ソフトウェア基礎論　16－8
プログラミング言語　4－8
（1986. 2. 7）

属性文法に基づくCコンパイラの系統的実現手法

神野俊昭*　森　敬安*　柏木有吾**　矢島　宏**
（*(株)日立製作所　システム開発研究所　**同　武蔵工場）

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目的で，D. E. Knuth によって提案されて以来，基礎，応用の両面で様々な研究が行われてきた．特に近年，コンパイラの自動生成との関連で注目を集め，実際に属性文法に基づくコンパイラ生成系もいくつか開発されている[1][2][3]．しかし，実用的なコンパイラを属性文法を応用して開発した例はまだ少なく，また，そうした例においても，属性文法の応用は意味解析部を中心とするコンパイラの一部分に限定されることが多かった．本稿では，属性文法に基づき，実用コンパイラを系統的に実現する手法を，主にコンパイラ仕様の形式的記述の問題を中心にして，Cコンパイラに即して述べる．C[4]を取り上げるのは，それが今日，実用言語として確かな位置を占めつつあるからである．

ここでは，次のようなコンパイラの処理モデルを想定している．コンパイラはフロント部，ミドル部，バック部から成る．フロント部は構文解析と意味解析を行って，中間語とシンボル・テーブル等を生成する．ミドル部は最適化を行って，改良された中間語を生成する．バック部はコード生成とピープホール最適化を行って，オブジェクトを生成する．中間語は一貫して属性付き抽象構文木の形式をとる．属性文法の手法は，フロント部とバック部に適用される．また，ミドル部にも部分的に適用される．本稿で述べるのは，フロント部とバック部への適用についてである．

以下，2で，属性文法に基づくコンパイラ仕様の記述方法について述べる．3と4で，それぞれ，フロント部(構文・意味解析)とバック部(コード生成)への属性文法手法の適用について述べる．

2．コンパイラ仕様の記述方法

我々が用いている，コンパイラ仕様の記述方法について，構文・意味解析仕様を基準にして述べる．このため，記述項目によってはコード生成仕様などでは用いないものもあるが，基本的な構造は共通である．

2.1　仕様の基本構造

属性文法は，そのままでは，コンパイラの仕様を記述するのに機能不足のところ(例えばエラー回復)がある．これを補完しながら，コンパイラの仕様の構造を次のように定める．

(1) 属性ドメイン定義部：属性がとる値の集合を定義し，それに名前を与える．
(2) 語彙定義部：文法記号に相続属性と合成属性を付与する．中間語ノードに固有属性を付与する．
(3) 属性変数定義部：属性変数を導入し，それに属性ドメインを対応づける．
(4) 意味記述本体部：生成規則ごとに属性評価則，文脈条件，エラー回復，中間語生成を定義する．

意味記述本体部は，次の形式をもつ記述単位の集合である．

生成規則形式
[where　生成規則適用条件]
[condition　文脈条件式の集合]
[evaluate　属性評価式の集合]
[generate　中間語仕様]

ここで，生成規則形式とは，生成規則においてその中に現れる文法記号の各属性ポジションに属性式を指定したものであり，拡張属性文法(Extended Attribute Grammar)[5]の記述スタイルに則って属

性評価則が定義される(拡張属性文法を用いるのは，比較的簡潔な記述ができ，また，無駄な属性名を導入しなくてすむ，という理由による)．where節では，この生成規則を適用する条件を記述する(詳しくは３．２で述べる)．condition節で指定される文脈条件の成立時と不成立時とで属性評価が変わるものについては，evaluate節に分離して記述する．evaluate節の属性評価式は次の形式をしている．

属性変数　＝　属性式１　｜　属性式２

文脈条件が全て成立したときは属性式１が，そうでないときは属性式２が属性変数の値となる(後者においてエラー回復の指定がなされる)．generate節では，この生成規則が適用されたときに生成される中間語の仕様を指定する．

２．２　属性ドメインの構成法

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[６]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．

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，BOOLEAN，NAME 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，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．ドメイン構成子には，直積，直和，シーケンス，ベキ集合，関数の5つがある．それぞれ，次のように表記する．

