属性文法に基づいた関数型言語ＡＧの プログラム作成・実行支援システム

篠田　陽一　・　橘　浩志　・　片山　卓也
（東京工業大学・情報工学科）

０．はじめに

属性文法はプログラム言語の意味記述のために、D.E.Knuthによって導入された形式システムであり(1)、プログラム言語やそのコンパイラの記述に有効であることが知られているが、その形式に多少の変更を加えると、一般の計算を記述するための計算モデルとして考えることができる(2)(3)。

プログラミング言語ＡＧは、この計算モデルに基づき、属性文法の各種応用に限らず一般的なプログラミングにも十分対応できる実用レベルの汎用核言語として設計された。

本報告では、ＡＧの言語仕様とエディタ、インタプリタ及びコンパイラを中心に構成されるプログラム開発環境ＳＡＧＥ (Support for AG Environment) について述べる。

１．ＡＧの言語仕様

1.1. データ型

1.1.1.　原始データ型

1) シンボル

シンボルは sunday や ABC などのような識別子（英字で始まる英数字列）であり、列挙型の要素、直積型の成分識別子、直和型のタグとして用いられる。

2) 整数型

整数の集合で定義される型であり、int で表す。

3) 実数型

実数の集合で定義される型であり、real で表す。

4) 列挙型

列挙型は、シンボルの列挙または、整数の部分集合の指定などにより、それらの値からなる集合を定義するものである。この集合の要素の間には定義の順番に従った順序関係があり、繰り返しモジュール分割を構成する際に利用されるが、その順序関係を陽に演算の中で用いることはできない。列挙型には、既定義のものと、ユーザ定義のものがある。

4-1) 論理型

論理型は論理値 true と false から成り、bool で表す。

4-2) 文字型

文字型は char で表現される列挙型であり、0 ～ 255 のコードに対応した文字の集合を表す。文字型の定数は、'a', '8' などのように文字の前後を“'”でくくって表現する。

4-3) ユーザ定義列挙型

これは利用者が定義する列挙型であり、その型に属する要素を指定することにより定義される。定義形態には、シンボルを列挙する場合(1)と整数または文字の範囲を指定する場合(2)の２通りがある。

```
type  T = { c1, c2, ..., cn }          ...(1)
      U = { r1 .. rn }                 ...(2)
```

5) ｎｉｌ型

ただ１つの値 nil から成る型であり、後述の再帰的な型定義の中で用いられる。

1.1.2.　構造データ型

1) 直積型

型定義の形態は、

```
type  T = ( s1 : T1, s2 : T2, ... sn : Tn )
```

である。型 T の要素は型 T1,...,Tn によって定められる直積空間の１点に対応する。ここで s1,...sn は成分識別子と呼ばれ、シンボルでなければならない。これら各成分の間には順序関係はない。

定数値は、

```
( s1 : c1, s2 : c2, ..., sn : cn )
```

のように成分識別子と成分値の対を並べて表現する。

直積型データ R の si 成分は R(si) によって取り出すことができる。

一般に複雑な入れ子構造を持つ構造型のデータに対して、そのある１成分だけを置換したデータを

```
R< p : e >
```

で表すことができる。p は R の中のある特定の成分を指定するもので通常は単なる成分識別子が書かれるが、R の入れ子構造に従い、一般に s1!s2!...!sn のような形式を持つ。e は式で、その値により p で指定した成分を置き換えたものが全体としての値となる。e の中では、R< p : &+1 > のように R の p で指定した成分の元の値を“&”で参照することができる。

2) 配列型

定義の形態は、

```
type    T = ( I : Ti )
```

I は配列のインデックスの型名で、列挙型でなければならない。Ti は配列の要素の型である。

定数値の表現、成分の取りだしについては、直積型の場合と同様であり、“(”，“)”をそれぞれ“[”，“]”に代えれば良い。

また、配列のある１要素だけを置換したものは、直積型の場合と全く同じ記法によって表現することができる。

3) 直和型

直和型は互いに素な集合の和集合を構成するのに用いられ、T1, T2, ..., Tn を任意の型とするとき、

```
type  T = tag1:T1 + tag2:T2 + ... + tagn:Tn
```

によって定義される。ここで tagi は型の識別に用いるタグで、互いに異なるシンボルでなければならない。この型

のデータは常にタグを伴った実体として認識される。したがって、直和型でないデータ D を直和型のデータに“変換”するには tagi:D のように必ずタグをつけなければならない。

逆に、直和型のデータ U からタグ t を取り去った生のデータを取り出すには、U(t)、また U の持っているタグを取り出すには U? なる記法を用いる。

直積型と直和型を組み合わせることにより、再帰的な型を定義することができる。例えばリスト構造は、

```
type  List = Nil:nil
                + node:( head: ListElem, tail: List )
      ListElem = elem:Element + list:List
```

