Asakusaソースコードリーディング #1 - Ashigel Compiler

2011/05/25

あらかわ (@ashigeru)

趣旨

- コンパイラの中身を説明
  - どこに何があるか
  - どんな流れか
- Asakusa DSLについては省略
  - http://www.slideshare.net/ashigeru/inside-of-asakusa-dsl

進め方

- Data Model Definition Language
  - データモデルの定義 (0.2.0~)
  - ウォーミングアップ
- Operator DSL
  - 演算子の定義
  - 中くらいのサイズ
- Flow DSL
  - データフローの定義
  - 大きすぎるので拾い読み

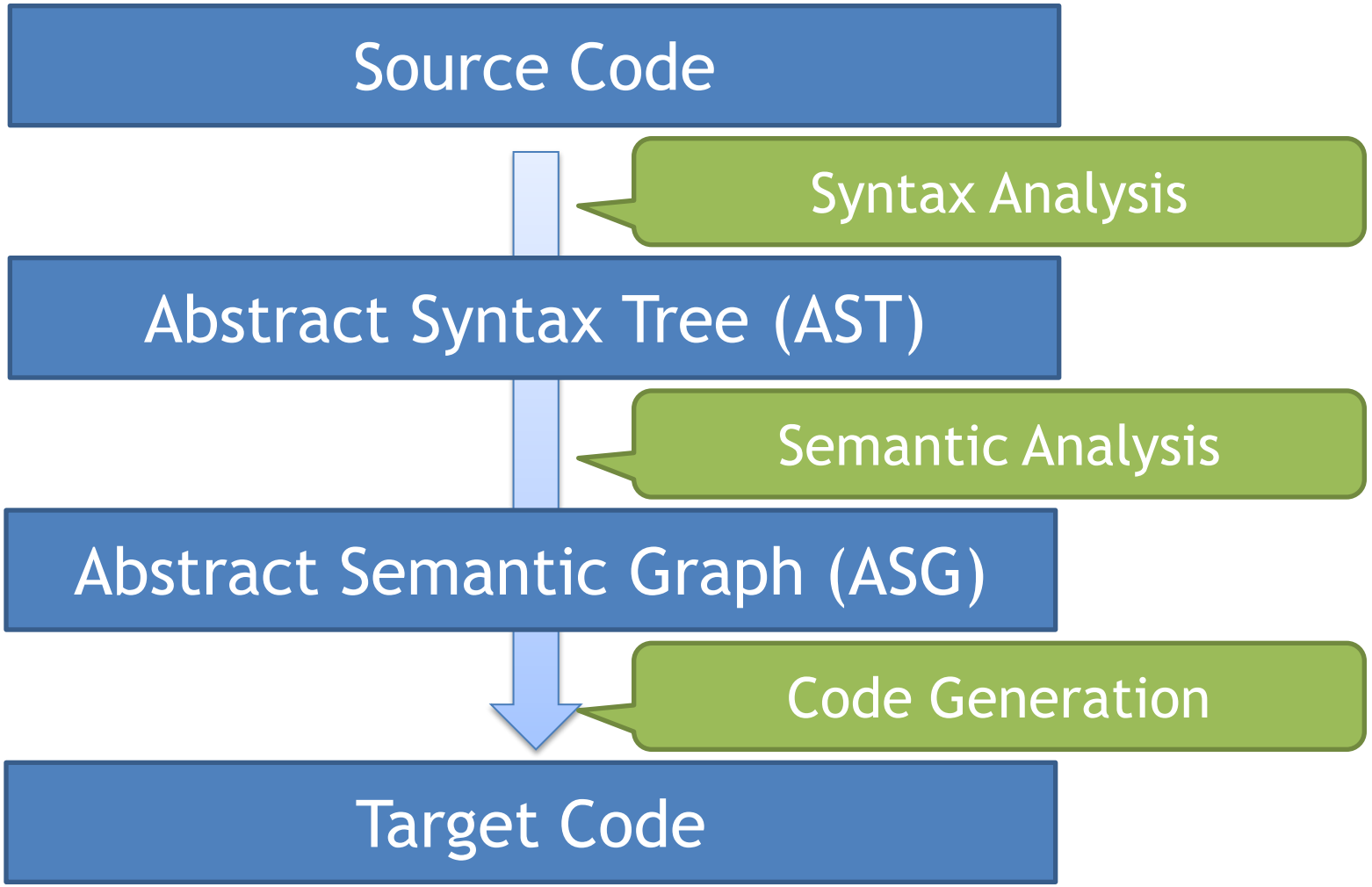
リソース

- Gistにメモ
  - https://gist.github.com/988810
  - 口頭の説明より詳しく
- GitHubのフォークにSCR-01タグ
  - https://github.com/ashigeru/asakusafw/tree/SCR-01/