- 直積　$P = D_1 \times \cdots \times D_n$
- 直和　$S = g_1[D_1] + \cdots + g_n[D_n]$
- シーケンス　$L = D^*$
- ベキ集合　T = set of D
- 関数　$F = D \rightarrow R$

２．３　仕様記述の手順

仕様記述は，概ね次のような手順で行われる．

(1) 生成規則を BNF 記法で記述する．
(2) 意味に関する言語の規定を摘出する．
(3) (2)の規定をチェックするのに必要な属性を抽出して，インフォーマルな定義を与える．
(4) (3)で抽出された属性のドメインを，上述のドメイン構成法を用いてフォーマルに定義する．
(5) 文法記号に属性を付与する．
(6) 各生成規則対応に以下の記述を行う．
　(a) 属性評価規則
　(b) 生成規則適用条件(ただしここは特定の構文解析法が前提となる)
　(c) 文脈条件
　(d) エラー回復法
　(e) 中間語変換仕様

３．構文・意味解析への属性文法手法の適用

３．１　仕様の条件

構文・意味解析の仕様を記述する際に，記述の自由度を高めるには，構文解析や意味解析(属性評価)の方法に制約を加えないことが望ましい．しかし，実用的なコンパイラとするには，その効率や実現のし易さが重要であり，そうした観点から，解析方法などに一定の制約を加え，それに合わせて仕様を記述してゆくことが必要な場合もある．

我々は，この立場に立って，解析方法に次のような制約を加えた．

(1) 構文木を作らずに１パスで構文解析と同時に意味解析(属性評価)を行えること

(2) 構文解析は下向き法で行えること

(1)はコンパイル効率を高めることを意図し，(2)はコンパイラを人手で作り易くすることを意図している．(2)は，(1)を満足させるには現存のパーサ・ジェネレータに頼るわけにいかず，従って，当面は人手で解析譜を作成せざるをえないという現実的要請からも来ている．上の２つの制約を別の表現で置き換えれば，「仕様がLL(1)文法に基づくＬ型属性文法となるように記述を行う」ということになる．この前提がＣに対して妥当なものであるかどうかは，実際に記述をしてみるまでは判らない．しかし，性能の要求される実用的なコンパイラを作るためには，できるだけ厳しい条件から出発して条件に合わなくなったところで制約を緩和する，といった試行錯誤的なアプローチもある程度までは必要と思われる．

３．２　LL(1)文法の拡張

構文解析と同時に属性評価を行う場合，属性評価の結果(属性値)を構文解析に反映させることが可能である．こうした考え方を定式化して，Watt[7]は属性文法における生成規則分割(rule splitting)と属性駆動構文解析(attribute-directed parsing)という概念を提案している．これは，適用すべき生成規則の決定に，属性を関与させようとするものである．生成規則分割の方法を，下向き構文解析に適した相続型生成規則分割と，上向き構文解析に適した合成型生成規則分割に分類しているが，そのいずれにせよ，Wattの方法は，生成規則分割を生成規則中の文法記号の属性に関して成立する条件にのみ基づいて行っていて，入力記号(先読み記号)に関しては考慮していない．しかし，実際にLL(1)文法に従った下向き構文解析あるいはLR(1)文法に従った上向き構文解析を前提とする場合には，入力記号の先読みをしているので，その属性も含めた形でWattの方法を拡張することができる．ここから，以下の拡張LL(1)文法の定義を得る(LR(1)文法の拡張も同様にできるが，ここでは直接扱わないので省略する)．

［定義］ 拡張LL(1)文法

文法Ｇにおいて，$N \rightarrow w_1$と$N \rightarrow w_2$を共通の左辺記号を持つ任意の生成規則とするとき，次の条件のいずれかが成立すれば，Ｇは拡張LL(1)文法であるという．

(1) $D = Director(N, w_1) \cap Director(N, w_2) = \phi$

(2) $\forall a \in D$ に対して，$\exists p_1, p_2 : p_1(inh(N), syn(a)) \Longrightarrow not\ p_2(inh(N), syn(a))$

