Mini Konoha: 究極のスクリプト言語にむけて

倉光　君郎†

Mini Konoha は、究極の、もう少し具体的にいえば人類が開発する最後のスクリプト言語処理系、つまりもう新しいスクリプト言語は設計しなくてもよいといえる仕様の実現を目指している。本稿では、Konoha/KonohaScript の開発経験からふりかえり、言語性能、応用領域、文法やプログラミング教育、ソフトウェア工学との関係など、様々な面から究極なスクリプト言語の設計を考えてみたい。

1. はじめに

Mini Konoha は、究極の、もう少し具体的にいえば人類が開発する最後のスクリプト言語処理系となることを目指している。もちろん、これは人類最後の有人飛行機と呼ばれた F104 に倣った目標設定であり、F104 が開発された 1950 年代以降も多くの航空機が設計開発を考えると、およそエンジニアリングにおいて究極というものはないだろう。しかし、我々が 2006 年以来続けてきた Konoha 言語の開発経験から、そろそろあらゆる要求に応えられる完成度の高いスクリプト言語処理系というものが実現できる、少なくとも技術的な素地は整ってきたと考えている。本稿では、それらについて論じたいと思う。

Konoha は、スクリプト言語に静的型付けの機構を統合することを目標としていたため、とくにアプリケーション領域を決めず、言語設計と開発を行った。そのため、様々なアプリケーション領域において、Konoha の活用を探求してきた。我々がとくに重視した領域は、組み込みシステム分野、HPC 分野への適用、プログラミング教育分野である。また、本稿ではあまり触れないが、ビッグデータ解析などのデータエンジニアリングを非常に有望な領域として調査を続けている。本稿では、HPC 分野、組み込み分野、教育応用、さらにディペンダブルコンピューティングの経験から、スクリプト言語への要望をまとめてみたい。

Mini Konoha は、これらの要望を実現するために新たに開発をはじめた第 3 世代の実装となる。既にオープンソースとして開発された Konoha, KonohaScript の実装経験を元に、パーサーからバーチャルマシンまで、全部新たに書き直している。Mini Konoha は、正式なリリースに向けてオープンソース開発の途中であり、本稿では Mini Konoha の設計思想がどのように究極のスクリプト言語を実現するか紹介する。

本稿は、以下のとおりである。第 2 節では、スクリプト言語と実行性能、コード生成に関して知見をまとめる。第 3 節では、組み込み分野での適用に関してまとめる。第 4 節では、教育分野での適用知見をまとめる。第 5 節は、JST/DEOS プロジェクトからのスクリプトの高信頼化についての知見をまとめる。第 6 節では、現在開発中の Mini Konoha の設計と実装方針についてまとめる。

† 横浜国立大学, Yokohama National University.

2. 実行性能

スクリプト言語は、従来、プログラミングしやすさを重視し、プログラミングの実行性能はあまり重視しない言語設計であった。そのため、C/C++ に比べると、10 倍から 100 倍ほどの性能差を示すことがあった。近年、Web アプリケーション開発分野を中心にスクリプト言語によるソフトウェア開発が大規模化すると、スクリプト言語処理系の性能も多くの関心が集まり、いつでも無視できる要件ではなくなった。

2.1 型付けと性能

従来、スクリプト言語は、動的型付けを採用した動的言語として実装されてきた。動的型付けは、実行時に型チェックが必要となり、これが実行時の性能低下の主因とみなされることもあった。しかし、Self /StrongTalk の開発経験により、動的言語でも適切な技法を導入することで、相当に実行性能を向上させられることが知られていた。実際、Google 社の Chrome ブラウザで採用されている V8 JavaScript エンジンでは、JIT コンパイラ、Hidden クラスなどの手法を導入し、動的言語であっても非常に実行性能がよいことが知られている。

Konoha/KonohaScript は、静的型付けを採用しており、「処理系が速いのは当然」という評価をされることも多かった。実際は、スクリプト言語処理系の性能は、インタプリタの方式、スタックの構造、オブジェクトのメモリ構造、メモリ管理方式など、複数の要因から決定されるため、それほど当然の結果ではない。

図 1 は、Konoha と各種スクリプト処理系との実行結果の経験である。C/C++の性能を 1 としている。Konoha (インタプリタ)は、Ruby, Python, Lua と比較したとき、平均的によい性能を記録しているが、Konoha (JIT)と V8 を比較したとき、V8 の方がよい性能を示すベンチマークも少なくない。これらから、型付けはスクリプト言語処理系の性能の決定的要因とはいえないことがいえる。

2.2 JIT コンパイラ技術

近年、JIT コンパイラ技術はより一般化し、多くのスクリプト言語でも導入が進められている。加えて、LLVM などオープンソースでよく整備されたコンパイラフレームワークも登場している。今後は、スクリプト言語であっても、インタプリタより、JIT コンパイラ技術の採用、さらに標準化された JIT コンパイラライブラリの活用が進むと考えられる。LLVM 等が採用している静的型付けの方が相性が

