GreenTeaScript: ユニバーサルスクリプトにむけて

井出 真広[†1]　関口 渚[†1]　松村 哲郎[†1]　倉光 君郎[†2]

本稿では、開発中の GreenTeaScript について紹介する。GreenTeaScript は、インライン・トランスパイラ技術を用いたスクリプト言語の実装で、ソースコード変換において様々なプログラミング環境においてスクリプト処理を可能にすることを目指している。本発表は、最初のステップとしてのインライン・トランスパイラ開発によるインライン・トランスパイラの開発を報告する。

GreenTeaScript: Toward Universal Scripting

MASAHIRO IDE[†1]　NAGISA SEKIGUTCHI[†1]
TETSUROU MATSUMURA[†1]　KIMIO KURAMITSU[†2]

This paper presents our ongoing implementation of GreenTeaScript. GreenTeasScript has been adopted an inline transpiler technique in order to run scripts on top of various programming languages. This presentation focuses as the first step attempt on self-hosting implementation of inline transplier.

1. はじめに

今日、ソフトウェア開発において、Perl, Python, Ruby, JavaScript, Lua など、様々なスクリプト言語は、Web アプリケーション開発、組み込みソフトウェア開発、ビックデータ処理など様々な分野で広く利用されるようになっている。これらのスクリプト言語の特徴は、動的型付き言語であり、プログラマは型宣言をしなくてよい代わりに、実行前に型検査が行えない問題があった。スクリプトのコード量が大きくなるにしたがい、ソフトウェア工学的な品質管理（リファクタリングやテスト負担）が難しくなることが問題視されてきた。

我々は、独自に静的型付けスクリプト言語 Konoha [1]を設計し、オープンソース公開を行ってきた。同様に Google 社が Dart 言語など、静的型付きスクリプト言語の開発や提案は増えている。しかし、新しいプログラミング言語やスクリプト言語を創造すると、既存のライブラリ資源が活用できなくなり、世の中に受け入れられにくいものであった。たとえば、JavaScript は Web 開発、R は統計処理など、他の言語に置き換えられないライブラリ群があり、言語文法に不満があってもよりライブラリの活用が重要となっている。

トランスパイラ(Transpiler) は、ソースコード変換をベースとしたプログラミング技法である。近年、LLVM 中間言語から JavaScript への変換，CoffeeScript から JavaScript へのスクリプト変換などにおいて、実用性が認められている [2, 3] 。我々は、トランスパイラ技術を応用することで、文法の自由度とライブラリの活用を高める言語設計を目指している。具体的には、スクリプト言語処理系 A は、ライブラリとしてトランスパイラを利用し、あるプログラミング言語 X をソースコード変換(X → A)し、変換されたスクリプトを Eval して実行する。

GreenTeaScript プロジェクトは、任意のプログラミング処理系で動作可能なトランスパイラの実現を目指している。我々は、制限された Java (RJava)でトランスパイラを作成し、具体的には、GreenTea 言語でトランスパイラを記述し、トランスパイラを用いてトランスパイラ自体を生成することを目指している。我々が当面ターゲットと考えているプログラミング言語は、Java(JVM), JavaScript, Python, Lua, R, C, Haskell である。

GreenTeaScript プロジェクトのもうひとつの目標は、トランスパイラのパーサに対し、文法拡張可能な仕組みを追加することである。文法拡張することで、様々なスクリプト言語文法のスクリプトを変換し、最終的にユニバーサルな文法変換が可能になる。

本稿の構成は、以下のとおりである。第 2 章では概要を述べる。第 3 章ではコード生成器について述べる。第 4 節では字句解析器について述べる。第 5 節は例を述べる。第 6 節は本論文をまとめる。

