Vol. 30 No. 4　　情　報　処　理　　Apr. 1989

解　説

5. 事　例

5.2 多重言語を指向する統合プログラミングシステムの開発経験†

佐　藤　　豊††

1. はじめに

1970年代末から80年代の前半にかけては，構造エディタ（言語指向エディタ）を中心とする統合プログラミング環境の研究が盛んに行われた時期である．カーネギーメロン大学のGandalf[1]，INRIAのMentor[2]，コーネル大学のCPS[3]，ダルムシュタッド工科大学のPSG[4]などをはじめ，多数のシステムがこの時期に開発されている[5),8),11)~16)]．

これらのシステムでは，利用者と環境との対話インタフェースを，構造エディタに統合している．統合環境における構造エディタの役割は，プログラムの編集中に誤りを即座に利用者に知らせることだけではない．利用者と環境内に統合された各種の道具との間を，言語レベル（ソースプログラムレベル）で結合するという，重要な役割を果たす．たとえば，実行系と結合して，言語の構文や意味を反映したシンボリックなデバッガの機能を提供したり[1),3)]，コンパイル，リンク，リビジョン管理といった作業を，プログラムの変更時に自動的に行う[1),8)]などである．構造エディタが維持しているプログラムの構文構造を利用して，変更にともなうさまざまな処理を，変更に影響を受ける最小の構文単位で“インクリメンタル”に行い，応答性を向上させることも試みられている．

このような統合環境では，構造エディタを中心に各種の機能を特定言語に専用化して結合する必要がある．専用化と結合とをともに容易にする手法として，生成系を用いた実現法が一般にとられている[1)~4)]．これはまず，言語独立な枠組みと，言語固有の部分とを分離する．次に，与えられた言語固有の構文と意味の定義から，生成系によっておのおのの構成要素を生成して要素間を結合するというものである．

生成系を用いて，複数の言語処理系を同一の枠組みで作ることは，複数の言語を混合して実行する環境の基礎を提供することになる．このような環境は，複数の言語を目的別に使い分けて一つのプログラムを構成するような開発を支援する．その応用領域は，プログラム言語に限定されない．多様な文書やコマンド言語への応用[2)]，利用者や実行環境向きのカスタマイズ[16)]などを，同様な生成系を用いて支援する試みがなされている．さらに，生成系の機能を一般のプログラマや末端の利用者に道具として解放することで，新たなプログラムの開発法や利用法がひらける可能性がある．以下ではそのような環境を，多重言語指向環境と呼ぶ．

本稿では，構造エディタを中核とする多重言語指向環境の機能，実現技術について，COSMOSプログラミングシステム[18)~25)]を例として解説する．COSMOSは上で述べたような認識のもとに，筆者が1984年から開発を行ってきたシステムである．以下，2.でその概要を述べ，3.ではプログラム開発のための支援機能と実現技術を，他のシステムとの比較も含めて概説する．4.では，複数言語の混合を前提とする言語の開発・実行環境と，その応用例について述べる．

2. COSMOSシステムの概要

COSMOSは，特定の言語向きのプログラム開発支援環境を複数の言語に対して統一的な枠組で提供すると同時に，言語そのものの開発支援環境も提供する．

2.1 プログラム開発支援機能

COSMOSは，手続き型の言語による中規模以上の実用プログラムの開発に使用できることを目標としている．支援するプログラム開発の形態としては，熟練したプログラマが頻繁にプログラムの修正を繰り返すような形の開発を考えている．このため，機能面では不要な対話を抑えた熟練者向きの利用者インタフェー

† An Experience of Development of A Multiple Language Oriented Integrated Programming System by Yutaka SATO (Electrotechnical Laboratory).

†† 電子技術総合研究所

スの実現を，性能面では，開発するプログラムの規模に依存せずに高速に変更・実行サイクルを行えることを目指した[23]．この方針に基づいて，以下のようなCOSMOSの特徴的な機能が実現されている．

（1） テキスト編集ベースの構造エディタ

エディタ中での対話的な構文エラーの修正を支援する構造エディタは，従来のテキストエディタの編集機能を包含している．これにより，プログラム言語に熟練した利用者は，構文を意識することを強制されることなく，自由な編集ができる．

（2） 構造エディタに結合されたインタプリタ

構造エディタで変更したプログラムは，インタプリタで即座に実行できる．プログラムの実行をシンボリックに，構文の単位や行単位でデバッグする機能や，構造エディタの画面上でブレークポイントの設定や実行のトレースを行う機能を提供している．