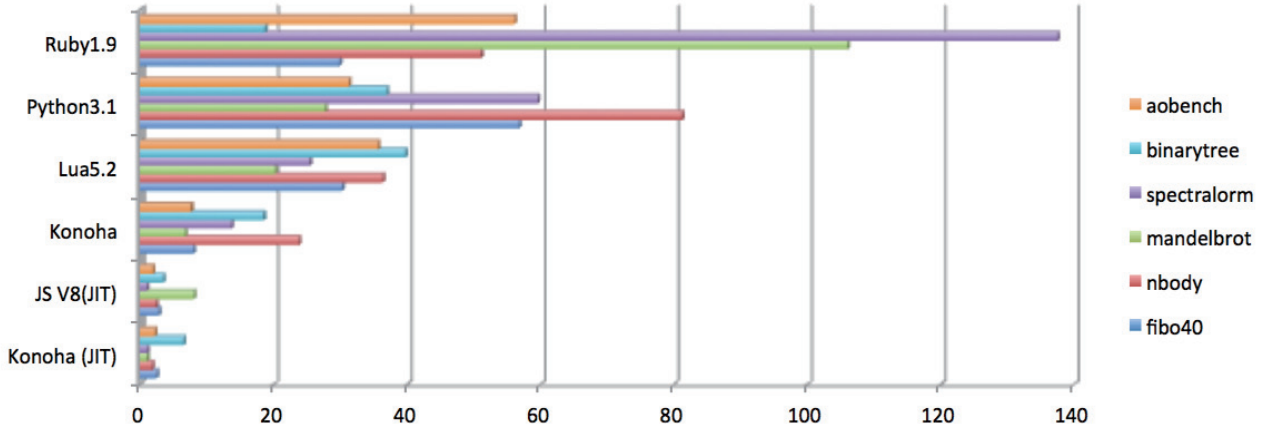

図 1 C++ の性能を 1 としたときのベンチマーク性能（smaller is better）

よい。

一方、JIT コンパイラ技術を用いてネイティブコードを生成しても、C/C++や Java との間にはオーバーヘッドが存在する。たとえば、もっとも簡単なフィボナッチベンチマークの場合、KonohaScript はソースコードレベルで C 言語と同じである。しかし残念ながら、C/C++と同等の性能を出すに至っていない。

```
int fibo(int n) {
  if(n == 1) return 1;
  return fibo(n-1) + fibo(n-2);
}
```

この原因は、メソッドを書き換えられるようにするなど、スクリプト言語特有の柔軟さを実現するためのオーバーヘッドにあるといえる。コンパイル時間は、それほど全体の実行時間に影響を与えない。

最後のオーバーヘッドをどうするかという問題は、言語仕様との調整が必要となる。もし C 言語と同じく完全にスタティックに動作を決めてしまうのなら、C 言語の性能に近づくが、そもそもスクリプト言語としての存在価値はなくなる。スクリプト言語設計と性能チューニングは、職人芸的なバランス調整が必要である。

2.3 HPC 適用

Konoha プロジェクトでは、静的スクリプト言語の設計と高速化へのチューニングに力を入れてきた経緯から、スクリプト言語における HPC 適用の可能性も調査した。ここでは、N 行 N 列の行列積計算をもとに GPGPU と MPI の活用を考えてみたい。

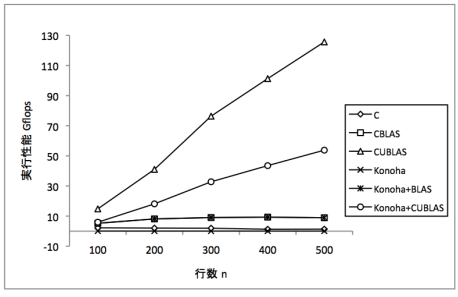

図 2 GPGPU (CUBLAS 活用)

GPGPU は、もともと 3D 画像処理用に開発された演算ハードウェアをより一般的な高性能演算に利用する手法である。OpenCL や CUDA などの計算ライブラリが知られている。もう一方、MPI は非共有メモリの分散計算ノード間の通信ライブラリであり、HPC 分野では標準的に使われている。

我々は、CUBLAS (CUDA 版 BLAS)と MPI ライブラリをそれぞれ Konoha 言語からライブラリとしてバインドして、活用できるようにした。図 2 は、CUBLAS ライブラリと GPGPU の活用したときの性能向上の効果、表 3 は非共有メモリの計算ノードに対して MPI ライブラリを活用し行列積を計算した性能評価である。

実験は、別々の時期に行われたため、そもそも直接比較することが難しいが、GPGPU はライブラリをうまく活用することができれば、インタプリタでも大きな効果が期待できる。しかし、スクリプト言語の書きやすさを活かして、GPGPU のコード生成を行うことは相当に技術的な壁がある。

MPI プログラミングの方は、台数効果（スケーラビリティ）が指標となるが、インタプリタは台数効果が高くとも、そもそもの計算ノードごとの計算性能が芳しくない。そもそも、実用的な結果とはいえない。

言語	単一ノード	48 ノード	台数効果
Ruby1.8.5	1.9Mflops	65Mflops	x35
Python2.7	7.6Mflops	76Mflops	x10
Konoha1.0(VM)	39Mflops	1.6Gflops	x42
Konoha1.0(JIT)	740Mflops	20Glops	x27
GCC4.1	1800Mflops	25Glops	x13
ICC	2Gflops	98Glops	x40

