環境反応型発光装置の開発
～GCを備えた高級言語による組み込み用マルチタスクOSの開発～

材料技術室　石川 隆朗

Development of Illumination System that Reacts to Environmental Conditions
～ Development of Multitasking OS Using High-level Language with GC for Embedded System ～

Takaaki ISHIKAWA

周囲の環境条件をセンサ等で読み取り，インテリジェントに反応し発光法を変化させるイルミネーション装置の開発を行う。

インテリジェントなイルミネーション装置を開発するにあたって，GC(Garbage Collector)を備えた組み込み開発用高級言語の開発を行った。併せて，マルチタスクOSの機能を備えた。本研究で開発した処理系，OSをルネサスエレクトロニクス株式会社製マイクロプロセッサ，RX63Nを搭載したGR-SAKURAマイコンボード上に実装を行った。5×5のダイナミック点灯を行うLEDマトリクス上に5-queenの解を表示させた。

１．はじめに

イルミネーションはそれだけで観光の名所になり，街角の広告，ディスプレイは我々の目を日夜楽しませている。

近年では情報機器の進歩により静止したディスプレイのみならず，デジタルサイネージとして，表示内容を時々刻々と変化させ，顧客に強い印象を与えることに成功している。飲料の自動販売機では，センサを用い，周囲の環境を読み取り，顧客に最適な情報を表示し，購買意欲を高めるものが実用化されている。

このように周囲の温度，音，人の様子等の環境情報をセンサ等で読み取り，それに対してインテリジェントに反応し，観客，顧客に強いインパクトを与える発光装置(イルミネーション装置)を作成することが，本研究の最終的な目標である。

このようなイルミネーション装置を作成するには顧客やデザイナーと試作品(プロトタイプ)を何度も作りながら打ち合わせを行い作成しなければいけない。従って開発環境はラピッドプロトタイピング性が重要となる。

本研究では，このようなイルミネーション装置を作成するに当たって以下の要求に応えることを目標とした。

まず，24時間風雨に曝される環境で，メンテナンス無しで稼動し続ける頑強性(robustness)を持つこととした。

次に，コントローラー自体は小型で，その存在がイルミネーション装置全体の中で目障りにならないことも要求される。

また，設置の自由度が要求されることから，AC100Vの電源が得られるとは限らないため，小さなバッテリーで動くよう低消費電力でなくてはならない。現在のインテリジェントなイルミネーション装置は制御にPC，もしくはそれに類するものが使用されており，一般に大消費電力である。

以上の要求を満たすためには，組み込み用マイクロプロセッサが最適である。

組み込み用マイクロプロセッサはI/OやRAM等の周辺装置がワンチップに収められているため，外部配線が少なくてすみ,非常に頑強である。また，同時に小型である。消費電力もLED等と比較すると無視できる程度である。

組み込みプロセッサ開発は主にC/C++言語で行われている。C/C++言語による開発は一般にGC(Garbage Collector)を利用せずに行われる。GCを使用しないことにより，プログラマは常にオブジェクト(ここでは計算機上の記憶領域を占有するデータ類のことを示す。所謂，オブジェクト指向開発でのオブジェクトのことではなく，それを含む

ものである。)の寿命を意識して開発を行う必要がある。このことは本研究で対象とする複雑なデータ構造を扱うインテリジェントなプログラムの作成では開発を阻害する要因となる。オブジェクトの寿命を適切に扱えなかった場合バグとなる。再現の困難な不可思議なバグはオブジェクトの寿命を適切に扱えなかったことが原因であることが多く，頑強なプログラム開発にはGCの存在が非常に有効である。

加えて，本研究で対象とする複雑なイルミネーション装置は一つ以上のセンサ等の情報源からデータを得て，一つ以上のアウトプット対象をコントロールする必要がある。そのため，プログラムの動作は複数の処理を時分割，同時並行で行うマルチタスク動作となる。よって，各タスクごとに適切にCPU時間や記憶領域等の計算機資源を分配する必要がある。また，I/Oポート等のハードウェア資源も抽象化する必要がある。

周囲の環境に反応しインテリジェントな反応を示す発光装置(イルミネーション装置)を開発するにあたり，GCを備えた高級言語の開発を行った。併せて，その言語処理系を動作させるOS機能も実装した。

２．実装

２．１　GCを備えた高級言語

PC上のソフトウェアの開発でのラピッドプロトタイピング性を高めるために所謂LL言語(Lightweight Language)が用いられている。LL言語では型情報をオブジェクトに持たせ，実行時に型解決させる等，実行させながらデバッグ等の問題解決を行うスタイルで開発が行われる。組み込み開発では動作は実機にプログラムをダウンロードさせる必要があり，プログラム実行のレスポンスはPC上のソフトウェアと比較して劣る。また，プログラムのバグにより実機の物理的損傷が起こる場合も考えられる。組み込み開発でラピッドプロトタイピングを行う場合，実機へのダウンロード前にできるだけのバグが除去できることが望ましい。

