第57回 プログラミング･シンポジウム 2016.1.8-10

DolittleからJavaScriptへのトランスパイル実行

本多 佑希[1] 長 慎也[2] 大村 基将[1,3] 久野 靖[4] 兼宗 進[1]

概要：Dolittle はプロトタイプ方式のプログラム言語であり，Class を定義せずにオブジェクトを扱えることから，中学校から大学などの多くの授業で利用されてきた．Dolittle は Java で開発され，Java アプレットにより Web での実行も可能である．教育利用においてはインストールせずに Web ブラウザから手軽に利用できることは重要な要件である．しかし，Java アプレットでは実習用の計算機に Java をインストールしたりアプレットの実行を許可したりする設定を管理者権限で行う必要があり導入の障害になってきた．そこで，CoffeeScript や TypeScript などの AltJS と同様に，Dolittle で記述されたプログラムを JavaScript に動的に変換（トランスパイル）して実行する方式を採用することで，Web ブラウザから手軽に利用できる環境を実装することにした．Dolittle と JavaScript の双方には，プロトタイプ方式であること，オブジェクトの実体がハッシュであること，動的型付け言語であること，メソッドを手続きオブジェクトの代入により定義することなど，多くの共通点がある．また，トランスパイラとして実装することにより，Dolittle のプログラムを実行できることに加え，JavaScript の各種機能を Dolittle から呼び出して利用することが可能になった．

キーワード：Dolittle JavaScript トランスパイル コンパイラ

1. はじめに

Dolittle[1,5] は教育用に設計された言語である．JavaScript のような prototype 方式のオブジェクト指向を採用することで，クラスやインスタンスなどの概念を意識せず，オブジェクト指向を学ぶことができる．

教育利用においては，ローカルにインストールすることなく Web ブラウザから実行できることは有用である．Dolittle は Java で開発されていることから，Java アプレットにより Web ブラウザで実行することが可能であった．しかし，近年では Java の実行環境を用意することが容易ではなくなってきたことから，JavaScript で実行できる Dolittle の実装を検討することにした．

実装にあたっては，JavaScript でインタプリタを実装することも考えられたが，今回は Dolittle と JavaScript の言語的な類似点に着目し，TypeScript[2] や CoffeeScript[3] 等の AltJS と同様に Dolittle を JavaScript に変換して実行するトランスパイル（source to source compile）する方式を採用した．

本稿ではある言語で書かれたプログラムを，抽象化レベルが同程度の別の言語に変換することをトランスパイルと呼ぶ[6]．TypeScript は JavaScript に静的型付けやモジュールといった概念を付加

1 大阪電気通信大学
Osaka Electoro-Communication University
2 明星大学
Meisei University
3 静岡大学
Shizuoka University
4 筑波大学
University of Tsukuba

しており，CoffeeScript は JavaScript にクラスの概念や構文の簡潔さを付加している．本研究では JavaScript に「プログラミング初学者でも扱いやすい」「教育利用に適している」といった，Dolittle の特徴を付加することを目的とする．

2. Dolittle 言語とその実装

図 1 に Dolittle の基本構文を示す [4,7]．基本的な構文は代入とメッセージ送信であり，簡潔に記述できることが特徴である．この章では，Dolittle の言語仕様について述べる．

```
プログラム ::= (文 ’ .’ )…
文 ::= [ 変数 ’=’ ] 式
変数 ::= [ 項 ’:’ ] 名前
式 ::= 単純式 | 送信
送信 ::= [ 項 ] ’!’ 電文
電文 ::= 単純式 … 名前 ( [’;’] 単純式 … 名前) …
括弧 ::= ’(’ 中置式 ’)’ | ’(’ 送信 ’)’
単純式 ::= 数値定数 | 文字列 | 括弧 | ブロック
ブロック ::= ’[’ [ ’ |’ 名前… ’ |’ ] 文 ( ’.’ 文)… ’]’
中置式 ::= 中置式 演算子 中置式 | 項
項 ::= 単純式 | 名前
```

図 1 Dolittle の基本構文

2.1 メソッド実行

Dolittle のメソッド実行（メッセージ送信）の例を示す．

```
obj ! 100 (x) (x+2) msg.
```

obj はオブジェクト，100 と (x) と (x+2) はパラメタ，msg はメソッド名である．変数名と中置式をパラメタに含める場合は括弧で囲んで記述する．メソッド実行はカスケードすることができる．

```
obj ! msg1 msg2.
```

