ユビキタス・組込みシステムにおける オブジェクト指向言語 Dolittle の実装

並木 美太郎[†1]　久野　靖[†2]
兼宗　進[†3]　早川 栄一[†4]

本発表では、小型の組込み機器、ユビキタスコンピューティングのノード向けのプログラミング環境、特に、省資源の小型機器向けの言語処理系について述べる。多くの組込み機器やユビキタスコンピューティングのノードの開発では、アセンブラや言語 C などが用いられてきたが、プログラミングが煩雑な上、機器へのプログラム転送など手間がかかるという問題があった。筆者らが設計開発したオブジェクト指向言語 Dolittle を ROM 数 10KB、RAM 数 KB〜数 10KB の 8 ビット、16 ビットマイクロプロセッサ上で稼働することを目標に仕様や処理系の構造などを見直し、ARM7、ATmel AVR、H8 などのマイクロプロセッサ上で動作する環境を構築した。本発表ではこれらについて報告する。

An Implementation of "Dolittle" Object-Oriented Language on Ubiquitous and Embedded Systems

Mitaro NAMIKI,[†1] Yasushi KUNO,[†2] Susumu KANEMUNE[†3]
and Eiichi HAYAKAWA[†4]

1. はじめに

ユビキタスコンピューティングのノード、組込みシステムのプログラミングでは、制御モデルの構築もさることながら、PC に比べ低い CPU 能力、少ないメモリなど、資源制約の厳しいプログラミングが要求されるほか、ROM ベースのコード、多種多様の I/O 制御など実装は、言語 C、C++などを用いてもプログラミングは煩雑である。オブジェクト指向プログラミング、クラスライブラリの利用、プラットホーム独立な実行環境から、近年組込みシステムでも Java が利用され始めており、例えば Lego MindStorms NXT の leJOS では、ROM が 128KB、RAM が 64KB のプロセッサに小型の JavaVM が移植された。だが、よりメモリの少ないプロセッサでの Java VM の移植は困難である。

また、MIT のメディアラボの Processing やオープンソースプロジェクトの Wiring では ROM 16KB、RAM 1KB 程度でも稼働する言語処理系が実装されているが、オブジェクト指向プログラミングはできない。

本稿では、ROM が数 10KB、RAM が数 KB の小型プロセッサ上で稼働するオブジェクト指向言語の処理系の実装について述べる。プログラミング言語として、共著者が設計開発を行ったプログラミング言語 Dolittle[1)2)3)] を採用した。Dolittle はプログラミング教育を目的に設計されたが、その本質はプロトタイプベースのオブジェクト指向言語であり、単純な文法と実行機構を有している。このため、単に教育に留まらず、小型軽量の機器に実装し、実用用途が可能な言語であると判断した。既存の Dolittle の言語処理系は Java で記述されており、J2SE のプロファイラ上で動作する。

本研究では、小型の組込みプロセッサを目標に、(1) VM 構成とすること (2) 言語 C により記述することとし、ROM が数十 KB、RAM が数 KB の小型の計算機上で動作できるようにした **μ Dolittle** を開発した。以降、詳細を論じる。

†1 東京農工大学 Tokyo University of Agriculture and Technology
†2 筑波大学 University of Tsukuba
†3 一橋大学 Hitotsuvashi University
†4 拓殖大学 Takushoku Univerisity

2. 対象とするシステムの概要

2.1 組込みシステムのハードウェア構成

本稿では主に小型の組込みシステム、特に 8bit、16bit の CPU を対象とした組込みシステムを対象とする。これらの組込みシステムは、一つのチップ上に ROM が 16KB〜256KB 程度、RAM が数 KB〜64KB、クロックも 8MHz〜数 10MHz であり、通常の PC と比べると省資源である。多くの CPU は、ROM、RAM を内蔵すると同時に、汎用 I/O(GPIO)、A/D コンバータ、シリアル通信の機能を有しており、オール・イン・ワンの設計となっている。CPU によっては外付けのメモリを接続することが可能となっている。国内ではルネサス社の H8、SH、R シリーズなどが、外国では Micro Chip 社の PIC、ATMEL 社の AVR などが利用されている。また、ARM 社の ARM7 などの組込み向けプロセッサもいくつかの組込みシステムで採用されている。

基本的にはチップ内の Flash ROM 上にプログラムを格納し、計算に必要なデータを RAM に配置する。一般に Flash ROM は数 10KB〜512KB 程度の容量を持つが、RAM は数 KB〜64KB 程度と少ないのが特徴である。また、組込みプロセッサのいくつかはハーバードアーキテクチャを採用しており、機械語を格納する ROM のアドレス空間と RAM は別空間となっている上、アクセス単位も ROM はワード単位、RAM はバイト単位とアドレスの意味づけが異なる CPU もある。

多くの CPU チップが GPIO や A/D コンバータを内蔵していることから、温度、光、加速度センサ類を直接接続できる。また、CPU チップが USART、UART、SPI、I2C などの汎用シリアル通信機構を持っているので、ホスト側との通信については古典的な RS-232C のほか、FTDI 社の USB-シリアル変換チップ、Lantronix 社の Xport のような Ethernet-シリアルアダプタ、BlueTooth、Zigbee などのモジュールを直接接続できる。I/O についても、D/A コンバータなどについては SPI に対応したものもあり、アナログ回路に精通した電子回路の専門家でなくとも、通常の計算機科学・工学の講義程度の知識があれば、システムを組み上げることが可能である。

この規模の CPU は近年センサネットワークなどのユビキタスコンピューティングのノードに利用されているほか、小型・小規模な計測機器、制御装置などに用いられている。表 1 に小型機器の組込み向け CPU の例を示す。

表 1 小型機器向け組込み CPU の例

CPU/MCU	クロック	Flash (B(W))	SRAM(B)	EEPROM (B)	通信
AVR					
ATtiny44		4K(2K)	256	256	UART,SPI
ATmega88		8K(4K)	1K	512	UART,SPI
AT90USB162	8/16MHz	16K(8K)	512	512	USB
ATtiny461		4K(2K)	256	256	SPI
ATmega128	16MHz	128K(64K)	4K +32K	4K	USART,SPI,I2C
ATmega32	16MHz	32K(16K)	2K	1K	USART,SPI,I2C
ATmega168	8−20MHz	16K(8K)	1K	512	USART,SPI,I2C
ATmega644	8−20MHz	64K(32K)	4K	2K	
AT90USB647	8/16MHz	64K(32K)	4K	2K	
PIC					
PIC18F2520		32K(16K)	1536		SPI
PIC18F2553	8MHz	32K	2K	256	USB
PIC24FJ64GA002	16MHz	64K	8K		USB
PIC16F690	8MHz	4KW	256		USART
PIC16F88		7K(4KW)	368	256	UART
PIC18F67J60	25MHz	128K	3008 +LAN8K		UART,SPI,I2C
dsPIC30F401230		48K	2K	1K	
HC08					
MC9S08QG8	16MHz	8K	512		USB(SPI,I2C)
R8/M16C					
R8C/15	20MHz	16K	1K		シリアル
M16C/29		128K+4K	12K		CAN,シリアル
H8/SH2					
H8/3664F		32K	2K		RS232C,I2C
HD64F3687FP	20MHz	56K	4K	ext.256K	USART,I2C
H8/3694F	20MHz	32K	2K		USART
HD64F7145F50V	50MHz	256K +2MB	8K +1MB		USART×4
ARM					
ARM9	180MHz	4M	256K		GPIO,Zigbee
LPC2138		512K	32K		SPI,I2C
AT91SAM7S32	16MHz	32K	8K		USART,SPI,I2C
AT91SAM7S256	18MHz	256K	64K		USB,UART
ARMCoretex-M3	8MHz	128K	20K		USB,USART
FreeScale(ColdFire)					
MCF52333	25MHz	256K	32K		USART
MCF5474	266MHz	16M	64M		USART