（3） コマンド言語としてのプログラム言語

テストや実行のために，関数にさまざまな引数を与えて実行したり，変数の値を設定・読み出しするコマンドは，プログラム言語の文そのものである[19]．デバッグのためのコマンド言語を新たに覚える必要はない．

（4） コンパイルコードの併用による実行

実行の高速化のため，現在デバッグしているモジュールだけをインタプリタで，それ以外は，コンパイルされたコードで実行する．エディタで変更したモジュールは，自動的にCOSMOS外部のコンパイラに渡される．コンパイルされたコードは自動的にCOSMOS内にロードされ，そのモジュールに関係するリンクだけが，インクリメンタルに更新される．コンパイラは既存のものを使用し，インクリメンタルリンカは，静的なリンクを想定して出力された標準のオブジェクト形式を処理する．

複数人による大規模プログラムの開発に固有な支援機能は，現在のCOSMOSには含まれていない．しかし，そのような開発環境における，対話的なプログラミングエンジン[17]を提供することができる．

2.2 多重言語指向の言語開発環境

規模の大きなプログラムでは，プログラム全体を単一の言語で書くのが困難なことが少なくない．プログラムの部品ごとに，その記述に最適な言語を使用することは，開発効率を高める上で効果があると考えられる．このような多重言語プログラムの開発を支援する環境として，COSMOSは次のような特徴をもっている．

（1） 言語独立な枠組み

構造エディタとインタプリタは，特定の言語に依存しない枠組みの上に構成されている．言語固有の部分は，与えられた言語の構文と意味の定義（言語定義）から生成され，枠組みの中に組み込まれる．

（2） 利用者による言語開発の支援

言語定義の開発を支援する機能は，プログラムの開発支援機能と同一の環境に統合されている．これにより，与えられたプログラム環境を利用するのと並行して，みずからの言語環境を拡張したり，新たな言語を作成することができる．

（3） 言語処理系の動的な生成

言語定義の開発は，プログラムの開発と同様な環境で，対話的に行うことができる．生成系は，実行時ルーチンとしてプログラム中からも起動することができる．生成された言語依存部品（パーサやインタプリタ）は，動的にロードされて即座に利用が可能になる．

（4） 複数の言語の混合実行

生成された複数の言語の部品は，COSMOSの単一プロセス空間内に多数混在することができる．これにより，複数の言語のプログラムを並行して編集できるとともに，異なる言語のプログラム間で呼び合うことが可能である．

このようなCOSMOSの多重言語指向環境（図-1）により，一般利用者（プログラマ）が容易に言語を定義してその処理系を得ることができる．利用者が作成する言語としては，プログラム全体を記述する汎用のプログラム言語ではなく，プログラムの一部分で扱う特定の応用向きの小規模な言語を想定している．たとえば，プログラムの外部から読み込まれるファイルの

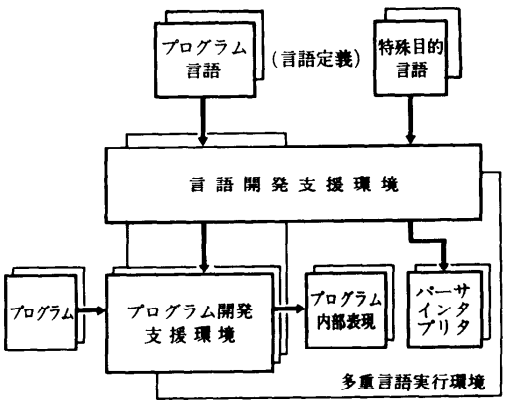


図-1 COSMOSの多重言語指向環境

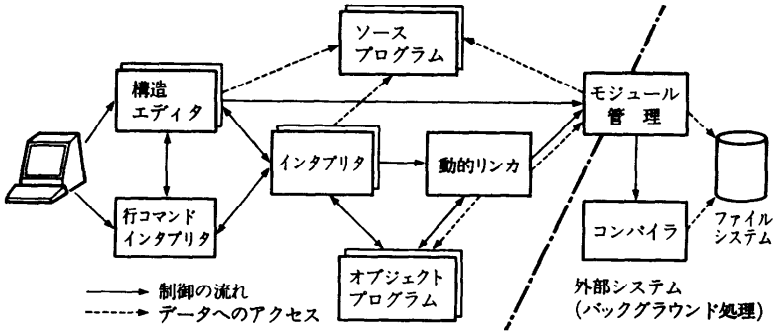


図-2 COSMOS のシステム構成

形式やコマンド形式を表す言語である．従来，これらの小規模な言語を解析する処理系は，手作りで実現されていることが少なくない．これを言語として形式的に定義することによって仕様を明確かつ柔軟にし，生成系による処理系の実現によって開発の効率化をはかることができる[24]．このような開発形態においては，一つのプログラムの部品として多数の言語を作成することになる．

