プログラミング言語処理をやってみよう

石尾 隆
奈良先端科学技術大学院大学
ソフトウェア工学研究室
ishio@is.naist.jp

本資料の内容

- Java のソースプログラムから情報を取り出す Java プログラムを書く方法
  - 必要なツールの使い方を練習するための演習課題
    https://github.com/takashi-ishio/ParserExample.git
  - Java の知識はある、言語処理系の知識はないものとした説明
- 言語処理の理論的な説明は含まない

プログラミング言語処理は
単純な文字列処理より少し難しい

例1. ある文字列が Java の10進数表現かどうかを知りたい

- Java では “64” や “1234_5678” が許される
- “0100” は8進数, “0x40” なら16進数

例2. プログラムに登場する if 文の数を知りたい

- 単に “if” で検索すると “specific” のような他の単語の中の出現や、文字列リテラル、コメント内の出現にもマッチしてしまう

この授業でやること

1. プログラミング言語の文法を知る
   - Java 言語仕様の読み方
   - オートマトンを使ったルールの可視化、分析
   - 正規表現でのルールの記述

2. 字句解析・構文解析の利用
   - トークンに対する字句情報の収集
   - 構文木を利用したデータ抽出

Step 1. プログラミング言語の文法を知る

- 何が許されているかは Java 言語仕様の文法 (Grammars) として書かれている
  - Java Language Specification https://docs.oracle.com/javase/specs/

10進数リテラルの文法

A: B C で「A とは B, C が並んだもの」
複数行の候補は「いずれか」と読む
[X] は X の 0回または 1回の出現
{X} は X の 0回以上繰り返しの出現

DecimalNumeral:

0

NonZeroDigit [Digits]

NonZeroDigit Underscores Digits

NonZeroDigit:

(one of)

1 2 3 4 5 6 7 8 9

Digits:

Digit

Digit [DigitsAndUnderscores] Digit

Digit:

0

NonZeroDigit

DigitsAndUnderscores:

DigitOrUnderscore {DigitOrUnderscore}

DigitOrUnderscore:

Digit

_

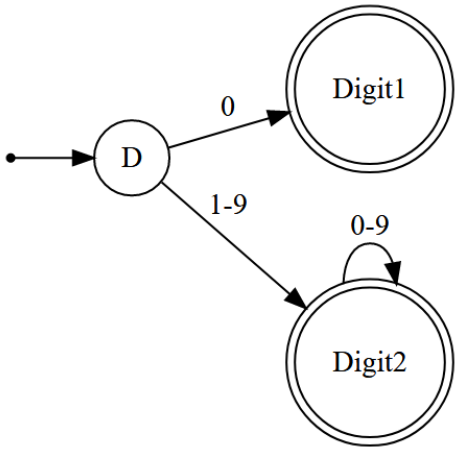
Underscores:

_ {_}

オートマトンによる文法の可視化

- 有限状態機械（Finite State Machine）あるいは有限オートマトン
  - 文字列を入力として受け取る
  - 初期状態からスタートして、１文字ごとに辺のラベルを見て次の状態へ進む
  - ◎ は受理状態。文字の並びを最後まで処理した結果、ここで停止すれば処理成功。
  - ○ は普通の状態。ここで停止したら処理失敗。