2.2 組込みシステムのソフトウェア開発

ソフトウェアシステムとしてみると、表 1 の CPU を用いた計算機システムでは、OS がない場合がほとんどであり、AP(応用プログラム) は装置制御主体のプログラムである。これら組込みシステムやユビキタスノードのソフトウェア開発の多くは、アセンブラ、言語 C などを用いてプログラムの開発を行う。小型機器向けのプロファイルである Java CLDC などを提供している例もあるが、資源が潤沢なハードウェアプラットホームに限られている。多くのシステムでは、ホスト計算機 (PC) でアセンブラや言語 C でプログラムを記述し、ISP、BDM、JTAG などの特殊なケーブルでネィティブコードをダウンロードする開発形態が多いが、手順は煩雑である。近年は、チップ上の ROM にブートローダと呼ばれるプログラムを格納しておき、手順を簡略したものが増えているが、やはり、それなりの手間がかかる。また、言語もアセンブラと言語 C

が多く、オブジェクト指向的なプログラミングは資源の豊かなシステムに限られる。

組込みシステムが単にスレーブデバイスとしてデータの入出力を行い、ホストまたはネットワークに流すようなシステム構築手法もありえる。この場合はホスト計算機と通信し合うソフトウェア開発を簡便に行えるプログラミング環境としたい。また、単にスレーブデバイスとしてだけではなく、個々のノードや組込みシステムが自律的に、例えば、ロボット自身で意思決定を行う、さらにはロボット同士が協調して動作を行わせる例のように、組込みシステムが自律分散処理を行えるプログラミング環境を提供したい。

このような組込みシステムやユビキタスコンピューティングの分散系のノード向けのプログラミング言語が提案され、処理系が実現されている。Processing[4] は Java のサブセット、Wiring[5] は C のサブセット、となっており、ATmel 社が提案した標準的なハードウェアプラットホームである Arduino[6] などで稼働する。Forth 言語のような後置記法に基づくプログラミング言語を採用した Forcy[7] では、コンパイラと VM が AVR 上で稼働している。しかし、Processing、Wiring、Forcy のいずれもオブジェクト指向ではない。Talktic[8] については MOXA と呼ばれる標準的なハードウェアプラットホーム上で JavsScript 風の言語を利用できる。しかし、他のプラットホームへの適用可能性は未知である。Konoha[9] と呼ばれるスクリプト言語もあるが、どのようなプラットホーム上で実装されているかは不明である。

3. オブジェクト指向言語 Dolittle の言語設計と実行モデル

3.1 オブジェクト指向言語 Dolittle 言語の特徴

Dolittle はもともとは初等中等教育を中心とする教育向けのプログラミング言語として設計された[1][2]。その特徴として次のものが挙げられる:

- プロトタイプ方式によるオブジェクト指向の採用 — 教育用であっても今後オブジェクト指向は必須の機能であるとの判断と、クラス方式に比べて道具だて (構文上の単位や言語上の概念) が少なく学習しやすいと考えたため。
- 字句上の工夫 — 予約語は無く、すべての名前は標準定義または利用者定義の識別子である。☆
- 構文上の工夫 — 基本となる構文単位はメッセージ送信式 (とその構文糖衣としての中置記法による式) のみであり、制御構造はすべてブロックを引数として受け取るメソッドで実現する。☆☆
- 実行モデル上の工夫 — すべての値はオブジェクトであり、値を保持する機構は極力オブジェクトのプロパティに統合する設計となっている。

これらはいずれも、コンパクトで生徒に教えやすく、オブジェクト指向を活かした教育を行えるようにするという教育用言語としての目標から選択した項目である。Dolittle は教育用プログラミング言語、日本語プログラミング言語として利用され、効果をあげているが、筆者はその本質を上記のプロトタイプベースのオブジェクト指向言語と理解しており、日本語プログラミングや教育用であることが本研究で重要な要素であるとは考えていない。無論、引き続き教育、例えば、組込みシステムの教育や新指導要領で導入された中学校の計測制御への応用も考えうるが、本稿では実用的視点、計算機科学的視点を重視する。プロトタイプベースのオブジェクト指向であること、メッセージ送信式などで分散のユビキタスノードのベースとなり得ること、簡潔な構文などから組込みシステム、ユビキタスノードのプログラミング環境の基盤となりえるのではないかと考えた。次に、Dolittle の紹介も兼ねて、主要なものを取り上げ解説する。

3.2 名前の束縛と解決

Dolittle ではプログラム中に出現する識別子を基本的にオブジェクトのプロパティのみに統一している。具体的な識別子の出現する場面ごとに列挙すると次の通り:

- プロパティ — 「式:識別子」は式を評価した結果のオブジェクトのプロパティを参照する。
- メソッド名 — 「式! … 識別子」は式を評価した結果のオブジェクトが持つプロパティを参照する (Dolittle ではメソッドはオブジェクトが持つプロパティに格納された値の一種であるため)。カスケード送信「式! … 識別子 1 … 識別子 2」等も同様。なお、「…」の部分は空またはパラメタの並びであり、パラメタとしてはリテラル (数値、文字列) またはかっこで囲まれた式が書ける。
- 変数名 — Dolittle ではすべての変数は何らかのオブジェクトのプロパティである。これには次の 3 つの場合がある:
  1. グローバル変数 — 実行プログラムに唯一ある「ルートオブジェクト」のプロパティ。

☆ ただし例外として `self` など少数の擬変数がある。

☆☆ この設計は Smalltalk に類似しているが、複数のブロックを必要とする構造を 1 つのメソッドではなく複数のメソッドのカスケード送信により実現している点は異なっている。

2. インスタンス変数 — そのインスタンス (オブジェクト) のプロパティ。
3. ブロックのパラメタと局所変数 — そのブロックの実行中だけ存在する無名の「環境オブジェクト」のプロパティ。

概念的にはこれらのプロパティを持つオブジェクトはプロトタイプ連鎖によってつながっており、それを手前から順に検索することで変数名に対応する変数の実体 (=オブジェクトのプロパティ) を参照できる (図 1)。

