拡張属性文法によるC言語の意味記述の一方法

神野俊昭（（株)日立製作所　システム開発研究所）

1．はじめに

属性文法は，プログラミング言語の意味を記述するために D.E.Knuth によって提案され［1］，その後今日に至るまで基礎，応用の両面で様々な研究がなされてきた．特に最近では，実用的なコンパイラ生成系を実現する立場から，コンパイラの意味解析部やコード生成部の仕様記述と自動生成に応用しようという研究も盛んに行われている［2］［3］．しかし，実用的なプログラミング言語の意味を属性文法で記述した例は意外に少なく，わずかに Pascal［4］や Ada[1]［5］をかぞえる程度である(文献［5］は正確には Ada 用の中間言語 Diana に対する意味記述である)．いうまでもなく，現実に使用されているプログラミング言語に属性文法を適用してみることは，その言語の厳密な仕様を定義するためだけでなく，属性文法自体の記述性を評価するためにも有効である．本論文では，C言語［6］をとりあげ，その静的意味(文脈依存の性質)を拡張属性文法(Extended Attribute Grammar)［7］とよばれる属性文法の1バージョンに基づいて記述する．C言語をとりあげるのは，それがシステム記述用の代表的言語であり，しかも，“特に高水準というわけではなく，また大きくもない”言語であるため，属性文法の記述性と適用性を評価するための素材として好適と考えられるからである．また，拡張属性文法に基づいて記述を行うのは，その表現の簡潔さから意味記述全体の見通しがよくなることが期待されるからである．以下，2で意味記述の体系について述べ，3でC言語の意味記述を行う．

2．拡張属性文法に基づく意味記述法

2.1　拡張属性文法

拡張属性文法は D. A. Watt と O. L. Madsen によって提案された．形式的な定義は文献［7］に述べられているので，ここでは，より直観的な定義を与える．通常の属性文法では，属性評価規則と文脈条件を生成規則とは分けて記述するのに対して，拡張属性文法では生成規則の中に直接埋め込んで表現する．そのために次のような表現機構が用意される．まず，各文法記号(非終端記号と終端記号)は有限個の属性ポジションをもち，各ポジションには属性ドメイン(属性のとりうる値の集合)が対応づけられる．生成規則中の各文法記号の任意の属性ポジションには対応する属性ドメインの要素を値とする属性式を記述する．属性式は定数か，属性変数か，属性式に関数を作用させたものである．属性ポジションには通常の意味での相続か合成かの区別がなされる．生成規則の左辺の記号の相続属性ポジションおよび右辺の記号の合成属性ポジションを定義ポジションといい，生成規則の左辺の記号の合成属性ポジションおよび右辺の記号の相続属性ポジションを被定義ポジションという．定義ポジション，被定義ポジションは生成規則を属性評価器とみた場合のそれぞれ入力属性，出力属性を表す．

拡張属性文法では，被定義ポジションに一般の属性式を許すことによって属性評価則を表現し，定義ポジションに一般の属性式を許すことによって文脈条件を表現する．例えば，下記の(1)は通常の属性文法による記述例，(2)はそれと同じことを拡張属性文法によって記述した例である(文献［7］より引用)．ただし，下向き矢印(↓)は相続属性，上向き矢印(↑)は合成属性を表す．また，(1)でevaluate節は属性評価則を，where節は文脈条件を表す．

1）Ada は米国政府の開発した言語であり，米国政府AJPO(Ada Joint Program Office)の米国における登録商標である．

```
(1)  assignment↓ENV
       ::=
     identifier↓ENV1↑MODE1  ":="  expression↓ENV2↑MODE2
     evaluate  ENV1  <-  ENV
     evaluate  ENV2  <-  ENV
     where  MODE1  =  ref(MODE2)

(2)  assignment↓ENV
       ::=
     identifier↓ENV↑ref(MODE)  ":="  expression↓ENV↑MODE
```

