解析表現文法をベースにした トランスパイラフレームワークの設計と実装

加瀬 豊[1,a)] 渡邉 遥輔[1,b)] 多田 拓[1,c)] 山口 大輔[1,d)] 倉光 君郎[2,e)]

概要：トランスパイラとは，ソースコード変換による言語処理系である．今日，JavaScript と Web アプリケーションのように実行環境が言語と一体化していることが多く，その場合，プログラミング処理系を実装する重要な実現手段として注目を集めている．我々は，解析表現文法をベースとして宣言的なルール記述によるトランスパイラ実現を目指している．解析表現文法を用いることで，様々な文法のソースコードを扱うことができ，抽象構文木によるルールによりターゲット言語を柔軟に切り替えることができる．本稿では開発中のトランスパイラを報告しながら，トランスパイラフレームワークとして目指すべき方向を議論したい．

キーワード：プログラミング言語処理系，トランスパイラ，解析表現文法

1. はじめに

トランスパイラ (Transpiler, Trans-Compiler) は，ソースコード変換による言語処理系である．f2c (Fortran-to-C)[1] や 2to3 (Python2-to-Python3) などのツールに見られるように，古くからソースコード資産の利活用ののため開発されてきた．最近では，JavaScript が Web ブラウザ上で唯一の実行可能形式であるため，トランスパイラ技法は Web ブラウザ上のプログラミング言語処理系の開発では一般になっている．JavaScript のトランスパイラ言語の例としては，Closure や Coccinelle, CoffeeScript, Dart, Haxe, Nim, TypeScript, Emscripten[7] など多岐にわたる．

[1] 横浜国立大学理工学部
[2] 日本女子大学理学部
[a)] kase-yutaka-wc@ynu.jp
[b)] watanabe-yosuke-wj@ynu.jp
[c)] taku-tada-jp@ynu.jp
[d)] yamaguchi-daisuke-bf@ynu.jp
[e)] kuramitsuk@fc.jwu.ac.jp

トランスパイラ技法は，今後ともソースコード資産の増加やスクリプト言語処理系の増加により，利用が増えることが予想される．しかし，ソースコード変換自体は，アドホックなパターンマッチングに頼ったり，伝統的なコンパイラ構成法にしたがって個々に実装される．異種データベース変換 [3] の研究が理論面で盛んに行われている点と好対照をなしている．

本稿の目的はトランスパイラに対して普遍的なフレームワークの実現を議論してゆきたい．議論のスタート地点は，我々が長らく開発してきた Nez パーサの発展形 [4], [5], [6] である TPEG (Typed Parsing Expression Grammars) である．TPEG は，宣言的なアノテーションだけで，型付きの任意の構文木に変換することができる．容易に想像つく通り，構文木からソースコードに変換する処理を加えれば，トランスパイラとなる．

本稿の構成は次の通り．第 2 節では，トランスパイラフレームワークの開発を始めることになっ

た NezCC の開発とその経験を述べる．第 3 節では，ORIGAMI フレームワークの開発状況を報告する．第 4 節では，トランスパイラフレームワークの研究方向性について議論する．第 5 節で本稿をまとめる．

2. 動機

本節では，トランスパイラフレームワークの開発を始めることになった NezCC の開発とその経験を述べる．

2.1 NezCC

NezCC は，TPEG の言語非依存性を活かし，様々なプログラミング言語のパーサソースコードを出力できるパーサジェネレータである．次は，NezCC によって生み出されたパーサの出力言語の例である．低水準なシステム言語からスクリプト言語，関数型プログラミング言語まで，多種多様な言語向けのパーサを出力することに成功している．

- C, Go
- Java8, C#, Kotlin, Scala
- Python, JavaScript, Lua, PHP
- F#, Racket, Haskell

NezCC は，多言語出力を言語テンプレート方式で実現している．一旦，TPEG パーサは構文木の形式でコード化され，言語テンプレートによってソースコードに整形される．したがって，言語テンプレートを定義すれば，原理的には対応する出力言語を増やすことができる．図 1 は，Python3 用の言語テンプレート（抜粋）である．