上で「概念的には」と書いたのは、実装上はこれと違う構成も可能だからである。プロトタイプ連鎖の構成については次節でさらに述べる。

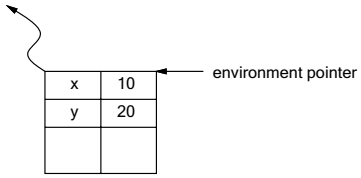


図 1 ブロックの基本的な環境

3.3 プロトタイプ連鎖

Dolittle はプロトタイプ方式のオブジェクト指向言語であり、ルートを除くすべてのオブジェクトは「親」オブジェクトを指すプロトタイプポインタを保持している (さらに、組み込みオブジェクトも含めてすべてのオブジェクトのプロトタイプ連鎖は最終的にルートにつながるようになっている)。任意のオブジェクトに対してメソッド `create` を呼ぶことで、そのオブジェクトを親として持つ新しいオブジェクトを生成できる。

環境オブジェクトは、ブロックの実行が開始されるごとに作り出される。各ブロックの先頭にはパラメタと局所変数の定義が記されており、環境オブジェクトはこれらをプロパティとして持つ形で生成される。

環境オブジェクトの親が何になるかについては、次の 2 つの場合がある:

- ブロックがメソッドとして実行される場合 (ブロックがオブジェクトのプロパティとして格納されており、そのオブジェクトに対するそのプロパティ名でのメソッド呼び出しがあった場合) — ブロック実行のために生成される環境オブジェクトの親は、そのオブジェクトになる (図 2)。
- それ以外の形で (ブロックが持つメソッド `exec` を呼び出すなどの形で) ブロックが実行開始される場合 — ブロック実行のために生成される環境オブジェクトの親は、そのブロックが書かれている位置を実行しているときに用いられている環境オブジェクトになる。

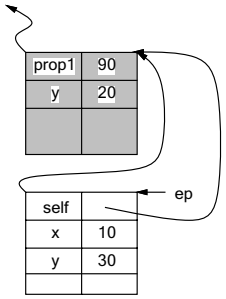


図 2 メソッドとして実行されるブロックの環境

後者が意味することは、ブロックの値はコードと環境の対から成るクロージャであり、ブロックを直接に (メソッドとしてでなく) 呼び出す場合はブロック中のコードはそれが作られた時点の環境をアクセスし得る、ということである。これは、ブロックが枝分かれや繰り返しなどの制御構造を構築する手段として使われるために必然的にこうしたものである。

これに対し、ブロックがオブジェクトのプロパティとして格納されていてメソッドとして呼び出される際は、ブロックの環境はそのオブジェクトになっている。

以上を整理すると、プロトタイプ連鎖は一番手前に現在実行しているブロックに対応する環境オブジェクトがあり、その先にはそれを囲むブロックに対応する環境オブジェクトが連なっている。連鎖をたどって行くとどこかで環境オブジェクトでない (通常の) オブジェクトにつながるが、それはそのオブジェクトに対するメソッド呼び出しが行われた箇所に相当する。その先は連鎖にはオブジェクトだけが連なり、末端にはルートがつながっている (図 3)。

プロトタイプ連鎖の利用方法は、変数の参照を行う時に手前から順次変数名と同名のプロパティを検索して行き、最初に見つかったものに対する値を取り出すという形で用いる。一方、変数に書き込みを行う場合は最も手前にある通常のオブジェクトに対して書き込みを行う (JavaScript などと同じ)。

この設計は、レキシカルスコープのための環境の連鎖とプロトタイプ連鎖を 1 つで兼ねているという点に特徴がある。このような連鎖の統合は、Dolittle を資源の限られた環境で動かす際の実装の簡潔さに貢献しうると考える。

3.4 軽量実装のための問題点

従来の Dolittle 言語の実装は Java で記述されており、前節までに述べた Dolittle 言語の実行モデルを忠

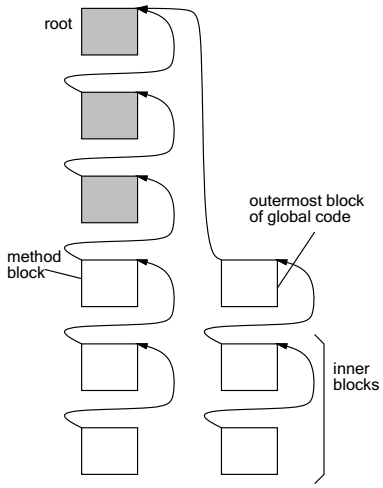


図 3 実行環境の連鎖

実にそのまま実現していた。例えば、環境オブジェクトは実際にブロックの実行開始ごとにヒープに割り当てられ、不要になったものは Java VM の GC により回収していた。この方式により、環境オブジェクトをそのまま保持するだけでクロージャが実装できる等の利点が得られた。

しかし、資源が限られた環境を前提とした Dolittle の実装では、このようなヒープと GC を前提とした設計は見直す必要がある。特に、現行の Dolittle は構文木を保持して実行時に木をたどる評価器になっている。対話的な処理系としては良好な方式であるが、資源の少ない計算機上では、記述言語が Java であることと重なって解決すべき課題である。

現行 Dolittle の持つ特徴をまとめると次のようになる。

(1) プロトタイプベースのオブジェクト指向言語
(2) プロトタイプと実行環境の連鎖
(3) クロージャ、GC
(4) 各種プロトタイプオブジェクト — GUI、ネットワークなどのプロトタイプオブジェクト
(5) Java による処理系記述 — アプレットによる版もあり、教育のみならず、ソフトウェア開発においてもプラットホーム独立なプログラミング環境の提供
(6) 構文解析木をたどる評価器

これらの設計上の選択は、教育目的、また、処理系実装について有利な点もあるが、本稿のような省資源の計算機上での実現については、言語仕様および実装方式の両方の観点から再検討が必要となる。次章以降に省資源向けの Dolittle 仕様と実装方式について述べる。

4. 本研究開発の目標と言語処理系の方針

前章までの議論をふまえて本研究開発は次の内容を目標とする。

- 組込み・ユビキタスコンピューティングのノード構築のためのオブジェクト指向プログラミング環境の提供

  組込みシステムやノードそれ自身でプログラムを自律分散的に実行できる実行環境を構築する。特に、組込みシステムやユビキタスコンピューティングのノードでは、生物にたとえれば眼や耳などの五感や手足に対応する各ノード、それから脳に対応する高度な情報処理を行うホスト計算機の間で、プロトコルを決めればシステム全体の統括制御を行えるプロトコル指向コンピューティング[10]の基盤を構築できる。個々の組込みシステムやユビキタスのノードをオブジェクトとして定義すると同時に、その中のプログラムもオブジェクト指向的なプログラミングを行えるようにすることで、自律・分散・協調システムのプログラミング基盤を提供する。
- 8bit、16bit マイクロプロセッサ上で Dolittle 処理系を稼働