このように，拡張属性文法では簡潔で読み易い意味記述が可能になるが，少し複雑な文脈条件を生成規則の中で表現しきろうとすると，新しい非終端記号とそのための新しい生成規則(多くはε-規則)を作り出さなければならないという欠点がある．そこで我々の意味記述法は，拡張属性文法をベースとしながら，その形式上の制約を緩めて文脈条件は生成規則と独立にcondition節で表現することにする．

2.2 意味記述の構造

ここでは，プログラミング言語の意味記述の全体構成を示し，属性ドメインの構成法について述べる．意味記述は次の各部から構成される．

- 属性ドメイン定義部
- 語彙定義部
- 属性変数定義部
- 意味記述本体部

属性ドメイン定義部は，属性がとる値の集合を定義しそれに名前を与える．語彙定義部は，文法記号の各属性ポジションに対応する属性ドメインを，相続，合成の区別をつけて定義する．属性変数定義部は，属性変数を導入し属性ドメイン(タイプ)を対応づける．意味記述本体部は，生成規則ごとに属性評価則と文脈条件を記述する．

属性ドメインの構成方法には表示的意味論のドメイン構成法[8]をとりいれて，記述の簡潔性と抽象性を向上させる．まず，基本ドメインが定義される．次に，基本ドメインから一般のドメインを作り上げるためのドメイン構成子が定義される．

(1) 基本ドメイン

基本ドメインは，標準ドメインと有限ドメインとから成る．標準ドメインは，

Integer = {…, -1, 0, 1, 2, …}

Bool = {true, false}

Name = {I | I は英字で始まる英数字の列}

などのように，予め定義されているドメインである．有限ドメインは，

Operator = {plus, minus, equal, unequal}

などのように，その要素を全て列挙することによって定義されるドメインである．

(2) ドメイン構成子

ドメイン構成子には，直積，直和，シーケンス，集合，関数の5つがある．

(a) 直積 $P = D_1 \times \cdots \times D_n$

Pの要素を$[d_1, \cdots, d_n]$と書く．オペレーションel_iが既定である．

el_i $[d_1, \cdots, d_n] = d_i$

(b) 直和 $S = g_1[D_1] \times \cdots \times g_n[D_n]$

$g_1, \cdots, g_n$は選択子であり，相異なる名前でなければならない．D_iが空の場合には$g_i[D_i]$と書く代わりに単にg_iと書く．Sの要素を$g_i[d_i]$と書く．オペレーションis_g_i，out_D_iが既定である．

$is\_g_i : S \rightarrow Bool$

$$is\_g_i\ d = \begin{cases} true & \text{ある} d_i \in D_i \text{に対して} d = g_i[d_i] \text{のとき，} \\ false & \text{そうでないとき} \end{cases}$$

$out\_D_i : S \rightarrow D_i$

$$out\_D_i\ d = \begin{cases} d_i & d = g_i[d_i] \text{のとき，} \\ undefined & \text{そうでないとき} \end{cases}$$

(c) シーケンス $L = D^*$

Lの要素を$d_1 \cdot \cdots \cdot d_n$と書く($d_i \in D$)．空シーケンスを$\varepsilon$と表記する．オペレーションel，hd，tl，nullが既定である．

$el : Integer \rightarrow S \rightarrow D$

$el\ i\ d_1 \cdot \cdots \cdot d_n = d_i$

$hd : S \rightarrow D$

$hd\ d_1 \cdot \cdots \cdot d_n = d_1$

$tl : S \rightarrow S$

$tl\ d_1 \cdot \cdots \cdot d_n = d_2 \cdot \cdots \cdot d_n$

$null : S \rightarrow Bool$

$$null\ s = \begin{cases} true & s = \varepsilon \text{ のとき，} \\ false & \text{そうでないとき} \end{cases}$$