2.2 多言語テンプレート

NezCC の目標は，可能な限り多くのプログラミング言語をサポートすることである．我々は，「最小主義」アプローチを採用し，言語テンプレートの設計を行った．

- (仮説) プログラミング言語は，チューリング完全であれば，共通する言語処理機能を備えている
- 共通する機能だけ選んでパーサを構成する

NezCC は，オリジナルの Java 版を，仮説に基づいて最小の共通機能で書き直した．

- if 文を使わない．全て 3 項演算子を用いる
- ループの代わりに再帰を用いる

JavaScript などのいくつかの言語は，最小主義の仮説どおり，ほとんど手を加えることなく，変

```
# Python nezcc file
extension          = py
comment            = # %s

# Type
String->Byte[]     = %s.encode('utf-8')
Byte[].get         = ord(%s[%s])
Byte[].slice       = %s[%s:%s]
Byte[]''           = '%s'
Byte[].quote       = '
Byte[].esc         = \x%02x

Array.size         = len(%s)
Array.get          = %s[%s]
Array.new          = [None] * %2$s

# Syntax
struct             = class %s :
init               = self.%s = %s
getter             = %s.%s
setter             = %s.%s = %s

const              = %2$s = %3$s
function           = def %2$s(%3$s):
param              = %2$s
return             = return %s

var                = %2$s = %3$s
assign             = %s = %s
if                 = if %s:
while              = while %s:
ifexpr             = (%2$s if (%1$s) else (%3$s))
lambda             = (lambda %s : %s)

and                = %s and %s
or                 = (%s) or (%s)
not                = not (%s)
true               = True
false              = False
null               = None
```

図 1 Python3 用テンプレート

```
function many7(px,f){
  var pos = px.pos
  var treeLog = px.treeLog
  var tree = px.tree
  var state = px.state
  while (f(px)){
    pos = px.pos;
    treeLog = px.treeLog;
    tree = px.tree;
    state = px.state;
  }
  return back7(px,pos,treeLog,tree,state);
}
```

図 2 While version of `many7` function in JavaScript

換できた．しかし，多くの言語では，最小主義の記述力であっても変換できなかった．

- Java 版 – オリジナル
- C – malloc/free, プロトタイプ宣言，配列宣言
- Python – 型アノテーション, `switch`/`case` サポートなし
- Scala – *null* なし (`Option[T]` 型が必要)
- Perl – 変数名の変換が必要
- Go – 条件式が存在しない
- F# – 再帰関数には 'rec' 修飾子が必要．暗黙的な型変換がない
- Lua – 配列の開始位置が 1 から
- Racket – グループ化の (e) が使えない．while ループが完全にない
- Haskell – ステートモナドが必要．レコード上の名前重複が使えない．
- Rust – 所有権 (ownership) を明示的にわたす必要がある

表 1 は，各言語のサポートする言語機能一覧をまとめたものである．NezCC の開発経験から，共通する言語機能はほとんど期待できないことがわかる．最終的にサポートしている言語機能にあわせ，複数の実装ベースを用意すること多言語パーサジェネレータを構築した．

```
(define (many7 px f)
  (define pos (get-field _pos px))
  (define treeLog (get-field _treeLog px))
  (define tree (get-field _tree px))
  (define state (get-field _state px))
  (if (f px) (begin (many7 px f))
    (begin (back7 px pos treeLog tree state)))
)
```

図 3 Recursion version of `many7` function in Racket

3. ORIGAMI

ORIGAMI は，当初，NezCC の言語テンプレートの部分を一般化し，パーサジェネレータ以外への応用も視野に入れて開発している枠組みである．

- TPEG に由来する ソースコードから

3.1 ソースコードから構文木へ

ORIGAMI は，純宣言的な木構築が自由に可能な TPEG をベースにしたツールである．TPEG は，旧 Nez オープングラマーの記法を整備した純宣言拡張した PEG[2] である．純宣言的拡張とは，セマンティックアクションなどのコード埋め込みを行わず，自由な形式な木構造データとして解析できる．

図 3.1 は，Nez 記法による構文定義の例である．PEG は，EBNF のように非終端記号と正規表現によく似た解析表現で構文パターンを記述する．つまり，*Expression* は *Numebr* もしくは *Variable* という意味になる．*Number* の定義に着目すると，`[0-9]+`は 1 個以上の digits の繰り返しの意味になる．Nez 記法は，パターンマッチした文字列のうち，木構造のノードとする部分を`{ }` のように囲む．`#IntExpr` はノードを識別するタグである．

図 3.1 は，C 言語風の If 文の構文定義である．まず全体を*#IfStmt* タグの木ノードとして囲んでいることに注意したい．内部の要素は，それぞれ *cond*:, `then`:, `else`: とラベル付けされた子ノードとして，それぞれ *Expression* や *Statement* パターンを受け付ける．`if` 文の `else` 節は省略可能である