  2 章で述べたようにシステムやセンサノードでは、8bit や 16bit のプロセッサが用いられる。これらのプロセッサを用いたシステムでは、通常の PC のような潤沢な資源を用いた処理系の実装は困難である。8bit/16bit の組込みプロセッサの ROM については数 10KB の容量があるが、RAM は極めて少ない。本稿では RAM が最低 2〜4KB 程度で稼働する処理系とする。そのため、Dolittle 自身の言語仕様をこれら資源の貧弱なプロセッサ上で実現できるように検討すると同時に、実装方式について再考する。

5. 軽量な Dolittle の言語仕様〜μ Dolittle

既存の Dolittle は Java で実装されており、第 3 章で述べた実行モデルを Java で忠実に実現していた。評価器は解析木をたどり木の節を実行する処理系である。環境オブジェクトはブロックの実行開始ごとにヒープに割り当て、不要になったメモリオブジェクトは Java VM の GC により回収していた。環境オブジェクトをそのまま保持するだけでクロージャが実装できた。このように、Java によりプラットホーム独立なプログラミング環境を得られた利点は大きい。シ

ステムを記述する基底言語が、記述される目標言語の実装に大きく寄与する一例である。特に、基底言語である Java はオブジェクト指向言語なので、同じくオブジェクト指向言語である Dolittle の実装には有利であった。ただ、既存の Java により記述された Dolittle 処理系は、起動時だけで数 10MB のメモリを消費している。利点と引換えに資源を、特にメモリ資源を必要とする。

メモリ資源の消費を抑え、RAM が数 KB で稼働するためには、

- 言語仕様の見直し。特に、クロージャと GC
- 処理系の方式。特に、木をたどる評価器

の部分について変更が必要であると考えた。現行 Dolittle の持つ特徴の (1)〜(6) の中で、プロトタイプベースのオブジェクト指向は基本中の基本である。また、(2) のプロトタイプと実行環境の連鎖は、プロトタイプベースのオブジェクト指向の実行系の観点からは必要不可欠である。(3) のクロージャ、GC はメモリを必要とする最大の要因である。ブロックの実行開始時にブロックが参照したオブジェクトの情報をクロージャとして保持することからメモリの使用量は多い。また、実行環境の情報もヒープにより保持することから全体的にメモリの使用量が増加すると同時に、一時的なオブジェクトが多数作られることから GC は必須となっていた。RAM の少ないシステムでは、このような GC なしでも長期間動作できる設計が不可欠である。

(5) の Java による処理系記述は今回対象とする計算機上では不可能である。そもそも組込みシステムでは JavaVM そのものを実現するのが困難だからである。TinyVM や KVM などの小型軽量の JavaVM も存在するが、最低でも RAM は数 10KB が必要となる。組込み C++の採用も一つの候補であるが、CPU によっては C++の処理系が用意されていないものもある。

上記の理由から、まず、組込み向け Dolittle 言語の仕様を既存の Dolittle 言語のサブセットとして策定した。プロトタイプベースのオブジェクト指向言語とプロトタイプ・実行環境の連鎖を継承することとし、クロージャと GC は含めない仕様を **μ Dolittle** とする。また、プロトタイプオブジェクトも組込み向けのオブジェクトを用意する。

記述言語については、多くの組込み向け CPU で用意されている言語 C とする。処理系も木をたどる方式ではなく、VM(Virtual Machine) による実行方式を採用する。Dolittle VM の機械語を定義し、その機械語を解釈実行する VM を組込みプロセッサ上に実現し、Dolittle プログラムを実行するものとする。

以上、本稿の処理系の方針と特徴は次のようになる。

(1) プロトタイプベースのオブジェクト指向言語

(2) プロトタイプと実行環境の連鎖

この 2 点については、現行 Dolittle の特徴を受け継ぐ。メモリ管理が複雑になるクロージャは実装しない。ただし、3 章で述べた連鎖によるサーチは実装する。GC はなく、プログラマが明示的にオブジェクトを管理する。

(3) 組込み向けプロトタイプオブジェクト

デバイス制御、ハードウェア割込みなどのプロトタイプオブジェクトを組込み、ユビキタスノード向けに用意する。また、制御構造に関するプロトタイプオブジェクトについては、基本的なものだけを提供し、必要に応じてプログラマが制御構造を記述できるようにし、メモリ量の制約に応じたプログラミングスタイルをとれるようにする。

(4) 言語 C による処理系記述

現行の Dolittle はプラットホーム独立性を高めるために Java で記述されている。しかし、多くの小型の組込みシステムでは JavaVM を搭載するのは困難であることから、言語 C により処理系を記述する。無論、可能な限り CPU 依存部を分離し、多種の組込みプロセッサで稼働するよう移植性を高める。

(5) VM 構成による実行系

現行の処理系は、パーサジェネレータである SableCC により生成されたパーサと、Java により記述された解析木をたどる評価器から構成され、パーサと評価器の言語処理系の核と GUI などのプロトタイプオブジェクト、エディタなどが統合された構成となっている。このオール・イン・ワンの処理系を Java のアプレットとして提供することで、Web ブラウザでも稼働する教育向けの環境を構築し、効果をあげてきたが、組込みシステムではこのような構成は望むべくもない。

本稿の処理系は、パーサと評価器を分離し、個々別々に稼働する構成を採用する。分離することで資源制約の厳しい組込みシステムやユビキタスノード上での構成を柔軟にする。

なお、このようなサブセットでもメモリ資源が不足するようなシステム向けに、プロトタイプベースのオブジェクト指向さえ含めない NanoDolittle の仕様も検討は行ったが、別の機会に論じたい。

6. 本処理系の機能と構成

6.1 処理系の全体構成

本稿の **μ Dolittle** の概念的な構成を図 4 に示す。基本的には、コンパイラと VM から構成され、いずれも言語 C で記述される。UNIX や Windows などの通常の OS が存在するマシンでは OS 上で、小型機器の組込みシステムでは OS を介さずハードウェア上で直接 VM を実行する。今回の発表では実装しなかったが、Java で VM を記述することにより SUN Spot や携帯電話などの Java CLDC 仕様の計算機でも Dolittle プログラムを実行できるであろう。

μ Dolittle はターゲットとなる組込み機器やユビキタスのノードで実行される。コンパイラについては、現在、ホスト計算機である PC 上で実行され、コンパイルされた VM コードを USB、RS-232C などでターゲットとなる組込み機器へ転送する。資源に余裕のある CPU では、コンパイラもターゲット機器上で稼働させる予定である。この場合は、Dolittle のソースプラグラムをホスト計算機上で作成し、ターゲット上でコンパイル&ゴーとなる。