2. GreenTeaScript

本節では、GreenTeaScript を構築する上で採用した実装方針について述べる。

2.1 アプローチ

GreenTeaScript は、インライン・トランスパイラ技術の上で構築される。インライン・トランスパイラは、ある言

語処理系の上でコード変換を行い、変換されたコードをそのまま評価して実行する処理系である。図 1 は、インライン・トランスパイラの概要を示している。

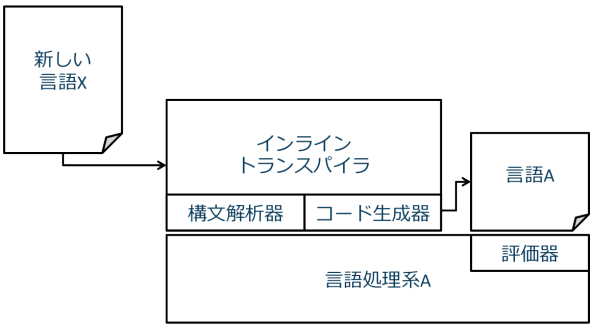

図 インライン・トランスパイラ

我々は、複数のプログラミング処理系で動作するトランスパイラの開発を目指している。本稿を執筆時点で対象と考えている言語は次のとおりである。

- Java (JVM)
- JavaScript
- Python
- Lua
- R
- C
- Haskell

最終的な目標は、任意のスクリプト言語を対象に考えているが、上記の処理系はプログラミング言語としての特徴がある処理系となっている。

我々は、複数環境のトランスパイラ開発を簡単にするため、コード生成器を抽象化し、トランスパイラによるコード生成によってトランスパイラの開発を行う。これをマルチターゲット・コード生成器と呼んでいる。また、最初のトランスパイラ実装言語として、制限された Java(RJava)を用いている。

我々は、同時に入力言語の自由度を高めるため、構文拡張可能なパーサの開発を提供する．詳細については 4 節にて述べる．

2.2 RJava

RJava は、GreenTea Transpiler 実装言語として導入された言語である。原則、Java 言語のサブセットであり、機械処理可能なアノテーションがつけられている。

実際には Java 言語を用いて言語処理系を記述しているが、次に示す機能を制限している。

- メソッドのオーバーロード禁止
  単純な変換では動的型付けを採用する言語への変換が困難になるため
- インターフェースを使用しない（多重継承を行わない）
- メンバ変数の変数名のみでのアクセス禁止
  変数名の名前解決の際に this や self などを記述することが必須の言語があるため
- 異なるスコープにおけるローカル変数の再定義
  言語間でレキシカルスコープや関数スコープなど異なるスコープを持つため
- アクセス修飾子を省略しない
- グローバル変数の禁止

また、ターゲット言語への変換処理には正規表現によるパターンマッチによる変換を用いた。また、パターンマッチに構文解析が不要になるよう RJava では一部構文にアノテーションを用いて単純な文字列処理になるよう文法設計を行っている。追加する記法はすべて Java のコメントなので、Java のコードとしても正しく動作する。コメントによるマークアップの一覧は次の通りである。

- Java 固有コード //ifdef JAVA ~ //endif JAVA
  //ifdef JAVA と//endif JAVA に囲まれた部分は、Java に依存したコードであるとして変換時に消去される。
- ローカル変数宣言 /*local*/
  /*local*/int n = 0; の形で型名の前につける。シンボルが 2 つ並ぶ構文は変数宣言のほかに関数宣言（int func(){...}）とフィールド宣言があるが、/*local*/のコメントによって識別が容易になっている。
- フィールド宣言 /*field*/
  ローカル変数宣言と同様に型名の前につける。ローカル変数宣言や関数宣言と区別するために必要である。
- キャスト /*cast*/
  int n = (/*cast*/int)1.5; の形でキャスト演算子の中かつ型名の前につける。本手法では、組み込み型を除いて、シンボルが関数名、変数名、クラス名のどれであるのかは判別できないため、このコメントでマークしている。
- コンストラクタ /*constructor*/
  public /*constructor*/Klass(){...} の形で、クラス名の前につける。コンストラクタは言語によってクラス名と同名のメソッドにする場合や、特定の名前を使う場合など、構文の差が大きく、特別な扱いが必要であるが、Java の構文では通常のメソッド宣言と文字列的に区別することができないため、このコメントでマークしている。

また、構文の変換だけでは対応できないものとして、Java 標準ライブラリがあるが、これについては Java の標準ライブラリをラップしたライブラリを、Java と JavaScript のそれぞれで実装することで対応した。

3. 抽象構文器とコード生成

GreenTeaScript 言語は各言語にて実装された処理系により、GreenTeaScript 処理系は典型的なコンパイラと同様に、次の手順によって抽象構文木を生成し、各コード生成モジュールにてターゲット言語のコードを生成する。

なおターゲット言語として、現在、実験的に Java バイトコード、JavaScript 言語、Python 言語, Bash 言語, C 言語のコードを生成するモジュールが実装されており、また Java 言語、JavaScript 言語上で動作する処理系が存在する。

1. 字句解析、構文解析を行い、ソース言語（GreenTeaScript）から抽象構文木を生成する
2. 抽象構文木に対し最適化を行う
3. 抽象構文木からターゲット言語のコードを生成する。