(d) 集合 T = set of D

一般の集合演算が既定のオペレーションとして用いられる．

(e) 関数 F = D → R

$d_1, \cdots, d_n$をDの相異なる要素とし，$r_1, \cdots, r_n$をRの要素とするとき，$\{d_1 \rightarrow r_1, \cdots, d_n \rightarrow r_n\}$は$d_1, \cdots, d_n$で定義され，それらを$r_1, \cdots, r_n$に写像する関数である．Dのいかなる要素でも定義されていない関数を{}で表す．オペレーション∪と\が既定である．

```
∪ : F × F → F
    (f ∪ g)(d) = if (f(d) and g(d) are defined) then undefined
                 else if f(d) is defined then f(d) else g(d)
\ : F × F → F
    (f \ g)(d)  = if g(d) is defined then g(d) else f(d)
```

3．C言語の意味記述

初めにC言語の属性文法を記述するのに必要な属性ドメインの定義を与え，次いで，意味記述本体を示し説明を行う．ただし，意味記述の全体を示すことは限られた紙数では不可能なので，C言語の特徴がよく現れており意味記述を行う上で特に意を用いた宣言の部分についてのみ述べる．なお，Cの言語仕様は文献［6］に基づいている．

3.1 属性ドメイン定義

```
Env       = Name → Mode                                    --環境属性
```

```
Mode      = var[Class × Type]                                  --名前モード属性
          + type[Type]
          + field[Type]
          + formal[Class × Type]
          + func[Class × Type × Types]
          + label
Type      = char                                               --型属性
          + int[Length × Sign]
          + float[Precision]
          + voidtype
          + array[Size × Type]
          + struct[Types × Env]
          + union[Types × Env]
          + pointer[Type]
          + func[Type]
          + bit[Integer × Sign]
          + typename[Name]
Length    = {short, normal, long}
Sign      = {signed, unsigned}
Precision = {single, double}
Size      = bound[Integer]
          + unbound
Class     = {auto, static, extern, register, typedef}          --記憶クラス属性
Types     = Type*
TAssoc    = Name × Type
OrgType   = char + short + int +long + unsigned                --基底型属性
          + float +double + shortint +longint
          + unsint + longfloat
          + struct[Types × Env]
          + union[Types × Env]
          + typename[Name]
Emode     = Type × Vmode × Cmode                               --式属性
Vmode     = {lvalue, notlval}                                  --左辺値モード属性
Cmode     = ce[Value]                                          --定数式モード属性
          + notce
Value     = int[Integer]
          + real[Real]
```

3.2 宣言の意味記述

語彙定義

```
declaration ↓Env ↓Env ↑Env                                                --宣言
decl-specifiers ↓Env ↓Env ↓Class ↓OrgType ↑Class ↑OrgType ↑Env            --宣言指定子リスト
decl-specifier ↓Env ↓Env ↓Class ↓OrgType ↑Class ↑OrgType ↑Env            --宣言指定子
sc-specifier ↓Class ↑Class                                                --記憶クラス指定子
type-specifier ↓Env ↓Env ↓OrgType ↑OrgType ↑Env                           --型指定子
```

```
declarator-list ↓ Env ↓ Env ↓ Class ↓ Type ↑ Env                          --宣言子リスト
init-declarator ↓ Env ↓ Env ↓ Class ↓ Type ↑ Env                          --初期宣言子
declarator ↓ Env ↓ Type ↑ TAssoc                                          --宣言子
initializer ↓ Env ↓ Class ↓ Type                                          --初期設定
initial-value-list ↓ Env ↓ Class ↓ Type ↓ Integer ↑ Integer               --初期値リスト
initial-value ↓ Env ↓ Class ↓ Type ↓ Integer ↑ Integer                    --初期値
expression ↓ Env ↑ Emode                                                  --式
```

