValueIndex);
    }

    /**
     * Return simple description of this object in a String.
     */
    public String toString() {
        return "ConstantValue_attribute { " + super.toString()
            + ", constantvalue_index = " + Integer.toString(constantValueIndex)
            + " }";
```

```
    }

    /**
     * Pretty printer.
     */
    public void print(IndentedPrintStream out) {
        out.println("ConstantValue_attribute {");
        out.inc();
        out.println("u2 attribute_name_index = "
                    + Integer.toString(attributeNameIndex) + ";");
        out.println("u4 attribute_length = "
                    + Integer.toString(attributeLength) + ";");
        out.println("u2 constantvalue_index = "
                    + Integer.toString(constantValueIndex) + "; // "
                    + classFile.constantPool
                          .resolveString(constantValueIndex));
        out.dec();
        out.println("}");
    }

    public int constantValueIndex() {
        return constantValueIndex;
    }
}
```

A.1.5.23 public class ExceptionsAttribute extends AttributeInfo

```
/*
 * $Id: ExceptionsAttribute.java,v 1.6 1998/12/15 14:40:36 maruyama Exp $
 */
```

```
package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>ExceptionsAttribute</code> represent
 * a set of exceptions that method containing it may throw.
 */
public class ExceptionsAttribute extends AttributeInfo {
    /**
     * Each content of </code>exceptionIndexTable[]</code> is an index
     * into ConstantPool that represent a Class of the exception.
     */
    protected int exceptionIndexTable[];

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public ExceptionsAttribute(int nameIndex, ClassFileInputStream stream,
                               ClassFile cF) throws IOException {
        super(nameIndex, stream, cF);
        int length = stream.readU2();
        exceptionIndexTable = new int[length];
        for (int i = 0; i < length; i++)
            exceptionIndexTable[i] = stream.readU2();
    }

    /**
     * Write this into </code>stream</code> with the style of Java classfile.
     */
```

```
public void write(ClassFileOutputStream stream) throws IOException {
    super.write(stream);
    int length = exceptionIndexTable.length;
    stream.writeU2(length);
    for (int i = 0; i < length; i++)
        stream.writeU2(exceptionIndexTable[i]);
}

/**
 * Return simple description of this object in a String.
 */
public String toString() {
    return "Exceptions_attribute { " + super.toString()
        + ", number_of_exceptions = "
        + Integer.toString(exceptionIndexTable.length)
        + ", exception_index_table[] }";
}

/**
 * Pretty printer.
 */
public void print(IndentedPrintStream out) {
    out.println("Exceptions_attribute {");
    out.inc();
    out.println("u2 attribute_name_index = "
                + Integer.toString(attributeNameIndex) + ";");
    out.println("u4 attribute_length = "
                + Integer.toString(attributeLength) + ";");
    int count = exceptionIndexTable.length;
    out.println("u2 number_of_exceptions = "
                + Integer.toString(count) + ";");
```

```
        if (count > 0) {
            StringBuffer buf = new StringBuffer();
            buf.append("exception_index_table[] = { ");
            buf.append(classFile.constantPool
                       .resolveString(exceptionIndexTable[0]));
            for (int i = 1; i < count; i++) {
                buf.append(", ");
                buf.append(classFile.constantPool
                           .resolveString(exceptionIndexTable[i]));
            }
            buf.append(" };");
            out.println(buf.toString());
        } else
            out.println("exception_index_table[];");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.24 public class FieldInfo extends MemberInfo

```
/*
 * $Id: FieldInfo.java,v 1.6 1998/12/01 20:27:48 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import java.io.IOException;

/**
 * Each instances of the class </code>FieldInfo</code> represent a
 * set of informations for a field contained in a classfile.
```

```
 * Its contents are just same as methods' information, so its are
 * treated as </code>MemberInfo</code> -- superclass of FieldInfo.
 */
public class FieldInfo extends MemberInfo {
    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public FieldInfo(ClassFileInputStream stream, ClassFile cF)
        throws IOException {
        super(stream, cF);
    }

    /**
     * Return simple description of this object in a String.
     */
    public String toString() {
        return "field" + super.toString();
    }

    /**
     * used by MethodInfo for pretty printing.
     */
    public String header() {
        return "field_info";
    }
}
```