一応

コンパイラの基礎

よくある構造


Source Code
Syntax Analysis
Abstract Syntax Tree (AST)
Semantic Analysis
Abstract Semantic Graph (ASG)
Code Generation
Target Code

中間表現

- Abstract Syntax Tree
  - ソースコードとほとんど同じ構造 (DOM like)
  - ソースコードをプログラムで取り扱いやすく
  - エラートラッキングの基盤
- Abstract Semantic Graph
  - ソースコードにまたがる参照情報など付与
  - 粒度をターゲットコードに寄せておく
  - 最適化の基盤

エンジン

- Syntax Analyzer
  - ソースコードをASTに (= Parser, Front-end)
  - 単純作業
- Semantic Analyzer
  - ASTをASGに
  - シンボルを実体にバインディング
  - グラフを組み替えて最適化
  - 一番がんばるところ
- Code Generator
  - ASGをターゲットコードに(= Back-end)
  - 命令レベル最適化がなければほぼ単純作業

Asakusa Frameworkでの構造

- Data Model Definition Language (DMDL)
  - 素直な構造
- Operator DSL
  - 注釈プロセッサ (JSR-269)として実装
  - フロントエンドがJavaコンパイラ
- Flow DSL
  - JavaホストのDSL
  - フロントエンドはDSLそのもの (実行可能)
- いずれもJavaのソースコードを生成
  - 厳密には「トランスレータ」

素直な構造でウォーミングアップ

Data Model Definition Language (DMDL)