上の例は次の例と等価であり，obj に msg1 を実行した戻り値に対して，msg2 が実行される．

```
(obj ! msg1) ! msg2.
```

2.2 ブロック

ブロックはコードを内包するオブジェクトであり，生成されたときの環境を保持している．

ブロックは単独で実行されるほか，オブジェクトのプロパティに代入することでそのオブジェクトのメソッドとして扱われる．また，後述するように，条件分岐や反復などの制御構造を記述するために用いられる．

次の例は，ブロックを実行する．

```
[...] ! execute.
```

次の例は，ブロックを 10 回実行する．

```
[...] ! 10 repeat.
```

次の例は，オブジェクト obj の msg というプロパティにブロックを代入することでメソッドを定義し，それを実行している．

```
obj:msg=[...].
obj ! msg.
```

引数は，次のようにブロックの先頭で取得する．

```
blk = [|x y| ...].
blk ! 100 200 execute.
```

繰り返しの場合は，何回目の実行かというカウンタの値が渡される．

```
[|n| ...] ! 10 repeat.
```

2.3 条件分岐

Dolittle には if や while のような制御構文はなく，ブロックを用いたメッセージ送信のカスケード実行で制御構造を表現する．

```
[...] ! then [...] execute.
[...] ! then [...] else [...]
execute.
```

これらは次のように実行される．

```
([...] ! then) ! [...]
execute.
((...] ! then) ! [...] else)
[...] execute.
```

ブロックに then が適用（`[...] ! then` が実行）されると，条件の成否等によって `True/False/Done` の中間オブジェクトが返る．

- `True ! [...] execute`
  は引数のブロックを実行する．
- `False ! [...] execute`
  は引数のブロックを実行せず undefined を返す．
- `True ! [...] else`

は引数のブロックを実行し，ブロックの実行結果を内包した Done を返す．

- `False ! [...] else`
  は引数のブロックを実行せず True を返す．
- `Done ! [...] execute`
  は引数のブロックを実行せず，自身が内包している値を返す．

2.4 オブジェクト間の継承

Dolittle はクラスではなくプロトタイプを用いてオブジェクト間の継承関係を持つ．root はすべてのオブジェクトの親 (プロトタイプ) であり，number, string, block などのオブジェクトがあらかじめ root の子として定義されている．

プログラム中で create を実行してオブジェクトを生成した場合，元のオブジェクト obj1 は生成されたオブジェクト obj2 の親になる．

```
obj2 = obj1 ! create.
```

定数や，計算結果などで暗黙的にオブジェクトが生成された場合は，string や number などのプロトタイプオブジェクトが親になる．

```
str = "abc".
x = 1 + 2.
```

2.5 プロパティの継承

オブジェクトのプロパティを参照した場合，そのオブジェクトに該当するプロパティが存在しない場合は親のプロパティを参照し，存在しない場合はさらに祖先のプロパティを参照していく．

メソッドもプロパティにブロックを代入する形で定義されているため，同様の探索が行われる．

すべてのオブジェクトの親は root であるため，root のプロパティはすべてのオブジェクトから参照可能なグローバル変数として扱われる．次の代入は，すべて root のプロパティ「x」を定義する．

```
root:x = 3.
:x = 3.
x = 3.
```

2.6 対応しているオブジェクト

表 1 に，現在 JavaScript 版で実装しているオブジェクトと定義されたメソッドを示す．

- array オブジェクトは，Array をラップしたラッパーオブジェクトである．
- string オブジェクトは，String.prototype にメソッドを追加し，メソッド拡張を行った．

表 1 対応しているオブジェクトの例

オブジェクト	メソッド	説明
array	create	配列を生成する
	size?	要素数を返す
	get	要素を参照する
	add	要素を追加する
	join	要素を連結した文字列を返す
	removepos	要素を削除する
	insert	要素を挿入する
	foreach	各要素に対してメソッドを実行する
string	add	文字列を連結する
	contain?	文字列が含まれているかを調べる
timer	create	タイマーを生成する
	interval	実行間隔を指定する
	times	実行回数を指定する
	execute	タイマーを実行する
block	then	ブロックを評価し中間オブジェクトを返す
	repeat	ブロックを指定回数実行する
	while	ブロックを繰り返し実行する
	execute	ブロックを実行する

3. トランスパイラの設計と実装

3.1 トランスパイルの方針

1 章で述べたように，今回は Dolittle を JavaScript で実装するために，インタプリタを作成して実行する代りに，JavaScript へのトランスレータを作成して実行する方式を採用した．

Dolittle と JavaScript は，構文は異なっているが，言語仕様には類似する点が多い．これらの性質を利用することで，JavaScript での実行が容易に実現できると期待した．