のように表現することができる。

4) 列型

これは同じ型のデータの列を扱うために用意され、ＡＧではファイルをこの型の上で扱っている。定義形態は、

```
type  T = sequence of ET
```

である。ET は列の要素の型を表す。

列型の式は、要素を e1, e2, ..., en とするとき、< e1, e2, ..., en > で表すことができる。空列は <> で表され、すべての列型に属する。

列型データに対する演算としては、“^”，“~”，“+”があり、直感的には次のような意味を持つ。

```
^ <e1,e2,...>    = e1
<...,en-1,en> ^ = en
~ <e1,e2,...>    = <e2,...>
<...,en-1,en> ~ = <...,en-1>
<e1,e2,...,ei> + <ej,...,en>
         = <e1,e2,...,ei,ej,...,en>
```

4-1) 文字列型

文字列型は文字型のデータを要素とする列で、

```
type  string = sequence of char
```

として定義される列型である。この型の定数値は、一般の列型の記法による <'a','b','c'> を "abc" と省略することができる。

5) べき集合型

これは基底集合のすべての部分集合から成る集合を定義するものであり、

```
type  T = powerset of BT
```

によって定義される。ここで BT は基底集合を表し、列挙型でなければならない。

べき集合型の式は { e1, e2, ..., en } で表すことができる。空集合は {} で表され、すべてのべき集合型に属する。

6) モジュール型

モジュール型はモジュールを高階データとして扱うためのものである。定義は次のような形式で行う。

```
type  T = ( t1, ..., tn ¦ s1, ..., sm )
```

t1,...,tn は入力属性の型、s1,...,sm は出力属性の型である。

この型の要素は、適当な入出力属性の型を持つモジュールの名前であり、無論そのモジュールは正しく定義されていなければならない。また、この型の属性はモジュールの参照において通常のモジュール名と全く同等に扱われる。

1.2. モジュールとモジュール分割

1.2.1. モジュール

ＡＧではある一定の処理の単位をモジュールと呼んでいる。モジュールはその入出力関係のみによって指定され、いわゆる副作用はない。モジュールの入力データのことを入力属性、出力データのことを出力属性と呼ぶ。入力属性が x1, ..., xn で、出力属性が y1, ..., ym のモジュール M を次のように表す。

```
M( x1, ..., xn ¦ y1, ..., ym )
```

1.2.2. モジュール分割

モジュールの実行する処理内容が複雑な時には、モジュール分割によってその処理を細分化できる。ＡＧでは、この細分化の作業を繰り返すことによりプログラムが記述される。モジュール分割は、一定の数のモジュールへの分割を行う場合と、複数個の同一モジュールへの分割を行う繰り返しモジュール分割がある。

1) モジュール分割の基本形

モジュール分割は基本的に次の形をしている。

```
M( x1, ..., xn ¦ y1, ..., ym )
  case Cond1( x1, ..., xn )
    => M11( x111, x112, ... ¦ y111, y112, ... )
       M12( x121, x122, ... ¦ y121, y122, ... )
       ...
    where
          x1,y1,...,x111,y111,...,x121,y121,...
          に関する属性定義式の列
  case Cond2( x1, ..., xn )
    => M21( x211, x212, ... ¦ y211, y212, ... )
       M22( x221, x222, ... ¦ y221, y222, ... )
       ...
    where
          x1,y1,...,x211,y211,...,x221,y221,...
          に関する属性定義式の列
           ......
  case otherwise
    => Mk1( xk11, xk12, ... ¦ yk11, yk12, ... )
       Mk2( xk21, xk22, ... ¦ yk21, yk22, ... )
       ...
    where
          x1,y1,...,xk11,yk11,...,xk21,yk21,...
          に関する属性定義式の列
```

以下で、M のある最初の行をモジュール頭部、“=>”以下のモジュールの並びをモジュール分割部、“where”以下の式の並びを属性関係定義部と呼ぶことにする。

このモジュール分割では、モジュール M は条件 Cond1 が成立する時はモジュール M11, M12, ... に分割され、また Cond2 が成立する時はモジュール M21, M22, ... に

分割される。もし、Cond1, Cond2, ... のいずれも成立しない時は Mk1, Mk2, ... に分割される。otherwise を含んだ分割はなくてもよく、またモジュールが１通りにしか分割されない時は“case”による場合分けは必要ない。

また、あるモジュールをそれ以上、下位モジュールに分割する必要がない時はモジュール分割部に“return”という、仮想的な終端モジュールを書けば良い。

x1, x111, y211 などは属性生起と呼ばれ、

＜型名＞ または ＜型名＞.＜修飾子＞

のいずれかの形式を持っている。修飾子は任意の英数字列である。

属性関係定義部では、各属性間の関係を示す属性関係式が列挙される。その各々は、

a = f(a1, a2, ..., an)