意味記述本体

```
(1) declaration ↓ GLOBAL ↓ LOCAL1 ↑ LOCAL3
      ::=
    decl-specifiers ↓ GLOBAL ↓ LOCAL1 ↓ void ↓ void ↑ CLASS ↑ OTYPE ↑ LOCAL2
      declarator-list ↓ GLOBAL ↓ LOCAL2 ↓ CLASS ↓ basetype(OTYPE) ↑ LOCAL3
    where basetype = λ x : OrgType.
                     ((x=void) or (x=int)) → int[normal, signed],
                     ((x=short) or (x=shortint)) → int[short, signed],
                     ……

(2) decl-specifiers ↓ GLOBAL ↓ LOCAL1 ↓ CLASS1 ↓ OTYPE1 ↑ CLASS3 ↑ OTYPE3 ↑ LOCAL3
      ::=
    decl-specifier ↓ GLOBAL ↓ LOCAL1 ↓ CLASS1 ↓ OTYPE1 ↑ CLASS2 ↑ OTYPE2 ↑ LOCAL2
      decl-specifiers ↓ GLOBAL ↓ LOCAL2 ↓ CLASS2 ↓ OTYPE2 ↑ CLASS3 ↑ OTYPE3 ↑ LOCAL3

(3) decl-specifier ↓ GLOBAL ↓ LOCAL1 ↓ CLASS ↓ OTYPE1 ↑ CLASS ↑ OTYPE2 ↑ LOCAL2
      ::=
    type-specifier ↓ GLOBAL ↓ LOCAL1 ↓ OTYPE1 ↑ OTYPE2 ↑ LOCAL2

    decl-specifier ↓ GLOBAL ↓ LOCAL ↓ CLASS1 ↓ OTYPE ↑ CLASS2 ↑ OTYPE ↑ LOCAL
      ::=
    sc-specifier ↓ CLASS1 ↑ CLASS2

(4) sc-specifier ↓ CLASS ↑ auto     ::=  "auto"      condition CLASS = void

    sc-specifier ↓ CLASS ↑ static   ::=  "static"    condition CLASS = void

    sc-specifier ↓ CLASS ↑ extern   ::=  "extern"    condition CLASS = void

    sc-specifier ↓ CLASS ↑ register ::=  "register"  condition CLASS = void

    sc-specifier ↓ CLASS ↑ typedef  ::=  "typedef"   condition CLASS = void

(5) type-specifier ↓ GLOBAL ↓ LOCAL ↓ OTYPE ↑ char ↑ LOCAL  ::=  "char"  condition OTYPE = void

    type-specifier ↓ GLOBAL ↓ LOCAL ↓ OTYPE ↑ short ↑ LOCAL ::=  "short" condition OTYPE = void

    type-specifier ↓ GLOBAL ↓ LOCAL ↓ OTYPE ↑ newint(OTYPE) ↑ LOCAL  ::=  "int"
```

```
where newint = λ x : OrgType.
               (x=void) → int,
               (x=short) → shortint,
               (x=long) → longint,
               (x=unsigned) → unsint,
               error
……
```

(6) declarator-list ↓ GLOBAL ↓ LOCAL1 ↓ CLASS ↓ TYPE ↑ LOCAL3
::=
init-declarator ↓ GLOBAL ↓ LOCAL1 ↓ CLASS ↓ TYPE ↑ LOCAL2 ","
declarator-list ↓ GLOBAL ↓ LOCAL2 ↓ CLASS ↓ TYPE ↑ LOCAL3

(7) init-declarator ↓ GLOBAL ↓ LOCAL ↓ CLASS ↓ TYPE1 ↑ (LOCAL ∪ newenv(NAME, CLASS, TYPE2))
::=
declarator ↓ (GLOBAL \ LOCAL) ↓ CLASS ↓ TYPE1 ↑ [NAME, TYPE2]
initializer ↓ (GLOBAL \ LOCAL) ↓ CLASS ↓ TYPE2
condition can_be_initialized(CLASS, TYPE2);
legal_combination(CLASS, TYPE2)
where newenv = λ (x, y, z):Name × Class × Type.
(y=void) → {x → var[auto, z]},
{x → var[y, z]}

init-declarator ↓ GLOBAL ↓ LOCAL ↓ CLASS ↓ TYPE1 ↑ (LOCAL ∪ newenv(NAME, CLASS, TYPE2))
::=
declarator ↓ (GLOBAL \ LOCAL) ↓ CLASS ↓ TYPE1 ↑ [NAME, TYPE2]
condition legal_combination(CLASS,TYPE2)
where newenv = λ (x, y, z):Name × Class × Type.
(y=typedef) → {x → type[z]},
(y=void) → ((is_func z) → {x → var[extern, z]}, {x → var[auto, z]}),
{x → var[y, z]}