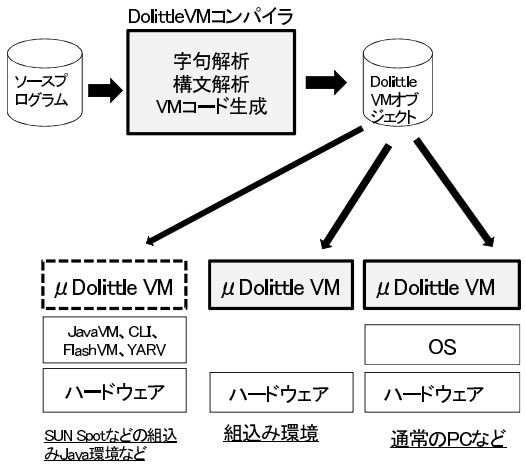


図 4 処理系の全体構成

6.2 μ Dolittle VM 命令語

現行 Dolittle の木をたどる実行方式とは異なり、**μ Dolittle** は VM 構成となっている。VM はスタックマシンを前提としており、次の命令 (ニーモニック) を有する。Dolittle コンパイラによって Dolittle ソースプログラムは、次の命令列に翻訳される。なお、機械語列そのものは、必ずしも **μ Dolittle** の制限された仕様ではなく、現行 Dolittle でも同様の VM 機械語に基づく VM を作成することが可能である。`[...]` は命令がオペランドとして消費する値スタック上の要素を底からトップへの順で示している。なお、名前については翻訳時に識別子 (整数である `nameid`) に変換され機械語のオペランドとして指定される。同様に、文字列定数、ブロックも識別子 (整数である `strid`、`blockid`) に翻訳され、それぞれ実行時に VM はその実体を参照する。

- `pushi #整数`
  整数を値スタックにプッシュ
- `pushs strid`
  文字列 (`strid`) を値スタックにプッシュ
- `push1 nameid`
  現在の環境にある名前 `nameid` で指定したプロパティの値を値スタックにプッシュ
- `[obj] push2 nameid`
  `obj` で指定されたオブジェクトにある名前 `nameid` のプロパティの値を値スタックにプッシュ
- `pushb blockid`
  ブロックオブジェクトとそのときの環境を対にした環境オブジェクトをスタックにプッシュする。後述するが、この環境オブジェクトは、ブロックオブジェクトとその時点での実行環境の対となったオブジェクトとなっている
- `[obj] [argN]...[arg1] send #N,nameid`
  `obj` の名前 (`nameid`) で指定したメソッドを `N` 個の引数を伴って呼び出す
- `ret`
  最後に評価された値を持ってブロック評価またはメソッド呼出しから呼出し元に戻る。ブロックの最後に必ず現れる
- `[anyN]...[any1] pop #N`
  スタックの N 個の先頭要素をポップして捨てる
- `[exp] store1 nameid`
  現在の環境にある名前 `nameid` で指定したプロパティの値として `exp` を代入
- `[obj] [exp] store2 nameid`
  `obj` の名前 `nameid` で指定したプロパティの値として `exp` を代入 `store1` は `p = ...` のような代入に、`store2` は `p:name = ...` のようなオブジェクトのプロパティ指定の代入に用いられる
- `para nameid`
  ブロックの引数を定義する。具体的には値スタック上の実引数の値を取り出し、環境スタック上のローカル変数に代入する。ブロックの先頭に連続して現れる

- `tmp nameid`
  ブロック内のローカル変数を定義する。ローカル変数が定義されたいたとき `para` 命令に続いて現れる

四則演算などの単純な計算、条件判定、分岐などがVM の機械語として現れないことに注意されたい。四則演算はもとより if〜then〜else、繰返しなどの制御構造もすべて決まった名前のメッセージ送信として処理される。単純で簡潔である。オブジェクトの定義の仕方次第で目的に合致した処理系を作ることができる。例として、Dolittle のサンプルプログラムを図 5 に、コンパイルした VM オブジェクトコードをダンプした例の一部を図 6 に示す (Dolittle ソースコードとの対応付は人手による)。ダンプでは、ツールで nameid、strid を文字列に変換しているが、機械語では id が埋め込まれている。

```
p:find = [
  |x;i cnt|
  i = 0. cnt = !len.
  [i < cnt] ! while [
    [x == (p!(i)ref)] ! ifthen
      [i ! return] exec.
    i = i + 1
    ] exec.
  UNDEF
].
```

図 5 Dolittle プログラムの例

μ Dolittle の VM では、これらの機械語、それから id に対応する実体を管理する表を ROM または RAM 上に配置する。VM のアドレシングはワード単位である。小型機器用の CPU では 1 ワード 16 ビット、高機能な組込み用の計算機では 1 ワード 32 ビットを想定し、それぞれ 16 ビットモード、32 ビットモードとしている。表 2 に VM の機械語の形式を示す。16 ビットモードでは、`INT` は 15 ビットの符号付整数、`nameid`、`strid`、`blockid` はその実体を管理する名前文字列表、文字列表、ブロック表のインデクスとして 10 ビット整数値をとる。`N` は引数の個数であり、3 ビットである。`M` はポップすべき個数で 8 ビットとなっている。表 6 は 16 ビットモードの例である。32 ビットモードではそれぞれの命令の MSB に 16 ビットが付加され、32 ビットで 1 ワード=1 命令語を構成する。

VM の機械語ではすべて id で実体を指定する。実体を管理する表のオーバヘッドを伴うが、複数の VM オブジェクトをファイルをマージする際に再配置が容易なこと、それぞれの表と実体については ROM に配置できるようにし、RAM を圧迫しないので、オペランドに VM コードの位置情報を埋め込まない設計と

```
Magic: 0xd11e
**Symbol Table  table = 13, body size = 0x0025
  ↓ nameid
  id \$offset  len:body  hex   ←識別子の表
   1 0000     1:p 7000
   2 0002     4:find 6669 6e64
   3 0005     1:x 7800
   4 0007     1:i 6900
   5 0009     3:cnt 636e 7400
   6 000c     3:len 6c65 6e00
   7 000f     5:while 7768 696c 6500
   (省略)
**Code  block table = 6, code size = 0x002c
Entry Block ID:0
        オフセット
blockid     |  機械語
   ↓        ↓  ↓
   0, 0  0x0028 0x0053 push1 p ← p:find=[...]
   0, 0  0x0029 0x004b pushb 1
   0, 0  0x002a 0x00ab store2 find
   0, 0  0x002b 0x0005 ret
   1, 1  0x0018 0x00f3 para x  ← [|x;i cnt|...]
   1, 1  0x0019 0x013b tmpvar i
   1, 1  0x001a 0x017b tmpvar cnt
   1, 1  0x001b 0x0000 pushi #0x0000
   1, 1  0x001c 0x0123 store1 i
   1, 1  0x001d 0xffd3 push1 <SELF>
   1, 1  0x001e 0x0181 send #0,len
   1, 1  0x001f 0x0163 store1 cnt
   1, 1  0x0020 0x008b pushb 2
   1, 1  0x0021 0x01c1 send #0,while
   1, 1  0x0022 0x00cb pushb 3
   1, 1  0x0023 0x02c9 send #1,exec
   1, 1  0x0024 0x0115 pop #1
   1, 1  0x0025 0x0313 push1 UNDEF
   1, 1  0x0026 0x0115 pop #1
   1, 1  0x0027 0x0005 ret
   2, 0  0x0000 0x0113 push1 i  ← [i < cnt]
   2, 0  0x0001 0x0153 push1 cnt
   2, 0  0x0002 0xf9c9 send #1,<<>
   2, 0  0x0003 0x0005 ret
   3, 0  0x000e 0x010b pushb 4  ← [[x== ...i=i+1]
   3, 0  0x000f 0x0241 send #0,ifthen
   3, 0  0x0010 0x014b pushb 5
   3, 0  0x0011 0x02c9 send #1,exec
   3, 0  0x0012 0x0115 pop #1
   3, 0  0x0013 0x0113 push1 i
   3, 0  0x0014 0x0002 pushi #0x0001
   3, 0  0x0015 0xfc09 send #1,<+>
   3, 0  0x0016 0x0123 store1 i
   3, 0  0x0017 0x0005 ret
   4, 0  0x0004 0x00d3 push1 x  ← [x==(p!(i)ref)]
   4, 0  0x0005 0x0053 push1 p
   4, 0  0x0006 0x0113 push1 i
   4, 0  0x0007 0x0209 send #1,ref
   4, 0  0x0008 0xf909 send #1,<==>
   4, 0  0x0009 0x0005 ret
   5, 0  0x000a 0x0113 push1 i  ← [i ! return]
   5, 0  0x000b 0x0281 send #0,return
   5, 0  0x000c 0x0115 pop #1
   5, 0  0x000d 0x0005 ret
```