2.3 システムの構成

COSMOS は unix 上に実現されており，図-2 に示すように単一のプロセス空間に各種の機能を統合している．プログラムはCで記述してあるが，動的リンカの実現のために機械依存な部分があり[18]，sun3(MC 68020)，sun4(SPARC)，vax8800，NEWS(MC 68020) 用のアセンブラコードを十数行ずつ含んでいる．現在のプログラム規模は本体が約4万行であり，この上に作成された，ニュースリーダやディレクトリビューアなどの応用向きエディタ群が1万4千行となっている．対象言語としては，Cの他に，Pascal や Lisp や Prolog などの言語の小さなサブセットが実験的に実現されている．

3. プログラム開発支援機能とその実現

プログラム開発を支援する道具としての COSMOS を他のシステムと比較すると，最大の特徴は構造エディタの編集機能にある．その特徴と実現技術を，他システムと比較しながら概観する．

3.1 利用者インタフェース

3.1.1 プログラムの編集と解析

(1) 入力方式

構造エディタは，その入力方式によって，“合成型”と“解析型”に大別できる[13]．文字列として入力されたプログラムを解析してエラーを知らせるものを，解析型と呼ぶ．Mentor[27] や Magpie[8] や COSMOS がこの型である．一方，合成型では，与えられた構文の“テンプレート”を選択しながら，トップダウンにプログラムを合成してゆく．合成型でも，式などの一部の構文に対してだけは，文字列としての入力を許すのが普通である．Gandalf[1] や CPS[3] をはじめ，多くのエディタがこの型である．解析型をコマンド言語型の利用者インタフェースにたとえると，合成型はメニュー選択型に相当する．合成型のものは初心者に，解析型は熟練者にそれぞれ適しているといわれている．熟練者を主な対象とする COSMOS では解析型をとっている．

(2) 編集機能

解析型，合成型によらず，一般の構造エディタでは，いったん構文木として作成されたプログラムに対しては，構文木単位の切り貼りという形での，構造的な編集だけしか提供していない[1)~3),13]．プログラムを文字列として自由に編集できるようにしているのは，Magpie[8] や COSMOS などの少数だけである．実際のプログラム開発では，初期入力よりも頻繁に繰り返される修正のしやすさが重要であり，熟練者は構文を意識しないことで効率的な編集ができる[19]．構造的な編集は，LISP などのように単純な構造をもつ言語以外では，実際に有用な場面が少ない．

このような観点から，COSMOS では従来のテキストエディタの編集機能を包含した，解析型の構造エディタを実現した．プログラムの編集にともなう構文の再解析は，従来の解析型のエディタでは，自動的に行われるのが一般的であり，一文字入力するごとにエラーを知らせるものもある[8]．しかしこれは 9) で指摘されているように，利用者にとっては逆に煩わしく，

① もとのプログラム
```
c=getchar();
if (c==' ')
      return (getchar());
else  return(c);
```

② do を挿入し，if を while に置換えて解析
```
do c=getchar();
while (c==' ')
      [!: return] (getchar());
else return(c);
```
→return で構文エラーになる→現在行

③ 現在行を削除して解析
```
do c=getchar();
while (c==' ');
[!: else] return(c);
```
→else で構文エラーになる→現在行

④ else を消去して解析
```
do c=getchar();
while (c==' ');
return(c);
```
→構文エラーがなくなり，変更完了

□は反転表示

図-3 変更と対話的なエラー修正

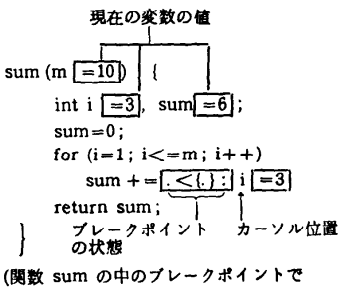
構文に縛られない自由な編集を阻害する．このためCOSMOSでは，利用者が構文を検査したいと思う時点で，構文解析を明示的に起動するようにしている．