10進数の数値： 1-9の後ろに任意の個数の0-9が続くか0である



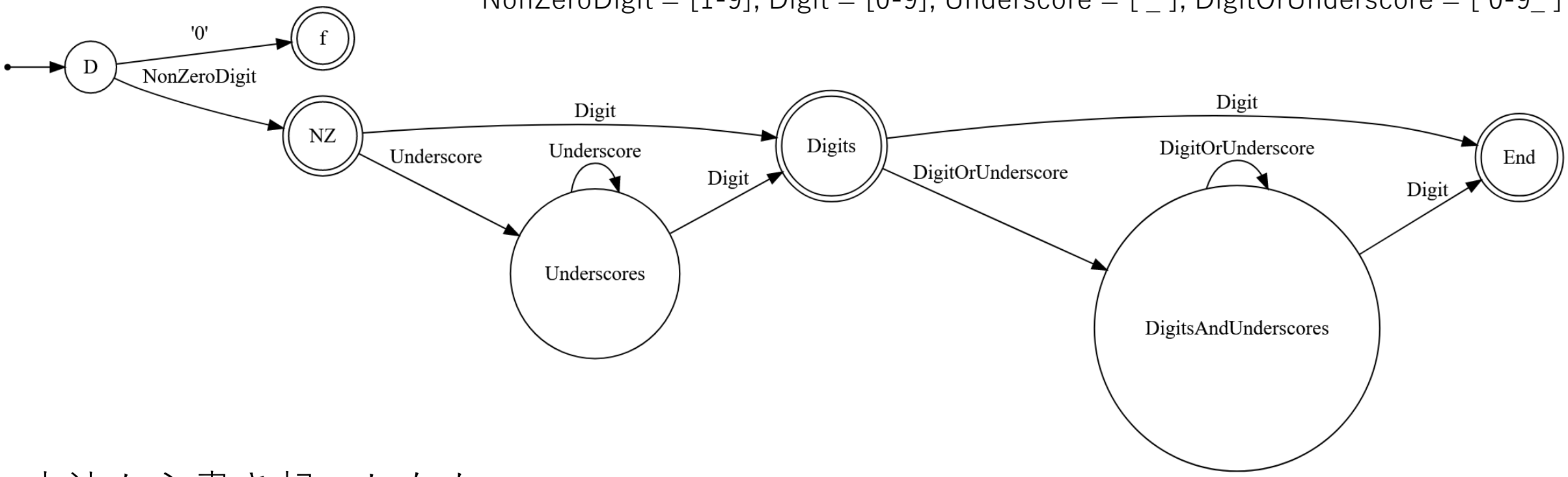
オートマトンは文字列集合を正確に表現する

- マッチのルールが決まっているので、オートマトンの定義と文字列が渡されれば、必ず受理か不受理が判定できる

- 「あいまいな」（非決定的な）記述があっても、等価変換で正確な表現が得られる

Java における10進数のオートマトン

NonZeroDigit = [1-9], Digit = [0-9], Underscore = [_], DigitOrUnderscore = [0-9_]

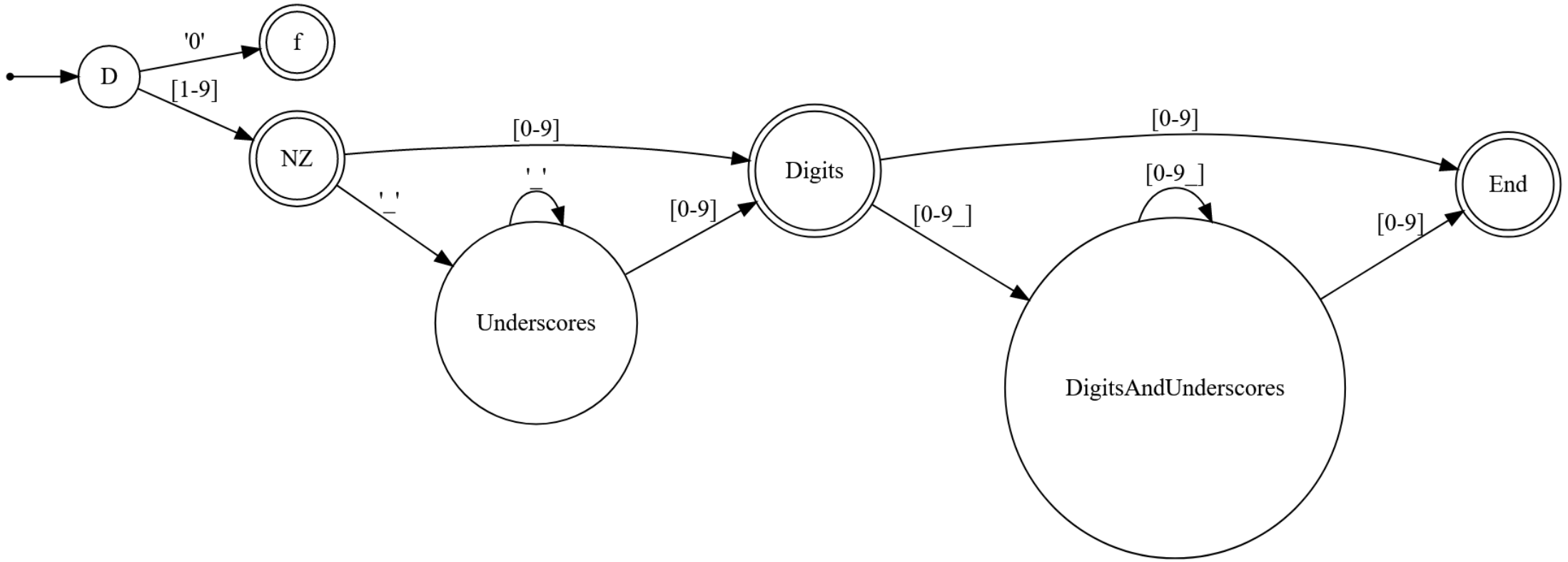


文法から書き起こしたもの。

文法の左辺にあるどの記号に進んだかを状態とみている。

同じ文字列に対して到達可能な状態が複数ある（非決定的である）。

オートマトンの簡略化 (原形)