(8) declarator ↓ ENV ↓ TYPE ↑ [NAME, TYPE]
::=
identifier ↑ NAME

declarator ↓ ENV ↓ TYPE1 ↑ [NAME, pointer[TYPE2]]
::=
"*" declarator ↓ ENV ↓ TYPE1 ↑ [NAME, TYPE2]

declarator ↓ ENV ↓ TYPE1 ↑ [NAME, func[TYPE2]]
::=
declarator ↓ ENV ↓ TYPE1 ↑ [NAME, TYPE2] "(" ")"
condition legal_return_type(TYPE2)

declarator ↓ ENV ↓ TYPE1 ↑ [NAME, array[bound[out_Integer CMODE], TYPE2]]
::=

```
    declarator ↓ ENV ↓ TYPE1 ↑ [NAME, TYPE2]
      "["  expression ↓ ENV ↑ [TYPE3, VMODE, CMODE]  "]"
    condition legal_element_type(TYPE2);
              integer_constant(TYPE3, CMODE)

    declarator ↓ ENV ↓ TYPE1 ↑ [NAME, array[unbound, TYPE2]]
      ::=
    declarator ↓ ENV ↓ TYPE1 ↑ [NAME, TYPE2]  "["  "]"
    condition legal_element_type(TYPE2)

    declarator ↓ ENV ↓ TYPE ↑ TASSOC
      ::=
    "("  declarator ↓ ENV ↓ TYPE ↑ TASSOC  ")"

(9) initializer ↓ ENV ↓ CLASS ↓ TYPE1
      ::=
    "="  expression ↓ ENV ↑ [TYPE2, VMODE, CMODE]
    condition legal_expression(CLASS, CMODE);
              type_compatible(TYPE1, TYPE2)

    initializer ↓ ENV ↓ CLASS ↓ TYPE
      ::=
    "="  "{"  initial_value_list ↓ ENV ↓ CLASS ↓ TYPE ↓ 1 ↑ NEXT  "}"

(10)initial_value_list ↓ ENV ↓ CLASS ↓ TYPE ↓ FIRST ↑ NEXT2
      ::=
    initial_value ↓ ENV ↓ CLASS ↓ TYPE ↓ FIRST ↑ NEXT1  ","
      initial_value_list ↓ ENV ↓ CLASS ↓ TYPE ↓ NEXT1 ↑ NEXT2

(11)initial_value ↓ ENV ↓ CLASS ↓ TYPE1 ↓ FIRST ↑ (FIRST+1)
      ::=
    expression ↓ ENV ↑ [TYPE2, VMODE, CMODE]
    condition legal_expression(CLASS, CMODE);
              not_too_many_initializers(TYPE1, FIRST);
              type_compatible(findscalar(TYPE1, FIRST), TYPE2)
    where findscalar = λ (t, n) : Type × Integer.
                       (is_scalar t) → t,
                       (t=array[s, te]) → findscalar(te, n-(n-1)/size(te)*size(te)),
                       (t=struct[tm, e]) → {(n <= size(hd tm)) → findscalar(hd tm, n),
                                            findscalar(struct[tl tm, e], n-size(hd tm))}

    initial_value ↓ ENV ↓ CLASS ↓ TYPE ↓ FIRST ↑ (FIRST+size(NEWTYPE))
      ::=
    "{"  initial-value-list ↓ ENV ↓ CLASS ↓ NEWTYPE ↓ 1 ↑ NEXT  "}"
    where NEWTYPE = findaggregate(TYPE, FIRST)
    where findaggregate = λ (t, n) : Type × Integer.
                          (n=1) → t,
```

```
                    (t=array[s, te]) → findaggregate(te, n-(n-1)/size(te)*size(te)),
                    (t=struct[tm, env]) → findaggregate(findelem(tm, n))
where findelem = λ (t, n) : Types × Integer.
                 (n<=size(hd t)) → [hd t, n],
                 findelem(tl t, n-size(hd t))
```