図 6 Dolittle プログラムのコンパイル例 (一部)

表 2 **μ Dolittle VM** の機械語形式

ニーモニック	機械語の形式 (MSB-LSB)		
pushi	INT		0
send #N,nameid	nameid	N	001
pushs strid	nameid	000	011
pushb blockid	nameid	001	011
push1 nameid	nameid	010	011
push2 nameid	nameid	011	011
store1 nameid	nameid	100	011
store2 nameid	nameid	101	011
para nameid	nameid	110	011
tmpvar nameid	nameid	111	011
ret	00000000	00000	101
pop #M	M	00010	101
(予約)	(13bit)		111

した。

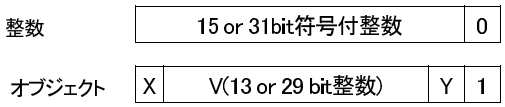
6.3 μ Dolittle VM のオブジェクト

μ DolittleVM では VM が管理する RAM 上のオブジェクトとして、次の 6 種類のオブジェクトが存在する。

(1) 整数オブジェクト
(2) ブロックオブジェクト
(3) 文字列定数オブジェクト
(4) 環境オブジェクト
(5) 組込みオブジェクト
(6) プロトタイプオブジェクトのインスタンス

(2) と (3) については、プログラム中のブロックおよび文字列定数に対するオブジェクトである。(4) の環境オブジェクトについては、3 章の説明にある実行環境を保持するための内部オブジェクトである。(5) の組込みオブジェクトは言語 C で記述されたプロトタイプオブジェクトとなっている。

μ DolittleVM のオブジェクトのタグを図 7 に示す。各オブジェクトは 1 ワードで表わされる。16 ビットモードのときは 16 ビット、32 ビットモードのときは 32 ビットとなり、整数オブジェクトは 15 ビットまたは 31 ビット符号付き整数、そのほかのオブジェクトについては、プロトタイプオブジェクトのインスタンス、環境オブジェクトについては、V の部分に VM のヒープ領域または環境スタック領域のワードアドレスが埋め込まれる。組込みオブジェクトについては文字列定数オブジェクトの後半を用いている。16 ビットモードでは V の部分が 13 ビットなので、最大 8K ワードのヒープおよび環境スタック領域を用いることができる。



XY V
00 オブジェクトのインスタンス（ワードアドレス）
10 ブロック(blockid)
11 文字列定数(strid) IDの後半は組込みオブジェクト
01 環境オブジェクト(ワードアドレス)

図 7 μ Dolittle におけるオブジェクトのタグ

プロトタイプオブジェクトのインスタンスの例を図 8 に示す。一つのインスタンスに最低 3 ワードのメモリをヒープから割り当てる。3 ワード中、1 ワード目は親のオブジェクトであり、図 7 の値が格納される。このプロトタイプオブジェクトへのポインタをたどることにより、プロトタイプオブジェクトの連鎖を実現している。2 ワード目はプロパティリストのポインタである。3 ワード目にオブジェクトの値が格納されるが、その内容はプロトタイプオブジェクトに依存する。例えば、ベクタオブジェクトや文字列についてはその内容へのポインタとなる。組込みオブジェクトについては、言語 C で記述されたメソッド表へのポインタとなっている。プロパティについても 3 ワードが割り当てられ、次のプロパティへのリスト、プロパティの名前 (`nameid`)、プロパティの値 (図 8 の値) が格納される。なお、後述するが、環境オブジェクトについてはスタックに割り当てられるため、プロパティの値として格納されない。**μ Dolittle では** GC がないためこれらのメモリは参照されなくなっても回収されないので、プログラマはメモリ管理を意識する必要がある。32 ビットモードについてはクロージャはないが GC を持つ版を開発する予定である。

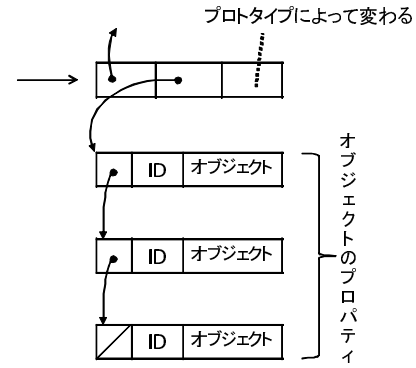


図 8 μ Dolittle におけるプロトタイプオブジェクトのインスタンス

現在、文字や文字列については ASCII コードを用い 8 ビット/文字としている (いずれは多国語化する予定である)。文字列中の文字は 1 ワードの中にパッキングされる。例えば、16 ビットモードでは 1 ワードに 2 文字がパックされている。これは、ROM 化を考慮したときに、CPU によっては ROM 領域がワードアドレスのアーキテクチャがあるため、ワードを基本語長とする形式にしたためである。

6.4 μ Dolittle VM の実行機構

3 章で Dolittle の実行モデルについて述べた。現行 Dolittle については、図 1〜3 の情報をブロックの実行環境として保持している。これらを環境オブジェクトとしてヒープに割り当て、実行制御を行うと同時にクロージャを実現していた。しかし、RAM の少ない

CPU ではヒープに割り当てるとメモリを消費する。そこで、本稿における実行方式としては、このように実行環境をヒープに割り当てるのではなく、環境スタック上に割り当てることとした。無論、プロトタイプオブジェクトの連鎖については、従来の Dolittle と同様の連鎖を持つが、ブロックがメソッドとして実行されるときは、その環境を環境スタック上に割当て、メソッドとしての実行が終了した時点でメモリを解放するようにした。この方式ではクロージャを実現できないが、クロージャを使わずにプログラミングできる局面も多いので、今回の実装ではこのような方式を採用した。