実現したエディタのコマンドは，画面指向テキストエディタviのサブセットに，構文木単位の編集機能と，デバッグ用のコマンドを加えたものである．図-3に，このエディタを用いたプログラムの変更とエラー検出・修正のようすを示す．

3.1.2 画面上のソースレベルデバッグ

多くのプログラミングシステムで，構造エディタをソースレベルのデバッグのための利用者インタフェースとして利用することが計画されている．実際にCPSでは，構文構造を反映した高機能なソースレベルデバッガを実現している[3]．

COSMOSにおいても，構造エディタで解析したプログラムは，インタプリタによって即座に実行できる．構造エディタの画面に表示されたテキストの上で，実行を制御したり，プログラムの実行の軌跡や変数の状態を表示する形でデバッグが行える．図-4に示すように，プログラムに設定されているブレークポイントの状態，変数の現在の値などは，表示されたソースプログラム中に埋め込まれて表示される．編集機能と同様に，構文の単位だけでなくテキストの行単位でも，実行を制御できることが特徴となっている．



(関数 sum の中のブレークポイントで実行が停止したときの画面)

□は反転表示

図-4 実行中のプログラムの表示

3.2 実現の技法

(1) プログラムの内部表現形式

利用者が構文木を直接作成する合成型では，プログラムを構文木としてのみ保存する[1),3)]．解析型でも，解析の過程を表す解析木は残さず，抽象構文木として保存するのが一般的である[2)]．この表現形式は記憶効率が良く，構造的な編集において利用者に不必要に複雑な構文を意識させずに済む．一方，表示用のテキストを表示の度に構文木から作成する必要があり，テキスト型の編集機能の実現も難しい．これらの問題点を避けるためにCOSMOSでは，解析木とともに，テキストと等価な字句解析後のトークン列を保存している[22)]．特別なパーサのアルゴリズムにより，解析木の表現を抽象構文木に近づけている[21)]．

(2) インクリメンタルパーサ

テキスト型の入力と編集機能を提供するために，構文を部分的に再解析することのできる，“インクリメンタルパーサ”が必要になる．これは，エディタでプログラムの任意の位置に加えられた変更に対して，その変更に影響を受ける最小限の構文の範囲だけを解析し直すパーサである．これによって，プログラムの全体の大きさに依存しない応答性を確保できる．LR型の実用可能なインクリメンタルパーサが提案されたのは，比較的最近のことである[6),7)]．Magpieでは解析木だけを用いた，LL(1)型のインクリメンタルパーサを実現している[8)]．COSMOSでは，4.2で述べるような利用者による言語定義の容易さを考慮して，再帰下降型とLL(1)を組合せたインクリメンタルパーサを，上述の内部表現を基礎に実現した[20)]．

(3) インタプリタ

対話的な実行のための実行系として，コンパイラを改良して用いるか，インタプリタを作成するかという

選択がある．Gandalf や Magpie などでは，“インクリメンタルコンパイル”と呼ばれる，関数や文単位の分割再コンパイル方式によって応答性を高めている[1),8)]．COSMOS では，上述のプログラムの内部表現（解析木）を直接，インタプリタで実行する．このインタプリタを生成系で作成することにより，シンボリックな構文単位のデバッグ機能を自然に実現している．コンパイラをベースにする場合に必要となるような外付けの特別なマッピングテーブルやメカニズムは不要である．

4. 多重言語指向プログラムの開発支援環境

プログラムの開発の支援環境と同様な対話的環境を，言語の開発に対しても提供することによって，一般のプログラマが，プログラムの部品として言語を作成するようなプログラミングを支援することができる．本章では，このような支援のために COSMOS が提供している環境について述べる．

4.1 言語の定義

言語を定義する記法は，記述の形式性よりも，簡便な記述と高速な生成が容易であることを重視して設計した[22)]．言語の定義では，言語の構文規則を骨組みとして，意味解析や実行のためのアクション，および構文木から表示用テキストを生成するための規則を与える．

4.1.1 構文規則と実行の記述

構文規則で記述できる文法のクラスは，LL(k) である．おのおのの構文規則に対して，その実行のためのアクションを，Cの実行文で記述する．以下は，Cの実行文の定義の一部である．

```
(1)  statement :
(2)      while ( $expression ) $statement
(3)          {while($1){$2; if (BREAK) break;}
                 BREAK=0;}.
(4)      break ;
(5)          {BREAK=1;}.
     ...
(6)  statement-list :
(7)      $statement @$statement-list
(8)          {$1; if (!BREAK) $2;}.
     ...
```