の形式を持つ。a は属性名であり、f は属性 a1, ..., an を含む任意の式である。

また、記述量の減少と了解性の向上のため、本来属性関係定義部に記述されるべき式、またはその右辺のみを、それに対応する入力属性生起の代わりに書くことができる。例えば、

```
case  x > 0
      => fact( x.minus1 | y.new )
      where  x.minus1 = x - 1
             y = y.new * x
```

と書くべきところで、

```
case  x > 0
      => fact( x.minus1 = x - 1 | y.new )
      where  y = y.new * x
```

のように省略して書くことができる（さらに x.minus1 も省略可能）。

2) 繰り返しモジュール分割

基本的なモジュール分割では、いくつかの固定されたモジュールに分割が行われていたが、繰り返しモジュール分割では、複数の同一の型のデータに対し同様の演算を施し、その結果を全体として利用する際に用いる。

一般形は、

```
=>  for { P1, P2, ... | C }
        M1(...)
        M2(...)
        ...
    until S
    where
        ...
```

である。“| C”（繰り返し分割制限条件と呼ばれる）と“until S”（繰り返し分割停止条件と呼ばれる）の部分はなくても良い。

P1, P2 などは繰り返し指定子と呼ばれ次のいずれかである。

<列挙型パラメタ>
<パラメタ>：<列型属性名>
<パラメタ>：<べき集合型属性名>

後二者ではパラメタの型が、その右側に指定した構造型データの要素の型と一致していなければならない。パラメタは形式上、通常の属性生起と全く同じだが、繰り返し分割におけるそれぞれの属性生起を代表した高階な意味を持つ。

繰り返し分割制限条件は、入力パラメタに関する述語である。この条件に合わないパラメタ（の並び）に対しては分割が行われない。ただし、後で述べるような配列データへの収集を行う場合はこの条件があってはならない。

繰り返し分割では、入力パラメタが、繰り返し指定子で指定された値をその順序に従って取りながら、順次それらを入力属性として渡し、分割が進む。入力パラメタが複数個ある場合は、後に書かれた方のパラメタによる繰り返しを優先的に行う。これら分割された複数のモジュールは独立かつ並列であり、それらの間の属性依存関係はない。

こうして次々に分割されたモジュールのそれぞれの出力パラメタ e の値は、

q = CL of e または a(i) = e

という形式の収集式を属性関係定義部に書くことで収集が行え、親モジュールに渡すことができる。ここで q は整数型、実数型、論理型、列型、べき集合型のいずれかのパラメタ、a は配列型のパラメタ、i は入力パラメタである。また CL は一般的に次のいずれかの形式を持つ。

(a) v0 op @
(b) @ op v0
(c) op(v0, @)
(d) op(@, v0)

op は組み込みの２項中置演算子（前２者）または２入力１出力属性のモジュール（後２者）、v0 は初期値、@はパラメタ e の入る位置を示す。この時、CL of e の値は、パラメタである e の値が繰り返し分割によって v1, v2, ..., vn と変化する時に次のように定義される。

(1) n > 0 のとき
(a) (...((v0 op v1) op v2)... op vn)
(b) (vn op (vn-1 op ...(v1 op v0)...))
(c) op(op(...op(op(v0,v1),v2)...),vn)
(d) op(vn,op(vn-1,op(...op(v1,v0)...)))

(2) n = 0 のとき　v0

ただし、標準的な収集のために、次のような別名を用意してある。

```
sequence      = rcons( <>, @ )
sum           = @ + 0
product       = @ * 1
conjunction   = @ and true
disjunction   = @ or false
set           = insert( {}, @ )
```

ここで、rcons, insert は次のような仮想的メタ・モジュールである。

```
type       seq = sequence of any_elem
module rcons( seq.old, any_elem | seq.new )
    => return
         seq.new = seq.old + < any_elem >
end rcons
```

```
type        set = powerset of any_enum
module insert( set.old, any_enum | set.new )
    => return
        set.new = set.old + { any_enum }
end insert
```

通常の繰り返し分割は、繰り返し指定子で指定されたデータの要素すべてに対して行われるが、繰り返し分割停止条件 S を付けることにより、分割を中止することができる。これは繰り返し分割記述中に現れるすべての属性およびパラメタを含んでよい。

1.3. プログラム

ＡＧのプログラムは基本的には定数、型の定義および、モジュール定義の集合であり、通常それは機能によりいくつかのブロックに分けられる。

1.3.1. ブロック

ブロックはいくつかの互いに何らかの関連を持ったモジュール群から構成される。その一般的な形式は、

＜外部ブロック参照宣言部＞
＜定数宣言部＞
＜型宣言部＞
＜モジュール宣言部＞

これらはいずれも省略可能だが、定数、型、モジュールの宣言の内、最低１つはなければならない。

各ブロックは、各々異なるファイルに格納され、そのファイル名がそのブロックの名前となる。また後述するように最小のライブラリ単位でもある。

外部ブロック参照宣言部は、他のブロックのモジュールを使用する場合に必要であり、

use ＜ブロック名＞, ...