表 3 MPI プログラミング（N=1920 行の場合）

Konoha は、LLVM コンパイラによる JIT を用いたとき、GCC と比較可能な性能を示すことができた。しかし、HPC 分野では Intel C++ (ICC)が標準的に利用されており、ICC との性能差はかなりある。GCC と ICC の性能差を埋めるのは、スクリプト言語処理系の開発者の技能では難しい。しかし、将来、LLVM が HPC 向けに性能チューニングが進めば、性能向上のメリットは得られる。

我々は、GPGPU と MPI をスクリプト言語との相性を検討してみた。これらの試みは、従来のプログラミングモデルをスクリプトで置き換えてみたといえる。もし、従来のプログラミングモデルを用いるのなら、スクリプト言語を使う強い動機付けはないこともわかった。一方、並列プログラミングは様々な課題に直面しており、スクリプトの記述による新しい計算モデルの提案が可能であれば、生産性に大きく貢献できるものといえる。

3. 組み込み領域への適用

Konoha プロジェクトは、もともと TRON プロジェクトの開発経験から出発しているため、当初から組み込み分野への適用、ITRON/T-Kernel 上での動作検証を行ってきた。

3.1 組み込み分野

組み込みソフトウェア開発では、次のような特殊な要望が求められてきたため、基盤ソフトウェア技術において常に新しいフロンティア領域となっている。

- 限られたメモリ容量、メモリ性能
- 限られた MPU 性能
- リアルタイム性
- ハードウェアの依存性が高い
- OS がないこともある
- 過酷な環境で動作しなければならない
- あり得ないエラーも想定する
- 汎用ライブラリに頼らない

スクリプト言語処理系は、とくにコンピューティング資源の制約面から、それほど積極的に活用されてこなかった。近年、組み込みアプリケーションの高度化にともない、スクリプト言語の活用が検討されはじめている。ひとつは、デバイス機器の制御を目的とした軽量なスクリプト処理系である。Lua や軽量 Ruby の導入がこれらに相当する。もうひとつは、デバイス独立を実現するため、HTML5 技術の適用である。

我々がもうひとつ着目するのは、オープン性と安全性である。今後、組み込みシステムは、Cyber-Physical システムのように、開放系を含めたソフトウェア開発が増えてくると予想される。従来のセンサーやアクチュエータが扱う物理現象は、ある程度、予想の範囲内で動作することができた。しかし、ネットワークに接続された開放系は、先方のサービスは極めて変化に富み、あらかじめ全て予測しておくことができない。このとき、ソフトウェアのアップデートが必要となるが、スクリプト言語処理系を組み込むことで、開放系の変化にあわせたソフトウェアが提供可能になる。

スクリプト言語は、組み込み分野に対し、ソフトウェア開発の効率化以外にも、まったく別の付加価値、あたらしい応用を提供する可能性があると考えられる。

3.2 組み込みボードへの移植評価

近年、組み込み分野でも Android など、Linux をベースとした OS の採用が広がっている。Unix ベースで開発されたスクリプト言語は、ほとんど移植の作業なしに、組み込み Linux で動作させることができる。よりメモリやプロセッサのリソース制約が厳しい状況では依然として組み込み Linux は採用されていない。その場合は、組み込み OS 向けの移植作業、さらにはスクリプト処理系のダウンサイズが必要となる。

我々は、パーソナルメディア社製の組込み評価ボードである TBMX1 を用い、Konoha の移植評価を行った。本ボードは，T-Engine フォーラムが標準化する T-Kernel/SE が搭載 OS となっている．性能は，CPU ARM 920 シリーズ 200Mhz, 2MB ROM/16MB RAM である．

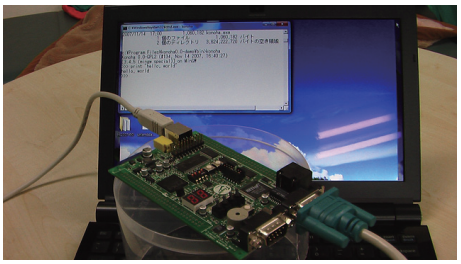

図 4. ARM920/TBMX1

Konoha ソースコードは、約 3 万行である。ARM-プロセッサ用の gcc4.3 のクロスコンパイル結果、コンパイル済みバイナリサイズは 240kb となった。パーサーを含めたスクリプト処理系の動作に必要なメモリは、200kb ほどであった。

スクリプト言語は、パーサやインタプリタなど、ソースコードを解釈し実行する開発環境が付随しているため、メモリ資源量がより厳しい資源制約となる。500kb が最低ラインとなる。

3.3 ET ロボコン走行体

ET ロボコンは、毎年、11 月にパシフィコ横浜で開催される世界最大の組み込み技術展 Embedded Technology に併設されるロボットコンテストである。組み込みソフトウェア開発の技量を競うため、ハードウェアは指定されている。指定されたハードウェアを「ET ロボコン走行体」と呼ぶ。

