ソフトウェア基礎論　27—10
プログラミング言語　19—10
（1988. 12. 9）

ソフトウェア開発環境記述用関数型言語の設計と処理系の試作

DESIGN OF THE PROCESS DESCRIPTION LANGUAGE AND ITS INTERPRETER

荻原剛志[+]　飯田元[+]　新田稔[++]　井上克郎[+]　鳥居宏次[+]
Takeshi OGIHARA　Hajimu IIDA　Minoru NITTA　Katsuro INOUE　Koji TORII

[+] 大阪大学基礎工学部　Osaka University
[++] ＳＲＡ　Software Research Associates, Inc.

あらまし　我々はソフトウェアの開発過程を形式的に記述し，実行するための関数型言語ＰＤＬ(Process Description Language)およびそのインタプリタを作成した．ＰＤＬでは開発過程をツールの起動やウィンドウ操作の系列として記述する．ＰＤＬはこれらの操作のための関数や複数の操作を並列実行するための関数を持つ．また，さまざまなマクロ機能があり，定義の記述を容易に行うことができる．ＰＤＬインタプリタは，実行中に検出した未定義関数をそのつどユーザに定義させる機能などを持ち，十分詳細化されていない記述も実行可能である．また，デバッグ機能やヒストリ機能，関数定義の画面編集機能などの機能も備えている．我々はすでにＪＳＤ（ジャクソンシステム開発法）など，いくつかの開発技法をＰＤＬで記述し，実行している．ＰＤＬインタプリタは現在，いくつかのUNIXワークステーション上で稼働中である．

Abstract　We have developed a functional language PDL(Process Description Language) and implemented its interpreter. PDL is designed for constructing support systems of various software development processes. PDL scripts(programs) are composed of tool activation and window control functions. They also have functions to evaluate those functions concurrently. Various macro notations are provided to improve readability and to help stepwise refinement of the abstracted scripts. The PDL interpreter allows to assign values or give definitions to undefined functions detected during execution. It also has various facilities for debugging, command history, screen editing of the definitions. PDL interpreter has been implemented on several UNIX workstations, and we executed relatively large programs such as JSD (Jackson software development).

1．はじめに

最近，ソフトウェアの開発過程を厳密に定義しようとする試みが盛んになっている[1],[2]．従来，ソフトウェア開発のための指針や考え方は示されていても，それらをどのように解釈し，適用するかはそれを用いる人によって異なる場合があった．これらの開発過程を定義することができれば，多くの人がその方法を理解し，曖昧なく用いることができよう．そして，作成されるソフトウェアがある水準以上の品質を持つことを保証したり，工程の管理を容易に行うことができると期待される．また，定義されたさまざまなソフトウェア開発技法を比較検討してそれぞれの性質を明らかにしたり，ソフトウェア開発を効率よく行える人の開発過程を定義して誰でも利用したりすることも考えられよう．

我々はこのような目的で作成されたソフトウェア開発法の定義から直接，その開発法を支援するシステムの生成を試みている[3]．すなわち開発法の定義を支援システムの一種の抽象的な仕様とみなし，それに順次具体的な情報を付加して詳細化し，目的とするスクリプト（プログラム）＊を得る．

我々はこれを代数的言語ＡＳＬ[4]という同一言語の枠組みで行っている．ＡＳＬはその意味が簡明に定義されており，いろいろな抽象化レベルで抽象化ができる．また検証等も比較的容易に行える．

本論文では，この詳細化の最後の段階で用いる，ＡＳＬの部分言語である関数型言語ＰＤＬ(Process Description Language) とそのインタプリタについて述べる．ＰＤＬインタプリタは現在，複数機種の計算機上で稼働中である．さらに，ＰＤＬを用いていく

＊ ツールの起動などを行うコマンド列の記述は習慣的にプログラムと呼ばず“スクリプト”と呼ぶ場合がある．本論文でもＰＤＬのプログラムをスクリプトと呼ぶ．

つかの開発技法の記述を進めているが，これらのスクリプトは実際にインタプリタ上で実行可能である．

2章ではＰＤＬとそのインタプリタの特徴を挙げる．次にＰＤＬについて，3章で構文と意味，4章で組込関数，5章でマクロ機能を説明する．ＰＤＬインタプリタについては，6章でインタプリタへのコマンド，7章でその他の特徴，8章でその実現方法を述べる．9章で簡単なスクリプトの例を示し，１０章でまとめを与える．

2．ＰＤＬとそのインタプリタの主な特徴

2.1 ＰＤＬの特徴

(1) 代数型言語ＡＳＬと同様の構文および意味定義を持つ関数型言語である．従って，あるＰＤＬ記述が元のＡＳＬの抽象的記述の詳細化になっているかなどの検証を比較的容易に行える．（3.2を参照）

(2) 基本関数としてツールの起動，ウィンドウのオープン，クローズなどの操作を行う関数を備えている．これらの関数を組み合わせて開発過程の支援システムを構築する．ただしこのシステムは，一人の人間が1台のワークステーション上で開発作業を行うことを前提としている．（4.1）

(3) 基本データ型として，ソフトウェアおよびハードウェアの状態を抽象的に表す“システム状態”がある．このシステム状態のある値に，順次ツールの起動，ウィンドウ操作等の遷移関数を施して目的のシステム状態にする．このような状態遷移をＰＤＬのスクリプト中で定義する．このシステム状態の他，整数，論理，文字列などのデータ型も扱える．（3.2）