プログラムの正当性を，プログラム実行時に動的に確かめるのではなく，記述されたプログラムを静的に確かめる手法として関数型プログラミング(functional programming)という考え方がある。

本研究では関数型プログラミングの手法から以下の3点の特徴を本実装言語に採用した。

1. 手続きが自由に変数に束縛できるファーストクラスのオブジェクトであること
2. 手続きがデータを束縛できるクロージャ(closure)であること
3. 末尾呼び出しの最適化(tail call optimization)を行うこと

手続きがファーストクラスのオブジェクトであることにより，既存の手続きを組み合わせて新たな手続きを記述する関数型パラダイムでのプログラムの記述が可能となる。手続きがクロージャであることにより手続きの記述の自由度が高まる。

頑強でバグの少ないプログラムを書くためには命令型(imperative)的な記述ではなく宣言型(declarative)的に記述した方が良いと考える。

階乗の計算を例に例えると命令型は

$$\mathrm{fact(n)} = \mathrm{n} \times (\mathrm{n}-1) \times \cdots \times 2 \times 1$$

という表記になり，宣言型は

$$\begin{cases} fact(1) = 1 \\ fact(\mathrm{n}) = \mathrm{n} \times \mathrm{fact}(\mathrm{n}-1) \end{cases}$$

のような記述になると言える。自然言語で表記すると命令型は

・階乗とは1から順にnまで一つずつ増やしながらかけていったもの

という記述になり，宣言型は

・階乗とは1の場合1
・nの場合n-1の階乗にnをかけたもの

という表記になる。一般に宣言型の方が表記が簡潔であると考えられる。

もう少し複雑な例としてユークリッドの互除法による最大公約数の計算を示す。ユークリッドの互除法の手順は入力される2つの数字をm,nとすると以下のとおりとなる[1]。

1. nがmより大きい場合mとnを入れ替える
2. nが0の場合，解がmと決定する
3. nをm，mとnの剰余をnとし再度計算を行う

これを仮想的な言語で命令型的に記述したものが図１，宣言型的に記述したものが図２となる。

```
GCD(m, n)
loop {
  if (m < n)
    swap(m, n)

  if (n = 0)
    return m

  m_next = n
  n_next = m mod n

  m = m_next
  n = n_next
}
```

図１　命令型的記述によるユークリッドの互除法

```
GCD(m, n) = m < n:      GCD(n, m)
            n = 0:      m
            otherwise: GCD(n, m mod n)
```

図２　宣言型的記述によるユークリッドの互除法

命令型的記述の場合，ブロックの最初から最後まで処理の流れを意識しなければならないが，宣言型的記述ではこの必要がなくなる。特に今回の例では命令型的記述で最後のmとnの代入の扱いを誤るとバグとなる。

宣言型的記述は命令型的記述ならばループ構文で表すところを再帰呼び出しで表現することとなる。末尾呼び出しの最適化の機構を持っていない場合，呼び出しごとにコンテキストの情報を記憶する必要があり，長大な繰り返しでは記憶領域の枯渇を招く。関数型プログラミング，宣言型的記述を活用するためには末尾呼び出しの最適化の機構を持つことが重要である。

以上の特徴を持った言語のプロトタイプとしてScheme言語のサブセットとなるLisp処理系の開発を行った。

作成にあたってはPeter J. LandinのSECDマシン[2]を拡張したSECDRマシン[3]であるAtsushi MoriwakiのMini-Schemeを参考にした。

Mini-SchemeのSECDRマシンと同様の動作を行うVM(Virtual Machine)をC言語で記述し，Schemeのコードを VMのコードの列に変換するコンパイラをHaskell言語で記述した。

以下に実装の概要を示す。

２．１．１ Garbage Collector

GCは今後の拡張を考え，最もシンプルなMark & Sweep法を採用した。

２．１．２ クロージャの生成機構

lambda構文により，その時点の環境(変数の束縛情報)と手続きのコードのエントリポイントがパックされたクロージャオブジェクトが生成される。define構文等で変数への束縛を行うことができる。

２．１．３手続き呼び出し

図３のSchemeのコードを実行することを考える。

本処理系のVMはS(Stack)，E(Environment)，C(Controlもしくは Code)，D(Dump)，R(Return)のレジスタを持つ。Sレジスタは評価が終わった引数を積むことに使われる。Eレジスタにはその時点での変数の束縛環境が保持される。Cレジスタは実行しているコードを指すプログラムカウンタである。