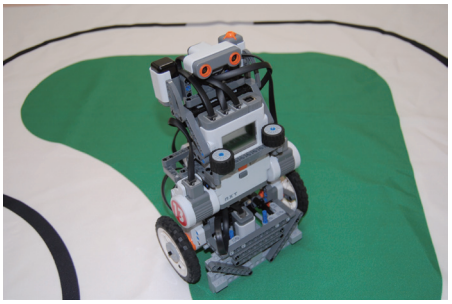

図 5　ET ロボコン走行体

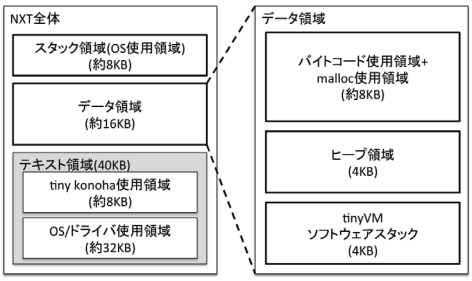

図 6　ET ロボコン走行体上のスクリプト言語処理系

ET ロボコン走行体は、Mind Storm Lego NXT をベースにしたハードウェアで、32 bit マイクロプロセッサ (ARM7) が内蔵されており，256kb のフラッシュメモリと，64kb の RAM が搭載されている。

我々は、ハードウェア環境に Konoha の移植を行った。まず、組み込み OS (Toppers)を搭載すると、スクリプト言語処理系が使えるヒープとスタックは 32kb 程度に制限される。この制約下では、パーサ機能やコード生成機能は取り除くことになる。(これは、Java と同じく、独立してバイトコードを生成することになり、スクリプト言語処理系と呼ぶのは難しい。) それでも、GC が管理するオブジェクト領域は 8kb しか確保できない。プログラミングに使えるオブジェクトが高々1000 個程度というような制約の中で、通常のスクリプト言語とはかなりメモリ制約を意識したプログラミングが必要となる。しかし、従来の組み込みプログラミングに比べ、開発イテレーションのサイクルは短くなり、開発者にとって新しいメリットは少なくない。

4. 言語文法と教育展開

Konoha は、大学の研究室で生まれたプログラミング言語として、プログラミング教育や演習からの要求は身近であり、常に教育分野への応用を検討してきた。教育分野への適応のポイントは、言語文法と演習環境となるツール支援といえる。

4.1 スクリプト言語が選ばれない理由

対話的なプログラミング環境を備えたスクリプト言語は，C 言語や Java のようなコンパイル言語に比べて習得しやすいと考える人は少なくない．これは，コーディングから実行までのサイクルが短く，コンパイルエラーによる初学者のつまずきが軽減される．実際，高校生向けの情報科目では，インタプリタ型の BASIC を用いている．しかしながら，プログラミング教育の世界では，C/C++や Java に比べると，米マサチューセッツ工科大学 (Python) や 東京大学教養学部 (Ruby) など，一部の大学で採用されることはあるが，それほど一般的な傾向となっていない．

現在のプログラミング教育では，プログラミング言語として C 言語や Java が多く採用されている．逆に，カリキュラムの一部として，プログラミング教育が導入されている場合 (つまり純粋な教養科目でない場合), C 言語や Java をまったく教えなくてもよいという大学は極めて少数である．

まず、情報系学科の場合，いくつかの科目群（コンピュータアーキテクチャ，アルゴリズム，ソフトウェア工学，コンパイラ構成法）を縦断的に学ぶため，プログラミング言語の選択はカリキュラムの視点から重要である．たとえば，オペレーティングシステムやアーキテクチャを理解するため，システム言語(C 言語)は必要となるなど制約がある．それらをふまえ，最初のプログラミングを習う言語として，C/C++, Java が選択されることが多い．一方，情報系を主体としない学科の場合は，プログラミング教育の言語選択の自由度はあがる．しかし，卒業研究などでシミュレーション，データ解析などで，プログラミング能力を必要とすることは増え，多くの教員からプログラミング教育への要望として実用性の高いスキルを求められる．その場合，C や Java など，実用性の高いプログラミング言語が選ばれることが少なくない．

4.2 C/Java 風と C/Java との違い

Konoha プロジェクトでは、スクリプト言語をプログラミング教育に導入しやすくするためのカギとして、複数の言語を学ぶのコストを軽減させ、C/C++, Java などの導入としてカリキュラムで運用しやすいようにすることと考えた。その点で、Konoha は、静的型付け言語して型宣言が必要であり、基本文法は Java から取っていたため、導入言語として使いやすいことと期待された。

我々は、過去、数年にわたり複数のプログラミング演習科目、アルゴリズムの講義などで、Konoha ベースの演習を試みてきた。その結果、スクリプト言語は、演習の進捗の早さ、学生からの積極的な取り組みなど、教育者として望ましい成果が観測された。これは、著者の本務校以外における演習成果でも同様といえる。

一方、C や Java ライクの文法というのは、微妙な文法な違いがあるため、少なからず学習者や教員側に混乱を与えることがわかった。たとえば、スクリプト言語は Python や JavaScript のように、[]でシーケンス（配列）の初期化を行えることが多く、C や Java の{}と異なる。

```
[1, 2, 3]  // JSON, JavaScript, Python 風
{1, 2, 3}  // C や Java の配列の初期化
```