可読性の都合上，Digit, NonZeroDigit などを
実際の対応する記号に置き換えている

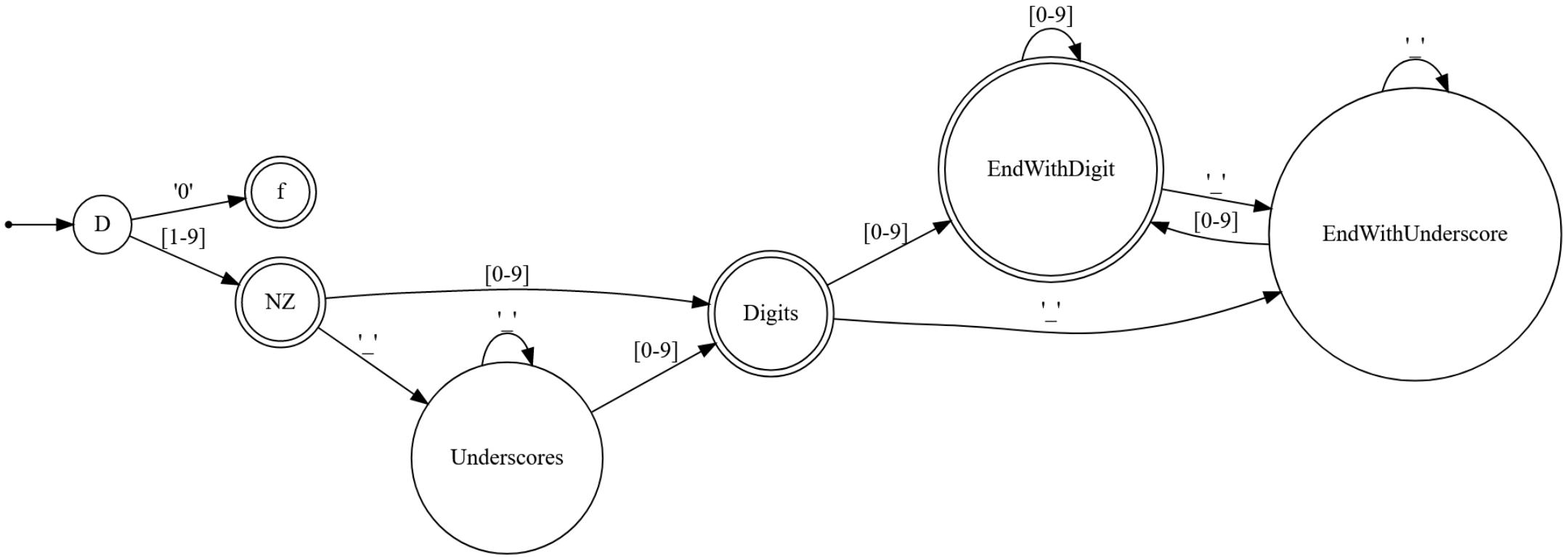
非決定性の除去

- 複数の状態に遷移可能なら，「どれに遷移できるか（遷移可能な状態の集合）」自体を状態とみなす

遷移前の状態	入力される記号	遷移後の状態
Digits	[0-9]	{DigitsAndUnderscores, End}
Digits	“_”	DigitsAndUnderscores
DigitsAndUnderscores	[0-9]	{DigitsAndUnderscores, End}
DigitsAndUnderscores	“_”	DigitsAndUnderscores

ここでは {DigitsAndUnderscores, End} を 1 状態とみなせる

オートマトンの簡略化 (1)



DigitsAndUnderscores に関する状態遷移を決定性に書き換えた。

- EndWithDigit ＝ Digit が来て {DigitsAndUnderscores, End} の両方にいられる状態
- EndWithUnderscore = Underscore が来て必ず DigitsAndUnderscores にいる状態

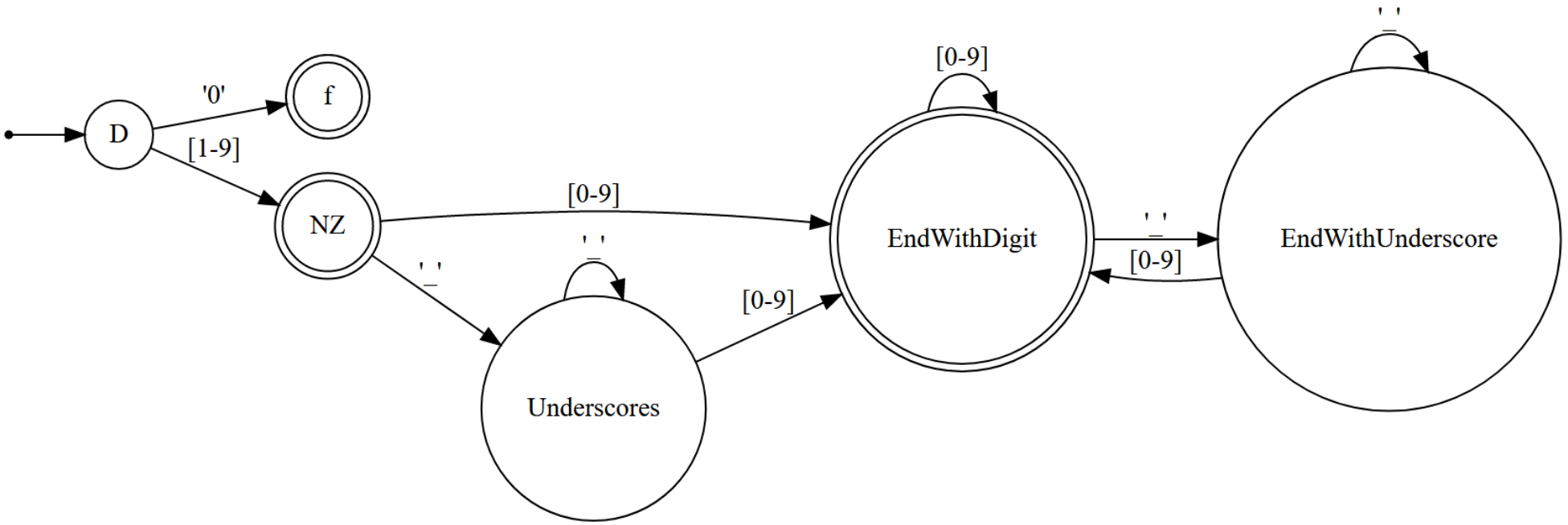
等価な状態のマージ

- ２つの状態から始まる状態遷移が，同一の入力系列に対して同じ受理・不受理の結果になるなら２つの状態は等価である

遷移前の状態	入力される記号	遷移後の状態
Digits [受理状態]	[0-9]	EndWithDigit [受理状態]
Digits [受理状態]	“_”	EndWithUnderscore
EndWithDigit [受理状態]	[0-9]	EndWithDigit [受理状態]
EndWithDigit [受理状態]	“_”	EndWithUnderscore