ここで，Director は中田[8]の定義に従う．inh(Ｘ) は(非終端)記号Ｘの相続属性の適当な組，syn(Ｘ) は(非終端または終端)記号Ｘの合成属性の適当な組である．また，$p_i(a_1,\ldots,a_n)$は属性 $a_1,\ldots,a_n$ に関して成立する制約条件(プレディケート)である(i = 1, 2)．

現実にある構文解析法を適用しようとして，そのままでは適用できない場合に，意味情報を用いることがあるが，上の定義はそれを LL(1) 文法による下向き構文解析に即して定式化したものである．この定義に従って，先述の仕様記述の方針を，「仕様が拡張 LL(1) 文法に基づくＬ型属性文法となるように記述を行う」と改める．

３．３　Ｃにおける仕様記述の実際

ここでは，関数定義とラベルに関する記述を取り上げ，上の方針がどのように具体化されているかを見る．なお，Ｃの属性文法記述については，別に文献[9]がある．

（１）関数定義に関する記述

Ｃの関数定義(function definition)の生成規則は，次のようになっている．

```
function_definition ::= type_specifier  function_declarator  function_body …… ①
                      | function_declarator  function_body                  …… ②
```

ここで，type_specifier は関数の結果の型，function_declarator は関数名とパラメタ名，funct

ion_body はパラメタの型と関数本体を，それぞれ規定する．type_specifier が省略されたとき(②)，結果の型は int 型と解釈される．①と②に対する Director を，それぞれ，D_1，D_2とすると，D_1とD_2は共通要素として識別子を持つ．従って，現在の入力記号が識別子の場合，①を適用すべきか②を適用すべきか判断できない．しかし，同じ識別子ではあっても，①ではそれは(ユーザ定義の)型名として作用し，②では関数名として作用する．このことは，その識別子がすでに宣言されているか，いないかということと同義である(正確には型名として宣言されているか全く宣言されていないかといわなければならないが，そのような意味的な整合性の検査はこれよりあとの問題であり，適用生成規則の選択に関してはこの程度の粗い場合分けで十分である)．

以上を基にして，①と②の選択条件を書き下してみると次のようになる．where節が選択条件である．ここでは，LL(1)構文解析法を前提として，現在の入力記号をlookaheadという名前で明示的に導入してある．

```
function_definition ↓ENVIN ↑ENVOUT ::=
    type_specifier ↓ENVIN ↑ENV ↑TYPE
    function_declarator ↓ENV ↓TYPE ↑ENVOUT
    function_body ↓ENVOUT
  where lookahead∈D1-{id} or (lookahead=id and ENVIN(lookahead.NAME) is defined)

function_definition ↓ENVIN ↑ENVOUT ::=
    function_declarator ↓ENVIN ↓int ↑ENVOUT
    function_body ↓ENVOUT
  where lookahead∈D2-{id} or (lookahead=id and ENVIN(lookahead.NAME) is undefined)
```

ここで，ENVINはこの関数定義の直前の環境属性，ENVOUTはその直後の環境属性を表す．環境属性はシン ボルテ-ブルを抽象化したもので，NAME→MODE即ち名前の集合NAMEから宣言によって名前に付与される性質(型，記憶域クラス等)の集合MODEへの関数空間をドメインとしてもつ．2つのwhere節で表現されている条件がお互いに排反することは明らかである．

(2)ラベル定義に関する記述

構文・意味解析仕様を記述する上で，エラ-検出が適切に行われるようにすることも重要である．このことと，先述の方針とは必ずしもかみ合わないことが多い．ここでは，ラベルに関する記述例をとり上げて，この問題を具体的に見てゆく．

Cでは，Pascalのラベル宣言に相当するものはない．ラベルの有効範囲は，ラベル定義を含む関数の全域である(ラベル定義がある複文の中にあっても，複文は有効範囲を区切る「壁」の役目をもたない)．ラベルはgoto文でのみ参照される．ここでは，次の２つの文脈条件をチェックする．

(1) 同じラベルを同じ関数の中で２回以上定義していないこと．

(2) 参照されるラベルは同じ関数内で定義されていること．

以下，２つの記述例を示し，その特徴と利害得失を検討する．ただし，ラベルに関係しないところの属性評価記述は全て省略してある．

[記述１]