(4) 複数のツールを並行して用いたりするために，基本関数として2つの状態遷移関数を同時に評価する関数 @ （並列演算子と呼ぶ）がある．（4.2）

(5) 記述しやすさの向上のみならず，スクリプトのライブラリ化に役立つ引数なしグローバルマクロ，引数付きグローバルマクロ，ローカルマクロの3種類のマクロ機能がある．（5.）

2.2 ＰＤＬインタプリタの特徴

(1) 抽象的な記述から得られた未定義関数を含むようなＰＤＬスクリプトでも一部実行できるよう，実行中検出された未定義関数の値をユーザが決めたり，関数定義を与えたりできる．（7.4を参照）

(2) システム状態は他の整数等のデータ型と異なり，その値をスタック等に保存せず，いわゆる大域データ[5]として扱う．ただし実行しようとするスクリプト中で最新のシステム状態ではなく，過去のシステム状態に対する操作の記述がある場合は，それを実行しようとする際に警告を出す．（7.1）

(3) 定義ファイルを取り込む機能 #include があり，それを用いて個人の好みのツールやその設定等を登録しておき，ライブラリ化できる．（6.2）

(4) 関数定義を画面編集したり，過去のコマンドの履歴を利用したりする，使いやすさのための種々の機能を持つ．（6.2）

3．プロセス記述言語ＰＤＬの構文と意味

3.1 構文

ＰＤＬスクリプトは関数定義の集合および変数を含まないひとつの評価すべき式から構成される．

ＰＤＬの関数定義は以下のように，定義する関数名と仮引数リストを '==' 記号の左辺に，関数を定義する式を右辺に記述する．

関数名 （ 仮引数1，仮引数2，... ） == 式 ；

定義は複数行にわたってもよい．定義の末尾には ';' を置く．また，'//' から行末まではコメントである．

式は組込関数（if関数，@関数を含む），定義関数，定数および左辺に現れる仮引数（変数）から構成される．主な組込関数の一覧を表1に示す．これらのうちで中置記法で用いるものは通常のプログラミング言語と同様な優先順位を持っており，式の記述をわかりやすく行うことができる．

条件判断を行うための if関数は

if 条件式 then 式1 else 式2

という形式をしており，条件が真の場合式1，偽の場合式2をこの if関数の値とする．

例として，階乗の計算を行う関数 factを以下に示す．

```
fact(n) == if n=0 then 1 else fact(n-1)*n;
```

現在，ＰＤＬでは以下のデータ型を使用できる．

システム状態　　整数　　文字列
論理　　タプル

文字列は " " で囲んで表す．論理定数にはtrueとfalseがある．タプルはいくつかのデータの組であるが，タプル自体を要素として含めることはできない．タプルは

［ 式1，式2，... ］

のように表現する．システム状態については次節で述べる．

なお，関数の引数やタプルの要素にシステム状態が含まれる場合には，習慣的にシステム状態を最後に記述する．

3.2 意味

スクリプトの意味は，式の合同関係を用いて定義される[6]．組込関数の定義（例えば 1+1==2, 1+2==3, ... ）および関数定義は，左辺と右辺の式が関数の適応という演算で閉じた合同関係を表すものとする．この合同関係から得られる最小の同値類分割を考え，評価すべき式の同値類中の構成子項（定数）をそのスクリプトの意味と定義する．

与えられた式は，関数定義の内側，左側から順次評価される．関数の引数には評価された値が渡される(call by value).

ＰＤＬはデータ型のひとつとしてシステム状態を持

つ．システム状態とは，ファイルシステムやその他の計算機資源の状態すべてを抽象的に表すもので，実行中のどの時点においてもある値が１つ存在する．システム状態はいくつかの組込関数（Ｃシェルコマンドの使用やウィンドウ操作など）によって状態が変化するが，これはシステムの状態遷移であると見なせる．従って，システムへの操作を行おうとするＰＤＬスクリプトは，システムの状態遷移関数の組合せであると考えることができる．

従来の手続き型言語では，手続き呼び出しの副作用としてデータ（変数）を書き換えたり計算機資源にアクセスしたりしていた．これに対し，ＰＤＬはシステム状態を遷移させる関数の集合としてスクリプトを構成する．

４．組込関数

4.1 システム機能を呼び出す組込関数

ＰＤＬは，ツールを起動したりウィンドウを操作するなど，定義関数で実現できない機能を実現するために組込関数をいくつか用意している．これらの機能はＣシェルおよびウィンドウシステムを利用して実現する．

Ｃシェルの機能を利用するには組込関数 exec を用いる． execは引数として与えられた文字列をＣシェルにそのまま渡して実行させる．結果の文字列を得る時には代わりに execstr関数を用いる．例えば，more というツールを用いてファイルの内容を表示させるには，