(1)〜(5)は，statement を左辺の非終端記号とする生成規則，while 文と break 文の構文と，それを実行するためのアクションを記述したものである．(1)が生成規則の左辺，(2)と(4)が右辺である．右辺の中で，先頭が '$' で始まるものが非終端記号，それ以外は終端記号である．(6),(7)は statement の並びを定義している．(7)の中で使われているような '@' が先頭に付けられている記号は，省略が可能であることを表す．“{ }.”で囲まれた(3),(5),(8)が，それぞれ while 文，break 文，文並びの実行のためのアクションである．この中で，$1, $2, ..., $N という記法は，対応する生成規則の中での N 番目の非終端記号を実行することを表す．

4.1.2 意味解析の記述

意味解析のアクションも同様にCの実行文で記述する．以下は，Cの複合文の記述の例である．

```
statement :
    \{ @$declaration-list @$statement-list }
      {? ASIZE=0; $1; #ASIZE=ASIZE;}.
      {ACTIVATION_RECORD(#ASIZE);
        $2;}.
```

“{? }.”で囲まれた部分が意味解析のアクションであり，最初の実行時に一度だけ実行される．ここでは，宣言部を実行して，局所変数を保持する励起レコードに必要な領域のサイズを調べている．意味解析のアクションの中で，名前の先頭に '#' の付いた変数は，この規則に対応する構文木のノードの局所的な属性であり，その他は広域的な変数である．

4.1.3 表示規則

表示規則は，構文木からテキストを作成するための規則である．通常の画面表示のためには不要であるが，プログラムを構造的に圧縮して表示したり，プリティプリントする場合のテキストの生成方法を記述する．以下は while 文の例である．

```
while ( $expression ) $statement
        {% 1 2 s 3 s 4 {>5<} }.
```

“{% }.”で囲まれた部分が表示規則である．この中で，1 2 ... N の数字は，N 番目の記号を表す．s は空白の挿入，'>' と '<' は字下げの制御，'{'と'}' は表示する構文木の深さを制御する．

4.2 生成系と支援環境

4.2.1 生 成 系

生成系は，与えられた言語定義から，構造エディタのための構文解析表，字句構文表，予約語表を作成し，インタプリタをCのソースプログラムとして生成する．スキャナとインクリメンタルパーサ，アンパーサ，インタプリタのデバッグ機能，構文木の切り貼

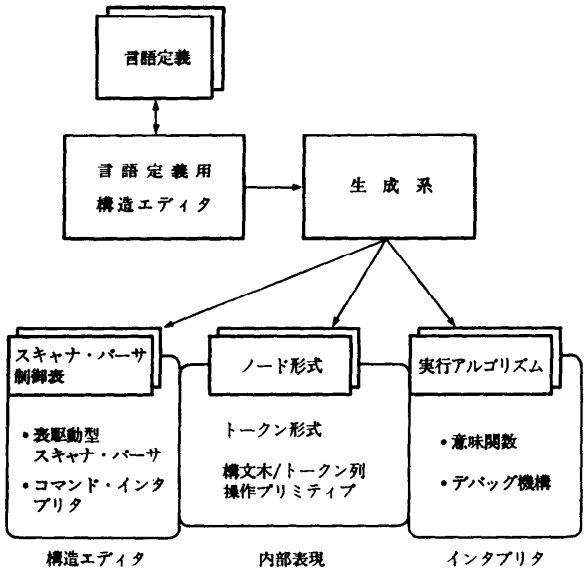


図-5　言語環境の枠組みと生成系

り，などのアルゴリズムは共通であり，実際に共有している(図-5)．

Cプログラムとして生成されたインタプリタは，外部のコンパイラでコンパイルされた後，ロードされて実行可能になる．スキャナやパーサの各種制御表は，主記憶上に作成される．これらの表は，その言語を実行する直前に毎回言語定義から作成してもよいが，規模の大きな言語に対しては，高速化のために，生成された表をオブジェクトファイル形式で出力し，リンクして使用する．これらの各種の制御表とインタプリタの関数には，言語ごとに異なる外部名が与えられる．