このような微妙な違いが学生だけでなく、教員からも教えにくいと指摘される結果になった。また、i++ や ++i の挙動の違いに関しては、教員によっては C 言語と同じように動いて欲しいと考える場合と、そこは C 言語とは同じでない方が教えやすいと考える場合があった。このように、プログラミング教育では、プログラミングの概念を教えるよりも、言語文法が大きな争点となりやすい。

4.3 演習環境としてのプログラミング環境

プログラミング言語の演習では、言語文法以外にも演習環境も大きな争点となる。多くの大学が採用している Microsoft 社の Visual Studio は、初学者の演習用にはオーバースペックで複雑すぎるし、Emacs などのエディタを使う場合は、情報リテラシーの補完も必要となる。また、在宅演習を期待する場合は、標準的な Windows パソコンで簡単にインストールできる実行環境も必要となる。

ひとつの解決策は、Web ブラウザを活用して、Web 上でプログラミング演習を行うことである。サーバ側でプログラムを実行し結果を表示する方法や、JavaScript を演習言語として用いることも可能である。初学者は無限ループを書きやすく、無限ループのケアは難しくなる。

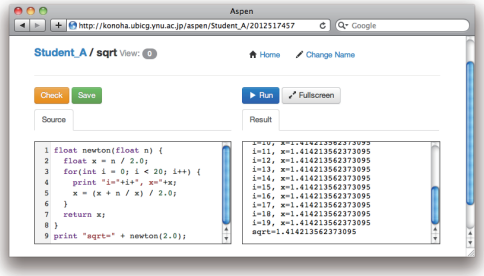

図 7 JavaScript ソースコード変換による
オンライン教育環境

我々は、図 7 に示すような Web ブラウザ上でプログラミング演習環境を実現するため、Konoha を JavaScript に変換して実行する ASPEN システムなどの開発し、プログラミング演習環境の向上を目指してきた。ASPEN システムは、短期間でより多くの演習を行う上で有用性は認められたが、エディタで編集し、実行結果を表示するというスタイルでは、コンパイル型言語と同じスタイルの演習となる。スクリプト言語らしさを活かした演習を行うためには、対話的プログラミング環境を備えた演習ツールが必要といえる。

5. ディペンダビリティと高信頼なスクリプト技術

我々は、日本科学技術振興機構「実用化を目指したディペンダブル組み込み OS 領域(DEOS プロジェクト)」の研究チームとして参加し、高信頼スクリプト処理系の研究開発を進めている。DEOS プロジェクトは、当初、高信頼なオペレーティングシステム技術の研究開発であったが、現在はソフトウェアプロセスや運用を含めたディペンダビリティの向上を論じている。要求工学やリスク分析の専門分野を超えた議論は、スクリプト言語のディペンダビリティにたいして大きな影響を与えてきた。

5.1 スクリプトの高信頼化とは何か？

ソフトウェアは、ソフトウェア開発過程のバグ（含む設計ミス）や運用時のソフトウェア老化など、様々な原因で不具合を発生する。ソフトウェアの高信頼化は、これらの不具合を発生しないことである。とくにソフトウェアのバグは、大きな関心事となるため、プログラミング言語の型理論やモデル検査、ソフトウェアテスト手法など、様々な観点から技術開発がされてきた。

最初の関心事は、スクリプト言語の言語設計がどの程度、その実行されるプログラムの信頼性に影響を与えるか？という点である。Konoha において静的型検査を導入したのは、まさにスクリプトの機械検証の実現を目指した試みといえた。

我々の静的スクリプト言語による開発経験からは、ソフトウェア工学的な定量評価は行っていないが、スクリプト言語であっても静的型検査ができることで、C/C++や Java 言語と同等なケアレスミスが検出可能となり、プログラミングの品質管理はしやすくなった。ただし、これらはソフトウェアテストの負担を軽減させる効果は期待できるもの、最終的なスクリプトの動作は実行させて検証する必要があり、型検査だけで何か信頼性を保証するものとはいえなかった。

スクリプト言語は、C/C++に比べ、より高度に抽象化されたプログラミング手段を提供している。型エラーは、仮に発生したとしてもスクリプト処理系で補足され、安全にレポートされる。そのため、プログラムのバグから予想困難な破壊的な結果を招きにくい。さらに、スクリプト言語処理系自体のクラッシュも大きな課題となるが、こちらもよくテストされ実績のある処理系であれば、今日、それほど大きな問題とならない。

DEOS 国際シンポジウムにおける海外からのディペンダビリティ技術の専門家と議論では、スクリプト言語の文法の特異さや言語仕様の複雑さが阻害要因となりえることが指摘された。これらは、エンジニアの理解不足を招き、スクリプト言語の動作に影響を与える。その点で、ディペンダブルスクリプト言語は、ディペンダビリティを向上させるため新しい言語機能を検討するよりは、シンプルな言語仕様と言語実装を目指すべきという助言をえた。