という形式を持つ。この宣言によって、そのブロックの中で参照可能とされている定数、型、モジュールを使うことができる。この宣言はそのブロックの中でのみ有効であり、他のブロックには影響を及ぼさない。

定数宣言は、各種の定数に名前を付ける機能である。形式は

export constant ＜定数定義式の並び＞
constant ＜定数定義式の並び＞

であり何れか片方でもよい。定数定義式は次の形式を持つ。

c1 = c2 = ... = cn = v

ci は定数名、v は定数式である。

“export” の側で宣言された定数は他のブロックからの参照が可能である。以下、“export” という prefix はすべて同様の意味で用いる。

型宣言は、型を定義するものである。形式は

export type ＜型定義式の並び＞
type ＜型定義式の並び＞

であり何れか片方でもよい。型定義式は次の形式を持つ。

t1 = t2 = ... = tn = t

ti は型名、t は型式である。これは、

t1 = t2
t2 = t3
...
tn-1 = tn

と書いたのと同じ意味を持つ。

ＡＧでは型の一致は、原則的に名前一致によって行っているので、２つの異なる型名 t1, t2 によって定義される型は構造的に同一であったとしても、直接または間接的に t1 = t2 という型定義式が現れない限りは、異なる型として扱われる。

モジュール宣言部は、入れ子になったモジュールの並びである。モジュールの入れ子は、

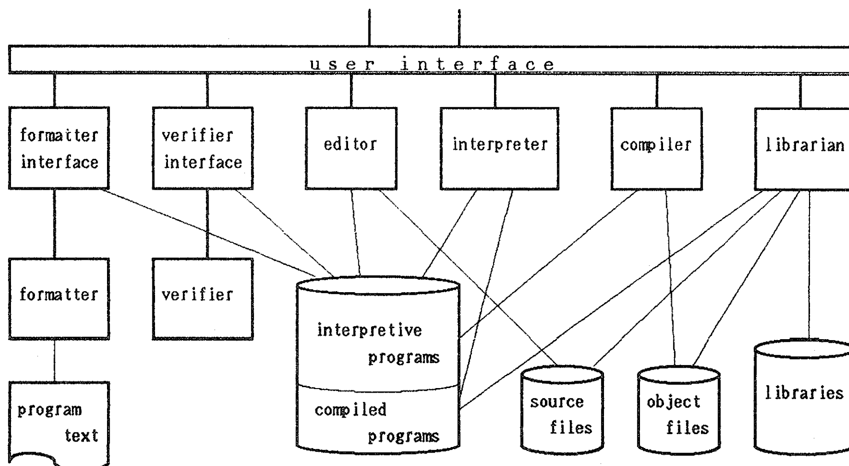

図2-1　ＳＡＧＥの構成

```
module A(...)
          ...
          module B(...)
                    ...
          end B
 end A
```

のように上位モジュールの“end”行の直前に、下位モジュールを挿入すればよい。各モジュールについては２章で述べた通りである。

“export”は入れ子モジュールの一番上位のモジュールにのみ前置することが許される。

２．統合プログラミング環境ＳＡＧＥ

2.1. 構成

ＳＡＧＥはＡＧのプログラムの作成・実行を支援する統合的な開発環境である。このシステムの機能単位の構成の概略は、図2-1に示すようであり、ユーザ・インタフェスを中心として、エディタ、インタプリタ、コンパイラ、ライブラリアン、ベリファイア、フォーマッタなどから構成される。

2.2. 実現

ＳＡＧＥは現在、Sun Workstation / UNIX 4.2BSD の上でインプリメントを行っている最中である。ベリファイア、フォーマッタの本体は既存のＵＮＩＸツールを利用しているが、その他は独自のもので記述言語はＣである。

３．プログラム作成支援環境

3.1. 外部環境

前章で述べたように、ＵＮＩＸをＯＳとするようなマシン上で動作することを仮定する。これは、

①入出力の最小単位が１文字である。

②入出力端末に対する依存度が小さい。

などの理由によるものである。

出力機器は、８０×２４文字のキャラクタ・ディスプレイ（vt100 端末相当）を標準とする。各文字には、反転、下線などの文字修飾機能があるものとする。また、入力はキーボードのみで、マウスなどのポインティング・デバイスは一切ないものとする。ただし、これは現時点での仮定で、将来的にはビットマップ・ディスプレイ、マウスをサポートするように拡張する予定である。