A.1.5.25 public class GenericAttribute extends AttributeInfo

```
/*
 * $Id: GenericAttribute.java,v 1.4 1998/11/26 12:36:09 maruyama Exp $
 */
```

```
package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

public class GenericAttribute extends AttributeInfo {
    protected byte info[];

    public GenericAttribute(int nameIndex, ClassFileInputStream stream,
                            ClassFile cF) throws IOException {
        super(nameIndex, stream, cF);
        info = stream.readBytes(attributeLength);
    }

    public void write(ClassFileOutputStream stream) throws IOException {
        super.write(stream);
        int length = info.length;
        for (int i = 0; i < length; i++)
            stream.writeU1(info[i]);
    }

    public String toString() {
        return "attribute_info { " + super.toString()
            + ", info[] }";
    }

    public void print(IndentedPrintStream out) {
        out.println("Generic_attribute {");
        out.inc();
        out.println("u2 attribute_name_index = "
```

```
                    + Integer.toString(attributeNameIndex) + ";");
        out.println("u4 attribute_length = "
                    + Integer.toString(attributeLength) + ";");
        out.println("u1 info[];");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.26 public class LineNumber

```
/*
 * $Id: LineNumber.java,v 1.3 1998/12/01 20:27:48 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>LineNumber</code> represent a set
 * of informations of line number.
 */
public class LineNumber {
    protected int startPC;
    protected int lineNumber;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public LineNumber(ClassFileInputStream stream) throws IOException {
```

```
        startPC = stream.readU2();
        lineNumber = stream.readU2();
    }

    /**
     * Write this into </code>stream</code> with the style of Java classfile.
     */
    public void write(ClassFileOutputStream stream) throws IOException {
        stream.writeU2(startPC);
        stream.writeU2(lineNumber);
    }

    /**
     * Return simple description of this object in a String.
     */
    public String toString() {
        return "line_number { " + Integer.toString(startPC)
            + ", " + Integer.toString(lineNumber) + " }";
    }

    /**
     * Pretty printer.
     */
    public void print(IndentedPrintStream out) {
        out.println(toString());
    }
}
```

A.1.5.27 public class LineNumberTableAttribute extends AttributeInfo

```
/*
 * $Id: LineNumberTableAttribute.java,v 1.4 1998/12/01 20:27:48 maruyama Exp $
```

```
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>LineNumberTableAttribute</code> represent
 * a set of informations for debugging purpose -- line number.
 */
public class LineNumberTableAttribute extends AttributeInfo {
    protected LineNumber lineNumberTable[];

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public LineNumberTableAttribute(int nameIndex, ClassFileInputStream stream,
                                    ClassFile cF) throws IOException {
        super(nameIndex, stream, cF);
        int count = stream.readU2();
        lineNumberTable = new LineNumber[count];
        for (int i = 0; i < count; i++)
            lineNumberTable[i] = new LineNumber(stream);
    }

    /**
     * Write this into </code>stream</code> with the style of Java classfile.
     */
    public void write(ClassFileOutputStream stream) throws IOException {
        super.write(stream);
```

```
    int count = lineNumberTable.length;
    stream.writeU2(count);
    for (int i = 0; i < count; i++)
        lineNumberTable[i].write(stream);
}

/**
 * Return simple description of this object in a String.
 */
public String toString() {
    return "LineNumberTable_attribute { " + super.toString()
        + ", line_number_table_length = "
        + Integer.toString(lineNumberTable.length)
        + ", line_number_table[] }";
}

/**
 * Pretty printer.
 */
public void print(IndentedPrintStream out) {
    out.println("LineNumberTable_attribute {");
    out.inc();
    out.println("u2 attribute_name_index = "
                + Integer.toString(attributeNameIndex) + ";");
    out.println("u4 attribute_length = "
                + Integer.toString(attributeLength) + ";");
    int count = lineNumberTable.length;
    out.println("u2 line_number_table_length = "
                + Integer.toString(count) + ";");
    if (count > 0) {
        out.println("line_number_table[] = {");
```