5.2 スクリプト言語設計と上流設計手法

スクリプト技術は、システム運用を支援するバッチ処理、Bourne シェルなどから発展した技術である。今日でも多くのスクリプトがシステム運用に活用されている。

これらのスクリプトは、一方、伝統的なソフトウェア工学の開発プロセス（要求分析から設計/実装、テスト）に比べると、現場のエンジニアがスクリプトを書き、それほどソフトウェアとして品質管理されていない。しかし、スクリプトは、システム運用を連携させるハブとして機能しているため、一旦、スクリプトの不具合による誤動作が起こると、システム障害は重症化するケースが少なくない。

たとえば、我々の分析したケースでは、サーバ負荷によって発生する IO ボトルネックを軽減するため、RAM ディスクを活用するスクリプトが書かれていた。しかし、RAM ディスクを用いることで、停電などの別の障害に対してはデータ消滅のより重大な障害につながることになる。

スクリプトの信頼性は、このようなケースをみてもわかるとおり、書かれたスクリプトが（バグなく）正しく実行されるかだけでなく、そもそもスクリプトの内容自体のリスクが検討されているかも重要になる。

伝統的なソフトウェア工学によれば、要求分析やリスク分析手法、それらにもとづくソフトウェア設計モデルによってカバーされ、プログラミング言語とは独立した課題といえる。しかし、スクリプト言語は、ある種のドメインに特化した記述を得意とする。スクリプトの記述の中にリスク分析を支援する記述を埋め込むことも可能である。

これらのテーマは、まだ結論に至るほどの議論は進んでいないが、スクリプト言語設計の中に、ソフトウェア工学の上流記述を取り込むことは検討に値する分野といえる。

5.3 説明責任性と障害診断

高信頼なスクリプト言語処理系は、非常に誤解されやすいことであるが、実行するプログラムを絶対に落とさないことではない。プログラムは、様々な要因によって実行できなくなることがある。むしろ、プログラムが落ちた要因を正しくレポートし、迅速にリカバリーすることを支援した方がよい。

我々は、スクリプト言語処理系を中心として、実行するプログラムが失敗する要因をフォルトマトリックスとして分類表を作っている。

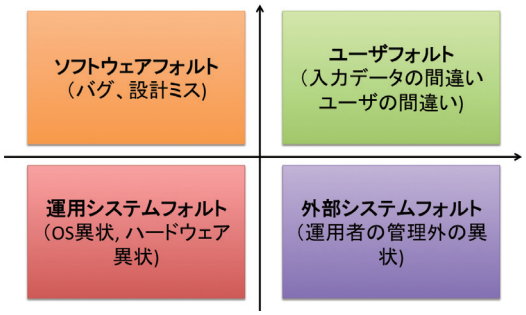

図８ フォルトマトリックス

ソフトウェアフォルト(Software Fault) - 対象とするプログラムが開発段階で混入することが要因で発生するフォルト、設計ミスやバグなど。フォルト除去には開発プロセスでの修正が必要であり、オンラインでのフォルト除去は不可能。
ユーザフォルト (User Fault) - ユーザの入力ミスや入力されたデータの不整合（エラー状態）を要因として発生するフォルト。入力データを修正することで、フォルトを除去できる。
システムフォルト (System Fault) - 言語ランタイムが動作するコンピュータシステムが、何らかの原因で通常とは異なる動作を行い、それがもとでアプリケーションのエラー状態が引き起こされるフォルト。システム状態を正常状態（初期化、バックアップ）に戻すことで、フォルトを除去できる。
外部フォルト (External Fault) - ネットワークや外部サービスなど,外部の管理ドメイン のエラー状態によって発生するフォルト．フォルトを除去するためには，外部の管理者の協力が必要となる．

スクリプトは、必ずしも実行時エラーのハンドリングが十分ではなく､システムフォルトや外部フォルトの場合は、スクリプト自体でエラーハンドリングを行うことができない。スクリプト処理系自身が、実行時エラーを検出したときに、障害原因（フォルト）を診断する技術開発を進めている。これにより、速やかに障害原因を取り除いて、スクリプト処理の継続が可能になる。少なくともスクリプトの実行が途中で止まって中途半端な状況に陥ることを防ぐことができると考えられる。

6. Mini Konoha の設計と実装

Mini Konoha は、第２節から第５節までで述べてきたスクリプトへの要望を満たす基盤として設計と実装が行われている。本節では、Mini Konoha の設計と実装方針を簡単に報告する。開発は、オープンソースプロジェクトとして行われ、開発経過はすべて以下のサイトからえることができる。

```
http://github.com/konoha-project/konoha.git
```

6.1 ソフトウェアアーキテクチャ

スクリプト言語は、ソースコードからコード生成するパーサ･コンパイラとしての機能だけでなく､プログラムを実行するランタイムも含まれている。ランタイムは、様々な実行環境に応じて再構成する必要がある。

我々は、OS 等のシステムアーキテクチャの概念を参考に、スクリプト言語のランタイム機能をモジュール化する方針とした。モジュール化することで、スクリプト言語処理系の基盤機能を切り替えられるわけである。図９は、Mini Konoha のソフトウェアアーキテクチャである。