4.2.2　言語の開発支援環境

言語の開発支援の多くは，プログラムの開発のための機能を利用して実現されている．

(1)　言語定義言語用構造エディタ

言語定義の作成，構文上のエラーチェックは，他のプログラム言語用のエディタと同様な枠組みで作られた構造エディタで支援している．

(2)　言語定義の変更に対する高速な再生成

言語定義からのパーサやインタプリタの生成が直接的であるため，生成は高速である．これによって，言語定義の変更とテストの繰り返しが効率的に行える．

(3)　動作の追跡機能

定義の内容に関する誤りの発見は，スキャナ，パーサ，インタプリタのそれぞれの動作のトレース機能で支援している．任意の非終端記号からの解析，実行により，言語の一部分ごとにテストできる．

COSMOS で用いているような下降型のパーサは，いわば文法を直接インタプリットするので，動作が直感的に理解しやすい．これは，言語処理系の専門家でないプログラマによる言語の開発を支援するうえで有利である．

おおまかなインタプリタのデバッグは，そのインタプリタの利用者に与えられるデバッグ機能を使用して，プログラムの実行を追跡することで行う．詳細なデバッグのためには，各種の属性の値の動きをトレースする機能を用意している．

4.3　言語の混合のための言語定義と実行環境

4.3.1　言語の外部定義

言語定義では，複数の言語の中から一つを選択するために言語に名前を与え，一つの言語を目的別に使い分けるための複数の入口を定義する．たとえばCに対しては，次のように定義してある．

```
LANGUAGE : c              /*言語の名前*/
PROGRAM : $external-definition
                          /*モジュールの開始記号*/
COMMAND : $statement-list
                          /*コマンドの開始記号*/
```

PROGRAM は，モジュール（ファイル）単位の開始記号を定義するもので，Cでは関数や広域変数などの外部定義の部分である．COMMAND は，この言語をコマンド言語として使用するときの開始記号で，Cでは実行文リストとしている．一般に手続き型の言語では，宣言文と実行文の構文のレベルが異なるので，このように用途別の入口を定義してやる必要がある．

4.3.2　異言語間の相互呼び出し

異なる言語のプログラムを混合して実行する方法は，ソースプログラムレベルと，オブジェクトプログラムレベルのものがある．

(1)　ソースプログラムレベルでの混合

ソースプログラムの文字列を，次のような組込み関数に渡して実行することによって，他の言語の文を実行することができる．

lang_eval(言語名，開始記号，プログラム片)

この関数は，利用者のプログラム中から直接使用してもよいが，言語定義の実行アクションの中に埋め込め

ば，言語の混合を利用者から隠すことができる．たとえば，ある言語の中で，Cの式を利用するには，下のようにすればよい．

```
LANGUAGE : sample
...
EXP : { $STRING }
    { @v=lang_eval("c", "expression",
      #%2); }.
```

ここで，'#%2' は，この生成規則の右辺の2番目の記号 ($STRING) に対応する文字列，'@v' はこの規則の実行結果として返される値を表す．この方法では実行の度に構文が解析されるので，繰り返して実行されるような構文に対しては効率が悪い．

(2) オブジェクトプログラムレベルでの混合

インタプリタにおけるデータ表現と関数呼び出しのインタフェースを，コンパイルされたコードでのものに合わせることによって，インタプリタとコンパイルコードの両者を混合した実行が可能である．おのおのの言語の定義の中では，関数あるいはそれに対応する構文単位に対して，一様な実行時インタフェースをもつ入口を定義する．これにより，異なる言語の構文木インタプリタ間，およびオブジェクトコード間で，相手の言語を意識しない呼び出しを行うことができる．このレベルでの共通のインタフェースとして，現在は，Cのコンパイルコードでのものを標準にしている．

言語の混合の最終的な方式は，他の言語の非終端記号を構文規則の中で参照できるようにすることであり，COSMOS をそのように拡張することも検討している．

4.3.3 言語環境の切り換え

ある言語を実行するための "環境" は，使用するスキャナやパーサの制御表とインタプリタの集まりで表される．一つの言語で使用する，各種の制御表やインタプリタへのポインタ群をまとめた構造体を，その言語の言語環境と呼んでいる．

言語環境の切り換えは，編集時に構造エディタのバッファを切り換えたとき，および実行時に関数呼びだしが起こったときに行われる．動的リンクのために各関数に対するディスクリプタを維持しているが，このディスクリプタ中にはその関数の記述言語も記録されている．ある関数が呼びだされると，対応する言語環境をセットアップし，終了時に前の言語環境を回復する．