3.3 補足説明

環境属性の評価

宣言に対する意味記述では，環境属性の評価が中心課題となる．これは，コンパイラの処理でいえば，宣言部を解析して記号表を作成することに相当する．規則（1）は，この宣言より前の宣言で作られた環境属性をdeclarationの相続属性として入力し，この宣言によって新たに作り出される環境属性をdeclarationの合成属性として出力する．「この宣言より前の宣言で作られた環境属性」は，さらに，大域環境属性(この宣言の置かれているブロックを囲むブロックの宣言に対する環境属性)と局所環境属性(この宣言の置かれているブロックでこの宣言より前にある宣言に対する環境属性)に分離する．こうすると，ブロック構造型言語の名前の有効範囲規則を自然な仕方で表現することができる．declarationに付与されている属性 GLOBAL, LOCAL1, LOCAL3 は，それぞれ，大域環境属性，旧局所環境属性，新局所環境属性を表現している．例えば，図1の宣言 float c ; に関して，規則（1）の各環境属性値は図2のようになる．

```
extern  int  a ;
int  f ( )
  {  char  b ;
     float  c ;
     ……
  }
```

図1

```
GLOBAL = {a → var[extern, int[normal, signed]],
          f → func[extern, int[normal, signed], ε }
LOCAL1 = {b → var[auto, char]}
LOCAL2 = LOCAL1
LOCAL3 = LOCAL2 ∪ {c → var[auto, float[single]]}
```

図2

実際に新しい環境属性を作る操作は規則（7）で行われる．すなわち，個々の初期宣言子に対する属性評価が終了したときに，その第5属性値として新環境属性が求められる．LOCAL ∪ newenv(NAME, CLASS, TYPE2)がそれである．

文脈条件と省略時解釈

C言語では，宣言における宣言指定子(型指定子と記憶クラス指定子)の指定の仕方にいろいろな制約を設けている．型や記憶クラスの指定の仕方によっては，初期設定が許されない場合や初期値式に制約が加えられる場合もある．また，宣言指定子が省略された場合の解釈も一様ではない．以下に，これらのことに関する規定の一部を，規定を記述している規則番号を対応させて示す．

① 型指定子の省略時解釈は int とする(1)．
② 2つ以上の記憶クラス指定子を指定してはならない(4)．
③ 型指定子の許される組合せは次のものだけである:short int, long int, unsigned int, long float. long float は double と同じ意味に解釈する(5)．
④ 記憶クラス指定子の省略時解釈は auto とする．ただし，宣言子が関数型の場合は extern とする(7)．
⑤ 記憶クラス指定子 register は算術型とポインタ型の変数に対してのみ指定できる(7)．
⑥ 記憶クラス指定子 auto と static を関数型に指定してはならない(7)．

⑦ 関数型の返すことのできる型は関数型や配列型であってはならない(8).
⑧ 外部変数および static変数の初期値は定数式でなければならない(9, 11).
⑨ 複合型(構造体型と配列型)の auto変数に初期値を与えてはならない(11).
⑩ 複合型変数に対し，その要素数よりも多い初期値を与えてはならない(11).

初期化規則

C言語の初期化規則を属性文法で厳密に記述する際に障害となるのは，2つの例外規定である．まず，初期化の原則と例外を整理しておく．

[原則]

1．初期化される変数がスカラー型のとき，初期値は exp または {exp} と書く．
2．初期化される変数が複合型のとき，初期値は {exp, …, exp} と書く．
3．初期化される変数が複合型で，それがさらに複合型の要素を含むとき，ルール2を要素に対して繰り返し適用する．

[例外]

1．初期化される変数が複合型で，その要素の数よりも少ない数の初期値を指定したとき，初期値と対応しない残りの要素には0が初期値として与えられる．
2．内側の括弧 { } は省略することができる．その場合の初期化規則は次のようになる．
   a) 複合型データに対応する初期値が { で始まらない場合，その全ての要素に対して順次初期値を与える．
   b) a)の対応づけの後さらに初期値が残っている場合は，それらの初期値はa)で初期値を与えられたデータの次の要素と対応づけられる．