3.2. 設計方針

①プログラムの内部表現は構文木として持つが、ユーザがそれを意識せずに編集できるようなコマンド体系にする。即ち、ＡＧで書かれたプログラムを構文木として表現する場合、その形状は大枠では一致するとしても、必ずしも一意には表現できない。したがって、システムがたまたま採用している構文木の形状がそのまま編集の操作に反映するのは望ましいとは言えない。ユーザの立場に立てば、編集しているのはあくまでプログラム・テキストであって、構文木ではない。そこで、テキスト上での前後関係に基づく編集操作を重視することになる。

②ユーザの思考の焦点がシステムとの対話の中で、あちこちに飛び回ることを想定し、これに柔軟に対処できるようにする。例えば、構文エラーを残したままデバッガにかけられたり、プログラムに対する直前の変更は必ず取り消せるということが挙げられる。

③ユーザがどのようなコマンドを与えても、必ず何らかの応答を返すようにし、ある画面の表示を見た時に、システムがどのような状態にあるか一目で判別できるようにする。このため、コマンドはすべて１ストロークになっている。

④マニュアルの要らないセルフ・ドキュメントなシステムにする。ただし、ＡＧという言語そのものについては、予め知識があるユーザが用いるという仮定を置いている。

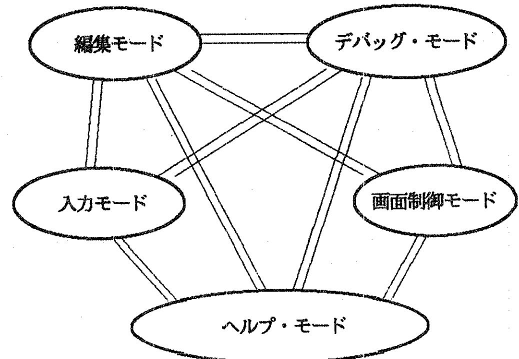

図3-1　主なモード間の移行関係

3.3. 概要

3.3.1 入力、編集方式

大まかな編集は、合成型構造エディタの編集方式に基づくが、式のレベルでは通常のテキストの編集に近い感覚での操作を行うようになっている。

コマンドはすべて１ストロークとし、編集モード、入力モード、ウィンドウ制御モード、デバッグ・モードなどのモードを設ける。主なモード間の移行関係は、図3-1 に示すようである。また、異なるモード間でもカーソルの移動、ヘルプなどの互いに共通の性質を持つものには、同じキーストロークを割り付けるなど、できるだけ一貫性の取れたコマンド体系にしている。

3.3.2 画面出力

画面分割方式によるマルチ・ウィンドウ方式を採用して

```
0: qsort (module)-----------------------1: qsort (declaration)----------------
module qsort( s | sorts )                  |type  s = sorts = sequence of elem
  case s == <> or ~s == <>                 |      elem = int
    => return                              |
         sorts = s                         |
  case otherwise                           |
    => less_half( ^s, ~s | s.less )        |
       great_half( ^s, ~s | s.great )      |
       qsort( s.less | sorts.less )        |
       qsort( s.great | sorts.great )      |
    where                                  |
      sorts = sorts.less + ^s +            |2: qsort (module)---------------------
              sorts.great                  |
                                           | module great_half( elem.key, s
 module less_half( elem.key, s             |                  | sorts.great )
                 | sorts.less )            |   => for ( elem:s | elem >= elem.key )
   => for ( elem:s | elem < elem.key )     |        return
        return                             |   where
   where                                   |     sorts.great = sequence of elem
     sorts.less = sequence of elem         | end great_half
 end less_half                             |end qsort
                                           |
Edit mode-----------------------------------------------------------------------
```

図3-2 編集モードにおける画面

```
0: qsort (module)-----------------------1: qsort (declaration)----------------
module qsort( s | sorts )                  |type  s = sorts = sequence of elem
  case s == <> or ~s == <>                 |      elem = int
    => return                              |
         sorts = s                         |
  case otherwise                           |
    => less_half( ^s, ~s | s.less )        |
       great_half( ^s, ~s | s.great )      |
       qsort( s.less | sorts.less )        |
       qsort( s.great | sorts.great )      |
    where                                  |
      sorts = sorts.less + ^s +            |
Debug mode----------------------------------------------------------------------
      qsort( s = <3,1,2> | sorts = ? )
 1      less_half( elem.key = 3, s = <1,2> | sorts.less = <1,2> )
 1      great_half( elem.key = 3, s = <1,2> | sorts.great = <> )
>1      qsort( s = <1,2> | sorts = <1,2> )
 2       less_half( elem.key = 1, s = <2> | sorts.less = <> )
 2       great_half( elem.key = 1, s = <2> | sorts.great = <2> )
 2       qsort( s = <> | sorts = <> )
 2       qsort( s = <2> | sorts = <2> )
```