表1. PDLの組込関数

関数と引数	結果	意味
I + I	: I	加算
C + C	: C	文字列の連結
I - I	: I	減算
- I	: I	負号化演算
I * I	: I	乗算
I / I	: I	除算
X = X	: B	等しい
X <> X	: B	等しくない
X > X	: B	大きい (1)
B | B	: B	論理和
B & B	: B	論理積
! B	: B	論理否定
S @ S	: S	並列演算
exec(C,S)	: S	Ｃシェルコマンドを実行する
execstr(C,S)	: [C,S]	Ｃシェルコマンドを実行し，結果の文字列を得る
status(S)	: I	Ｃシェルのステータスコードを得る
wopen(S)	: S	ウィンドゥを開く
wclose(S)	: S	ウィンドゥを閉じる
read(S)	: C	ウィンドゥから一行読み込む
write(C,S)	: S	ウィンドゥに文字列を書き出す
readc(S)	: C	コンソールから一行読み込む
writec(C,S)	: S	コンソールに文字列を書き出す
term_tool(C,S)	: S	ウィンドゥツールを指定する
break(X)	: X	ユーザ設定ブレーク
element1([...])	: X	タプルの１番目の要素を取り出す (2)
substr(C1,I1,I2)	: C2	C1のI1文字目から長さI2の部分文字列を取り出す
index(C1,C2)	: I	C1中でC2の現れる位置（左から探索）
rindex(C1,C2)	: I	C1中でC2の現れる位置（左から探索）
strlen(C)	: I	文字列の長さを得る
stripnl(C)	: C	文字列中の改行や連続した空白を取り除く
atoi(C)	: I	文字列を整数に変換する
itoa(I)	: C	整数を文字列に変換する
field(C,I)	: C	空白を区切りとするＩ番目のフィールドを得る
tempnam(S)	: C	一時ファイル名を作成する
time(S)	: C	時刻を表す文字列を得る

引数と結果の型は以下の通りとする．

S : システム状態　　I : 整数　　C : 文字列

[...] : タプル　　B : 論理　　X : 任意

(1) 大小比較には >, <, >=, <= がある． X は整数あるいは文字列．

(2) element1,element2,...,element10まで用意されている．

```
exec("more "+file, S)
```

とすればよい．ただし，fileはファイル名，Sはシステム状態を表す仮引数である．

ウィンドウを操作するために wopen, wclose などの関数がある．

スクリプトではこれらの基本的な操作をもとに，より抽象度の高い記述を行うことができる．例えば，

```
list(file,S) == exec("more "+file,S);
edit(file,S) == exec("vi "+file,S);
```

と定義しておけば，実際のツール名を陽に用いないでスクリプトを構成することができ，後からの変更が容易である．

4.2 作業を並行に実行する組込関数

ソフトウェア開発では，複数の作業を並行に行いたいことがある．例えば，別のプログラムやオンラインマニュアルを見ながらプログラムの作成を行うような状況は頻繁に生じると考えられる．

このような並列した作業を記述するために，ＰＤＬでは新たに組込関数 @ （並列演算子と呼ぶ）を導入した．この関数の引数はシステム状態であり，新しいシステム状態を返す．

いま，以下のように並列演算子 @ を用いた記述を考える．

```
F(S) == G(S) @ H(S);                                  (1)
G(S) == wclose(exec("vi tmp",wopen(S)));              (2)
H(S) == wclose(exec("view dat",wopen(S)));            (3)
```

関数 G はウィンドウを開いて vi を起動し，作業が終了するとウィンドウを閉じる．関数 H は同様にしてウィンドウ内で view を起動する．

このとき関数 F を起動すると，ウィンドウが２つ開いて一方で vi が，他方で view が起動される．両方の作業を終了させるとウィンドウは閉じられ，関数 F は新しいシステム状態を返して終了する．

関数 F の値は，引数 S の状態に対して関数 G で得られる値と関数 H で得られる値の“合成”と考える．ここでいう合成された状態とは，G の状態遷移関数（wopen, exec, wclose）と F の状態遷移関数（wopen, exec, wclose）が互いに重なり合って起こった結果の状態である．重なり方は引数のシステム状態 S に依存するものと考え，ここではその定義を陽に与えない．通常，@ の左右に記述される状態遷移はそれぞれ独立した意味を持つ．スクリプトの書き手は @ の左右の状態遷移が互いに影響を与えるような記述を行わないものとする（例えば，一方であるファイルを消去し，一方でファイルの一覧を表示するなど）．

この関数は，例にも示したようにいくつかのウィンドウを開いて別々の作業を行う場合に有用である．

なお，式の評価が複雑になるのを防ぐため，この関数呼び出しを他の関数の実引数として記述することはできない．

5．マクロ定義

ＰＤＬはわかりやすくかつ変更が容易なスクリプトを記述するために，さまざまなマクロ機能を持つ．ここではこれらについて説明する．

5.1 グローバルマクロ

グローバルマクロはいったん定義されると定義が取り消されるまで有効である．関数定義内に現れたマクロ名は定義した文字列で置換される．

引数なしグローバルマクロの定義は次の例のように行う．

```
#let MAXIMUM : 20
#let YES OK : true
```

はじめの行では MAXIMUM というマクロ名を定義している．これは関数定義内部では 20で置換される．複数のマクロ名に同じ定義を与えることもできる．２行目の例では YES と OK というマクロ名をともに論理定数 true で定義している．