①２回のパスで属性評価を行う(Ｌ型属性文法の条件に反する)．

②１回目のパスでは「縫い糸」式に定義ラベルを収集し，あわせて，文脈条件(1)をチェックする．

③２回目のパスでは収集した定義ラベル全体を下方に伝播(相続)し，文脈条件(2)をチェックする．

④ラベルに関する属性のドメインはベキ集合ドメインset of NAMEとする．

```
function_statement ::=                    ┌ パス１ ┐ ┌パス２┐
```

```
    "{" declaration_list  statement_list ↓{} ↑DEF_LAB ↓DEF_LAB  "}"

statement_list ↓DEF_LAB_IN ↑DEF_LAB_OUT ↓ALL_DEF_LAB  ::=
    statement ↓DEF_LAB_IN ↑DEF_LAB ↓ALL_DEF_LAB
    statement_list ↓DEF_LAB ↑DEF_LAB_OUT ↓ALL_DEF_LAB
  | statement ↓DEF_LAB_IN ↑DEF_LAB_OUT ↓ALL_DEF_LAB

statement ↓DEF_LAB_IN ↑DEF_LAB_OUT ↓ALL_DEF_LAB ::=
    identifier ↑NAME  ":"  statement ↓DEF_LAB ↑DEF_LAB_OUT ↓ALL_DEF_LAB
  condition NAME∉DEF_LAB_IN {"$NAME is redefined"}
  evaluate  DEF_LAB = DEF_LAB_IN ∪ {NAME} | DEF_LAB_IN

statement ↓DEF_LAB_IN ↑DEF_LAB_IN ↓ALL_DEF_LAB ::=
    "goto"  identifier ↑NAME  ";"
  condition NAME∈ALL_DEF_LAB {"$NAME is undefined"}
```

［記述２］

①１回のパスで「縫い糸」式に定義ラベルを収集しながら，文脈条件(1)，(2)をチェックする．

②ラベルに関係する属性のドメインは，関数空間ドメイン NAME → set of {defined, referred} とする．

```
function_statement ::=
    "{" declaration_list  statement_list ↓{} ↑ALL_LAB  "}"
  condition if ALL_LAB(NAME)∋referred then ALL_LAB(NAME)∋defined
            {"$NAME is undefined"}

statement_list ↓LAB_IN ↑LAB_OUT  ::=
    statement ↓LAB_IN ↑LAB
    statement_list ↓LAB ↑LAB_OUT
  | statement ↓LAB_IN ↑LAB_OUT

statement ↓LAB_IN ↑LAB_OUT ::=
    identifier ↑NAME  ":"  statement ↓LAB ↑LAB_OUT
  condition LAB_IN(NAME) = ⊥ or LAB_IN(NAME)∌defined {"$NAME is redefined"}
  evaluate  LAB = set_defined_mark(LABIN, NAME) | LABIN

statement ↓LAB_IN ↑set_referred_mark(LAB_IN, NAME) ::=
    "goto"  identifier ↑NAME  ";"
```

［記述１］は，文脈条件(1)，(2)に対するエラーがその発生箇所で遅れなしに検出できる利点をもっているが，２回のパスを必要とし，Ｌ型属性文法の条件に反するという欠点をもつ．［記述２］は，パスが１回ですみ，Ｌ型属性文法の条件にも合致するが，文脈条件(2)に対するエラーの検出が遅れる．しかし，この問題に関しては，Ｌ型条件を満たすことと，エラー検出を適正化することとは，両立しないから，結局いずれか一方を選ぶほかない．我々は，先述の方針に則り，［記述２］を採用した．

一般にＬ型にすると，１パス化できる，実現がし易い，ということのほかに，次の利点がある．

(1) 属性を大域化し易い(さらに条件を強めて強Ｌ型(strongly L-attributed)　にすれば，全て

の属性が大域化可能である)．

(2) 考え易い．

一方，次のような欠点もある．

(3) 記述量が増える．

(4) 属性の種類が増える，あるいは，個々の属性が担うべき情報が増える(ドメインの構造が複雑になる)傾向がある．

(3)に関しては，右正則属性文法(Regular Right-Part AG)[10]などの提案が最近なされているが，新しい記述法とそれに基づく属性評価系の構成法の研究が必要となろう．