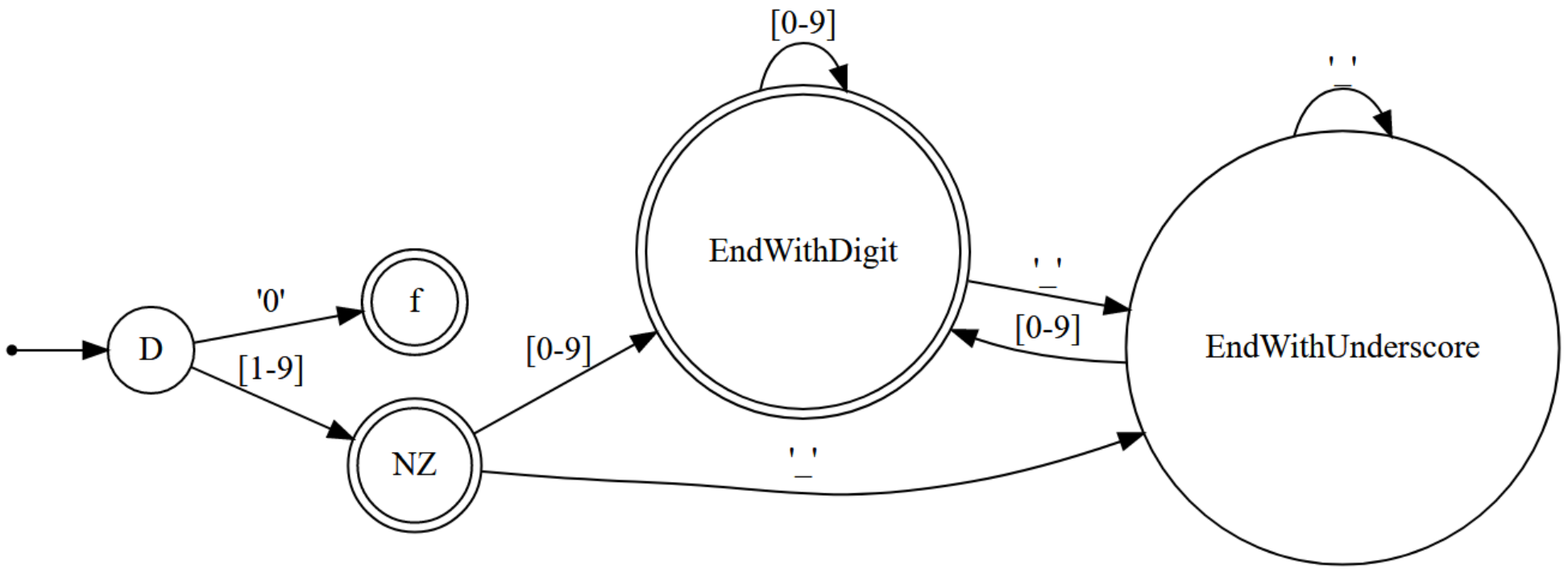
Digits と EndWithDigit はどちらも受理状態で，
同一の入力に対して同一の遷移をするので等価である

オートマトンの簡略化 (2)



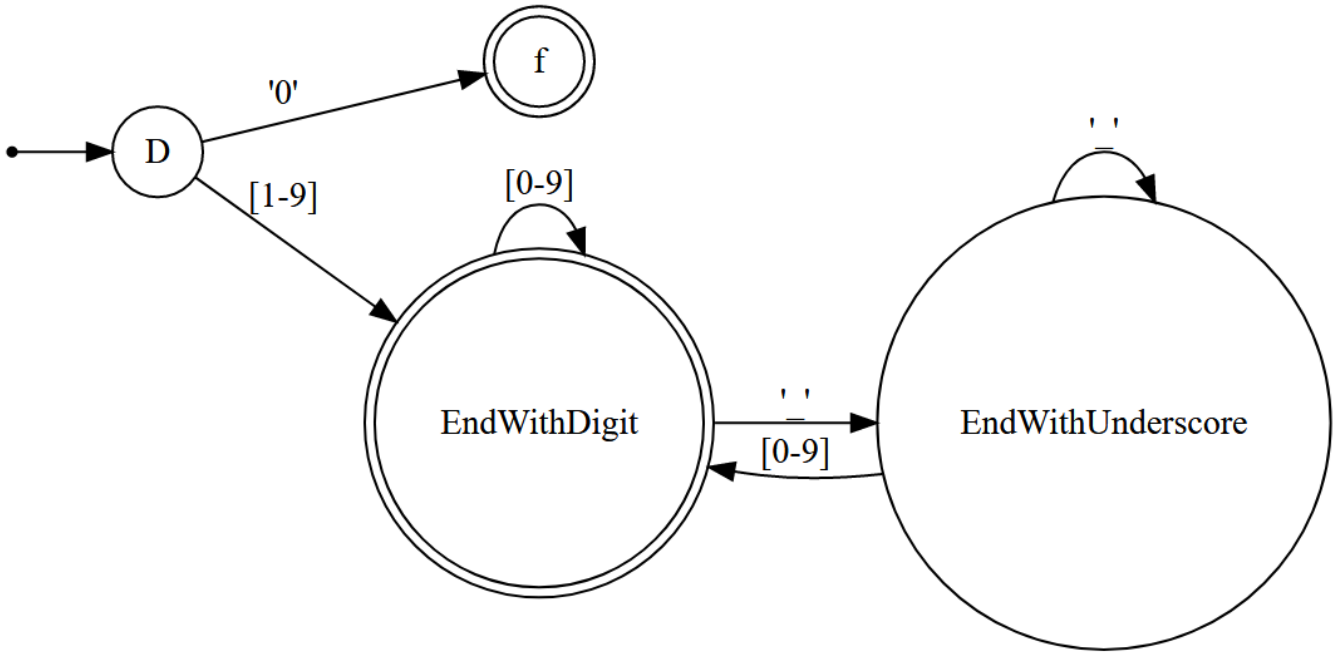
EndWithDigit と Digits が等価だったので 1 状態にマージした結果