引数付きグローバルマクロは次のように定義する．

```
#let TWICE(f,p) : f(f(p))
square(n) == n*n;
```

このとき，TWICE(square,2) は square(square(2)) と展開される．

グローバルマクロはどちらの場合も，

```
#unlet マクロ名
```

でマクロ定義を消去できる．

関数定義内のマクロは，インタプリタが関数定義を読み込む時に展開される．従って，マクロ定義の変更はすでに読み込まれた関数定義には影響を及ぼさない．

5.2 ローカルマクロ

ローカルマクロはひとつの関数定義内でのみ有効なマクロである．ローカルマクロは式の記述の中で以下のように定義する．

```
因子 : マクロ名
```

マクロ名は指定した因子によって置き換わる．ただし，因子とは仮引数，関数呼び出し，または括弧でくくられた式のことである．':' はローカルマクロの定義であることを示し，他のどの演算子よりも結合力が強い．

ＰＤＬは関数型言語であり，手続き型言語でいう変数の概念を持たない．このような言語では，同一の式の値を複数箇所で利用する場合，それぞれの箇所にその式全体を記述したり，別の関数にしたりする必要がある．例えば，仮引数 x に対し，test(x)が 0 以下ならば 0 を，そうでなければ test(x) の値を返すという関数 u を考える．この定義では以下に示すよう

に， test(x)を if文の条件判定部およびelse部にそれぞれ記述する必要がある．

```
u(x) == if test(x) <= 0
        then 0
        else test(x);
```

ＰＤＬではこのような煩雑さを解消するために，ローカルマクロを用いて式の記述を簡便に行うことができる．ローカルマクロを用いて上の関数 u を定義するとこのようになる．

```
u(x) == if test(x):t <= 0
        then 0
        else t;
```

これは上で記述したものと同様の定義に展開される．

6． インタプリタへのコマンド

ＰＤＬインタプリタには端末，あるいはファイルからスクリプトを入力することができる．インタプリタはスクリプトを逐一解釈実行する．前章で述べた関数定義やマクロ定義のほかに，表２に示すようなインタプリタに対するコマンドがある．

ここではまずインタプリタに式を評価させる方法を述べ，次いでいくつかのコマンドについて説明する．

6.1 式の評価

インタプリタは式の記述を直接与えられるとそれを評価し，結果の値を返す．与えられた式の記述に，引数を持たない未定義名がひとつだけ含まれていた場合，インタプリタはこれを現在のシステム状態を表す引数であると見なす．

例えば， 1+1 を与えれば 2 が返され， time(x) を与えれば x をシステム状態と見なし，現在の時刻を表す文字列を返す．

定義関数では型宣言は行っていない．実引数が組込関数の型と適合しているかどうかは実行中に検査する．

6.2 コマンド

(1) スクリプトファイルの読み込み

#include ファイル名

という行をスクリプト内に記述しておけば，その行は指定されたファイルの内容で置き換えることができる．指定したファイル内に別の #include があってもよい．

この機能を利用すればスクリプトをモジュール化して記述できる．

また，頻繁に使われる関数定義をファイルにまとめてライブラリとして利用すれば，スクリプト記述をわかりやすく容易に行うことができる．使用しているＯＳやウィンドウシステムに依存する部分をライブラリ化することにより，特定の環境に依存しないスクリプトを作成することも容易である．

インタプリタの起動時に，読み込むスクリプトファイル名を指定することができる．また，ユーザのホームディレクトリに .pdlrc というファイルがあれば，これが自動的に読み込まれる．

表2. PDLインタプリタのコマンド

#let マクロ名... : トークン列	マクロの定義
#unlet マクロ名	マクロ定義の消去
#unletAll	すべてのマクロ定義の消去
#undef 関数名	関数定義の消去
#undefAll	すべての関数定義の消去
#new	すべての関数，マクロ定義の消去
#ifdef 関数名	条件ブロックの開始（関数が既定義なら実行）
#ifndef 関数名	条件ブロックの開始（関数が未定義なら実行）
#iflet マクロ名	条件ブロックの開始（マクロが既定義なら実行）
#ifnlet マクロ名	条件ブロックの開始（マクロが未定義なら実行）
#else	else部の開始
#endif	条件ブロックの終了
#include ファイル名...	ファイルからのスクリプト入力
#list [関数名...] [> ファイル名]	関数定義の表示
#show [マクロ名...] [> ファイル名]	マクロ定義の表示
#history	ヒストリの表示
#n	ヒストリの呼び出し（n は数字）
#edit 関数名...	関数定義の編集
#exit	ＰＤＬインタプリタの終了
#help	インタプリタへの指令の一覧を表示

(2) 条件付き解釈

スクリプトの一部を条件付きで解釈させることができる.

```
#iflet マクロ名
  (1)
#else
  (2)
#endif
```

スクリプト内にこのように記述しておくと，与えたマクロ名が定義されていた場合に (1) の部分が解釈され，定義されていなければ (2) の部分が解釈される. #else および (2) の部分はなくてもよい. 逆に，マクロ名が定義されていない場合に (1) の部分を行うには，#iflet の代わりに #ifnlet を用いる. 例えば，

```
#ifnlet FILES
#let FILES: 10
#endif
```

と記述しておけば，マクロ名 FILES が定義されていない場合のみ，2行目の定義が有効になる. #iflet - #else - #endif の構造はネストさせることができる.