Mini Konoha では、言語ランタイムの主要な部分は、あらかじめモジュール化され、それらは標準化された Mini Konoha API を通して書かれている。あらかじめ、複数の実装を切り替えられるように設計されている。

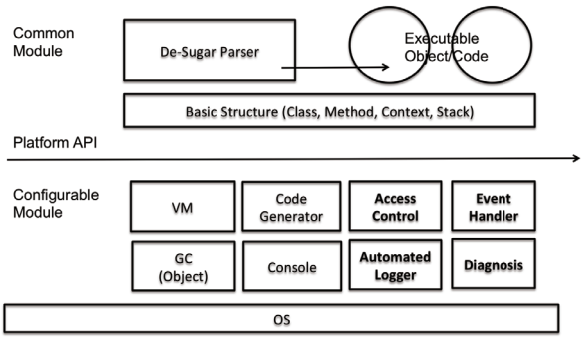

図９ Mini Konoha ソフトウェアアーキテクチャ

次のリストは、Mini Konoha のモジュール化された機能である。

- GC（メモリ管理）
- VM (インタプリタ)とコード生成
- 非同期イベント処理
- ユーザインタラクション (非 GUI)
- 多国語対応
- デバッグ支援（トレース）
- アクセス制御
- 障害診断

これらの機能は、新しい Mini Konoha インスタンスを生成するときにオプションとして指定することで、選択可能になっている。

```
minikonoha –M MSGC –M Diagnosis –M IConv
minikonoha –M TraceVM
```

実行環境とコード生成は、デフォルトのモジュール以外に名前空間ごとにユーザが切り替えられるようにモジュール設計されている。これにより、複数のターゲットに対しコード生成と実行が可能となる。

6.2 最小文法と文法ライブラリ

Mini Konoha のもうひとつの特徴は、最小限の文法セットに制限した点である。Mini Konoha は、原則として次の言語文法しかサポートしない。

- 関数定義
- 変数宣言，`if /else` 文，`return` 文
- 変数代入、関数呼び出し、メソッド呼び出し
- `boolean` 型と論理演算子
- `int` 型 と四則演算子、比較演算子
- `String` 型と文字列連結
- `Func` 型と関数オブジェクト
- `new` 演算子

Mini Konoha は、たとえば、制御構造のループも存在しない。このような最小文法セットでは、ほとんどのプログラミングシナリオで不便である。そのため、文法ライブラリの概念を導入し、言語文法を自由に拡張可能にした。たとえば、C 言語風の while 文を使いたければ、ライブラリからインポートして使うことができる。

```
import(“cstyle”, “while”);

void f(int n) {
  while(n < 100) {  // while 文が利用可能に
    ..
  }
}
```

このような while 文の拡張は、スクリプト言語自身で拡張することができるようになっている。次は、while 文を拡張する文法ライブラリの例である。まず、syntax 文でシンタックスを拡張し、続いて型付けを記述する。

```
import(“konoha.desugar”);
syntax “while” “(“ $Expr “)” $Block;

boolean WhileStmt(Stmt stmt, Gamma gma) {
   if(Stmt.typeCheckByName(”$Expr", gma,
Boolean)) {
     Block block = stmt.getBlock(“$Block”);
     return block.typeCheckAll(gma);
   }
   return false;
}
// 新しい文法を登録する
AddStatement(“while”, WhileStmt);
```

6.3 最小文法と文法ライブラリ

Mini Konoha のもっともコアの部分は、図 10 に示す DeSugar Parser である。新しい拡張された言語文法は、最小セットの AST に変換できるようなっており、Sugar API がフックポイントを提供する。

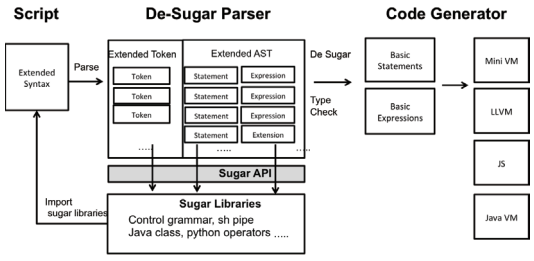

図 10 De Sugar パーサ機能

一方、コード生成は、最小セットの AST からターゲットに対して行い、コード生成自体はモジュールとして切り替えられるように設計されている。これにより、コード生成を独立させることができるように設計されている。現在、Built-In された Mini VM, JavaScript (V8), LLVM IR へのコード生成が試作されている。

7. まとめ

Mini Konoha は、Konoha プロジェクトの第 3 世代となるスクリプト処理系の実装である。実行性能、アプリケーション領域、ソフトウェア工学からの要望を取り入れながら、それらの要望を満たすことが可能な拡張性をもった基盤設計となっている。また、スクリプト言語では、ユーザが書きたい記述を認めることが必須であり、文法拡張機能を持たせたパーサを採用している。

Mini Konoha は、現在、オープンソースプロジェクトとして開発を進め、正式なリリースの準備を進めている。最終的に、本稿の題が示すとおり、究極のスクリプト言語となるかどうかわからないが、現時点で必要な要求にすべて応えられるスクリプト言語処理系の基盤となるように開発が進められている。