```
            out.inc();
            for (int i = 0; i < count; i++)
                lineNumberTable[i].print(out);
            out.dec();
            out.println("}");
        } else
            out.println("line_number_table[];");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.28 public class LocalVariable

```
/*
 * $Id: LocalVariable.java,v 1.5 1998/12/15 14:40:36 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>LocalVariable</code> represent a set
 * of informations of local variable.
 */
public class LocalVariable {
    protected ClassFile classFile;

    protected int startPC;
    protected int length;
```

```
protected int nameIndex;
protected int descriptorIndex;
protected int index;

/**
 * Constructor.  This is used to construct from parts of classfile.
 */
public LocalVariable(ClassFileInputStream stream, ClassFile cF)
    throws IOException {
    classFile = cF;
    startPC = stream.readU2();
    length = stream.readU2();
    nameIndex = stream.readU2();
    descriptorIndex = stream.readU2();
    index = stream.readU2();
}

/**
 * Write this into </code>stream</code> with the style of Java classfile.
 */
public void write(ClassFileOutputStream stream) throws IOException {
    stream.writeU2(startPC);
    stream.writeU2(length);
    stream.writeU2(nameIndex);
    stream.writeU2(descriptorIndex);
    stream.writeU2(index);
}

/**
 * Return simple description of this object in a String.
 */
```

```
    public String toString() {
        return "local_variable { " + Integer.toString(startPC)
            + ", " + Integer.toString(length)
            + ", " + Integer.toString(nameIndex)
            + ", " + Integer.toString(descriptorIndex)
            + ", " + Integer.toString(index) + " }";
    }

    /**
     * Pretty printer.
     */
    public void print(IndentedPrintStream out) {
        out.println("local_variable {");
        out.inc();
        out.println("u2 start_pc = "
                    + Integer.toString(startPC) + ";");
        out.println("u2 length = " + Integer.toString(length) + ";");
        out.println("u2 name_index = " + Integer.toString(nameIndex) + "; // "
                    + classFile.constantPool.resolveString(nameIndex));
        out.println("u2 descriptor_index = "
                    + Integer.toString(descriptorIndex) + "; // "
                    + classFile.constantPool.resolveString(descriptorIndex));
        out.println("u2 index = " + Integer.toString(index) + ";");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.29 public class LocalVariableTableAttribute extends AttributeInfo

```
/*
 * $Id: LocalVariableTableAttribute.java,v 1.4 1998/12/01 20:27:49 maruyama Exp
```

```
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>LocalVariableTableAttribute</code>
 * represent a set of informations for debugging purpose -- local variable.
 */
public class LocalVariableTableAttribute extends AttributeInfo {
    protected LocalVariable localVariableTable[];

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public LocalVariableTableAttribute(int nameIndex,
                                       ClassFileInputStream stream,
                                       ClassFile cF) throws IOException {
        super(nameIndex, stream, cF);
        int count = stream.readU2();
        localVariableTable = new LocalVariable[count];
        for (int i = 0; i < count; i++)
            localVariableTable[i] = new LocalVariable(stream, cF);
    }

    /**
     * Write this into </code>stream</code> with the style of Java classfile.
     */
    public void write(ClassFileOutputStream stream) throws IOException {
```

```
    super.write(stream);
    int count = localVariableTable.length;
    stream.writeU2(count);
    for (int i = 0; i < count; i++)
        localVariableTable[i].write(stream);
}

/**
 * Return simple description of this object in a String.
 */
public String toString() {
    return "LocalVariableTable_attribute { " + super.toString()
        + ", local_variable_table_length = "
        + Integer.toString(localVariableTable.length)
        + ", local_variable_table[] }";
}