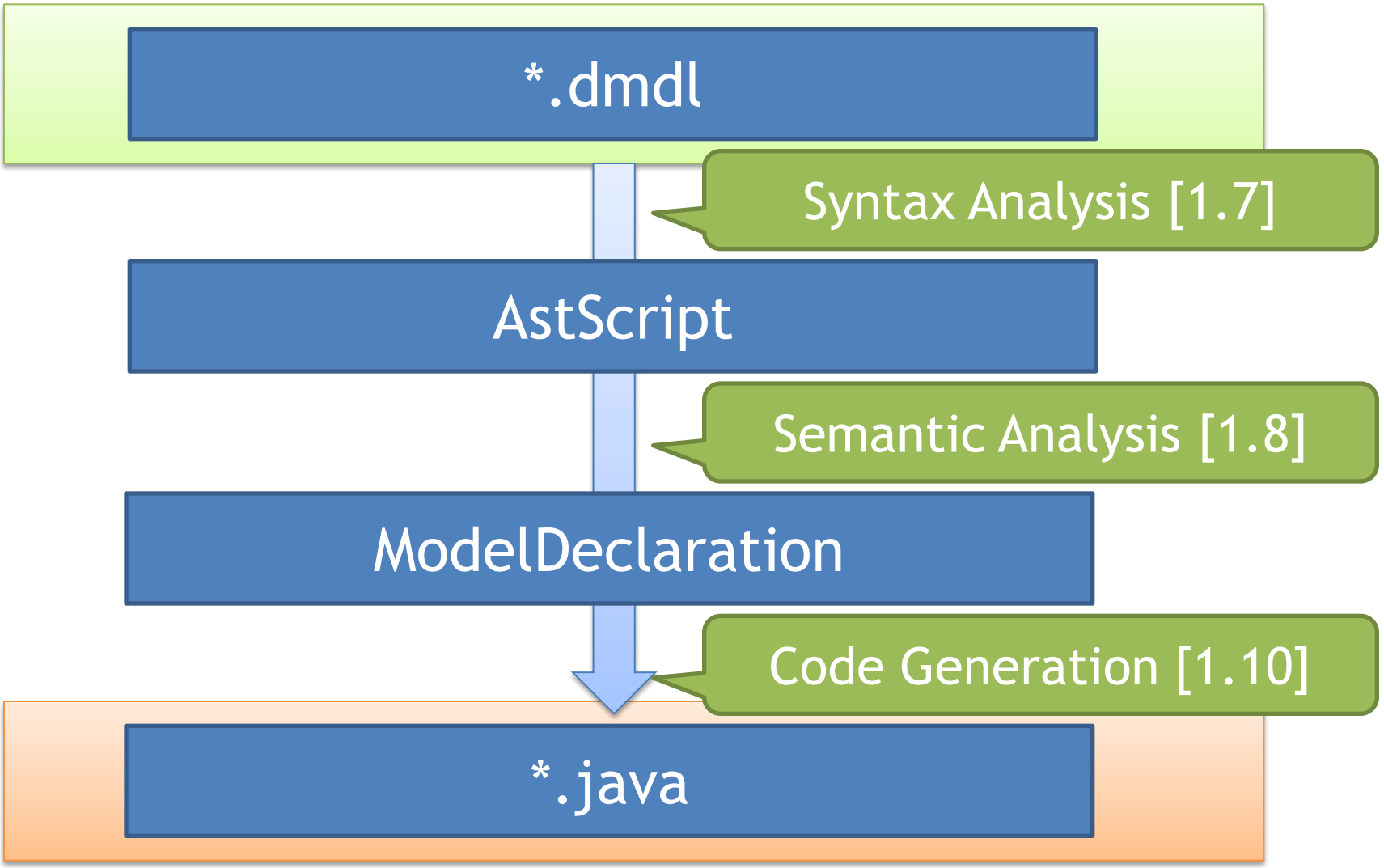
DMDLとは

- データモデル定義DSL (0.2.0~)
  - レコード構造をテキストで記述

```
@namespace(value = com.example)
hoge = {
    number : INT;
    value : TEXT;
};
@namespace(value = com.example)
foo = hoge + {
    extra : DATE;
};
```

他のデータモデルを参照

DMDLコンパイラ [1.3]



中間表現

- AstScript
  - DMDLスクリプト一つ分を表すASTノード
  - 記載された構造をほぼそのまま保持
- ModelDeclaration
  - DMDLのモデル定義一つ分を表すASGノード
  - 参照データモデルのプロパティは展開済

エントリポイント [1.4]

- com.asakusafw.dmdl.java.Main#main
  - オプション引数の解析
  - コンパイラタスクの起動

構文解析 [1.7]

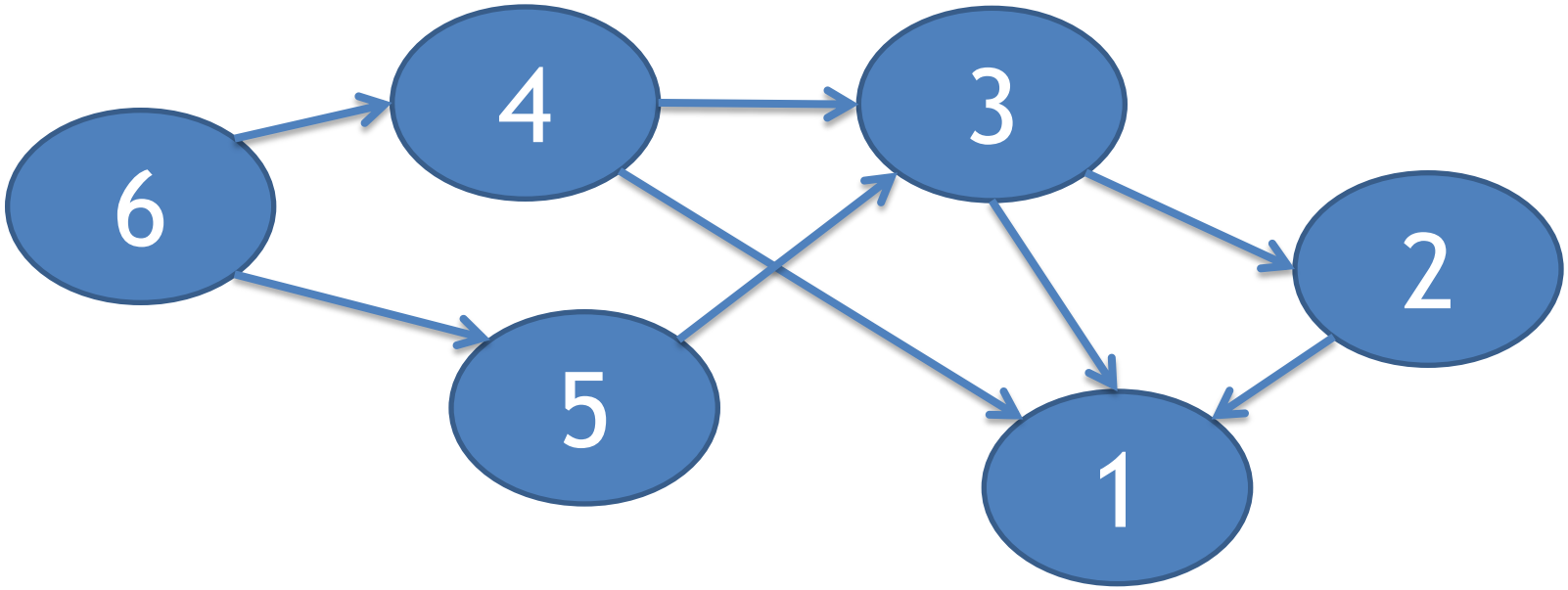
- DmdlParser
  - Java CCで記述したパーサを呼び出す
  - AstScriptというASTオブジェクトを生成