表 1 各言語のサポートする機能一覧

Language	constructor	λ 式	ifexpr	iteration	switch	null	mutation
C	malloc	x	o	o	o	o	o
C#	new	o	o	o	o	o	o
F#	record	o	o	o	match	x	o
Go	new	o	x	o	o	o	o
Haskell	record	o	o	x	match	x	x
Java8	new	o	o	o	o	o	o
JavaScript	function	o	o	o	o	o	o
Lua	record	o	x	o	x	o	o
Perl	record	o	o	o	x	x	o
PHP5/7	new	o	o	o	o	o	o
Python2/3	new	o	o	o	x	o	o
Racket	record	o	o	x	match	o	o
Scala	new	o	o	o	match	x	o
Swift	record	o	o	o	o	x	o

```
Expression = Number / Variable
Number = { [0-9]+ #IntExpr }
Variable = {
   [A-Za-z_] [A-Za-z0-9_]*
   #NameExpr
}
```

図 4 Example of Expressions in TPEG

ため，? で省略可能となっている．

```
IfStmt = {
   "if" "(" cond: Expression ")"
   then: Statement
   (else: Statement)?
   #IfStmt
}
```

図 5 Example of If Syntax in Nez

ORIGAMI の利用者は，TPEG を用いることで，入力ソースコードをタグとラベル付けされた半構造的な抽象構文木に変換できる．

3.2 構文木からコード

ORIGAMI のフロントエンドは，TPEG による抽象構文木の構築である．バックエンドは，抽象構文木からコードに再変換する部分であり，こちらがメインである．

木からコードに再変換するのは，コードマップという規則で与えられる．コードマップは，２つのことができる．

- 型付け規則により，入力言語の仕様（解釈）を与えること（オプショナル）
- 変換規則により，出力言語の構文を変えること

コードマップは，ユーザが定義可能な変換ルールである．次は，#IfExpr に対するコードマップの例である．

```
#IfExpr
   = if ${1} then ${2} else ${3}
```

${1} は，抽象構文木の最初の部分木を表す．ユーザは、自由にコードマップを修正し、ソースコードを変換できる．

```
#IfExpr
   = ${1} ? ${2} : ${3}
```

解釈はビックステップの評価規則というよりも，表示的意味論によって意味を与えるのに近い．

$$\frac{[\#\mathsf{IfExpr}\ e_1\ e_2\ e_3]}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3} \quad (\texttt{\#IfExpr})$$

コードマップは，多相性が認められて，型付け

することもできる．Bool は論理値型，a は型変数である．

```
#  IfExpr: (Bool, a, a) → a
   = if ${1} then ${2} else ${3}
```

型付けルール (*#IfExpr*:: (Bool, a, a) → a) は，もう少しフォーマルにかけば次のような型規則に相当する．

$$\frac{\Gamma \rhd e_1 : \mathsf{Bool} \qquad e_2 : a \qquad e_3 : a}{\Gamma \rhd [\#\mathsf{IfExpr}\ e_1\ e_2\ e_3] : a}\ (\texttt{\#IfExpr})$$

型環境 Γ において，e_1 が Bool 型, e_2 と e_3 が a 型を持つならば，$[\#IfExpr\ e_1\ e_2\ e_3]$ は a 型を持つ．ここで a 型は型変数である．

具体的にみていく．例えば，型変数 a の代わりに空型 () に変更すれば，式ではなくステートメントとして解釈される．

```
#IfExpr:: (Bool, (), ()) → ()
   = if ${1} then ${2} else ${3}
```

3.3 共通構文木

ORIGAMI は，このようなコードマップを文法開発者にカタログとして提供する．表?? は，ORIGAMI が提供しているカタログの例である．トランスパイラ開発者は，ORIGAMI の提供するルールを選んで，TPEG の構文木を構築する．カタログから選んだ構文木であれば，あとは ORIGAMI が出力言語に最適なコードを生成してくれる．

4. 議論

我々は，NezCC 多言語パーサジェネレータの開発経験にもとづき，ORIGAMI トランスパイラフレームワークの開発を進めている．

ORIGAMI の利点は，TPEG の文法記述力による入力言語の自由度が高さである．共通構文木の上で，様々な入力言語を簡単にトランスパイルできることになる．一方，入力言語の文法を切り替えることにどれだけの有用性があるかは慎重に考察しなければならない．なぜなら，ORIGAMI の実装でも述べたとおり，抽象構文木と言語テンプレートへの対応付けは型付けが必要になるためである．構文木の型付けは入力言語の型システムに強く依存しているため，フレームワークで一般化することは難しい．この点を考えると，ORIGAMI は型付き構文木のソースコード変換フレームワークになる．