/**
 * Pretty printer.
 */
public void print(IndentedPrintStream out) {
    out.println("LocalVariableTable_attribute {");
    out.inc();
    out.println("u2 attribute_name_index = "
                + Integer.toString(attributeNameIndex) + ";");
    out.println("u4 attribute_length = "
                + Integer.toString(attributeLength) + ";");
    int count = localVariableTable.length;
    out.println("u2 local_variable_table_length = "
                + Integer.toString(count) + ";");
    if (count > 0) {
```

```
            out.println("local_variable_table[] = {");
            out.inc();
            for (int i = 0; i < count; i++)
                localVariableTable[i].print(out);
            out.dec();
            out.println("}");
        } else
            out.println("local_variable_table[];");
        out.dec();
        out.println("}");
    }
}
```

A.1.5.30 public abstract class MemberInfo

```
/*
 * $Id: MemberInfo.java,v 1.8 1998/12/15 14:40:36 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>MemberInfo</code> represent a
 * set of informations for a field/method contained in a classfile.
 */
public abstract class MemberInfo {
    protected int accessFlags;
    protected int nameIndex;
    protected int descriptorIndex;
```

```
protected Attributes attributes;
protected ClassFile classFile;

public MemberInfo(ClassFileInputStream stream, ClassFile cF)
    throws IOException {
    accessFlags = stream.readU2();
    nameIndex = stream.readU2();
    descriptorIndex = stream.readU2();
    classFile = cF;
    attributes = cF.newAttributes(stream, cF);
}

public void write(ClassFileOutputStream stream) throws IOException {
    stream.writeU2(accessFlags);
    stream.writeU2(nameIndex);
    stream.writeU2(descriptorIndex);
    attributes.write(stream);
}

public String toString() {
    return " { access_flags = " + Integer.toString(accessFlags)
        + ", name_index = " + Integer.toString(nameIndex)
        + ", descriptor_index = " + Integer.toString(descriptorIndex)
        + " };";
}
abstract public String header();

public void print(IndentedPrintStream out) {
    out.println(header() + " {");
    out.inc();
    out.println("u2 access_flags = 0x"
```

```
                + Integer.toHexString(accessFlags) + ";");
        out.println("u2 name_index = " + Integer.toString(nameIndex) + "; // "
                + classFile.constantPool.resolveString(nameIndex));
        out.println("u2 descriptor_index = "
                + Integer.toString(descriptorIndex) + "; // "
                + classFile.constantPool.resolveString(descriptorIndex));
        attributes.print(out);
        out.dec();
        out.println("}");
    }
    /**
     * Accessors
     */
    public int accessFlags() {
        return accessFlags;
    }
    public int nameIndex() {
        return nameIndex;
    }
    public int descriptorIndex() {
        return descriptorIndex;
    }
    public Attributes attributes() {
        return attributes;
    }
    public ClassFile classFile() {
        return classFile;
    }
}
```

A.1.5.31 public class MethodInfo extends MemberInfo

```
/*
 * $Id: MethodInfo.java,v 1.6 1998/12/01 20:27:49 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import java.io.IOException;

/**
 * Each instances of the class </code>MethodInfo</code> represent a
 * set of informations for a method contained in a classfile.
 * Its contents are just same as fields' information, so its are
 * treated as </code>MemberInfo</code> -- superclass of MethodInfo.
 */
public class MethodInfo extends MemberInfo {
    public MethodInfo(ClassFileInputStream stream, ClassFile cF)
        throws IOException {
        super(stream, cF);
    }

    public String toString() {
        return "method" + super.toString();
    }

    public String header() {
        return "method_info";
    }
}
```

A.1.5.32 public class ResolveError extends Error

```
/*
```

```
 * $Id: ResolveError.java,v 1.2 1998/12/15 14:40:36 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

public class ResolveError extends Error {
    Throwable e;

    public ResolveError(String msg) {
        super(msg);
        this.e = this;
    }

    public ResolveError(Exception e) {
        super(e.getMessage());
        this.e = e;
    }
}
```

A.1.5.33 public class SourceFileAttribute extends AttributeInfo

```
/*
 * $Id: SourceFileAttribute.java,v 1.6 1998/12/15 14:40:36 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

import OpenJIT.frontend.util.IndentedPrintStream;
import java.io.IOException;