図3コード中のgの呼び出しが行われる状況を考える。gの呼び出しはfの呼び出しが行われる際の引数解決の状況で行われる。

gが呼び出される前，定数4の評価が終了した後のS,E,C,Dレジスタの状況を図４(a)に示す。(本実装は，現在，引数解決は後ろから順に行われる。) 引数が解決されるたびにレジスタSから連なるリンクトリストに引数が連なることとなる。g呼び出しの引数解決を行う前に現在のSレジスタから連なるリンクトリストがDレジスタから連なるリンクトリストに退避され，Sレジスタを空とする。この状態から定数3，2の解決，Sレジスタへのリンクが行われる。(図４(b)) gの呼び出しはDレジスタに呼び出し前のEレジスタ，gからの帰り先のコードポイントが退避され，Eレジスタにはgに束縛されたクロージャの環境が，Cレジスタにはgに束縛されたクロージャのエントリポイントが束縛され，呼び出された手続きの実行が始まる。手続き実行時の変数束縛

環境はクロージャ生成時のものとなるため，レキシカルスコープとなる。g呼び出し直後のレジスタの状況を図４(c)に示す。

手続きから戻るのはDレジスタに積まれたS,E,Cレジスタの内容を戻すことによって行われる，返り値はRレジスタに保持される。

```
(f 1 (g 2 3) 4 (h 5))
```

図３　Schemeコード例

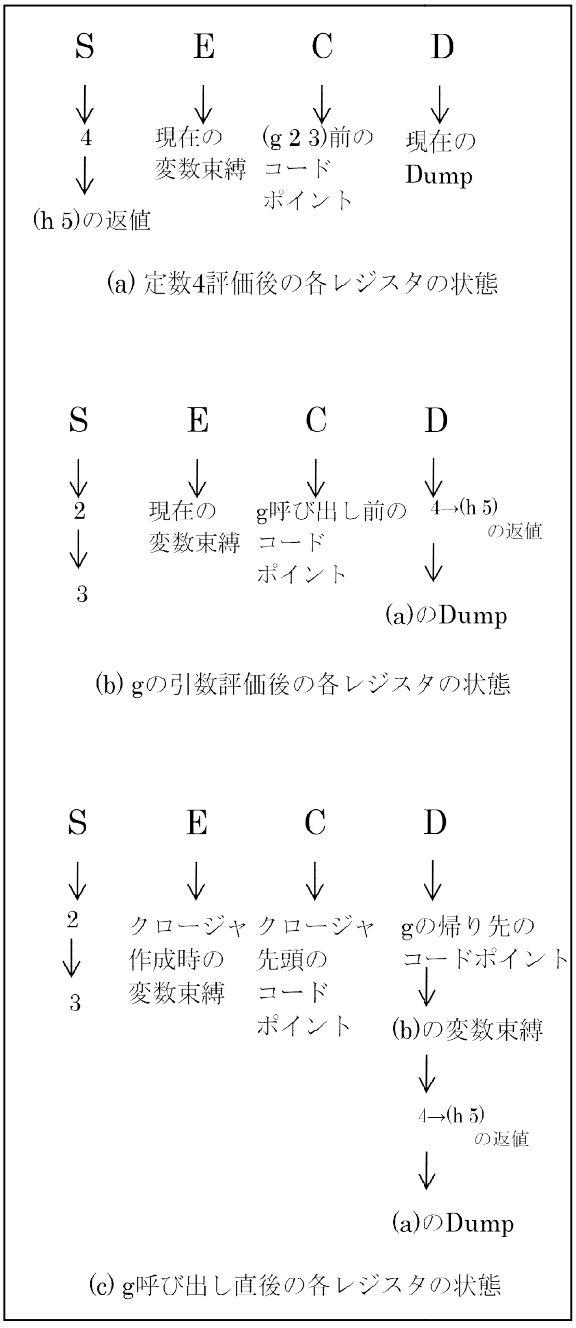

図４　関数呼び出し時の各レジスタの変化

２．１．４ 末尾呼び出しの最適化

末尾呼び出しの最適化はSECDRマシンでは2.1.3の手続き呼び出しでのS, E, CレジスタのDレジスタへの退避を行わないことによりなされる。

本処理系ではコンパイル時に手続き呼び出し後，何もせずに手続きから抜けることが判明したとき,Dレジスタに各レジスタを積む処理を除去することによって行われる。

２．２ OS機能

実在の計算機を動かすためには,CPU時間や記憶領域等の計算機資源を適切に割り振る必要がある。また，ハードウェアを抽象化し，ユーザに利用し易くすることも重要である。