木パターン	言語機能
#FuncDecl[*name type? param, body*]	関数定義
#LetDecl[*name type? value*]	定数定義
#VarDecl[*name type? value?*]	変数宣言
#IfStmt[*cond* then else?]	if 文
#WhileStmt[*cond body*]	while 文
#ForEachStmt[*init iter body*]	for 文
#BreakStmt[*label?*]	break 文
#ReturnStmt[*expr*]	return 文
#TryStmt[*body catch? finally?*]	try-catch 文
#ThrowStmt[*expr*]	throw 文
#AssignExpr[*name expr*]	代入
#IfExpr[*cond* then else?]	条件式
#Infix[*left right*]	2 項演算子
#Unary[*recv*]	単項演算子
#ApplyExpr[*param*]	関数適用
#MethodExpr[*recv param*]	メソッド適用
#GetterExpr[*recv name*]	Get field
#GetIndex[*recv param*]	Get index
#CastExpr[*recv*]	Casting
#NameExpr[]	変数
#NullExpr[]	Null
#TrueExpr[]	True
#FalseExpr[]	False
#IntExpr[]	整数値
#ArrayExpr[]	配列リテラル

表 2 共通構文木

もう少しフレームワークの守備範囲を広げる鍵は，具体的なトランスパイラ言語（もしくは型システム）を定義することである．しかし，ここで新たな疑問が生じる．そもそも，トランスパイラ言語とはどのような言語であるべきか？何をゴールにするべきか？　トランスパイラならでは言語機能はありえるのか？

4.1 Konoha6

我々は，ORIGAMI の開発と平行して，Konoha6 と呼ぶ新しいトランスパイラ言語を設計している．

```
assume coin : uint256
assume MyToken : { coinMap }

@contract(MyToken)
MyToken(initialSupply) = {
  update(coinMap, msg.sender, initialSupply)
}

@contract(MyToken)
transfer (to : address, coin) = {
  require (coinMap[msg.sender] >= coin)
  decrease(coinMap, msg.sender, coin)
  increase(coinMap, to, coin)
}
```

図 6 Konoha6 によるスマートコントラクト（仮想コイン）

目標は，NezCC と同等の記述力，つまり多言語 TPEG パーサを出力可能な多言語サポートである．次は，主な Konoha6 の設計方針である．

- 静的型付き
- Minimum な関数型言語
  - かけないことがあっても仕方がない
- デコレータ
- インラインコード変換

図**??**は，ブロックチェーン言語 Solidity へのトランスパイルを実現する開発中の Konoha6 のサンプルコードである．出力言語のコンストラクタにあわせて，*@constract* などのデコレータで出力形式を調整できるように工夫をしている．

5. むすびに

トランスパイラとは，ソースコード変換による言語処理系である．本稿では，解析表現文法をベースとして宣言的なルール記述によるトランスパイラ実現を目指し，様々な文法のソースコードを扱うことができ，抽象構文木によるルールにより出力言語を柔軟に切り替えることができる．

現在，ORIGAMI は Python ベースの新しい実装でオープンソース開発が進められている．より汎用的にトランスパイラフレームワークとして提供できることを目指してゆきたい．

参考文献

[1] Feldman, S. I., Gay, D. M., Maimone, M. W. and Schryer, N. L.: Availability of F2C&Mdash;a Fortran to C Converter, *SIGPLAN Fortran Forum*, Vol. 10, No. 2, pp. 14–15 (online), DOI: 10.1145/122006.122007 (1991).

[2] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, New York, NY, USA, ACM, pp. 111–122 (online), DOI: 10.1145/964001.964011 (2004).

[3] Kolaitis, P. G., Panttaja, J. and Tan, W.-C.: The Complexity of Data Exchange, *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, New York, NY, USA, ACM, pp. 30–39 (online), DOI: 10.1145/1142351.1142357 (2006).

[4] Kuramitsu, K.: Nez: Practical Open Grammar Language, *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, New York, NY, USA, ACM, pp. 29–42 (online), DOI: 10.1145/2986012.2986019 (2016).

[5] Kuramitsu, K.: A Symbol-Based Extension of Parsing Expression Grammars and Context-Sensitive Packrat Parsing, *Proceedings of ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017 (2017).

[6] Yamaguchi, D. and Kuramitsu, K.: CPEG: A Typed Tree Construction from Parsing Expression Grammars with Regex-Like Captures, *Proceedings of the 34th Annual ACM Symposium on Applied Computing*, SAC '19, New York, NY, USA, ACM, p. (to appear) (2019).

[7] Zakai, A.: Emscripten: An LLVM-to-JavaScript Compiler, *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, New York, NY, USA, ACM, pp. 301–312 (online), DOI: 10.1145/2048147.2048224 (2011).