図 9 に実行環境を示す。スタックとしては、値スタックと環境スタックの二つがある。値スタックには、push 命令の値が積まれるほか、メソッドの計算値が値スタックに積まれて呼出し元に返される。環境スタック上にはメソッド呼出し元の戻り番地のほかに、ブロックの実行環境である self オブジェクト、ローカル変数が格納される。ブロックに対してメソッド呼出しを行った場合の self はブロックのオブジェクトとそのブロックが呼びだされた時点での実行環境の対を指し示すようになっている。対のうちの一つであるブロックオブジェクトで処理を実行し、もう一つの値である環境オブジェクトをたどることにより実行環境の連鎖を実現している。メソッド一回の呼出しで最低 4 ワードの環境スタックを消費する。引数およびローカル変数が N 個ある場合は、(4+2N) ワード分環境スタックを利用するが、再帰などが深くなければそれほどメモリを浪費しないと考えている。

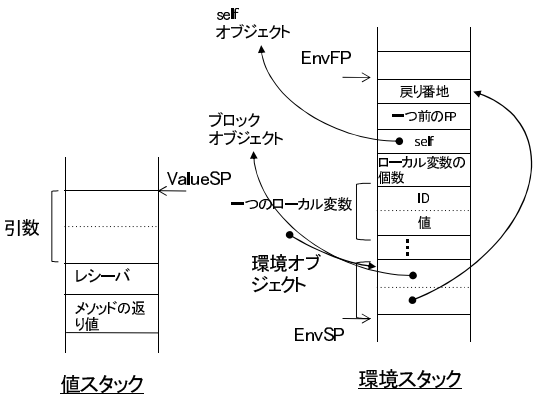


図 9 Dolittle の実行環境

6.5 μ Dolittle におけるプロトタイプオブジェクト

現行の Dolittle は、通常の PC が有する各種資源を用い、GUI、ネットワーク、音楽などのプロトタイプオブジェクトを提供している。しかし、**μ Dolittle** は組込みシステムでの稼働を目標としていることから、実行に必要最低限のプロトタイプオブジェクトを提供するにとどめている。加減算などの基本的な演算なども、処理系組込みのメソッドで処理される。これら処理系組込みのメソッドも基本的なものに限定し、VM のサイズを小さくする設計とした。必要ならば、Dolittle 自身で必要なメソッドを記述すればよい。次のプロトタイプオブジェクトと処理系組込みメソッドを用意した。基本的なオブジェクトについては、次のものがある。

- 整数オブジェクト
  四則演算 (加減乗除)、比較、ビット演算 (積和、排他的論理和、ビット反転、シフト)、端末への出力
- 文字列オブジェクト
  インスタンス生成 (`create`)、最大長・長さの取得、文字列複写、文字設定、文字取出し、文字列比較、端末への出力
- ベクタ (配列) オブジェクト
  インスタンス生成 (`create`)、最大長・長さの取得、要素の値取得、要素へ書込み

これら以外に、ブロック、ルート、未定義オブジェクトなどがあるが、これらは主に制御構造に用いられる。

Dolittle は、すべてオブジェクトのメソッド呼出し (メッセージ送信) で実行が行われる。条件実行、繰返しなどの制御構造も基本的にはブロックオブジェクトへのメッセージ送信で実現されている。例えば、選択構造については、次のメソッド呼出しで記述できる。なお、次の `[...]` はブロックオブジェクトである。

- `[COND] ! then [BLOCK`$_{\texttt{then}}$`] exec`
  `[COND]` が成立したとき (非ゼロの整数値) は、`[BLOCK`$_{\texttt{then}}$`]` を実行する
- `[COND] ! then [BLOCK`$_{\texttt{then}}$`] else [BLOCK`$_{\texttt{else}}$`] exec`
  `[COND]` が成立したとき (非ゼロの整数値) は `[BLOCK`$_{\texttt{then}}$`]` を実行し、成立しないとき (整数値ゼロ) は `[BLOCK`$_{\texttt{else}}$`]` を実行する。メソッドの戻り値は、どちらかのブロックを評価した値となる
- `[COND1] ! then [BLOCK`$_{\texttt{then1}}$`] else [COND2] then [BLOCK`$_{\texttt{then2}}$`] else [BLOCK`$_{\texttt{else}}$`] exec`
  `if(COND1) then [BLOCK`$_{\texttt{then1}}$`] elseif(COND2) [BLOCK`$_{\texttt{then2}}$`] else [BLOCK`$_{\texttt{else}}$`]` の条件実行を行う

繰返し構造については、`while` メソッドを提供している。繰返しについては、

`[COND] ! while [BLOCK`$_{\texttt{body}}$`] exec`

となっており、[COND] が成立している間 [BLOCK$_{body}$] を実行する。現行 Dolittle では repeat と呼ばれる N 回繰返しを行う

```
[BLOCKbody] ! (N) repeat
```

メソッドが組込みメソッドとして用意されているが、μ **Dolittle** では、組込みメソッドとして提供していない。図 10 のようなメソッド定義により同等の機能を記述できる。

```
BLOCK: repeat = [
  |count ; i e |
  e = self. i = 0.
  [i < count] ! while [
    e ! (i) exec.
    i = i + 1] exec.
  i
].
```

図 10 while により repeat メソッドを定義した例

制御構造については、これら以外に、ブロックの途中で処理を中断しそのブロックを抜ける last メソッド、繰返しを途中で中断する break、オブジェクトのメソッド呼出しを終了する return などのメソッドがある。さらに、論理式の結合である and、or のメソッドも実装した。

6.6 デバイス操作のオブジェクトとメソッド

本稿の大きな目的の一つに装置制御がある。VM は OS 上で動くのではなく、ハードウェア上で直接動作し、入出力装置を処理するプログラムを記述する。装置制御のプログラム記述の観点からは、もっとも基本的なオブジェクトとして、入出力装置の I/O ポートを操作するオブジェクトとして PORT オブジェクトを提供する。

- PORTinstance = PORT ! (I/Oaddress) create.
  PORT オブジェクトをプロトタイプオブジェクトとし、I/Oaddress を I/O ポートとするオブジェクトを生成する
- val = PORTinstance ! in8.
- val = PORTinstance ! (VECTOR) in16.
- val = PORTinstance ! (VECTOR) in32.
  PORT のインスタンスからデータを読み込む。整数オブジェクトの表現範囲を越えるので 16 ビット以上のデータについては VECTOR オブジェクトに格納する (いずれは、バイトストリームのプロトタイプオブジェクトを導入する予定である)。
- PORTinstance ! (INT) out8.
- PORTinstance ! (VECTOR) out16.
- PORTinstance ! (VECTOR) out32.
  PORT のインスタンスにデータを書き込む。