意味解析 [1.8]

- DmdlAnalyzer
  - addModelメソッドでASTのモデルを追加
    - 内部で依存関係のグラフを構築
  - 全部追加したらresolveメソッドでASGを構築
    - 全部追加し終わるまで参照解析を始められない

Topological Sort

- DAG上の要素を依存関係順に並べる
  - 番号の若い順にノードを分析
  - 「準備ができていない」ことがなくなる



ServiceLoader [1.6, 1.9]

- Java標準のService Provider Interface (SPI) を利用するAPI
  - "META-INF/services/<interface>" を作り、実装クラスの一覧を記載
  - ServiceLoader.load(<interface>.class)で実装クラスの一覧をとれる
- コンパイラの拡張ポイントとして利用
  - DMDLでは「属性 (@...)」に対するSemantic Analyzerのフックとして使う

コード生成 [1.10]

- ConcreteModelEmitter
  - ASGからJavaのクラスを生成
  - Java DOMのライブラリを利用 [1.11]
    - テンプレートエンジンで書くと死ぬ
  - SPIを使って生成するクラスを拡張 [1.12]
    - hashCode, equals, toStringの実装
    - Writableの実装
    - などなど

DMDLの設計方針

- データモデル系の中心
  - MySQLのリバース結果もDMDLを経由
  - Excelもここから生成する
  - Data Model Manipulation Language?
- 拡張はSPI経由
  - 追加属性でデータモデルの表現を拡張
  - Javaのドライバで生成コードを拡張
  - MySQLのリバース結果でも利用