上のことを行う機構，プログラムはOSと呼ばれる。PCでは所謂UnixやMicrosoft社のWindowsが用いられる。

小規模な組み込み開発ではOSを利用せずアプリケーション開発者が自身で制御コードを記述する場合もあるが，組み込み用OSが利用される場合もある。それらのOSはC言語で記述され，C言語を用いて利用されることを期待して開発されている。

本研究で開発される言語処理系を動作させることを念頭に置いたOS機構の作成を行った。以下にその特徴を記述する。

２．２．１ コンテキストスイッチング

本OS機構は手続き呼び出し等を行うとき言語処理系のVMがコンテキストスイッチング(タスクスイッチング)を行う。タイマ割り込み等を使い，処理の途中で強制的にコンテキストスイッチングを行うプリエンプティブマルチタスクではない。

これにより,CPUレジスタ等の情報を退避する必要がなくなり，最低限の情報を保持すればよくなる。

ノンプリエンプティブマルチタスクではユーザプログラムがOSに処理を返さないことにより,コンテキストスイッチングが行われなくなることが問題となるが，本OS機構ではVMにより細かい粒度でコンテキストスイッチングが行われるためそのようなことが起こらない。

２．２．２ コンテキスト情報の保持

手続き呼び出しによって入れ子的に実行されるプログラム情報の保持はスタックと呼ばれるデータ構造によって行われる。本処理系ではDレジスタがそれとなる。

スタックはCPUのスタックポインタ等を用い連続したアドレス空間上に構築する処理系が多い。その場合ベースアドレスを起点とする無限に伸びる半直線状の記憶領域を原理的に必要とし，その大きさを見積もることは困難である。各タスクごとにこのスタックの保持が必要となる。PCでは仮想記憶という機構を用い，各タスクごとに仮想的に無限のアドレス空間を与えることによりこの問題を解決しているが，小規模な組み込み開発ではそのようなことを行うことはできない。

本言語処理系はコンテキスト情報を，ヒープ上に不連続なリンクトリストとして保持している。このことにより，ヒープ領域に空きがある限り，各タスクはコンテキスト情報を保持することが可能となる。

３． 処理系の実装

本処理系はScheme言語のsubsetで記述されたプログラムをHaskell言語で記述されたコンパイラによってVM用のコード列に変換し，C言語で記述されたVMによって実行される。

３．１ PC上での実行

本処理系をPC(1.7GHzのCore i5 CPUを搭載したApple社製MacBook Air)上で動作させ8-queen問題の亜種である10-queenを解かせたところMini-Schemeの約2倍の速度で動作した。日本国内で最も広く利用されているScheme処理系であるGaucheと比較すると約1/13の速度となった。Ruby言語としては不利なアルゴリズムでの比較ではあるがmruby処理系と比較した場合約1/2の速度であった。

３．２ 組み込みマイクロプロセッサへの実装

本処理系をルネサス エレクトロニクス株式会社製RX63Nマイクロコンピュータを搭載したGR-SAKURAマイコンボード上に実装した。

5×5のLEDマトリクスを用い5-queen問題の解の表示を行った。タクトスイッチを押すことにより，5-queenの10の解を順に表示する。5-queenは最初に1度解を求めればよいが，本サンプルプログラムでは繰り返し，解を求め続けるものとした。

上記の動作をさせるためには

1. 5×5のLEDマトリクスのダイナミック点灯
2. ソフトウェアによるチャタリング除去を行ったスイッチ入力
3. 5-queenのソルバ

を非同期で動作させなくてはならない。

以上のことを行うプログラムが空行，コメント行を含めて130行程度で記述することができた。5-queenのソルバを除いたハードウェア制御部分は80行程度である。

４．まとめ

鑑賞者に強い印象を与えるために周囲の環境情報を集め，インテリジェントな反応を示す発光装置(イルミネーション装置)の開発を行うため，GCを備えた高級言語と，それを動作させるOS機構を作成した。その処理系をルネサスエレクトロニクス社製RX63Nマイクロコンピュータを搭載したGR-SAKURAマイコンボード上で動作させた。

ある程度複雑な動作である5-queenのソルバ，ハードウェアの制御が130行程度のプログラムで記述することができた。

今後は，型推論を備えた静的型システム，代数的データ型等，関数型言語の進歩によって得られた知見を取り込み，組み込み開発でより簡潔で頑強なプログラミングを行うことができる環境を構築したい。

参考文献

1) Wikipedia:ユークリッドの互除法
2) P. J. Landin: The Computer Journal, 6 (1964), 308-320
3) http://storage.osdev.info/pub/idmjt/diaryimage/1204/secdrscm.pdf