- プロトタイプ方式である
- オブジェクトの実体がハッシュである
- 動的型付け言語である
- メソッドを手続きオブジェクトの代入で定義

できる
- グローバルなオブジェクトは，あるオブジェクトのプロパティである
- 関数 (ブロック) は，実行された外側のスコープを持つ

3.2 トランスパイルの実装

3.2.1 ブロックの変換

ブロックは，JavaScript の匿名関数を用いて実装を行った．Dolittle においてのブロックは実行されたスコープを自らのスコープとして持つ．これは，JavaScript の匿名関数においても同様の動作をする．宣言時には実行されない点も同様である．図 2 と図 3 に，ブロックとメソッドの変換例を示す．

```
// Dolittle
[100].
[|param| param + 100].
[|param; local_param|
  local_param=100.
  local_param+param
].
```

```
// JavaScript
(function(){return 100;});
(function(param){return param!100 add;});
(function(param){
  var local_param;
  local_param=100;
  return local_param+param;
});
```

図 2 ブロックの変換

```
// Dolittle
obj ! 100 (param) "str" msg .
obj ! [func] msg.
```

```
// JavaScript
obj.msg(100,param,"str");
obj.msg(function(){return func;});
```

図 3 メソッドの変換

3.2.2 継承の実装

親子関係は，JavaScript の prototype を利用した．create メソッドが実行されると，Object.create を使用し，自分を親とする子オブジェクトを生成し，返す．親のプロパティを探しに行く仕様は，JavaScript の prototype チェーンを利用することで対応した．

3.3 制御構造の実装

3.3.1 条件分岐

条件分岐は，JavaScript の Function.prototype に then メソッドを追加し，Function オブジェクトの prototype 拡張を行うことで対応した．then メソッドを呼び出し，カスケードすることで条件分岐を行う．

3.3.2 繰り返し

繰り返しに対しても，JavaScript の Function.prototype に repeat メソッドを追加し，Function オブジェクトのメソッド拡張を行うことで対応した．繰り返し文では，引数として渡された回数分，呼出し元のブロックを実行する．その際には，呼出し元のブロックに繰り返し回数を引数として与える．呼出し元のブロックで引数を指定している場合，この繰り返し回数を引数として取ることができる．

4. 議論

いくつかの機能に関しては，言語仕様の違い等の理由から，互換性について検討を行っている．

4.1 タイマーオブジェクトの互換性

Dolittle のタイマーオブジェクトの繰り返し実行は，スレッドにより実行される．しかし，JavaScript にはスレッドが存在しない．本研究で実装したタイマーオブジェクトの繰り返し実行も，JavaScript の setInterval 関数を用いて実装したため，スレッドでの実装にはなっていない．

Dolittle のタイマーオブジェクトには，wait メソッドが存在する [4,5]．これは，呼び出し元のタイマースレッドが処理を終えるまで，メインスレッドの動作を止めて待ち合わせる．しかし，この wait

メソッドを JavaScript で実装することは困難である．

そこで，「タイマーの終了までメインスレッドを止める」のではなく，図 4 のように「タイマーの終了時に実行するメソッド」を指定する機能を提供することで，実用上は wait に近い用途を実現することを検討している．

```
clock = timer ! create.
clock ! [...] execute.
clock ! wait.
label ! "hello" create.
```

```
clock = timer ! create.
clock ! [...] execute.
clock ! [label ! "hello" create] after_ececute.
```

図 4 タイマーの wait の対応

4.2 prototype 汚染

JavaScript において，Function オブジェクトや String オブジェクト等の prototype にメソッドを追加してメソッド拡張を行うことはあまり良いものではないとされている．この問題は，ラッパーオブジェクトを設計し，提供することで解決が可能である．しかし，その場合はトランスパイルが複雑になってしまう．そのため，本研究ではメソッド拡張を行うことにした．

以下に，Function の prototype を拡張した場合と，ラッパーオブジェクトを実装した場合の，ブロックのトランスパル例を示す．

- トランスパイル前

```
// Dolittle
obj:func=[|param| self:myField
].
```

- トランスパイル後

```
// JavaScript:prototype 拡張
obj.func=(function(param){
 return this.myField;
}));
// JavaScript:ラッパー
obj.func=(new Block(function(param){
 return this.myField;
}));
```

上の obj.func を呼び出す文は以下のようにトランスパイルされる．

- トランスパイル前

```
//Dolittle
obj:func!(param) execute.
```

- トランスパイル後

```
//JavaScript:prototype 拡張
obj.func(param);
//JavaScript:ラッパー
obj.func.execute(param);
```

この呼び出しは，Block オブジェクトに execute メソッドを追加しておくことで実現可能である．しかし，Dolittle においては，次の例のような方法でもメソッドを呼び出すことができる．

```
//Dolittle
obj!(param) func.
```