図3-3 デバッグ・モードにおける画面

いる。ウィンドウは１つのメッセージ用ウィンドウまたはデバッグ用ウィンドウと、最大１０個までの編集用ウィンドウから成る。編集用ウィンドウはさらに、モジュール定義用と外部参照・型・定数定義用の２種類に分けられる。これらの各々には０～９までの識別番号が付けられている。この番号は、カーソルの他のウィンドウへの移動や画面制御モードにおけるウィンドウの指定に用いられる。図3-2 および図3-3 はそれぞれ、編集モード、デバッグ・モードにおける典型的な画面配置を示している。

編集用ウィンドウにおけるプログラム・テキストの出力フォーマットは、ウィンドウの大きさ、現在の編集位置などに応じて自動的に制御する。

現在のモード、プログラムの未入力部分、編集対象となるテキストの範囲などは、反転、下線などの文字属性を利用して、わかりやすい表示を心がけている。

3.3.3 プログラム作成のための機能

プログラム中に出現する個々の式や名前は、編集位置がそれらから離れる際に、通常構文解析が行われ、構文エラーは直ちに報告しユーザに修正を促す。タイプの不一致などの意味エラーについては、編集位置が他のモジュールに移る際に解析され、もしエラーがあればやはり修正を促す。しかし、ユーザがプログラムのある部分を意識的に不完全な状態にしたまま、他の部分の編集に移ることは常に可能である。この場合、構文的に不完全な部分は下線を用いた強調表示を行っている。

また、既知のモジュールや型を参照する場合、その情報を用いて自動的に入力の補助を行う機能がある。例えば図3-4 では、6,7 行目の編集を終わって“where”以下の編集に移ろうとする時に、モジュール頭部（1行目）とモジュール分割部（6,7行目）の属性生起から定義されるべき属性を 9-11 行目に自動的に挿入する。この時、これらの属性関係式の右辺は、未入力部分であるので反転の“?”で表示されている。

3.3.4 ヘルプ機能

どのようなモードにおいても、同じコマンドによって、その時点で有効なコマンドの一覧を一時的なウィンドウの上で見ることができる（ヘルプ・モード）。このモードで個々のコマンドについて更に詳しく知りたければ、そのコマンドのキーストロークを打鍵すればよい。

3.3.5 デバッガ

プログラムの編集とデバッグは切っても切れないものであり、ＳＡＧＥにおけるデバッガは、独立したサブシステムというよりは、エディタと融合しているといった方がよい。

デバッガは、デバッグ・モード（図3-3 ）で動かすこと

```
 1  module fib( int.x | int.fib )
 2      case int.x < 2
 3          => return
 4               int.fib = 1
 5      case otherwise
 6          => fib( int.x1 | int.fib1 )
 7             fib( int.x2 | int.fib2 )
 8          where
 9              int.x1 = ?
10              int.x2 = ?
11              int.fib = ?
12  end fib
```

図3-4 テキストの自動入力補助

ができる。デバッグ・モードの編集用ウィンドウの上では、編集モードにおけるコマンドの他に、実行制御コマンド、ブレイク・ポイントやスナップ・ポイントの設定などを行うことができる。また、デバッガと関連して編集モードでも使えるコマンドとして、プログラムの一部を削除せずに実際上無効にする機能がある。

ソース・コードがあるモジュールはすべて、どれほど不完全であっても、デバッガを通してインタプリタで実行することができる。不完全であればある程、実行中に何らかのエラーが発生する度合いが大きくなるのは言うまでもない。実行時に発生するエラーとしては、

①構文または意味エラーのある箇所に到達した。

②未定義の型、定数、モジュールを参照している箇所に到達した。

③属性の依存関係が完全に決定されていないモジュールを参照している箇所に到達した。

④演算上のエラーやスタック・オーバーフローなどにより計算が続行不可能になった。

などが主なものである。③のエラーは、モジュール分割部に列挙された子モジュールが実行される順番というのは一般には任意であり、実行系がそれらの子モジュールの属性生起の依存関係を調べて自動的に順番を定めるために、生ずるものである。

実行中に上記のようなエラーが発生した場合、またはユーザが設定したブレイク・ポイントに到達した場合には、インタプリタからエディタに制御が戻り、デバッグ用ウィンドウにスタック・トレースと停止した理由のメッセージが表示される。

④のエラー以外ならば、ユーザは適当な処置をしてデバッグを続行できる。通常はプログラムを編集することになるが、未定義のモジュール参照を解決するのに、そのモジュールの出力属性の値を強制的に与えたり、③のエラーの場合に、そのモジュールの依存関係だけを強制的に定めたりすることもできる。プログラムの編集を行うときは、編集領域が狭いので、一旦編集モードに戻ることにもできる。

4．実行系

4.1. 実行系の構成

ＳＡＧＥの実行支援系は、デバッガの管理下で動作するインタプリタ、及びコンパイラから構成され、エディタによって作成されたＡＧのプログラムの内部表現を対象としてその動作を行う。コンパイラは厳密な意味での実行系ではないが、 そのコード生成方式は、インタプリタの構成と密接な関連を持つため、あえて実行系としての解説を行う。