3.1 GtNode

我々は、C、Java、C#、Python、JavaScript の 5 つの言語について、それぞれの文法の和集合を取ることで抽象構文木を設計した。和集合を取る際に各言語の意味論に基づいて類似する構文を共通要素として、そうでないものはそれぞれ異なる要素として扱った。

なお、抽象構文木からターゲット言語の生成を行う際、必ずしもすべての抽象構文木を任意のターゲット言語へ変換できるわけではない。例えば、Java 言語のソースコードから抽象構文木を生成し、Python 言語を生成する場合、例えば、Java 言語と C 言語にはループ文の 1 つとして for 文（for(初期化式; 条件式; 変化式) ）があるが、Python 言語には存在しないないため、 1 体 1 の対応付けとして生成処理を実装できない。

このような場合の対応を行うため、抽象構文木を、ほぼすべてのプログラム言語で表現可能な抽象構文木（基本 AST）と、基本 AST の組み合わせで表現可能な構文木(拡張 AST)の 2 つに分類し､入力された抽象構文木とターゲット言語に応じて抽象構文木を変換することで対応している。基本 AST の種類について表 1 に示す。

表 1 基本 AST の種類

種類	例	意味
定数式	(CONST int 10)	整数の定数 10
配列初期化式	(ARRAY int[] a b)	配列[a, b]を生成
変数定義式	(let a INIT (BLOCK))	BLOCKのスコープで有効な変数aを定義
ローカル変数操作	(getlocal A a)	変数aの値を取得
メンバ変数操作	(getmember int (getlocal A a) b)	a.Bの値を取得
配列操作	(getindex int (getlocal int[] a) 10)	a[10]の値を取得
単項演算	(not (getlocal int a))	!a
二項演算	(+ (getlocal int a) (getlocal int b))	a + b
関数適応式	(apply int f ARG1 ARG2)	関数f(ARG1, ARG2)を呼び出す
If文	(if void COND THEN ELSE)	if(COND) THEN else ELSE
Return文	(return void)	関数から抜ける
While文	(while void COND BODY)	While(COND) BODY
Jump文	(break void)	ループを中断する

```
// (a) GreenTeaScript ソースコード
for(int i = 0; i < 10; i++) { print(i); }

;; (b) 構文解析直後のAST
(let a (const 0)
        (Ext:for (< (getlocal i) (const 10)
                (Ext:PrefixIncrement (getlocal i))
                (apply print (getlocal i)))))

;; © Pythonコード生成用AST
(let a (const 0)
        (while (< (getlocal i) (const 10) (
                (apply print (getlocal i))
                (setlocal i (+ (getlocal i) (const 1))
                ))))

// (d) Javaコード生成モジュールで生成されるコード
int i_0 = 0;
for(; i_0 < 10; i_0++) {
        print(i_0);
}

// (e) Pythonコード生成モジュールで生成されるコード
i_0 = 0
while (i_0 < 10):
  print i_0
  i_0 = (i_0 + 1)
```

図 1 GreenTeaScript 処理系によるプログラム変換例

3.2 コード生成

本節では前述の例で挙げた GreenTeaScript プログラムをターゲット言語へ変換例を図 1 に示す．まず入力されたプログラム（図 1-a）は GreenTea パーサによって，構文木に変換される（図 1-b）．構文解析直後の構文木は，基本 AST と拡張 AST が混在した構文木となっている．

for 文は拡張 AST に分類され､for 文が存在しないターゲット言語への変換の際には if 文と while 文など基本 AST を組み合わせた AST に変換される。最後に Java 言語、Python 言語をそれぞれターゲット言語コード変換によりコードが生成される最終生成コードを図 1-c, 図 1-d に示す。

4. GreenTea パーサ

GreenTea パーサは，字句解析，構文解析，型付けにフックポイントを入れた構文拡張可能なパーサである．本節では，GreenTea パーサの概要について述べる．

4.1 概要

GreenTea パーサは，通常のパーサと同じく字句解析，構文解析のあと，型付けとコード生成を行う．このうち，字句解析，構文解析，型付けのステージにおいて，フックポイントが用意されており，パーサの処理は組み込み文法の解析も含め，フックポイントを経由して行われる．ユーザーは，フック関数を差し替えることでパース処理を自由に変更することができる．図 2 は，GreenTea パーサの概要と