オートマトンの簡略化 (3)



Underscores と EndWithUnderscore が等価なのでマージした

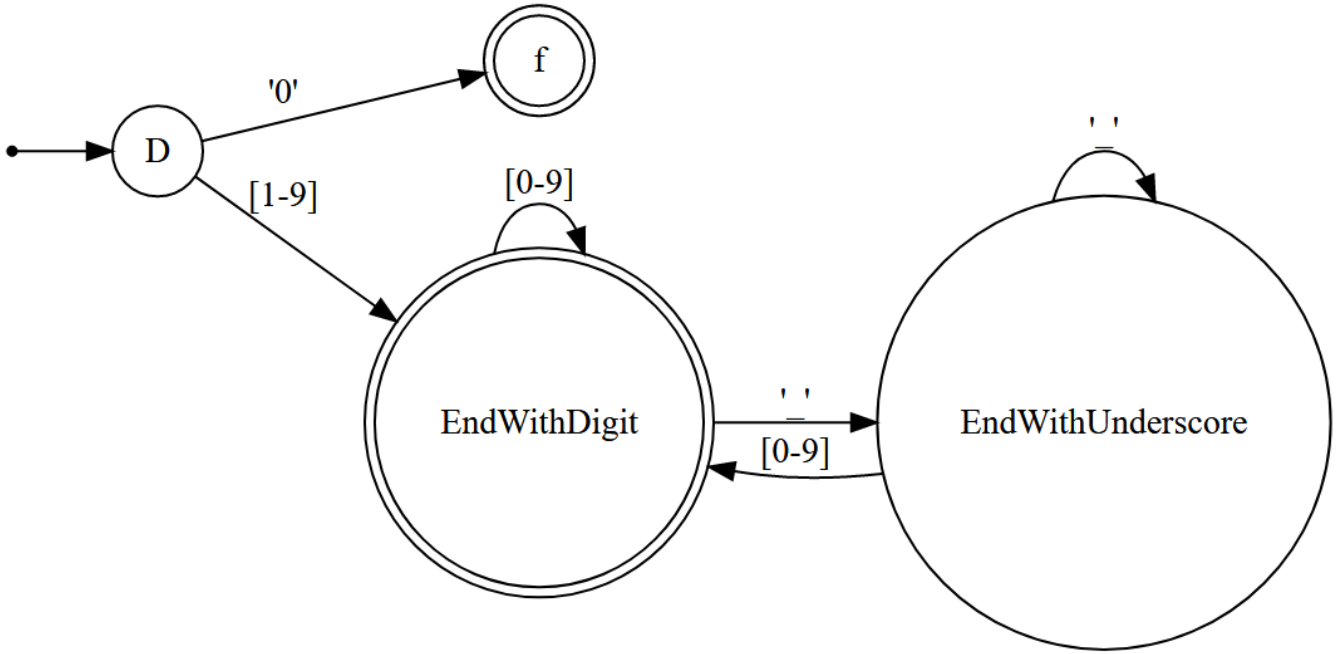
オートマトンの簡略化 (最終形)



NZ と EndWithDigit が等価なのでさらにマージした

オートマトンから見える Java における10進数リテラルの定義

- 2 桁以上の整数は 1-9 で始まらないといけない
- 2 桁目以降は “_” が登場できる，いくつ並んでもよい
- 整数は 0-9 で終わらないといけない



[注意] 文法は意味を定義しているわけではない

- 文法そのものは，　1____2 が10進数の記述であることは示すが，それが 12 という値として解釈されるということは規定していない
  - 言語仕様として，そういう意味解釈が定義される

- 文法上は 123456789012345678901234567890 のような巨大な数値も許される
  - コンパイラにとって構文エラーにはならないが，意味エラーになる

正規表現によるオートマトンの利用

- オートマトンをテキストで表現したもの
  - 何らかの文字列が与えられたとき，マッチする（受理状態になる）か，マッチしない（受理状態にならない）かのどちらかになる

- 使い方
  - Java の String.matches(regex) メソッド：オブジェクトが正規表現にマッチするなら true, そうでなければ false を返す
  - 高機能なテキストエディタの検索機能：正規表現にマッチしたすべての文字列を検索できる

「メタ文字」による正規表現の記述

- 文字集合の表現 “[ ]” など
  - [0123456789] は「0, 1, 2, 3, 4, 5, 6, 7, 8, 9 のどれか 1 文字」
  - [0-9] は「0から9までのどれか 1 文字」（Digit に相当）
  - [0-9_] は「0から9まで，あるいは “_” のどれか 1 文字」
  - ¥d は [0-9] と同等

- 出現回数の表現 ?, *, +
  - X? と書くと，X が出現してもよいし，しなくてもよい
  - X* は X の 0回以上、X+ は X の 1回以上の出現

- 正規表現のグループ化 “()”
  - (123)+ は， 123, 123123, 123123123, … に対応する

- OR の表現 “|”
  - a|b は [ab] と同様。 a|(bc) とすると a または bc のどちらか。

- カッコ自体などの記号を使いたいときは “¥” でエスケープする
- Java の文字列リテラルとして書くときは “¥” 自体をエスケープする必要がある

演習(1) 正規表現を書いてみる

Java の10進数整数にマッチする正規表現を記述せよ.