インタプリタは、内部表現を追いながら評価動作を行い、3.3.5.で示した状態を検出した場合はその動作を停止あるいは中断する。

コンパイラによるオブジェクト・コードの生成は、属性評価を手続きに変換する方法(4)を用い、テール・マージは当然のことながら、サブシステムであるグローバライザを用いた属性の大域化(5)や、モジュールのインライン展開等を行って良質のコードを生成する。また、プログラムをコンパイル時に解析し、ソースレベルでのプログラム変換を行ってモジュール数の減少を図り、生成されたコードにおける手続き呼び出しのオーバーヘッドを減らす REDUCER の研究も進められている。

4.2. 基本設計方針

コンパイラ／インタプリタ系の設計を行うに際して次の各点に留意した。

a) 最終的には全てのモジュールがコンパイルされた状態での実行が行われる。このときの実行効率を出来るだけ優先するような構造を持つこと。

b) 実行の為の環境は、ターゲット・マシンのネイティブ・モードを極力使用すること。 つまり、人為的なスタック等を用いる事を避けること。これは、インタプリタの実行にも当てはめられる。

c) デバッガの管理下で、インタプリタによるソースコード中のモジュール（ＳＭ：Source Module）の実行とオブジェクトコード中のモジュール（ＣＭ：Compiled Module）の混合実行が可能なようにする。

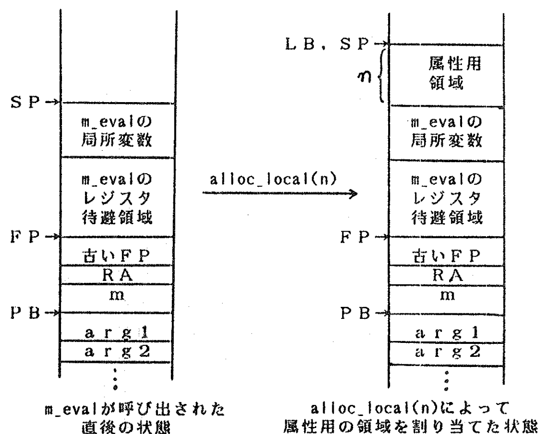

図4-1 関数m_evalのスタックフレーム

4.3. 実行環境

4.3.1. ＣＭのスタック構成

実行系はＣ言語に準じた環境で動作する。Ｃ言語は、静的な変数のスコープ規則を持たず、Ｐａｓｃａｌのように静的なスコープ規則を持つような言語の実行環境（スタック構成）と比較して、実行時の手続き呼び出しのコストが若干安いことが期待されると共に、コンパイラの出力をＣ言語にすることによるコンパイラ開発のプロトタイピングの容易さや、各種のシステムモジュール等がＣ言語で記述できるといった利点がある。

4.3.2. ＳＭのスタック構成

ここでのスタック構成とは、ソースコードの内部表現を解釈して評価を行うインタープリタの核関数（これを m_eval と呼ぶ）の実行時のスタック構成のことである。

m_eval は、

```
m_eval( m, arg )
  struct modd *m ;
  struct value *arg ;
```

のように宣言される関数で、開発言語であるＣ言語の処理系によって決定されるスタック構成は静的に決定され、このままでは評価に必要な領域をスタック上に割り付けることができない。そこで、m_eval はスタック上に局所変数領域を割り当てる特殊な関数 alloc_local を呼び出し、必要な作業領域を獲得する。この様子を図4-1に示す。

m_eval の局所変数であるＰＢ，ＬＢはそれぞれ parameter base, local base で、ＰＢは入出力属性の並びの最低位アドレスを、ＬＢは alloc_local によって獲得されたスタック上の局所変数領域の最低位アドレスを指すようにセットされる。これら２つの値は、m_eval の内部において局所変数や入出力属性の値を参照するために用いられる他、式の評価を行う関数 e_eval を呼び出す際に、現在評価が行われているモジュールの環境を正しく伝達するためにも用いられ、m_eval の動作において重要な役割を持っている。

4.3.3. 属性の表現

スタック上における属性値は全て型ディスクリプタとのタップルで表現されているものと考える。すなわち、属性値＝<型ディスクリプタ，値>のように表現されていると考えてよい。さらに、実際に実行系が扱う属性値は、次の２つのクラスに大別される。

a) Small Object と呼ばれる、属性の値の表現形式が４バイト以下のもの。これには int型及びそのsub-range 型、char型の値等が含まれる。このクラスの属性値では、値のフィールドに属性値そのものが埋め込まれる。

b) Large Object と呼ばれる、属性の値の表現形式が４バイトより大きなもの、又はその大きさが不定もしくは可変であるもの。このクラスには、配列型、列型のインスタンスが含まれ、値のフィールドには実際の値を保持する領域へのポインタが格納される。