やっとAsakusa DSLファミリ

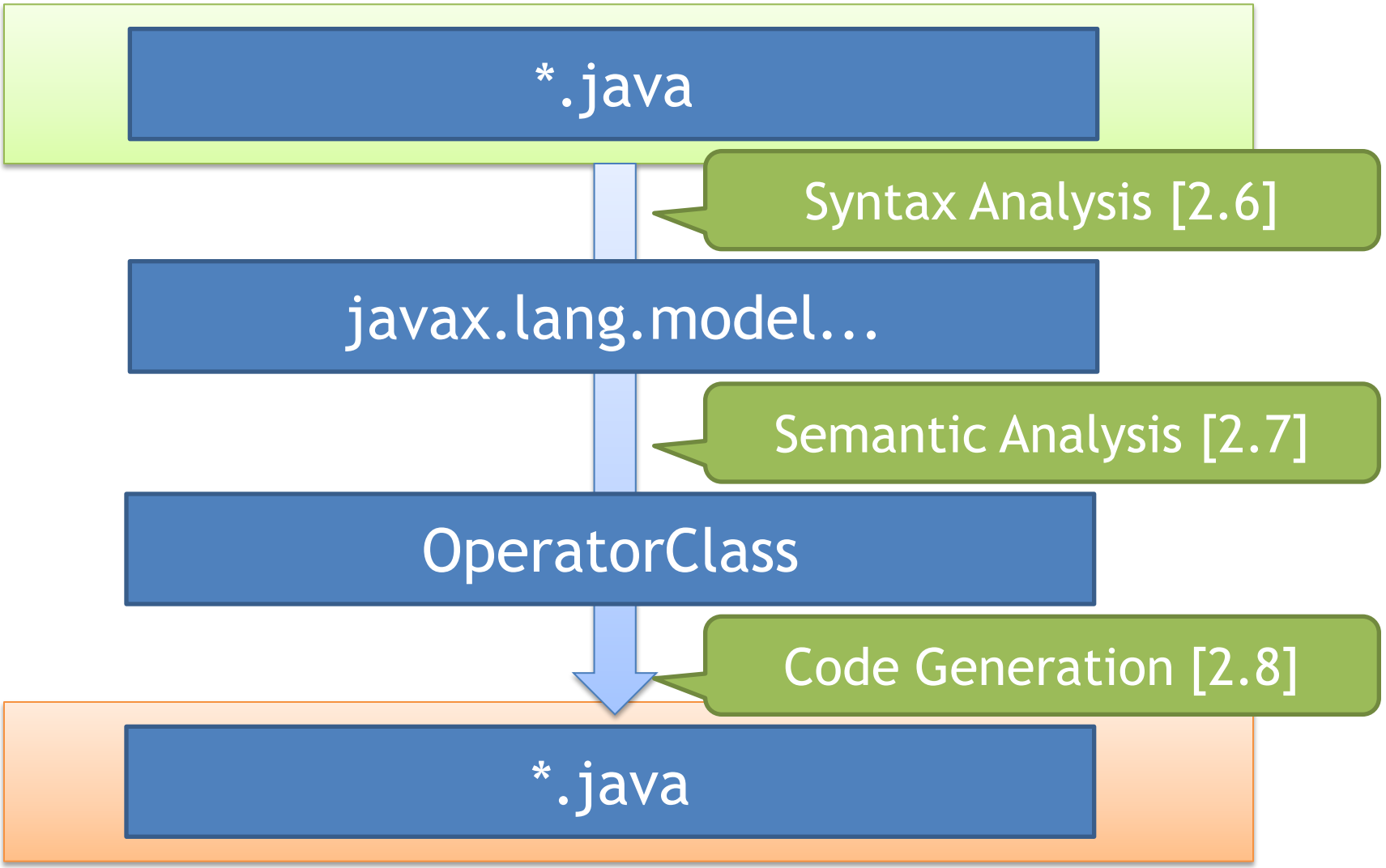
Operator DSL

Operator DSLとは

- 演算子記述DSL
  - Javaメソッド + 注釈
  - 演算子ファクトリを生成
    - Flow DSL向け
  - 演算子実装クラスを生成
    - Operator DSLでは抽象クラスとして記述

```
@Update
public void op(Hoge hoge) {
    hoge.setValue(100);
}
```

Operator DSLコンパイラ [2.2]



中間表現

- javax.lang.model.element.TypeElement
  - Javaソースコードの型宣言モデル
  - 注釈プロセッサを使うと簡単に取れる
  - Java上の参照は解決済み
- OperatorClass
  - 演算子メソッドの定義だけを持つ
  - 演算子ごとに情報が独立しているため、ASGというほど複雑な構造はしていない

注釈プロセッサ [2.3]

- 標準のjavacに組み込んで注釈処理に介入できる (Java 6~)
  - javacの中で動く
  - javax.annotation.processing.Processorのサブタイプをサービスとして登録すると利用可能

エントリポイント [2.4]

- OperatorCompiler
  - 演算子プロセッサの初期化
    - 演算子の種類ごとに個別のプロセッサ
    - SPI経由で自由に追加可能
  - 演算子メソッドの分析
  - ターゲットコードの生成

- 注釈プロセッサとしてjavacから呼ばれる

構文解析 [2.6]

- 注釈プロセッサで代用
  - 「特定の注釈がついたメソッド一覧」などをすでに取り出せる状態

- まじめにJavaコンパイラを書く気はない

意味解析 [2.7]

- OperatorClassCollector
  - 演算子プロセッサをaddメソッドで追加
    - 対応する演算子メソッドを裏側で収集
    - メソッドごとのDSLエラーをここで解析
  - 全部追加したらcollectメソッドでASGを構築
    - 全部追加し終わるまで参照解析を始められない
    - クラスごとのDSLエラーをここで解析

コード生成 [2.8]

- OperatorClassEmitter
  - 演算子ファクトリクラス (*Factory)を生成
  - 演算子実装クラス (*Impl)を生成
- OperatorFactoryClassGenerator [2.8.1]
  - 演算子ごとに演算子プロセッサを実行
  - 実行結果を元に演算子ファクトリを生成
    - Flow DSLで「自分自身のデータフローのノードを構築するプログラム」を生成する

Operator DSLの設計方針

- Flow DSL用のプログラムを生成
  - 「データフローのノード」を表すクラス
  - Flow DSLで記述したデータフローを解析するためのコードを生成物に含めてある
- 拡張はSPI経由
  - 演算子プロセッサを追加すると新しい演算子を定義できる
  - 実際には、Flow DSL用の演算子プロセッサも別途必要