これを JavaScript にトランスパイルする場合，prototype 拡張とラッパーを用いた場合では次のようにトランスパイルの方法を変える必要がある．

```
// JavaScript:prototype 拡張
obj.func(param);
// JavaScript:ラッパー
obj.func.execute(param);
```

しかし，ラッパーを用いた場合には次のような問題がある．

- execute に渡される this の値が obj ではなく，obj.func になってしまう．
- obj.func が本当の Function オブジェクトの場合は実行できなくなる（つまり，Function オブジェクトをすべてラップをする必要がある．JavaScript から生のデータを受け取るのが難しくなる．）

これらの問題を考慮して，Function オブジェクトでは prototype 拡張を行うことにした．

ただし，他の組み込みオブジェクトに関しては prototype の拡張は可能な限り行わず，ラッパーを作成することを設計の指針としている．

例えば，配列については JavaScript の Array クラスを直接利用するのではなく，ラッパーを作成した．理由としては次のような点が挙げられる．

- JavaScript の配列は，要素の添字が 0 から始まる (0-origin である) が，ドリトルの配列は，1-origin である．このため，Array クラスのメソッドを拡張してしまうと，添字の仕様が 0-origin のものと 1-origin のものが混在してしまう．
- ドリトルには JavaScript における a[i] のような，配列の要素にアクセスするための特別な構文がなく，単なるメソッド呼び出しで要素にアクセスするため，ラッパーの実装が簡単である．

このような，組み込みオブジェクトの性質を踏まえて，ラッパーを利用するか prototype 拡張をするかを臨機応変に採択する方針で設計を進めていく．

5. 現状と今後の対応

5.1 実装の状況

Dolittle を JavaScript にトランスパイル実行することで，言語仕様のコア部分を実装することができた．性能は，基本的なベンチマークでは，Firefox, Chrome 等の代表的なブラウザで実用的な速度で動くことを確認した．今後は，グラフィックスや GUI 部品などの UI 関係の拡張を行う予定である．

5.2 JavaScript で実行することの利点

Dolittle を JavaScript で実装したことで，端末に Java や Dolittle をインストールすることなく，手軽に Web ブラウザで実行できるようになった．これは教員や生徒が端末へのインストールを許されていない一般的な学校において大きな利点になると考えている．

Dolittle から JavaScript を呼び出して使えることも利点になる．今後は，Dolittle からの HTML/CSS 等を生成する機能も検討したい．

5.3 プログラムの保存方法

Web ブラウザでプログラムを記述した場合，作成したプログラムを保存する方法は検討が必要である．ひとつは Web ブラウザ内に Web Storage として保存することが考えられる．同じ端末で実行する必要があるが，容易に実現できることが利点である．もうひとつはサーバー側でファイル保存機能を提供することが考えられる．これについては，個人利用の場合は ID 登録や認証，学校利用の場合は教員の登録や授業ごとの管理などが必要になる可能性があり，引き続き検討を行う予定である．

5.4 I/O についての検討

JavaScript から，周辺装置やネットワーク，OS などへのアクセスについては引き続き検討が必要である．ローカルファイルへのアクセス，サーバーとの通信，MIDI 等を利用した音楽演奏，接続したセンサーやロボット等の外部機器との通信などは，今後の検討が必要になる．

6. おわりに

Dolittle のプログラムを JavaScript にトランスパイルする，トランスパイラを実装した．このトランスパイラを使用することで，Dolittle のプログラムを Web ブラウザで動作させることが可能になった．また，Dolittle のプログラムから，JavaScript の各種機能を使用できることになった．

本研究を進めることで，Dolittle は，より教育現場に適したプログラミング言語になる．そのため，4 章で述べた問題点の解決に努め，改良を進めていきたい．

参考文献

[1] プログラミング言語「ドリトル」. http://dolittle.eplang.jp/
[2] TypeScript : http://www.typescriptlang.org/
[3] CoffeeScript : http://coffeescript.org/
[4] 兼宗進, 御手洗理英, 中谷多哉子, 福井眞吾, 久野靖. 学校教育用オブジェクト指向言語「ドリトル」の設計と実装. 情報処理学会論文誌, Vol.42, No.SIG11, pp.78–90, 2001.
[5] 兼宗 進, 久野 靖, ドリトルで学ぶプログラミング [第 2 版], 2011.
[6] スティーブ・フェントン, TypeScript 実践プログラミング, 2015.
[7] 兼宗進, 久野靖. プロトタイプ階層を持つ教育用オブジェクト指向言語「ドリトル」. コンピュータソフトウェア, Vol.28, No.1, pp.43–48, 2011.