謝辞　本研究は，JST/CREST「実用化に向けたディペンダブル組込みオペレーティングシステム」領域の研究助成を受けて行われてきた. また、開発者としてユーザとして、Konoha プロジェクトに関わって頂いた全ての皆さんに感謝致します。

Reference

1) Kimio Kuramitsu: KonohaScript: Static Scripting for practical use. OOPSLA Companion 2011: pp. 27-28, 2011.
2) 倉光君郎「Konoha: ハイブリッドな型検査システムを備えたスクリプティング言語」日本ソフトウェア科学会プログラミング言語シンポジウム PPL2008, 2008 年 3 月
3) 倉光君郎, "KonohaScript: 静的スクリプト言語の実装と進化", 日本ソフトウェア科学会第 28 回大会,2011.
4) 井出真広，志田俊介，倉光君郎. スクリプト言語 KonohaScript における LLVM を用いた Just-in-time コンパイラの実装と評価, 情報処理学会プログラミング論文誌(PRO89), 2012.
5) 若森 拓馬、倉光君郎. 静的単一代入形式表現にむけたスクリプティング言語のバイトコード拡張, 日本ソフトウェア科学会 (JSSST)、津田塾大学　小平キャンパス、2010 年 9 月 13 日
6) 志田 駿介, 井出 真広,菅谷 みどり, 倉光 君郎. SSA 変換を用いた JavaScript 難読化コードの生成. 第 90 回プログラミング研究発表会 (PRO-2012-2), 2012
7) Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In Proc of PLDI2010, 2010.
8) 平岡佑太郎, 菅谷みどり, 倉光君郎, "KonohaScript による MPI 統合," 第 86 回プログラミング研究発表会 (PRO-2011-3), 2011/11/1, 神奈川.
9) 倉光君郎. 拡張性のある組み込みアプリケーションを実現するスクリプティング言語の開発, 情報処理学会論文誌. 51(12), pp. 2185-2194, 2010
10) 志田 駿介, 井出 真広,菅谷 みどり, 倉光 君郎. TinyKonoha: ET ロボコン向けのスクリプト処理系の簡素化. 情報処理学会組み込みソフトウェアシンポジウム 2012.
11) Roberto Ierusalimschy , Luiz Henrique de Figueiredo and Waldemar Celes. The Implementation of Lua 5.0. Journal of Universal Computer Science. Vol 11. no 5. pages 1159-1176, 2005.
12) 倉光君郎.「ユビキタス環境のためのスクリプト言語の設計」情報処理学会ユビキタスコンピューティングシステム研究会報告 (UBI16), 2007 年 11 月
13) 井出真広, 中田晋平, 倉光君郎. 「スクリフティンク言語 Konoha によるカーネル拡張」情報処理学会システム研究会, 2010 年 1 月.
14) 倉光君郎. プログラミングをはじめて学ぶための スクリプト言語の開発と教育実践. 情報処理学会 情報教育シンポジウム Summer Symposium in Setouchi 2011
15) 菅谷 みどり, 若森 拓馬, 倉光 君郎. ASPEN: 自動データ収集機能を備えた Web ベースプログラミング学習システム. 情報処理学会 情報教育シンポジウム Summer Symposium in

Setouchi 2012.
16) 倉光君郎, 大学における新しいプログラミング言語の創造, 日本応用数理学会, 国立情報学研究所, 2010
17) 平岡佑太郎, 倉光君郎. D-Intent: 原因特定できないエラーに対する分散リカバリーサービスの提案, 日本ソフトウェア科学会ディペンダブルシステム研究会 第 8 回ディペンダブルシステムワークショップ, 函館大沼プリンスホテル, 2010 年 7 月 21 日
18) 倉光君郎, 若森拓馬, 菅谷みどり, “D-Script: 分散システムのディペンダビリティ管理を行うスクリプト言語フレームワーク,” ディペンダブルシステムワークショップ&シンポジウム (DSW & DSS 2011), 2011/12/13-14, 京都.
19) Adam Dunkels : A Low-Overhead Script Language for Tiny Networked Embedded Systems, Technical Report T2006:15, Swedish Institute of Computer Science, September 2006.
20) Midori Sugaya, Hiroki Takamura, Youichi Ishiwata, Satoshi Kagami ,Kimio Kuramitsu, Online Kernel Log Analysis for Robotics Application. Journal of Information Processing, 2012
21) Takuma Wakamori. Masahiro Ide, Midori Sugaya, Kimio Kuramitsu. Reconfigurable Scripting Language with Programming Risk. Workshop on Open System Dependability 2012, in conjunction with ISSRE2012, to appear, 2012.
22) Sebastian Erdweg, Tillmann Rendel, Christian Kastner, and Klaus Ostermann. SugarJ: Library-based syntactic language extensibility. In Proc of SPLASH2011, 2011.
23) 菅谷 みどり, 岡本 悠希, 若森 拓馬, 倉光 君郎. 言語ランタイムによる自動的なログ収集機構. 情報処理学会プログラミング研究会 PRO91. 2012