- ParserExample の src/test/java ディレクトリ以下にある jp.naist.se.parser.RegexTest クラスの以下のメソッドとして記述すると JUnit でテストできる

```
public static boolean isDecimalLiteral(String text) {
    // TODO 1. Write a regular expression that matches a decimal number literal.
    return text.matches("Write a correct regular expression here!");
}
```

- https://regex-testdrive.com/ja/dotest でも練習可能
- 時間に余裕のある人は：
  - Java の10進数整数リテラルにマッチする正規表現（long 型を意味する l/L が末尾に付いても付かなくてもよい版）
  - 2進数，8進数，16進数をサポートする版 …などに挑戦してみるとよい

正規表現の便利なところ

- “()” を利用すると，パターンの一部にマッチした部分文字列を抽出できる（賢いテキスト置換などに使える）
  - 例1: GitHub の URL https://github.com/user/repo/.../filename という形式から https://github.com/user/repo の部分だけを取り出す
    - (https:¥/¥/github¥.com¥/[^¥/]+¥/[^¥/]+)¥/
  - 例2: https://regex-testdrive.com/ja/dotest に電話番号の分解例がある
- Ruby や Python などの文字列処理が得意な言語 ≒ 正規表現が利用しやすい言語

正規表現を詳しく知るためには

Jeffrey E. F. Friedl 著「詳説 正規表現」（オライリー）を読もう！

- パターンの記述能力，高速なマッチができる書き方など，たくさんの知恵が詰まっている

正規表現の限界：
プログラミング言語の構文をすべては扱えない

- “{ }” などの入れ子関係は表現できない
  - 直観的な説明でいうと，N個の開きカッコが出現したことを状態として記憶しないと対応する閉じカッコが処理できない．状態数が有限のオートマトンでは，それより大きい数のカッコの出現を扱えない．
  - つまり，カッコの対応を正規表現でチェックしようとしてはいけない

- プッシュダウンオートマトンという拡張によって，多くのプログラミング言語の「文脈自由文法」が表現可能になる
  - 状態を移動するときにスタックに処理中のトークン列を積んでおけるもの
  - 例： Java のクラス宣言は、内部にクラス宣言を含むことができる。「現在処理中のクラス宣言」が処理できた後、外側のクラス宣言を処理できるような処理系が必要。

Step 2. 字句解析・構文解析の利用

- やること
  - ANTLR を使って字句解析器・構文解析器を生成する
  - Java のソースプログラムに対して字句解析を実行する
  - 字句解析の結果をもとに構文解析を実行する

ANTLR とは

- ANother Tool for Language Recognition
- Terence Parr が開発した Parser Generator
  - 文法定義ファイル (.g) を与えると、字句解析・構文解析器を生成する
    - Adaptive LL(*): non-left-recursive な文脈自由文法はすべて扱える
      - Terence, P. et al.: Adaptive LL(*) Parsing: The Power of Dynamic Analysis, In Proc. OOPSLA 2014.
  - Java 以外に C++ 用などの解析器も生成できる
    - ただし文法定義ファイルの書き方によっては、Java 専用のものなどもある

ANTLR4 が生成するファイル

- サンプルでは Maven を使った方法が設定済みなので自動でファイルが生成された状態になっている
  - target/generated-sources/antlr4 ディレクトリ

- **Java9Lexer.java**: 字句解析器
- **Java9Parser.java**: 構文解析器
  - 構文木を表現する “Context” クラス群を内部クラスとして定義する
- Java9Visitor.java: 構文木訪問の Visitor インタフェース
- **Java9BasicVisitor.java**: Visitor インタフェースの標準実装
  - 木構造オブジェクトに再帰呼び出しで「訪問」していくイメージの動作をする。
- Java9Listener.java: 構文木訪問の Listner インタフェース
- Java9BasicListener.java: Listener インタフェースの標準実装
  - ANTLR4 の TreeWalk クラス等と組み合わせて、構文木の各ノードを訪問したときにメソッド呼び出しを受け取る。

字句解析のステップ

```
CharStream stream = CharStreams.fromFileName("MyExample.java");
Java9Lexer lexer = new Java9Lexer(stream);
CommonTokenStream tokens = new CommonTokenStream(lexer);
```

ソースファイル = 文字列(CharStream)

```
import java.io.IOException;
import java.util.ArrayList;

/** comment */
public class JavaParseExample {
...
```

字句解析結果 = トークン列 (CommonTokenStream)

```
import  java  .  io  .  IOException  ;
import  java  .  util  .  ArrayList  ;
public  class  JavaParseExample  {
...
```

- 各トークンに属性が付与される
  - Index: 順番
  - Type: import, class, Identifier, ...
  - Line: 行番号
  - CharPositionInLine: 桁位置
  - Channel: コメントなど無視するものかを識別する