処理の流れを示している．

続いて，字句解析，構文解析，型付けのフック関数を説明する．

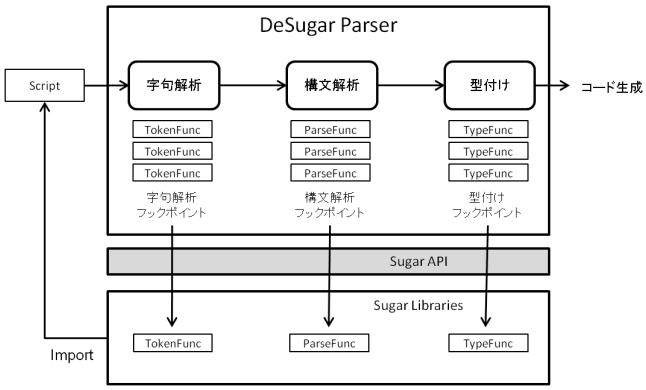

図 2 GreenTea パーサの概要と処理の流れ

4.2 字句解析

GreenTea パーサは､受け取ったソースコードを字句解析にて字句列に分割する．ソースコードの文字は，40 種類の GreenTea 内部コードに分類され, それぞれの内部コードに対応するフック関数が登録されている．

字句解析のフック関数は，TokenFunc と呼ばれる．TokenFunc は，次のインターフェースで定義される．

```
int TokenFunc (
    TokenContext,
    String Source,
    pos)
```

ここで第 1 引数 TokenContext は，字句解析器のコンテクストであり，第 2 引数 Source は字句解析の対象となるソースコード, 第 3 引数 pos は読み込みを行う位置である．切り出した文字列を字句オブジェクトに設定し，消費した文字数を返す.

GreenTea パーサの特徴は，TokenFunc を多重に登録することができる点である．GreenTea パーサは，まず最後に登録された TokenFunc を実行し，正しく字句をマッチして切り出せなければ，返り値は-1 を返す．すると，ひとつ前に登録されている TokenFunc を実行し, マッチするまで繰り返す．このような多重登録により，既に登録された字句解析のルールを一部再利用しながら，字句解析を拡張することができる.

4.3 構文解析

GreenTea パーサは，字句の列を受け取り，構文木を生成する．このとき，字句列のパターンから抽象構文木を構成するフック関数が呼ばれる．この関数のことを，ParseFunc と呼ぶ.

ParseFunc は，次のインターフェースで定義される.

```
SyntaxTree ParseFunc(
    NameSpace,
    TokenContext,
    LeftTree,
    Pattern)
```

ここで第 1 引数 NameSpace は型環境を含めたパーサ情報，第 2 引数 GtTokenContext はパースする字句列，第 3 引数 LeftTree は構文解析済の左側の構文木, 第 4 引数 Pattern は構文解析パターンである. 返り値は構文された木であり、このとき null を返すと，字句解析と同じく，多重登録された次の関数が呼び出される.

4.4 型付け

GreenTea パーサの構文解析器は, 型付けされていない構文木を生成する．このとき，構文解析器は，各構文木のノードごとに，型付けを行う関数を登録する．この関数を TypeFunc と呼び，次のインターフェースを持つ.

```
Node TypeFunc(
    TypeEnv Gamma,
    SyntaxTree ParsedTree,
    Type ContextType);
```

ここで第一引数 Gamma は，型環境であり、第 2 引数は型付けされていない構文木、第 3 引数 ContextType はコンテクストの型である．戻り値は型付ノードで作られた構文木である.

型付ノードの数が限られているため，型付けでは制御構造以外の文法は，関数呼び出しを表す ApplyNode ノードに変換し，ApplyNode を評価することで，構文からターゲットのモデルへの変換を実現する（糖衣構文の処理と同じである）.

5. 現状と今後の課題

本節では，GreenTeaScript を用いた応用例と，インライン・トランスパイラを用いた JavaScript アプリケーションの例を紹介し，また今後の課題について述べる.