でかい

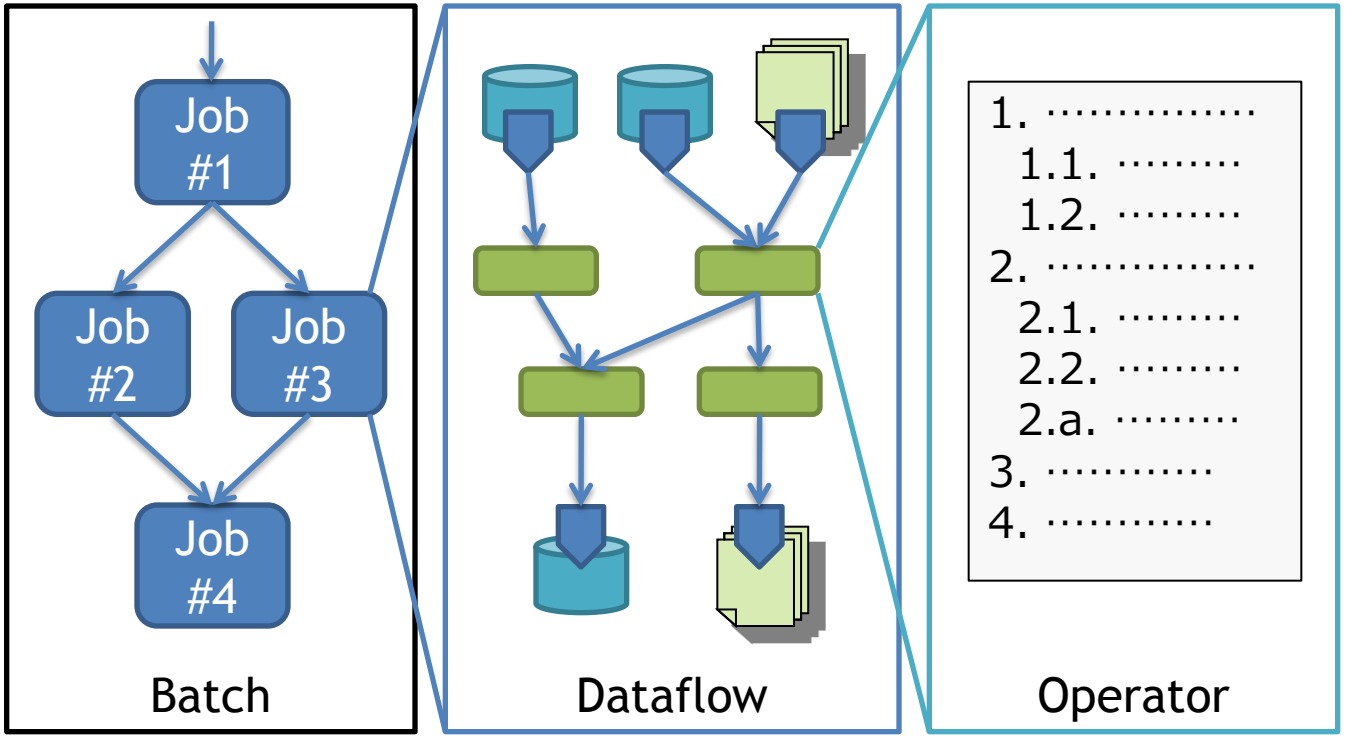
Flow DSL

Flow DSLとは

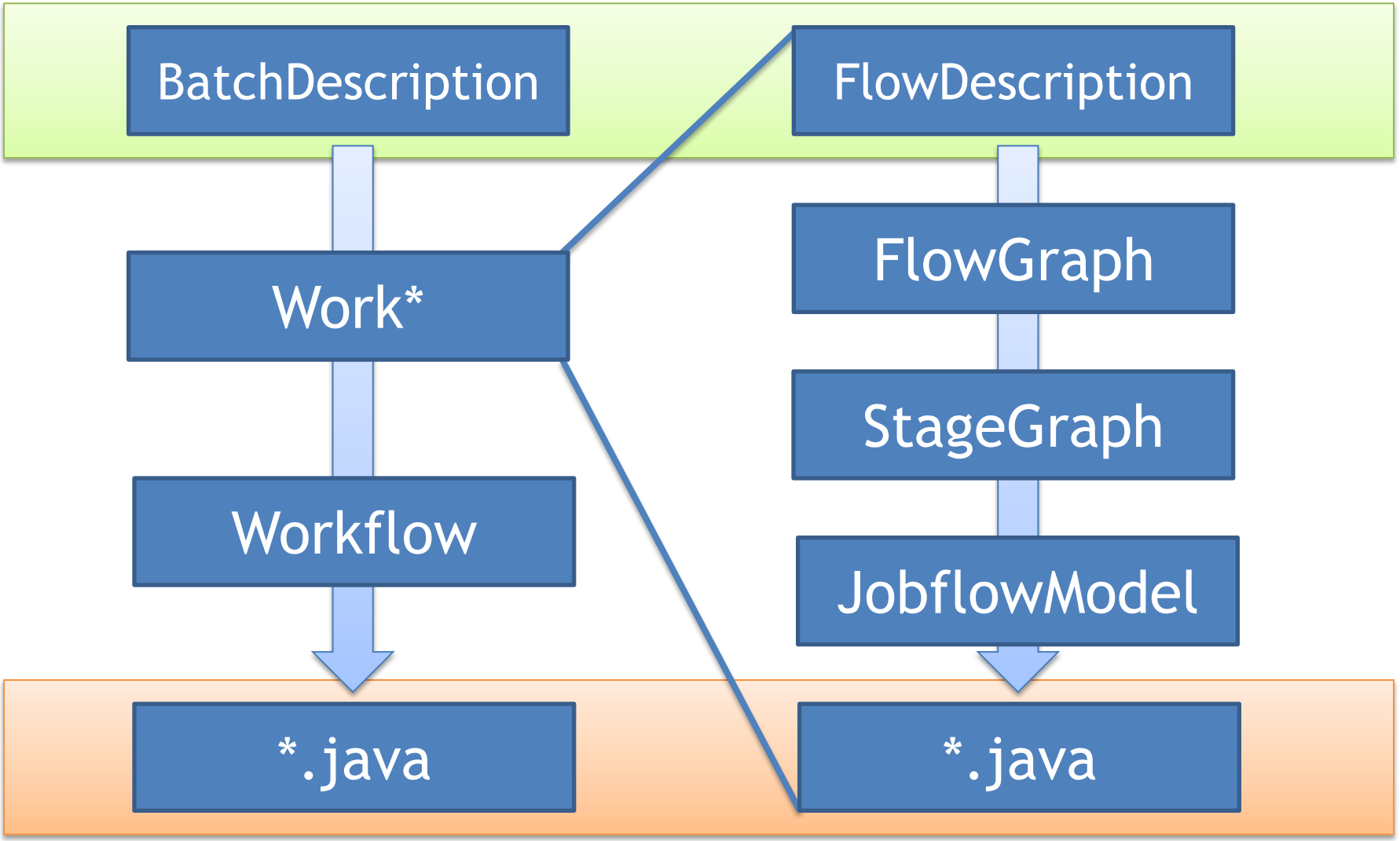
- データフロー記述DSL
  - Operator DSLの演算子をグラフ状にくみ上げ
  - ここからMap Reduceジョブネットを構築