４．　コード生成への属性文法手法の適用

木構造中間語に対するコード生成の仕様を属性文法に基づいて記述する．中間語の各ターミナルノードには，構文解析，意味解析の際に計算された属性がすでに与えられている．これを固有属性という．　属性文法によるコード生成仕様の記述においては，最終的に合成属性としての生成コードを得るように，伝播属性を決定してゆかなければならない．

ここでは，Ｃ言語の式について，各伝播属性の内容，属性評価方式について考察する．

４．１　木構造中間語と固有属性

Ｃ言語の式における木構造中間語の構成を図４．１に示す．

ここで，expは式，b_opは二項演算子，u_opは単項演算子，primaryは変数，定数，文字列の一次式を示す．

各演算子には固有属性として式の結果のデータ型(TYPE)が与えられる．一次式(praimary)には，固有属性として，式のデータ型(TYPE)とその割りつけ情報(LOCATION)が与えられる．

これらのドメインの定義は以下のとおりである．

```
exp ::= "?:" exp exp exp
exp ::= b_op exp exp
exp ::= u_op exp
exp ::= "call" exp exp_list
exp ::= primary
```

図4.1 木構造中間語の構成

```
TYPE = int[SIZE×SIGN] + real[PRECISION] + pointer + array + struct[INTEGER]
SIZE = {char,int,short,long}
SIGN = {signed,unsigned}
PRECISION = {float,double}

LOCATION = const[VALUE] + global[ID] + local[OFFSET] + parameter[OFFSET]
         + register[REG_NO] + l_value[LOCATION]
VALUE = int[INTEGER] + real[REAL]
OFFSET = INTEGER
```

ここで，TYPE属性は，コード生成に必要最小限のデータを表現するため，構文･意味解析部で持つ型情報に比べて簡略化を行っている．

LOCATION属性は，宣言情報の割りつけ結果を反映する．この中で，l_value[LOCATION]というのは，変数の実体に対してその左辺値を示すものである．

４．２　属性評価規則

ここでは，図４．２のプログラム例に対して，コード生成に必要な属性評価規則を与えて，本方式によるコード生成方式について説明する．

```
int      a,b;
main ( )
{        int    c,d;
         a = b ? c : d ;
}
```

図4.2 プログラム例

図４．２の式に対応する木構造中間語を図４．３に示す．

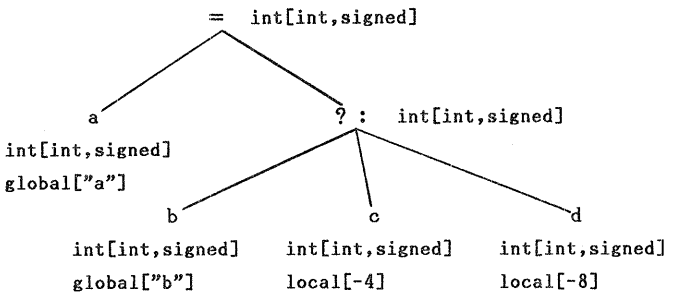

図4.3 図4.2の式に対する木構造中間語

コードを直接生成するためには，以下の情報が必要である．これらを伝播属性として，属性評価規則を組み立ててゆく．

(1) ラベル情報(LABEL_NO)

?:，&&，||演算子においては，式のコード生成のためにラベルとブランチの生成が必要である．そのため，ラベル生成のためのラベル番号の情報を伝播させる．

LABEL_NO = INTEGER

(2) レジスタ情報(REG_STAT)

各ノードにおける使用可能なレジスタの情報．

REG_STAT = REG_NO -> BOOLEAN

(3) ノード割りつけ情報(DESTINATION, RESULT)

各ノードの式の結果の値の割りつけ場所を示す情報(RESULT)．

式の評価においては，オペランドを特殊なレジスタに置かなければならない場合もあるので，合成属性RESULTの他に相続属性DESTINATIONによって，オペランドの割りつけ場所を指定する必要がある．