(3) 定義の表示

関数定義，マクロ定義はそれぞれ #list, #show で表示させたりファイルに出力させたりできる. 関数名，マクロ名を与えないとすべての定義が表示される. インタプリタの持っている定義をすべてファイル savefile に保存するには，

```
#show > savefile
#list >> savefile
```

のようにすればよい.

(4) エディタの呼び出し

関数名を指定してエディタを呼び出し，定義の編集を行うこともできる.

```
#edit 関数名...
```

とすると，指定した関数を画面上で編集できる.

(5) ヒストリ機能

インタプリタに対する以前のコマンドを繰り返し用いることができる.

```
#history
```

とするとこれまでのコマンドの履歴が番号つきで表示される. ここでその番号（例えば15）を使って #15 と入力すれば，対応するコマンドを再び入力したのとおなじことができる.

(6) インタプリタの終了

```
#exit
```

と入力するとＰＤＬインタプリタを終了できる. その時開いているウィンドウは自動的に閉じられる.

7. ＰＤＬインタプリタのその他の特徴

7.1 部分式の評価

関数の定義中にまったく同じ部分式が現れた場合，この部分式の値は一度評価されると保存され，その値が再び必要になったときには保存された値が参照される. すなわち，この部分式の評価はたかだか1回である.

7.2 システム状態の取扱い

システム状態は通常，引数として受け取って順次その値を更新し，その更新した値を次に伝えるように記述する. なぜなら，他の整数型などのデータ型と異なり，その値のコピー，保存ができないからである.

例えば以下の (1)では，関数execを実行したあとのシステム状態を使って関数writeを実行するように記述している（この記述例ではローカルマクロが使用できる）. これに対し，(2)の関数writeでは関数execの結果ではなく，仮引数のシステム状態を使っている. しかし，システムの状態は保存されないため，関数execを実行する前の状態で関数writeを実行したことにはならない.

ＰＤＬインタプリタはつねに最も新しいシステム状態を用いて関数を実行する. ただし，(2)の例のように，スクリプト上で指定されたシステム状態と実行時の実際のシステム状態が異なる場合（いわば古いシステム状態を指定している場合）には警告を出すことができる.

```
compile(src,S) ==
  if status(exec("cc "+src,S))=0
    then write("success",exec("cc "+src,S))
    else write("error",exec("cc "+src,S));     (1)

compile(src,S) ==
  if status(exec("cc "+src,S))=0
    then write("success",S)
    else write("error",S);                     (2)
```

ＰＤＬスクリプトはシステム状態の状態遷移であると見なせるが，システム状態の遷移をこの意味定義に従って厳密に記述するのは困難であり，書き手に負担がかかる. しかし作りやすさや実行効率のみを重視すると，スクリプトの意味と実行結果が異なってしまうという問題がある. このため，ローカルマクロなどを導入して記述をしやすくすると同時に，スクリプトの意味と結果が異なる場合には実行中にインタプリタが警告を発するようにしている.

7.3 ウィンドウ操作

ＰＤＬインタプリタでは通常Ｃシェル上でツールを起動する場合，ウィンドウを開いてその中で作業を行う. ウィンドウは組込関数 @ と組み合わせて使えばいくつも同時に開いて作業をすることができる. このようにＣシェル上のツールを起動するウィンドウを作業ウィンドウと言う. インタプリタのコマンドなどを入力するウィンドウをコンソールと呼ぶが，特に断わ

らない限り，単にウィンドウと言った場合は作業ウィンドウのことを指す．

ウィンドウを開くためには wopen，閉じるためには wclose という組込関数を用いる．wopen関数を用いてウィンドウの位置や大きさなどを指定することができるが，これらの指定方法は実際に使用するウィンドウシステムに依存する．

7.4 未定義関数の処理

スクリプトを実行中に未定義関数に実行が及んだ場合，ＰＤＬインタプリタは実行をそこで一時中断してブレークモードに入り，ユーザに次のいずれかの動作を選択させる．

(a) その関数が返す値を与え，実行を再開する
(b) その関数を定義する
(c) 実行を中断する

ブレークモードとは，実行が中断している時のインタプリタの状態をいう．ユーザはこのモードでも通常とまったく同様に式を評価したり，インタプリタにコマンドを与えたりすることができる．

(a)の場合，ユーザは

#cont 値

というコマンドを与える．(b)の場合は関数の定義をしてから

#cont

とする．(c)の場合には

#top

とすると実行は中止され，ブレークモードも終了する．

この機能により，未定義関数を含むスクリプトでも，ユーザが実行中に値を決めたり関数定義を行いながら実行を進めることができる．この機能はプロトタイプのように抽象度が高く，未定義関数を多く含むスクリプトの実行に有効であると考えられる．

またこの機能を利用し，ユーザの判断を必要とする場面では関数を未定義にしておくこともできる．ソフトウェア開発過程の記述のように，開発者の判断に依存する部分のあるスクリプトでは有効な機能である．

7.5 デバッグ

ブレークモードに入るには以下の３つの場合がある．

(a) 未定義関数を評価した場合
(b) 端末から中断文字（通常 control-C）が入力された場合
(c) 組込関数 break が実行された場合．

ブレークモードでは，引き続く実行を関数単位で行わせることができる．これをステップ実行という．

#step

このコマンドを与えるとインタプリタは関数をひとつ評価してその関数名と値を表示し，再びブレークモードに戻る．

この機能により，スクリプトのデバッグを容易に行うことができる．しかし，ユーザインタフェースがあまりよくないため，ウィンドウを活用したデバッグ専用のインタフェースを作成することを検討している．