バッチの構造

- 複数のデータフローを並べたもの
  - 基本的にはバッチ単位でコンパイル
  - データフローごとに階層的にコンパイル



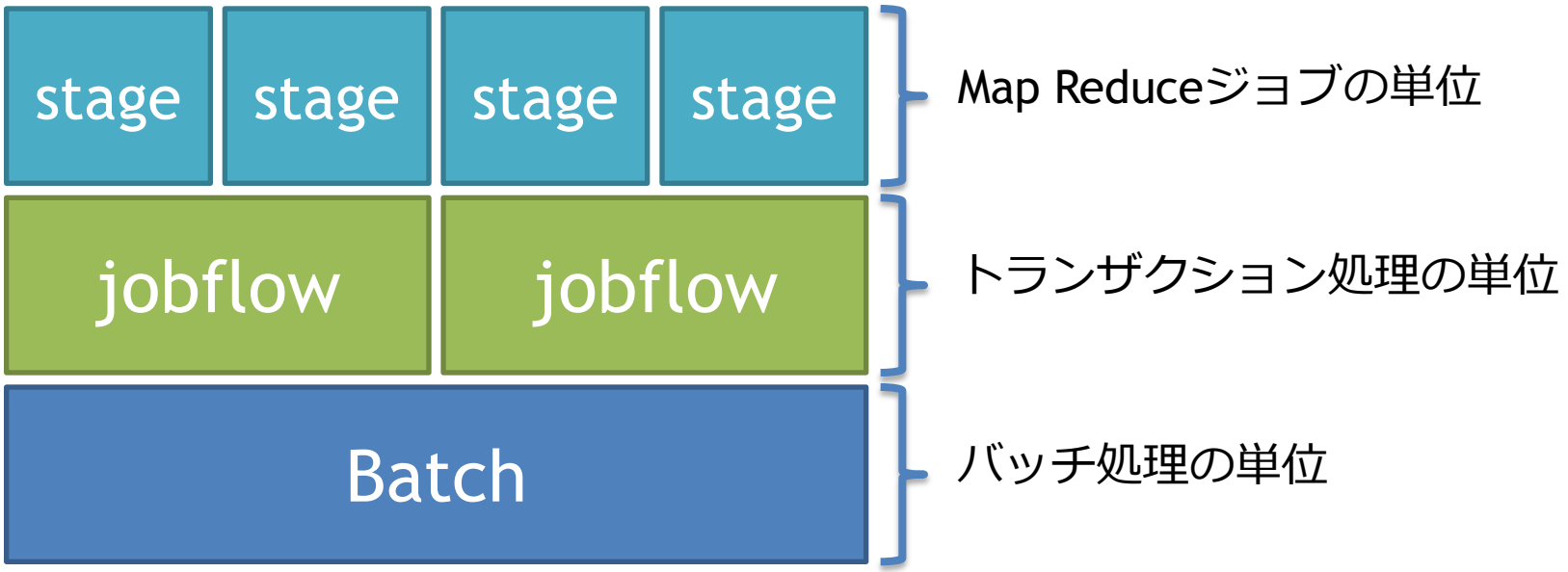
Flow DSLコンパイラ [2.2]