Large Object において値を保持する領域は、スタック上とヒープ内の２箇所が考えられるが、これらを場合によって使い分けることにしている。（大きさが固定である配列型等の属性値は、スタック上に配置してもかまわず、記憶管理のためのオーバーヘッドがヒープ内に配置する方法に比べて少ない。一方、列型の属性値のように大きさが静的に決定できない場合はヒープ領域を用いなければならない。）

属性値の参照は、次の方式によって行われる。

a) 入力属性は実パラメータとして、属性値へのポインタを持つ。

b) 出力属性も入力属性と同様に、実パラメータとして属性値へのポインタを持つ。

c) 局所変数は、ＬＢをベースとした領域に直接格納されているものとする。

4.4. 混合実行方式

以下に、インタープリタによるＳＭの実行と、既にコンパイルされたＣＭの実行を混在させるときに生ずるＳＭ／ＣＭ間の呼び出しを処理する方法について述べる。

4.4.1. ＳＭ→ＳＭの呼び出し

ＳＭ→ＳＭの呼び出しは、

```
m_eval( m, i_atr1, i_atr2, ... , i_atrn,
        o_atr1, o_atr2, ..., o_atrn )
```

の形でモジュールの評価関数を再起的に呼び出すことによって行われる。但し、引数が可変個であるので、入出力の仕様をたどりながら 特別な関数 push_arg （一引数の関数で、呼び出しから戻ったときに引数がスタック上に残るような関数）を繰り返し呼び出して呼び出し環境の設定を行う。

4.4.2. ＳＭ→ＣＭの呼び出し

ＳＭ→ＣＭの呼び出しは、モジュール・ヘッダから、デマンド・リンクを行ったときに決定されるロードアドレスをみて、

```
load_addr( i_atr1, i_atr2, ... , i_atrn,
           o_atr1, o_atr2, ..., o_atrn )
```

の形でオブジェクトモジュールへの制御を移行することによって行う。呼び出し環境の設定は、ＳＭ→ＳＭの場合と同様に行われる。

4.4.3. ＣＭ→ＣＭの呼び出し

ＣＭ→ＣＭの呼び出しは、

a) リンクを行った際にアドレスを埋め込むことによって解決する。

b) 外部参照を解決するようなモジュールを仲介する。

の２つの方式を混在させる。

a)の方式は、通常のリンケージと同様な方法である。この方法は、混在実行時においてもＣＭ→ＣＭの呼び出しでは最高速度で動作するという利点を持つ。

b)の方式は、シンボリック・リンキング（外部関数の参照を行っている場所に、参照を解決する関数のアドレスを

埋め込み、その関数がロードアドレスのテーブルやＣＡＬＬアドレスのテーブルを参照して実際に要求されているモジュールを呼び出す。）とも呼ばれる。シンボリック・リンキングを用いると、柔軟なリンケージが行える。

4.4.4. ＣＭ→ＳＭの呼び出し

ＣＭ→ＳＭの呼び出しは、リンクを行う際に、参照されているモジュールのモジュール・ヘッダを付加した後、モジュール評価関数 m_eval を呼び出すような特殊なコードを生成して、このインターフェースコードの入口アドレスを埋め込むことによって対応する。

4.5. 子プロセスによる実行

エディタの管理下におけるプログラムの開発の段階では、全ての評価動作はその時点でのシステムのプロセスと全く同じメモリイメージを持った子プロセスをforkし、その子プロセス上で行われる。

親プロセスではデバッガが、fork した子プロセスのインタープリタに対して各種の命令を発行し、評価動作を行わせその動作を監視する。
これは、被実行プログラムによるシステムの破壊を防ぎ、システム全体の integrity を保つために重要である。

５．まとめ

統合プログラミング環境ＳＡＧＥは、本報告で述べたような設計思想に基づいて基本部分の制作がすすめられているが、列型の属性が外部ファイル、特に端末に対して結合されたときの扱いについてなど、解決しなければならない問題も幾つか残されている。

［参考文献］

(1) Knuth,D.E.: Semantics of Context-Free Language, Math. Sys. Theory, J.2 pp.127-145 (1968).
(2) Katayama,T.: HFP : A Hierarchical and Functional Programming Language Based on Attribute Grammar, Proc. of 5th International Conference on Software Engineering, pp.343-353 (1981).
(3) 片山卓也：属性文法による在庫管理システムの記述、情報処理,Vol.20, No.5, pp.478-485 (1985).
(4) Katayama, T.: Translation of Attribute Grammars into Procedures, ACM Transaction on Programming Languages and Systems, Vol.6, No.2, pp345-369 (1984).
(5) Katayama,T.,Sasaki,H. : Global Storage Allocation in Attribute Evaluation, Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages, pp.26-37 (1986).