8．インタプリタの実現方法

ＰＤＬインタプリタはシステムの機能を利用するためにさまざまな組込関数を用意している．ここではツールを起動する組込関数とウィンドウを操作する組込関数，並列演算子の実現方法について述べる．

これらの実現方法はUNIXの標準の機能のみを用いており，さらに特定のウィンドウシステムにも依存していない．

8.1 ツールとウィンドウを操作する組込関数の実現

ＰＤＬインタプリタは，ウィンドウシステムを利用して自由にウィンドウを開き，その上でＣシェルからツールを起動する．このためにはウィンドウとＣシェルの設定を行ってからＣシェルにコマンド列を与えればよい．

ＰＤＬインタプリタはウィンドウを開くためにまず子プロセスとしてウィンドウプログラム（X-Windowではxtermなど）を起動する．新しいウィンドウを開く時に，ウィンドウ内部で実行されるプログラムを指定することができるが，これでＣシェルを指定すればよい．（図１）

ところが，ＰＤＬインタプリタからこのＣシェルにコマンドを送るのは非常に困難である．プロセス間の通信にはパイプやソケットを利用するのが普通であるが，このＣシェルはＰＤＬインタプリタからは孫のプロセスになるため，パイプで結合することはできない．また，Ｃシェルにはソケットを通してコマンドを実行する機能はない．

この問題を解決するため，Ｃシェルとの通信にUNIXの擬似端末(psudo terminal)を利用した．擬似端末とは仮想的なデバイスであり，端末デバイスと同様の振舞いをするが，それへの入出力はプログラムによって制御することが可能である．

ＰＤＬインタプリタは，まず擬似端末を一組確保し（通常 /dev/ttyp? および /dev/ptyp? が組になっている），これをＣシェルの入力ファイルとしてウィ

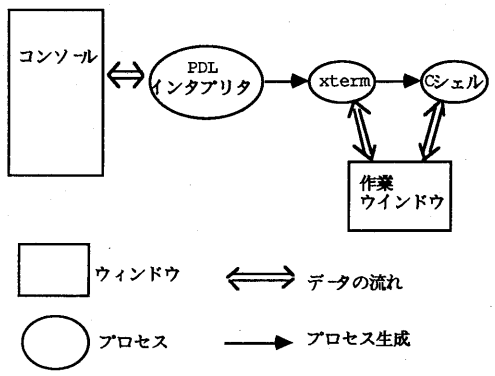

図1. インタプリタ,ウィンドウとCシェルプロセスの関係

ンドウを起動する．ＰＤＬインタプリタが擬似端末にコマンドを書き込むと，Ｃシェルはそこからコマンドを読み取り，実行する．（図２）

8.2 並列演算の実現

組込関数 @ の引数が表す作業はそれぞれ並列に実行できなければならないが，このような複数の作業をひとつのＰＤＬインタプリタで同時に制御するのは大変困難である．そこで，並列に動作する作業のそれぞれについてＰＤＬインタプリタのプロセスをひとつずつ作って作業を制御させるという方法を用いる．

いま，

```
f(S) == g(S) @ h(S);
```

と定義された関数 f を実行すると，ＰＤＬインタプリタはまず子プロセスをひとつ作り，自分（親）は関数 g の実行に移る．子プロセスは関数 h を実行し，実行が終了すると消滅する．親プロセスは関数 g を実行し終わると子プロセスの終了を確認し，新しいシステム状態を作って関数 f の値とする．

組込関数 @ を使わずに実行している場合，ＰＤＬインタプリタ，作業ウィンドウ，Ｃシェルの関係は図２で説明した通りである．関数 @ を使って複数のウィンドウで作業を行う場合，ＰＤＬインタプリタ，作業ウィンドウ，Ｃシェルは図３のように複数組生成される．

9．スクリプト例とその実行

ここでは簡単な例として，Ｃのプログラムの編集とコンパイルの過程をＰＤＬを用いて定義した．これを図４に示す．

このスクリプトでは，まずウィンドウが開いてエディタが起動される．ユーザが編集を終了させると自動的にコンパイルが開始され，エラーなくコンパイルできればオブジェクトプログラムが実行される．コンパイルあるいは実行が正しく行えなかった場合は，再びウィンドウを開いてプログラムを修正し，コンパイル，実行を繰り返すことができる．さらに，これらの作業とは別のウィンドウでオンラインマニュアルが参照できる．ユーザは正しいプログラムが作成できるまで，エラーメッセージとオンラインマニュアルを見ながらプログラムの修正とコンパイルを繰り返すことができる．

行3-5はマクロ定義である．行7-10ではこれを用い，それぞれエディタの呼び出し，コンパイル，オブジェクトプログラムの実行，オンラインマニュアルの参照を行う関数の定義を行っている．関数compileはステータスコードとシステム状態からなるタプルを返すが，コンパイルがエラーなく終了したかどうかはこのステータスコードによって判断できる．