5.1 D-Shell

GreenTeaScript を基盤とする新たなシェル処理系 D-Shell [5]を開発している。D-Shell はシステム運用の高信頼化を支援するためのシェル処理系として開発が進められている。D-Shell は、コマンドラインツールとの連携が容易である、という既存のシェル処理系の利点を備えるとともに、GreenTeaScript の高度な言語機能を利用することができる。

D-Shell の代表的な機能として、例外処理機能が挙げられる。コマンドの実行に失敗した際、そのエラー原因を推定し、それに基づいた例外を発行する機能を備える。コマンドのエラーの多くは、内部で発生したシステムコールエラーに起因するという点に着目し、システムコールエラー

に基づいた例外発行を行っている。図 3 に示すように、D-Shell はトレーサ、エラー原因推定機構、例外発行機構、を備えており、それぞれ、システムコール情報の取得、エラー原因の推定、エラー原因に対応する例外の発行、を行っている。例外処理を用いることで、従来のシェル処理系では記述することが困難であった、エラー原因に応じた処理を記述できるようになった。

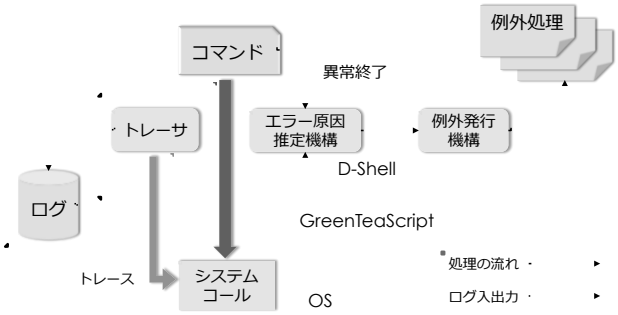

図 3 DShell 処理系の全体像

5.2 PlayGround

PlayGround[6]は，我々がコード変換器と標準ライブラリのラッパーを用いることで、図 4 に示すように Konoha 処理系を JavaScript に変換し、Web ブラウザ上で動作させることが可能となっている。

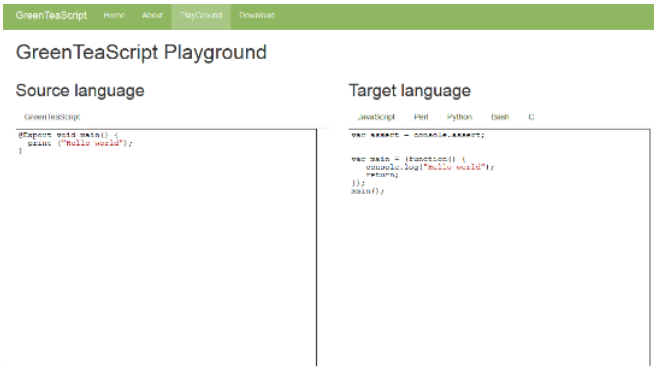

図 4 PlayGround 動作図

5.3 課題

GreenTeaScript の言語仕様は、現在、Java スタイルの文法をベースに設計された言語仕様であるが、パーサのコード変換が成功するように言語仕様を調整・変更する必要があると予想している。そのため、複数の対象スクリプト言語 (Python, Lua, R, JavaScript)を同時に進めることが重要となる。

6. むすびに

GreenTeaScript 言語処理系は、静的型付けされたスクリプト言語 GreenTeaScript を構文解析し、型検査を行ったのち、抽象構文木を生成する。本論文では、抽象構文木から、Java Virtual Machine のバイトコードを生成し、プログラムを実行することができる。この試作バージョンは、既に世界のオープンソース開発者が集まる GitHub ソースコードレポジトリにおいて公開され、利用可能になっている。

http://github.com/GreenTeaScript/GreenTeaScript

謝辞 本研究は，JST/CREST「実用化を目指した組込みシステム用ディペンタブル・オペレーティングシステム」領域の研究課題「実行時の安全性を確保する SecurityWeaver と P-SCRIPT」の一部として行われた.

参考文献

1) Kimio Kuramitsu. Konoha - implementing a static scripting language with dynamic behaviors. In Workshop on Self-sustaining Systems (S3) ACM. ACM Press, 2010.
2) Eclipse Foundation, Xtext, available from http://www.eclipse.org/xtend/
3) Alon Zakai, Emscripten: an LLVM-to-JavaScript compiler. In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion (SPLASH '11), 2011.
4) 松村哲郎, 志田 駿介, 井出 真広, 倉光 君郎. シェル文法の拡張可能な自己文法拡張スクリプトを言語, 第 94 回プログラミング研究会発表(PRO-2013-1)
5) 関口 渚, 若松 悠樹, 倉光 君郎. エラーハンドリングを備えたシェルスクリプトの提案, 第 96 回プログラミング研究会発表 (PRO-2013-3)
6) 松村哲郎, 倉光 君郎. Konoha PlayGround: Web 上で動作する多言語ソースコード変換系, 第 96 回プログラミング研究会発表 (PRO-2013-3)