演習(2) 字句解析の実行

Java9Lexer を使って、Java ソースファイルのトークン列を出力するプログラムを作成せよ。

- src/main/java ディレクトリ以下にある jp.naist.se.parser.JavaLexerExample クラスが素材プログラム。
- 対象ファイルは特定の１つでよい。example-source ディレクトリにサンプルがあるものを選んで使う。
- トークンの読み込みは必要ない限り行われないので、CommonTokenStream.fill() を実行して CommonTokenStream にトークンを読み込ませる必要がある。
- CommonTokenStream 自体は、ArrayList に似ている。size() でトークン数を得て、get(int) で指定位置の Token を読み出せる。

演習(3) 字句解析の利用

トークン列から、識別子だけを “_” に置き換えた（“正規化”した）ソースコードを出力するプログラムを作成せよ。

- 演習(2)と同じ JavaLexerExample クラスが素材プログラム。
- トークンを表現するクラス Token は、トークン種別を返す getType メソッドを持つ。戻り値が Java9Lexer.Identifier と等しい値であれば、そのトークンは識別子である。
- Token は行番号情報や桁位置を持っているので、それを使えばキレイに成形することが可能。ただし、今回はそこまで考慮しなくてもよい。
- Example のファイルをコピーしていくつか識別子を変更するなどして、正規化結果が変わらないことを確認せよ。

演習(2), (3) の内容の研究における応用

- プログラム中で使われているコメントやリテラルの列挙
  - 論文の Online Appendix として公開しているツールが、演習(2)の内容相当をもう少し細かく作りこんだものに対応する： Hata et al.: 9.6 Million Links in Source Code: Purpose, Evolution, and Decay. Proc. ICSE 2019.

- コードクローン検出
  - 演習(3) の内容がこれに相当する
  - 他人にコードをコピーして、変数名を置き換えた程度なら、先ほどの識別子の “正規化” によって同一テキストになる（SHA-1 ハッシュなどを使えば同一性の確認は簡単）
  - Longest Common Subsequence (いわゆる diff 計算) などを適用すれば、２つのソースファイルが「識別子を無視すると」同一か計算可能になる
    - Kamiya et al. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 2002.

構文解析

- 登場した “Identifier” でも、パッケージ名、クラス名、import 文など、色々ある
  - その字句がどの文に所属し、どのような役割を持つかは、前後の構文を見なければわからない

構文解析の実行

先ほどの CommonTokenSteram

```
Java9Parser parser = new Java9Parser(tokens);
CompilationUnitContext c = parser.compilationUnit();
```

構文解析結果 = 構文木 (“Context” オブジェクト群)

```
(compilationUnit
  (ordinaryCompilation
    (importDeclaration
      (singleTypeImportDeclaration import
        (typeName
          (packageOrTypeName …
    (importDeclaration …
    (typeDeclaration
      (classDeclaration
        (normalClassDeclaration
          (classModifier public)
          class
          (identifier JavaParseExample)
          (classBody …
```

左の出力は、ANTLR のテキスト出力機能（以下のコード）で得られたものを手動で成形したもの。

```
System.out.println(
  Trees.toStringTree(c, parser));
```

個別のノード名は、 Java9.g ファイルの構文内部の名称に対応する。

構文解析結果は木構造のオブジェクト

- 各ノードは children として子要素のリストを持つ
  - c.ordinaryCompilation() のように、文法定義ファイルでの名前でも子ノードを参照できる

- 特定の種別のノードだけを訪問する場合は Java9BasicVisitor を継承して、目的のノードに到着した時点で処理を行えばよい
  - visitCompilationUnit() などの visit 系メソッドをオーバーライドして，そのノードに到達したときの処理を記述する
  - c.accept(visitor); のように Visitor を引き渡して実行を開始する
    - c は自分のノード種別に対応するメソッド，たとえば visitor.visitCompilationUnit() を呼び出す
    - visit 系メソッドの標準の実装は，c の子ノードの accept メソッドを呼び出すので，オーバーライドした場所以外は，構文木に対応する visit メソッドが順番に実行される

演習(4) メソッド宣言の認識

ソースファイルごとにメソッド宣言の数を数えるプログラムを作成せよ。

- ここから JavaParserExample クラスが素材プログラム。
- Java9BasicVisitor を継承したクラスを作成する。
- 文法上のメソッド宣言（MethodDeclaration）に対応する visitMethodDeclaration メソッドをオーバーライドし、呼び出されるたびにカウンタを +1 するコードを記述する。
- 初期値として、オブジェクトが作られた時点でのカウンタを 0 としておけば、visit が終了した時点でメソッドの個数がカウンタに入る。
- visit メソッドは戻り値の型を BasicVisitor の型パラメータで自由に指定できるが、通常（使わない場合）は Void 型を指定して、戻り値には null を使う。
- ファイルごとの値が、実際に数えた結果と合致するかを確認すること。
  - コンストラクタ宣言を数えたい場合は、visitConstructorDeclaration メソッドを使う。
  - インタフェースのメソッド宣言も数えたい場合は、visitInterfaceMethodDeclaration メソッドを使う。
  - Static Initializer もメソッド宣言とみなす場合は、visitStaticInitializer を使う。