図3に初期値の対応例を示す．宣言の下に書かれた数字の列は，その順に，x[0][0], x[0][1], x[1][0], x[1][1]に初期値として与えられる．

```
int  x[2][2] = {
   {1, 2},
   {3, 4}
} ;
 1, 2, 3, 4
    (a)
```

```
int  x[2][2] = {
 1, 2, 3, 4
} ;
 1, 2, 3, 4
    (b)
```

```
int  x[2][2] = {
   {1}, {2}
} ;
 1, 0, 2, 0
    (c)
```

図3

我々は，初期化規則をトップダウン型で属性文法表現する．すなわち，初期値全体の解析木の根(initializer)に宣言された変数の型を渡し，それを下向きに伝えてゆく．括弧が現れたら，それより下の部分木には，対応する要素の複合型を切り出して渡す．そのようにして葉(初期値)に到達したときにはスカラー型が切り出されるように型の細分と伝播を繰り返す．しかしここで注意を要するのは，括弧が省略された場合でも最終的にスカラー型が切り出されなければならないという点である．そこで，型の切り出しを制御する整数属性Ｉを型属性Ｔとともに伝播させる．Ｉは部分木を左から右に辿っていくときに最初に出会う初期値が，ＴのＩ番目のスカラー要素に対応するような性質を満たす．もし（Ｔ，Ｉ）が伝播されたところで括弧が現れたら，ＴのＩ番目のスカラー要素を第一要素にもつような最上位の複合型が切り出されて部分木に渡される．ＴとＩは相続属性であるが，Ｉの値を決めるには今までに何番目のスカラー要素まで初期値の対応がなされたかという情報もいる．これは合成属性である(Ｊとする)．initial-value-list および initial-valueの第３，第４，第５属性は，それぞれ，Ｔ，Ｉ，Ｊに対応している．規則（９），(10)，(11)は以上述べたことを，形式的に記述したものである．規則(11)で定義されている関数 findscalar, findaggregate はそれぞれ，ＴのＩ番目のスカラー型，ＴのＩ番目のスカラー要素を第一要素とする最上位の複合型を計算する関数である．

4．おわりに

拡張属性文法に基づくC言語の意味記述の一方法を述べた．本稿では，C言語の属性文法の特徴が最もよく反映されている宣言に関してのみ論じた．C言語の静的意味は属性文法で完全に表現できることを確認した．しかしプログラミングの便をはかるために例外規定を多く盛り込んだところほど，当然のことながら形式的記述は困難であった．我々が記述したC言語の属性文法は，定義ラベル属性の伝播を除いて L-attributed のクラスに属する．拡張属性文法は，記述量が少ない，属性間の依存関係が容易に見てとれる，属性の流れの直観的な把握がし易い等の点で"純粋"の属性文法よりも扱い易かった．

参考文献

[1] Knuth, D. E. : Semantics of Context-free Languages, Mathematical Systems Theory 2, 127-145, 1968

[2] Kastens, U., et al. : GAG : A Practical Compiler Generator, LNCS 141, Springer, 1982

[3] 石塚治志，佐々政孝 : 属性文法によるコンパイラ生成系，第26回プログラミング・シンポジウム論文集，69-80，1985

[4] Watt, D. A. : An Extended Attribute Grammar for Pascal, SIGPLAN Notices 14, 2, 60-74, 1979

[5] Uhl, J., et al. : An Attribute Grammar for the Semantic Analysis of Ada, LNCS 139, Springer, 1982

[6] Kernighan, B. W. and Ritchie, D. M. : The C Programming Language, Prentice-Hall, 1978

[7] Watt, D. A. and Madsen, O. L. : Extended Attribute Grammars, Computer Journal 26, 2, 142-153, 1983

[8] Gordon, M. : The Denotational Description of Programming Languages, Springer, 1979