RESULT = no_result + result[EFFECTIVE_ADDRESS]

DESTINATION = no_destination + destination[EFFECTIVE_ADDRESS]

(4) 文脈情報(CONTEXT)

各ノードの式の値の使用形態を示す情報．以下の４つの場合がある．

(a) 式の値を使用しないもの(no_context)

(b) 式の値を左辺値として使用するもの(lvalue_context)

(c) 式の値そのものを使用するもの(value_context)

(d) 式の値を真偽値として分岐のために用いるもの(flow_context)

CONTEXT = no_context + value_context + lvalue_context + flow_context[LABEL,LABEL]

LABEL = no_label + label[ID]

flow_contextの２つのLABEL は，真の時のジャンプ先，偽の時のジャンプ先を示す．分岐しない時はno_labelを指定する．

上記の属性のうち，LABEL_NO，REG_STATについては式の評価前，評価後の属性値が必要であり，相続属性，合成属性の２つの属性値がノード毎に必要である．また，DESTINATION，CONTEXT は相続属性として，RESULTは合成属性として伝播される．

最終的に生成されるコードの属性ドメインは，命令あるいはラベルの列であり，次に示すとおりである．

```
CODE = (label[ID] + instruction[INSTRUCTION])...
```

４．３　コード生成の実際

上記の例に従って，68000用のコードを生成する例を図４．４に示す．

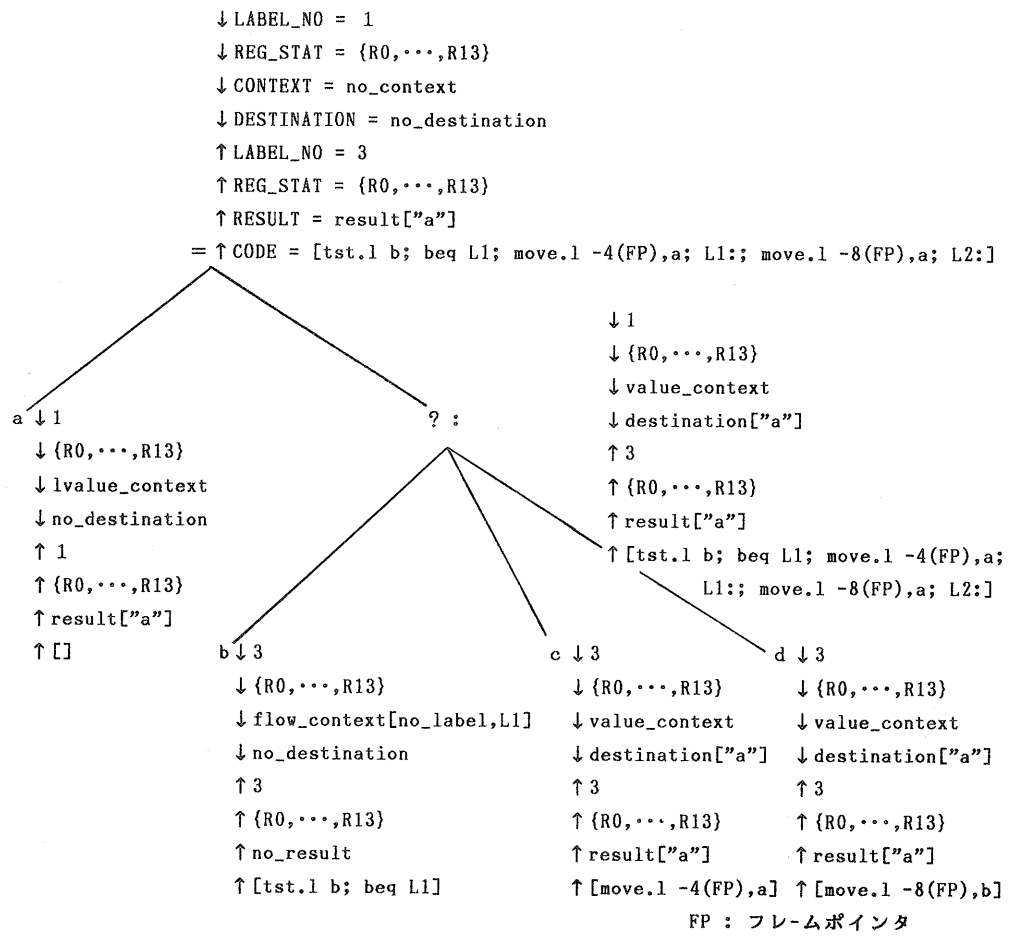

図4.4 68000 用コード生成

４．４　現実のコンパイラへの適用

(1) パスの分割

現実のコンパイラに適用するためには，属性の依存関係により木のトラバースのパスが何回必要であるかを見極めなければならない．

今回の例では基本的に１回のパスでコード生成が可能であるが，レジスタ割りつけ，最適化(

ローカルなレベル）を行なうためには属性の詳細化が必要であり，依存関係が増えることによりパス数を増やす必要が出てくる．しかしながら，ターゲットマシン毎にパス数が変わるとすると移植性を大いに損ねることになる．

本コンパイラにおいては，上記の最適化と移植性を考慮してパス数をプログラム全体で２パス，式の中ではさらに１パストラバースするようにして多くのターゲットに適用可能なようにしている．

(2) 属性の内部表現の最適化

属性文法の記述では，各ノードに属性を持たせる形式になっているが，実際の実現上では，このままでは効率が悪い．

属性の依存関係から最適な内部表現を見極めねばならない．本稿で紹介した各属性に対する最適化について以下に考察する．

LABEL_NO：

属性は木のトラバースの方向に沿って順送りで伝達される．したがって，原則的に１つのグローバル変数で実現可能である．

しかしながら，条件演算子等においては，コード生成のパスでラベル生成のためにLABEL_NOを参照する．したがって，このようにラベルを生成するノードではLABEL_NOの値を保持しておかなければならない．