演習(5) 内部クラスの区別

メソッド宣言の数を、内部クラス・匿名クラスの中でのメソッド宣言を除いて数えるように修正せよ。

- Java 文法上では、 ClassBody の中や特定のメソッド宣言の途中にもクラス本体（ClassBody）の出現が許容されている。
- BasicVisitor は、放っておくとどんどん子要素に訪問を続けていく。
- ある ClassBody の処理中には別の ClassBody の中には訪問を進まないように visitClassBody の内容をオーバーライドすれば、内部クラスや匿名クラスの定義範囲を visit しないようにできる。
- enum の内部は visitEnumBody が、インタフェースの内部は visitInterfaceBody が対応する。

演習(6) 構文に所属するトークンの取得

ソースファイルで宣言されたメソッドの名前を出力するプログラムを作成せよ。

- カウント処理と同時にメソッド名の情報を取り出すのが１つの方法である。
- MethodDeclarationContext オブジェクトから見ると methodHeader に含まれる methodDeclarator に含まれる identifier に対応する。
- 各ノードからは getStart(), getStop() で、元になったトークン列を取り出せる。
- コンストラクタや Static Initializer は名前を持たないことに注意。
  - 内部的には（Java のバイトコードレベルでは） <init>, <clinit> という名前が割り当てられているので、それを出力してもよいし、しなくてもよい。
- 練習なので引数情報までは含めなくてもよい。

演習(7) IF文の数え上げ

ソースファイルに対して if 文の数を数えるプログラムを作成せよ。

- 文法定義 IfThenStatement は Else が省略された if 文の形。
- 文法定義 IfThenElseStatementNoShortIf が「else が省略されない if 文」の形。
- IfThenElseStatement は Then 節の中で else が省略された if を許さないという形。もし else がないと、つづく else は Then 節内での if に結び付いてしまうため。
- 対応する visit メソッドをオーバーライドして、カウント処理を記述するところは演習(5)と同様である。
- 字句解析で列挙できる if の数を数え上げても if 文の数え上げだけは実現可能だが、たとえば内部クラスを無視するなどの条件を加えたい場合や、条件式の内容を知りたい、開始・終了行番号を知りたい場合には構文解析が必須である。

演習(8) 構文解析を使ったリテラル判定

入力されたテキストが Java のリテラルかどうかを判定するメソッドを作成せよ。

- src/test/java ディレクトリにある jp.naist.se.parser.ParserTest クラスの isLiteral メソッドとして実装すると JUnit でテストできる。
- Java9Parser の compilationUnit は、「コンパイル単位が来る」ことを前提にした解析用メソッドであるが、同様に literal() メソッドを使うとリテラルに関する構文ルールを使った解析が行える
  - Java の文法定義ファイルの各要素にほぼ対応する解析メソッドがある
- CharStreams.fromString メソッドを使うと、任意の文字列を Lexer に渡せる
- 構文解析途中にエラーが起きた（つまり構文的におかしい）場合は exception フィールドに例外が格納される

構文解析の応用

- メソッドや関数単位の情報収集、コードクローン検出
  - メソッドごとの行数や条件分岐の数を数える（サイクロマティック複雑度）などの処理が実現できる
  - 演習(8)のようにソースコード断片も処理できるので、StackOverflow に書かれた「メソッドの中身っぽい」コード片なども構文を取り出せる

- Eclipse JDT などの開発支援機能
  - たとえばメソッド宣言の折り畳み、カッコの対応の強調表示
  - 実際には意味解析を組み合わせて、さらに詳しい情報を使っている
    - 変数名などの宣言の処理、制御フロー解析、データフロー解析

実装のコスト

- あらゆる情報を集めようとすると言語によっては実装の手間が大きい
  - C/C++ はマクロ定義などで構文が崩れてうまく処理できないことがある。
  - Ruby 用文法は標準ライブラリ Ripper のものが今のところまともそう（ANTLR4 用の文法はきちんと作られていない）

- 代替ツール
  - メソッド名の一覧がほしいだけなら Universal Ctags のような軽量ツールが利用できる
  - コンパイル可能な環境なら統合開発環境やコンパイラから情報を引き出すことも可能である
  - SciTools Understand のようなメトリクス計測向け商用ツールも存在する

実行のコスト

- 正規表現検索、字句解析結果に対する検索、構文解析結果となるほど実行時間がかかるので、できるだけ簡単な方法を選びたい
  - 行単位の解析なら正規表現で十分なことが多い
  - プログラミング言語の文字列リテラルなどを取り出す場合は、字句解析に頼る
  - “{}” のような階層構造を扱う場合は構文解析を使う

- ソースファイルは「変なもの」が出てくることもあるので、勝手な仮定は厳禁
  - ほとんどのソースファイルは小さいが、きわめて大きいものもある（例: SQLite 単一ファイルパッケージ）

おわりに

- 正規表現はプログラマとして覚えておくと非常に便利
  - 特定の文字列を検索するなどの処理を素早く書ける
  - エディタ上での正規表現を使った置換での出番も実は多い
- 字句解析はコメントの除去やテキストの切り出し時に便利
  - プログラミング言語を相手にした解析は、これだけでもけっこう色々できる
- 構文解析は入れ子を含む複雑な文字列を扱うときに便利
  - 「AまたはBまたはC」などを自然に書ける
  - プログラミング言語に限らず、ルールのある言語に有効
- 文法や言語仕様を読むと、プログラマとしてのレベルアップにもつながる
  - プログラミング言語の機能を知る良い機会になる