4.4 電子メイル/ニュースリーダへの応用例

COSMOS の多重言語環境の応用例として，vin と呼ぶ電子メイル/ニュースリーダを作成した[25]．

4.4.1 多重言語環境を活用した実現

vin の実現においては，COSMOS の多重言語環境を利用したいくつかの試みを行っている．vin は，COSMOS の構造エディタのコマンドセットをニュースリーダ用にカスタマイズして実現したものである．このエディタの生成のための言語定義では，ニュースやメイルの記事のヘッダの構文や，記事内容中の字句レベルの構文（他の記事への参照の記号など）を記述している．こうした構文情報を利用して，記事の圧縮表示や検索の機能を実現している．

多様な入力の形式や，利用者によるカスタマイズに対応するために，いくつかの小規模な言語を定義して部品として使用している．たとえば，記事中の多様な日付の記法を整数値へ変換するための言語や，画面に表示する記事のリストを利用者の定義した形式で作成するフォーマッタの生成言語などがある．

4.4.2 実行時ルーチンとしての生成系の応用

vin のベースにある COSMOS のインタプリタ生成機能を利用者レベルに開放することによって，多様な情報や動作をメイルで転送する実験を行っている．これは記事の中に，ある言語の定義とその言語で書かれた内容を合わせて転送するものである．記事の中に次のような形式，

[言語L]{ … }[言語L]

が含まれていると，vin の実行コマンドにより，…の部分を言語Lの文として実行することができる．この形式が続けて書かれていると，上から順に続けて実行される．たとえば，次のような内容の記事，

```
[language]{
  LANGUAGE : animation
  …(アニメーション記述言語)
}[language]
[animation]{
  アニメーションの記述
}[animation]
```

では，最初に language という名前の，言語を定義するための組込み言語で，animation という名前の言語を記述している．これを実行すると，animation 言語のパーサとインタプリタが生成され，これを用いて，続いて書かれているアニメーションの文が実行される．この機能を用いて，簡単な描画言語によるアニメーションの転送を始め，文書整形言語により内容を整形して画面に表示する記事，対話的にアンケートを行い結果をメイルで回収する記事などの実験を行っ

た．このニュースリーダは現在，電子技術総合研究所内外の数十人の利用者に利用されている．

5. おわりに

COSMOS ではその開発の初期から，COSMOS 自身を開発環境として利用してきた．最初に実現した動的リンカと，Cの実行文のインタプリタは，COSMOS の構成モジュールに対する小刻みな変更とテストの繰り返しを効率的に支援している．これは，中から大規模のプログラムとなりつつある COSMOS の開発には，不可欠な機能となっている．実際に，現在の COSMOS の全モジュールを通常のリンカでリンクするには，sun 4/280 の上で1分近くを要するが，変更したモジュールだけの動的リンカによる再リンクは，最大のモジュールで1秒程度である．言語定義の記法と生成系は，実際にそれを用いて複数の言語を記述し，実用した結果をフィードバックしながら，拡張を続けている．構造エディタを応用したニュースリーダなどの特殊用途向きエディタは，その実現と使用の経験が COSMOS の基本的なメカニズムの拡張や洗練に役だっている．このような統合環境では，一つの機能をさまざまなサブシステムから利用できるため，利用価値の高い部品の開発を促進する効果があると感じている．

従来の多くの研究によって，プログラムの作成支援環境のための各種の要素技術は，個々の技術としては完成しつつあるように思われる．しかし，プログラミングのように非定型な要素の多い作業を実用レベルで支援するには，要素技術を有効に結合し，統合されたシステムとしての完成度を高めてゆく努力を今後も継続することが必要である．また，ここで得られた技術は，プログラムの開発以外の用途，特に，定型的な文書の作成のような応用に対しても有効に利用できると考えられる．

構造エディタを中心とする統合環境は，本稿で述べたような多重言語指向の環境に発展させることで，新たなプログラムの開発法，構成法を提供できる可能性がある．筆者は，これまでに COSMOS の上で行ったいくつかの実験を通じて，そのような方法が，仕様の多様性や頻繁な変更への対応を容易にする効果をもつことを認識した．利用者による言語の開発を支援するためには，言語を定義するための記述性の高いメタ言語の導入，言語定義の対話的なテストやデバッグの支援，言語の部品化と再利用を支援する環境を充実してゆく必要がある．