行12で定義している関数doitはこのスクリプトを起動するための関数である．スクリプトを起動するには，作成するＣソースプログラム名とシステム状態を実引数とした関数呼び出しをＰＤＬインタプリタに与えればよい．この関数は２つの関数 CCwin と MANwin を並列に起動する．関数CCwinは編集とコンパイルを繰り返し，関数MANwinはオンラインマニュアルの参照を繰り返す．

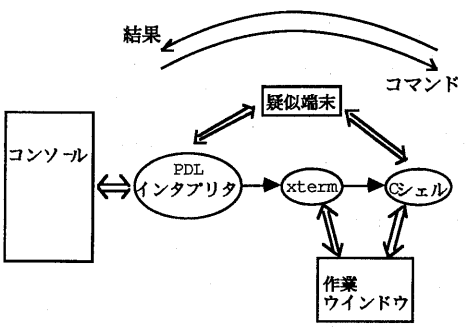

図2. 疑似端末の利用

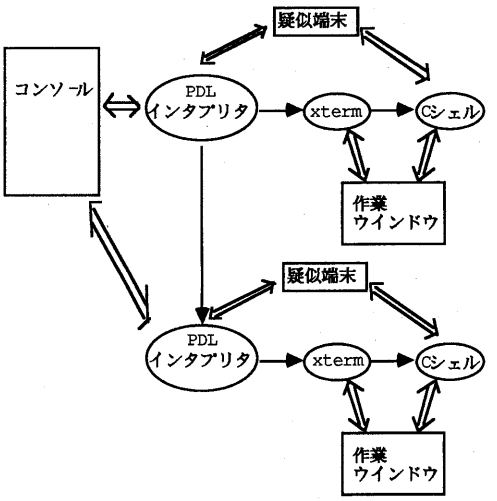

図3. 並列演算の実現

編集とコンパイルを繰り返すかどうかは，組込関数read（行17）を使ってユーザに尋ねる（関数readの第一引数の文字列はプロンプトとして表示される）．一番最初の編集作業の前にはこの質問は必要ないため，関数CCwinで編集とコンパイルをまず１回行い，次からの編集とコンパイルを関数CCloop内で繰り返すように記述してある．

関数CCwin,CCloop内で使用している関数について説明する．関数EDnew（行20）は現在の作業と並行して関数EDopenを呼び出す．関数EDopen（行21）は新たにウィンドウを開いてエディタを起動する．関数makeit（行22）はソースプログラムをコンパイル

し，コンパイルがエラーなく終了するとオブジェクトプログラムを実行する．

関数MANwinはウィンドウを開いて関数MANloopを呼び出す．関数MANloopでは端末から文字列を読み込み，その文字列を引数としてオンラインマニュアルを参照する．文字列が "q" のとき繰り返しは終了する．

このスクリプトを実行した時の画面の例を図5に掲げる．左上のウィンドウがＰＤＬのコンソールである．ここから関数doitを起動して実行を開始する．残り3つのウィンドウは左からそれぞれ，エディタ用，コンパイル用，オンラインマニュアル用のウィンドウである．（この画面は Sun3 上で実行した例である）

１０．おわりに

本報告ではプロセスプログラミングの考え方に基づくソフトウェア開発環境記述言語ＰＤＬを提案し，その言語仕様および処理系の概要について述べた．

ＰＤＬインタプリタは，約6人月の作業量をかけて作成された．使用言語はＣで，ソースコードはおよそ7500行である．Ｃシェルへのコマンド列を制御するという目的には十分な実行速度を持つ（スクリプト実行中に最も時間がかかるのは主にウィンドウ操作やツールの起動である）．

8章で述べたように，ＰＤＬインタプリタは UNIX 上のウィンドウシステム内で実行できる．ウィンドウの大きさや位置の指定方法など，ウィンドウシステムに依存する部分を除き，大変移植性がよい．ＰＤＬインタプリタは現在，IBM RT/PC(ACIS) および Sony NEWS の X-Window 上，Sun3 の Suntools 上で稼働している．これらのシステム上へのインストールは，ツールの Path の書換え程度の変更で容易に行うことができる．

ＰＤＬを用いて記述した例としては，ソフトウェア開発技法であるＪＳＰおよびＪＳＤがある．これらのスクリプトは実際にインタプリタ上で実行させることができ，この開発技法に従ってソフトウェアを開発することが可能である[3]．

現状では開発過程の記述，すなわちＰＤＬスクリプトの作成は人間がすべて手で行わなければならない．そこで，抽象度の高い記述から実用的なレベルの記述を導出するシステムの作成を行っている．この方法ではまずソフトウェア開発過程をいくつかの段階に分解し，それらの間の関係をプロセスとプロダクトという観点から記述する．次に各段階内部について同様な方法を適用して段階的な詳細化を進め，最終的にＰＤＬスクリプトを導出するのである．また，人間が実際に行った作業過程の記録を収集し，その情報からソフトウェア開発過程をＰＤＬスクリプトとして一般化するという方法も検討している．

現在，ＰＤＬインタプリタのユーザはＰＤＬスクリプトをある程度知っているということを前提に開発を進めているが，ＰＤＬスクリプトを知らないユーザでも開発が容易に行えるような方法についても検討が必要である．