/**
 * Each instances of the class </code>SourceFileAttribute</code> represent a
```

```
 * file name of the source file.
 */
public class SourceFileAttribute extends AttributeInfo {
    /**
     * It holds an index into ConstantPool that represent the filename.
     */
    protected int sourceFileIndex;

    /**
     * Constructor.  This is used to construct from parts of classfile.
     */
    public SourceFileAttribute(int nameIndex, ClassFileInputStream stream,
                               ClassFile cF) throws IOException {
        super(nameIndex, stream, cF);
        sourceFileIndex = stream.readU2();
    }

    /**
     * Write this into </code>stream</code> with the style of Java classfile.
     */
    public void write(ClassFileOutputStream stream) throws IOException {
        super.write(stream);
        stream.writeU2(sourceFileIndex);
    }

    /**
     * Return simple description of this object in a String.
     */
    public String toString() {
        return "SourceFile_attribute { " + super.toString()
            + ", sourcefile_index = " + Integer.toString(sourceFileIndex)
```

```
            + " }";
    }

    /**
     * Pretty printer.
     */
    public void print(IndentedPrintStream out) {
        out.println("SourceFile_attribute {");
        out.inc();
        out.println("u2 attribute_name_index = "
                    + Integer.toString(attributeNameIndex) + ";");
        out.println("u4 attribute_length = "
                    + Integer.toString(attributeLength) + ";");
        out.println("u2 sourcefile_index = "
                    + Integer.toString(sourceFileIndex) + "; // "
                    + classFile.constantPool.resolveString(sourceFileIndex));
        out.dec();
        out.println("}");
    }
}
```

A.1.5.34 public class UnknownFileException extends Error

```
/*
 * $Id: UnknownFileException.java,v 1.2 1998/11/24 18:14:01 maruyama Exp $
 */

package OpenJIT.frontend.classfile;

public class UnknownFileException extends Error {
    Throwable e;
```

```
    public UnknownFileException(String msg) {
        super(msg);
        this.e = this;
    }

    public UnknownFileException(Exception e) {
        super(e.getMessage());
        this.e = e;
    }
}
```

A.2 OpenJIT バックエンドシステム

A.2.1 メソッド情報受け渡し試験用クラス

```
package OpenJIT;

class TestMethod extends Compile {
    void parseBytecode() {}
    void convertRTL() {}
    void optimizeRTL() {}
    void genNativeCode() {}
    void regAlloc() {}

    public boolean compile() {
        System.err.println("Method:" + this);
        System.err.println("access:" + access);
        System.err.println("nlocals:" + nlocals);
        System.err.println("maxstack:" + maxstack);
        System.err.println("args_size:" + args_size);
        System.err.println();
        return false;
    }
}
```

A.2.2 バイトコード読み出し試験用クラス

```
package OpenJIT;

class TestBytecode extends Compile {
  void parseBytecode() {}
  void convertRTL() {}
  void optimizeRTL() {}
  void genNativeCode() {}
  void regAlloc() {}

  public boolean compile() {
    int i;

    System.out.println("Method:" + this + "(" + bytecode.length + ")");
    for(i = 0; i < bytecode.length; i++) {
      System.out.print(Integer.toHexString((int)(bytecode[i] & 0xff)));
    }
    System.out.println();
    System.out.println();
    return false;
  }
}
```

A.2.3 バックエンド中間コード変換試験用クラス

```
package OpenJIT;

class TestParse extends ParseBytecode {
    void convertRTL() {}
    void optimizeRTL() {}
    void genNativeCode() {}
    void regAlloc() {}

    public boolean compile() {
        int pc;
        bcinfo = new BCinfo [bytecode.length];

        parseBytecode();
        System.out.println("Method:" + this);
        for (pc = 0; pc < bytecode.length; pc++) {
            BCinfo bc = bcinfo[pc];

            if (bc == null) continue;
            System.out.println(pc + "\t" + opcName(pc));
            for (ILnode il = bc.next; il != null; il = il.next) {
                System.out.println("\t" + il);
            }
        }
        return false;
    }
}
```