REG_STAT：

本属性は，通常は木のトラバースの方向に沿って順送りに伝達されるが，処理の分岐がある場合は，相続属性の伝播が分岐する．したがって，一段の木のネスト毎に深まるスタックとして実現しなければならない．

CODE：

本属性は，後のパスでの参照は一切なく，また親ノードでの評価関数は，子ノードにおけるCODE属性のコンカテネーションのみである．したがってコードは生成する順に出力してしまってかまわない．

５．おわりに

属性文法手法の，構文・意味解析およびコード生成への適用について，主として仕様記述の面から，Ｃコンパイラに即して述べた．

(1) 属性文法にエラー回復等の記述機能を組み込んだ，コンパイラの仕様記述法を示した．

(2) 生成規則左辺記号の相続属性と先読み記号の合成属性によって，LL(1)下向き構文解析における多義性の解消ができることに着目して，これを拡張LL(1)文法として定式化した．

(3) Ｃの構文・意味解析仕様は，拡張LL(1)文法に基づくＬ型属性文法で記述できるが，これを具体例で説明した．

(4) コード生成に属性文法を適用する場合の，属性の与え方，属性ドメインの定義の仕方，属性評価規則の組み立て方について，具体例を用いて示した．

属性文法の適用によって，コンパイラの高品質化，言語仕様の変更に対する即応性の向上，リターゲッタビリティの向上などの効果が期待される．

参考文献

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

[2] Koskimies, K., et al. : Compiler Construction Using Attribute Grammars, Proc. SIGPLAN '82 Symposium on Compiler Construction, 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] Kernighan, B. W. and Ritchie, D. M. : The C Programming Language, Prentice-Hall, 1978

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

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

[7] Watt, D, A. : Rule Splitting and Attribute-Directed Parsing, in Jones, N. D. (ed.) : Semantics-Directed Compiler Generation, LNCS 94, Springer, 1980

[8] 中田育男 : コンパイラ, 産業図書, 1981

[9] 神野俊昭 : 拡張属性文法によるＣ言語の意味記述の一方法, 情報処理学会プログラミング言語研究会資料1, 1985.6

[10]Jullig, R. and DeRemer, F. : Regular Right-Part Attribute Grammars, SIGPLAN Notices, 19-6, 1984