今後はデバッグ用のユーザインタフェースの開発など，ＰＤＬインタプリタの使い勝手の向上を図るとともに，これらの課題についての検討も進める予定である．

```
1  // C program development
2
3  #let EDIT   : "vi "
4  #let CC     : "cc "
5  #let MANUAL : "man "
6
7  edit(src,S) == exec(EDIT+src,S);
8  compile(src,S) == [status(exec(CC+src,S):S1),S1];
9  run(S) == exec("a.out",S);
10 manual(obj,S) == exec(MANUAL+obj,S);
11
12 doit(src,S) == CCwin(src,S) @ MANwin(S);
13
14 CCwin(src,S) ==
15     wclose(CCloop(src,makeit(src,wopen(EDopen(src,S)))));
16 CCloop(src,S) ==
17     if read("continue(y/n)?",S)="y" then
18         CCloop(src,makeit(src,EDnew(src,S)))
19     else S;
20 EDnew(src,S) == S @ EDopen(src,S);
21 EDopen(src,S) == wclose(edit(src,wopen(S)));
22 makeit(src,S) ==
23     if element1(compile(src,S):T)<>0 then element2(T):S1
24     else run(write("**OK! RUNNING**",S1));
25
26 MANwin(S) == wclose(MANloop(wopen(S)));
27 MANloop(S) ==
28     if read("online manual>",S):X="q" then S
29     else MANloop(manual(X,S));
```

図4. Cプログラム開発用PDLスクリプト

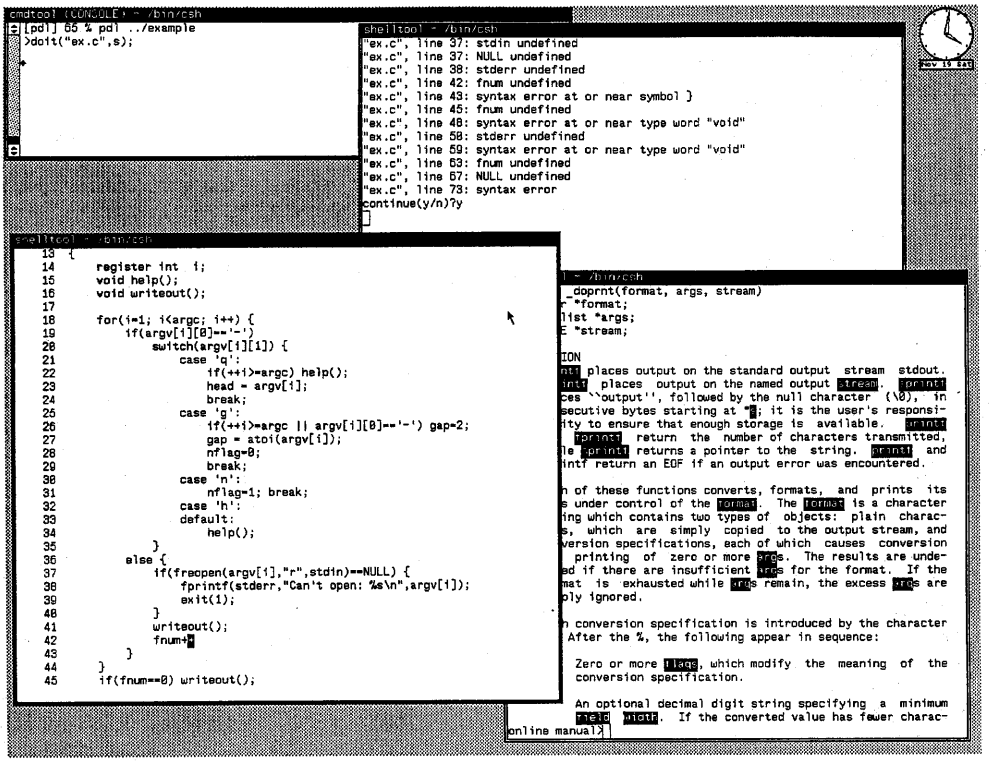

図5. PDLスクリプトを実行した例

参考文献

(1) L.Osterweil: "Software processes are Software too", Proc.of 9th ICSE,pp.2-13(1987).

(2) G.E.Kaiser and P.H.Feiler : "An Architecture for intelligent assistance in software development", Proc. of 9th ICSE,pp.180-188 (1988).

(3) 稲田, 荻原, 井上, 菊野, 鳥居 : “ジャクソン開発法の形式的記述の詳細化とその実行”, 信学技報, (発表予定) (昭 63).

(4) 東野, 関, 谷口 : “代数的仕様から関数型プログラムの導出とその実行”, 情報処理,Vol.29, No.8, pp.881-896 (昭 63).

(5) 関, 井上, 谷口, 嵩 : “関数型言語ＡＳＬ／Ｆのコンパイル時における最適化”, 信学論(D), Vol.J67-D, No.10, pp.1115-1122 (昭 59).

(6) 嵩, 谷口, 杉山:“代数的言語の設計と処理系”, 榎本編, ソフトウェア工学ハンドブック, オーム社, pp.1066-1073 (昭 61).

(7) 井上, 関, 谷口, 嵩 : “関数型言語ＡＳＬ／Ｆとその最適化コンパイラ”, 信学論(D), Vol.J67-D, No.4, pp.458-465 (昭 59).