謝辞 本研究の機会を与えていただいている，電子技術総合研究所情報アーキテクチャ部棟上昭男部長，草稿に対して貴重なコメントをいただいた，同部情報ベース研究室真野芳久室長，小島功氏，そして，COSMOS システムの利用を通じて開発に協力してくださっている利用者の方々に感謝します．

参考文献

1) Medina-mora, R., and Feiler, P. H.: An Incremental Programming Environment, IEEE Trans. Softw. Eng., Vol. SE-7, No. 5, pp. 472-482 (1981).
2) Donzeau-Gouge, V. et al.: Document structure and modularity in Mentor, Proc. of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, Vol. 19, No. 5, pp. 141-148 (1984).
3) Teitelbaum, T. and Reps, T.: The Cornel Program Synthesizer: A Syntax Directed Programming Environment, Comm. ACM, Vol. 24, No. 9, pp. 563-573 (1981).
4) Bahlke, R. and Snelting, G.: The PSG System: From Formal Language Definitions To Interactive Programming Environments, ACM TOPLAS, Vol. 8, No. 4, pp. 547-576 (1986).
5) Teitelman, W.: A tour through Ceder, IEEE software, pp. 181-195 (1984).
6) Getzzi, C. and Mandrioli, D.: Incremental Parsing, ACM TOPLAS No. 1, pp. 58-70 (1979).
7) Agrawal, R. and Detro, K. D.: An Efficient Incremental LR Parser for Grammars with Epsilon Productions, Acta Inf., 19, pp. 369-376 (1983).
8) Schwartz, M. D., Delisle, N. M. and Begwani, V. S.: Incremental Compilation in Magpie, Proc. ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 19, No. 6, pp. 122-131 (1984).
9) Ross, G.: Integral C-A Practical Environment for C Programming, Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices, Vol. 22, No. 1, pp. 42-48 (1987).
10) Ballance, R. A., Butcher, J. and Graham, S. L.: Grammatical Abstraction and Incremental Syntax Analysis in a Language-Based Editor, Proc. SIGPLAN '88 Conf. Programming Language Design and Implementation, pp. 185-198(1988).
11) 海尻：構文指導型プログラム開発システムにつ

いて，ソフトウェア工学研究会 27-4 (1982).
12) 中所：プログラミング言語とその会話型支援環境，情報処理 Vol. 24, No. 6, pp. 715-721 (1983).
13) 原田：構造エディタ，情報処理 Vol. 25, No. 8, pp. 767-775 (1984).
14) 浜田，宮内：文法指向エディタの自動生成，電子通信学会論文誌，Vol. J69-D, No. 1, pp. 61-71 (1986).
15) 原田(編)：構造エディタ，共立出版 (1987).
16) 田中，岸本，仲原，中所：言語適応型プログラミング環境用構造エディタ PARSE，(原田編) 構造エディタ，pp. 48-57 (1987).
17) 斎藤：ソフトウェア開発環境，情報処理，Vol. 28, No. 7, pp. 887-897 (1987).
18) 佐藤，板野：動的複合実行方式-直接実行系と翻訳実行系を統合した対話型実行方式，コンピュータソフトウェア，Vol. 2, No. 4, pp. 19-29 (1985).
19) 佐藤，板野：C言語指向構造エディタ SSE，(原田編) 構造エディタ，共立出版，pp. 59-70 (1987).
20) 佐藤，板野：構造エディタのためのインクリメンタル LL パーサの一構成法，情報処理学会論文誌，Vol. 28, No. 6, pp. 668-671 (1987).
21) 佐藤，板野：構造エディタにおける下降型パーサのための構文木の圧縮化技法，情報処理学会論文誌，Vol. 28, No. 3, pp. 310-313 (1987).
22) 佐藤，板野：構造エディタとインタプリタの統括的記述とその生成系，コンピュータソフトウェア，Vol. 4, No. 2, pp. 39-50 (1987).
23) 佐藤：COSMOS プログラミングシステム，情報処理学会，夏のシンポジウム，「究極のプログラミング環境」シンポジウム報告集，pp. 49-59 (1987).
24) 佐藤：COSMOS 上の構文指向プログラミング，情報処理学会第 36 回全国大会論文集，pp. 1085-1086 (1988).
25) 佐藤，田沼，小島，植村：内容を解釈実行する電子メイル/ニュースリーダとその応用，電子情報通信学会データ工学研究会，DE 88-23 (1988).

(平成元年2月21日受付)