コンパイルのレイヤ

stage
jobflow
batch

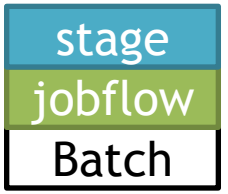
- 分割統治
  - それぞれのレイヤで独立してコンパイル
  - スライドの右上に現在のレイヤを表示



中間表現

- BatchDescription
  - ユーザーが記述したBatch DSL
  - 複数のデータフローをまとめて取り扱う
- Work*
  - ユーザーが記述したBatch DSLを実行した結果
  - データフロー1つ分のFlow DSLを持つ
  - 依存先のWorkに関する情報を持つ
- Workflow
  - コンパイル済みのバッチ
  - ジョブネット構造とジョブのコマンド情報

中間表現



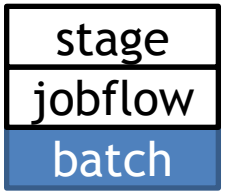
- FlowDescription
  - ユーザーが記述したFlow DSL
  - 単体のデータフローを取り扱う
- FlowGraph
  - ユーザーが記述したFlow DSLを実行した結果
- StageGraph
  - 実行計画によりジョブ単位に区切ったデータフロー
  - Map ReduceジョブごとにStageModel
- JobflowModel
  - コンパイル済みのデータフロー
  - ジョブネット構造とステージごとのコマンド情報
  - 外部入出力の情報なども

stage
jobflow
batch

エントリポイント [3.3]

- BatchCompilerDriver
  - コマンドライン引数の解析
  - DirectBatchCompilerに処理を委譲
- DirectBatchCompiler
  - バッチDSLの解析
  - テストドライバからも利用する

バッチのコンパイル



- バッチの解析
- データフローのコンパイル
- ワークフロー情報の出力

記述解析 [3.4]

- BatchDriver
  - ユーザーの記述したDSLを実行
    - BatchDescription
  - DSL自体がDSLで表記した内容を解析
    - ホスト言語を利用する強みのひとつ
- WorkDescriptionの依存グラフができる

意味解析 [3.5]

stage
jobflow
batch

- BatchCompiler
  - WorkDescriptionを一つ一つ処理
  - データフローを処理するコンパイラに委譲
- ワークフローの詳細な計画を立てる
  - ワークフロー内のノード (データフロー)に対する処理は次の層で行う

エントリポイント [3.6]

- JobFlowWorkDescriptionProcessor
  - データフローひとつ分を処理

stage
jobflow
batch

データフローのコンパイル

- データフローの分析
- 実行計画 -> ステージグラフ
- Map Reduceジョブごとに:
  - シャッフル構造の分析
  - Map, Reduceプログラムの生成
  - Shuffle情報の生成
  - ジョブクライアントプログラムの生成
- 入出力情報の生成

記述解析 [3.7]

stage
jobflow
batch

- JobFlowDriver
  - ユーザーの記述したDSLを実行
    - FlowDescription
  - バッチDSLの解析とほぼ同じ
- FlowGraphができる
  - この時点ではフローグラフは多層化したまま
  - 後のステージで平坦化する

意味解析 [3.8]

stage
jobflow
batch

- StagePlanner
  - 結線構造の検査
  - グラフの書き換え
  - フローグラフの標準化
  - 多層化したフローグラフの平坦化
  - ステージ区切りの抽出
  - Map, Reduceブロックの抽出
  - ステージブロックの抽出
  - ステージグラフの構築

プラグインで
書き換えの種類を
増やせる

コンパイル [3.9]

stage
jobflow
batch

- StageCompiler
  - シャッフル構造の分析
  - ステージ構造の分析
  - シャッフル情報の生成
  - 演算子グラフの生成
  - Mapper, Reducerの生成

プラグインで
演算子の種類を
増やせる

- ジョブクライアントの生成は後回し
  - この時点では入出力のパスが未確定

stage
jobflow
batch

コード生成 [3.10]

- JobflowCompiler
  - 入出力の分析
  - 入出力情報の出力
  - ジョブクライアントの出力

プラグインで
入出力の種類を
増やせる

- 対応している入出力
  - Hadoop FileInputFormat/FileOutputFormat
  - ThunderGate

コード生成 [3.11]

stage
jobflow
batch

- BatchCompiler
  - ワークフロー情報の出力

プラグインで
ワークフロー情報の
形式を増やせる

- 対応している形式
  - Monkey Magic v0.9.7
  - Experimental Shell Script