以上が入出力ポート操作であるが、割込み処理については SIGNAL オブジェクトを導入する。

- SIGinstance = SIGNAL ! (SIGID) create.
  SIGNAL オブジェクトをプロトタイプオブジェクトとし、SIGID を割込み番号とするオブジェクトを生成する
- SIGinstance ! mask
- SIGinstance ! unmask
  その割込みを許可・不許可にする
- SIGinstance ! OBJ [handlerBLOCK] sethandler
  割込み発生時の割込みハンドラを登録する。割込み発生時、OBJ をプロトタイプ連鎖の起点として、[handlerBLOCK] を実行する。なお、このブロックには引数として、割込みを発生した SIGinstance を渡して実行する

PORT オブジェクト、SIGNAL オブジェクトともに原稿執筆段階では未実装であるが、近日中に実装する予定である。

なお、現行 Dolittle では、タイマオブジェクト、スレッドを提供しているが、μ **Dolittle** ではスレッドの実装は資源が不足することから困難ではないかと考えている。タイマについては、実タイマと精度とのトレードオフであるが、数個分の仮想タイマを提供する必要があるかもしれない。

7. 処理系の実装と評価

原稿執筆時点でμ **Dolittle** は、コンパイラ、VM ともに稼働している。表 3 の実行環境上で実現を行った。

表 3 μ **Dolittle VM** で実装で利用した CPU

CPU	OS	モード	クロック	ROM	RAM
x86	Cygwin, Linux, FreeBSD	16,32	1-2GHz	-	1GB
ARM7	-	16	18MHz	256K	64K
ATmega128	-	16	16MHz	128K	32K
H8/3687F	-	16	20MHz	56K	4K

ARM7(AT91SAM7S256)、ATmega128、H8/3687F では、コンパイラは PC の Cygwin 上で稼働、VM をターゲットの ROM に格納している。PC とターゲットは、USB または RS-232C で接続されており、コンパイルされた VM コードを転送した上でプログラムを実行する形態となっている。現在四つの CPU 上で μ **Dolittle** VM が稼働している。コンパイラは言語 C で約 1000 行、VM については約 2000 行程度の大きさである。ROM、RAM の使用量を表 4 に示す。

表 4 μ **Dolittle VM** のコードサイズ

CPU	テキスト	データ
ARM7	26KB	46KB
ATmega128	28KB	24KB
H8/3687F	21KB	3KB

現時点では ROM 上にコンパイルされた VM コードを格納するルーチンを作成していないので、コンパイルされたコードはすべて RAM 上に配置されている。もっとも条件が厳しいのは H8/3687F であるが、VM コード領域を 384 ワード、ヒープ＋値スタック＋環境スタック合わせて 320 ワードを割り当てて動かしている。それほど大きなプログラムを実行できないと思われるが、簡単なプログラムなら一応実行はできている。ARM7 については 16 ビットモードのほぼ上限に近いサイズのヒープ、スタックを割り当てている。ATmega128 は 16 ビットモード上限の 2/3 程度を割り当てているが、いずれも RAM の使用量とほぼ一致している。ROM については、三つともまだ余裕がある。コンパイルされた VM コードを ROM に格納する方式が有効であるが、ARM7 と ATmega128 は ROM の書換え回数上限が 1 万回と VM コードを書いても問題はないのに対し、H8/3687F は ROM の書換え回数上限が 1000 回と少ないことから、ROM に VM コードを置かざるをえない。

実行速度の評価については現在詳細を計測中であるが、基本的な実行時性能として、繰返しのブロックを空にして、(1)10000 回の `while` ループ、(2)10000 回の `repeat` ループ、(3)100x100 の 2 重の `repeat` ループの実行時間を計測した。表 5 に測定結果を示す

表 5　μ Dolittle VM の実行速度 (10000 回の繰返し)

CPU	(1)	(2)	(3)
ARM7	5.0	4.7	4.7
ATmega128	13.6	13.6	13.6
H8/3687F	15.2	16.5	16.8

(単位は秒)

ループ一回あたり 500 μ s〜1.5ms 程度の実行時間となっている。三つともクロックに大きな差はないが、実行速度は 3 倍程度の開きがある。プロセッサの内部アーキテクチャ、AVR については外部メモリを用いたことで性能に差が生じている。いずれにせよ、繰返しのオーバヘッドが大きいのは、連鎖をたどる処理が線形探索になっていること、プロトタイプオブジェクトのメソッド探索は連鎖の先にあることなど、検索を高速化する工夫が必要である。

今回は、これら 4 種類の CPU で実装を行ったが、評価に用いた CPU を搭載した Lego Mindstorms NXT への移植、他の CPU である ColdFire、PIC への移植作業に着手したところである。

8. ま と め

本稿では、ROM が数 10KB、RAM が数 KB の組込みシステム向けのプロセッサで実行可能なオブジェクト指向言語 Dolittle の一実装について示した。Dolittle の言語仕様で資源を必要とする部分を見直し、小型計算機向けの言語仕様とした。また、VM 構成の処理系とし、VM の機械語を定め、その機械語を解釈実行する VM を、ターゲットとする組込み向けプロセッサ上に実装した。プロトタイプベースのオブジェクト指向言語、実行環境の連鎖などの機能を提供し、RAM が 4KB の計算機システム上でも稼働している。この結果、組込みシステムやユビキタスコンピューティングのノードの開発環境を整えることができた。ただし、プロパティ検索や連鎖をたどる処理などのオーバヘッドが若干大きいので、今後性能解析を行い、実行時オーバヘッドをより減らすことを検討すべきであろう。

実装としてはまずは最初の一歩が動いた、と言うところであり、各種プロトタイプオブジェクトの整備、分散協調を行うためのホスト PC ないしは他ノードとの通信機構の導入、各種 AP の整備と評価などが今後の課題である。

参 考 文 献

1) 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:プログラミング, vol. 42, No. SIG 11 (PRO 12), pp. 78-90, 2001.
2) 中谷多哉子, 兼宗　進, 御手洗理英, 福井眞吾, 久野　靖, オブジェクトストーム：オブジェクト指向言語による初中等プログラミング教育の提案, 情報処理学会論文誌, Vol.42, No.6, pp1610-1624, 2002.
3) 兼宗　進, 久野　靖, ドリトルで学ぶプログラミング — グラフィックス、音楽、ネットワーク、ロボット制御 —, イーテキスト研究所, 2008.
4) http://processing.org/
5) http://wiring.org.co/
6) http://www.arduino.cc/
7) http://www.recursion.jp/mitou17/
8) http://uc.sfc.keio.ac.jp/xtel/
9) 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, 情報処理学会研究報告, Vol.2007-UBI-16, pp.51-55, 2007.
10) 日本ロボット学会, 人工知能学会, 日本人間工学会, ロボット分野に関するアカデミック・ロードマップ報告書, II, 情報系複合領域のアカデミック・ロードマップ, 経済産業省,(株)KRI, 2008.