拡張1パス型属性文法によるコンパイラ生成系の実現

中川裕之[†1] 金谷英信[†2] 星野秀之[†3]
中田育男[†4] 山下義行[†4]

文脈自由文法に意味規則を付加した属性文法はコンパイラの仕様記述に向いており，コンパイラ生成系への応用が広く研究されている．また，属性文法の記述性や生成されるコンパイラの性能を改善する研究も続けられている．本論文は，分かりやすい記述法から，効率の良いコンパイラを生成する1つの方式として，構文規則を正規右辺文法とする属性文法の記述から，1パスコンパイラを生成する方式を提案している．正規右辺文法の場合，右辺に不定回出現しうる構文要素に対応する意味規則を独立に書くことが難しいという問題がある．そこで，構文規則内の文法記号に直接付随した形で意味規則を記述する方法を提案する．1パス型の属性文法（L属性文法）では，属性値は，プログラムの上から下，左から右，の順番で決まっていかなければならないが，意味規則を素直に表現しようとすると，後で決まる属性値を先に使うような形になることが多い．本論文では，このようなL属性の範囲を越えるものも素直に記述できるようにして，それに対しては，バックパッチする処理系を自動的に生成する方式を提案する．従来の1パスコンパイラを生成する属性文法処理系では，実際には，if文などの分岐先の番地を後で解決するために，アセンブラのパスを設けているものが多いが，本方式によれば，そのパスも不要になる．

A Generator of One-Pass Compilers Based on Regular Right Part Attribute Grammars

HIROYUKI NAKAGAWA,[†1] HIDENOBU KANAYA,[†2] HIDEYUKI HOSHINO,[†3]
IKUO NAKATA [†4] and YOSHIYUKI YAMASHITA[†4]

An attribute grammar, which is a context-free grammar with semantic rules, is suited for the description of a compiler, and the researches on compiler generator systems based on attribute grammars have been done widely. The readability of the description of a grammar and the efficiency of the generated compilers are the main topics of these researches. In this paper, we propose a method of generating an efficient one-pass compiler from a regular right part attribute grammar, which has consice and easy to read semantic rules. One of the main problems of describing semantic rules in a regular right part grammar is the difficulty of independent description of a syntactic rule and the corresponding semantic rule because a right part regular expression may represents indefinite number of occurrences of syntactic elements. Therefore, we propose to describe semantic rules by appending them directory to the corresponding syntactic symbols. In a one-pass attribute (L-attributed) grammar, basically, the right-to-left attribute dependence is not allowed, that is, the attribute value which has not been yet computed cannot be used. However, it is often desirable that we can express semantic rules directly by using such forms. In this paper, we propose a description method that allows the rules with right-to-left dependency, and a generation method of evaluators that process such rules by the backpatch method of hand-written compilers. Most of the coventional one-pass compilers generated by attribute grammar processors have a back-end assembler pass in order to solve the problem of branch targets, for example, to decide the start address of the "else" part of an "if" statement. Our method dispenses with such a pass.

†1 キヤノンソフトウェア(株)
Canon Software inc.
†2 ソニー(株)
Sony Corporation
†3 サンデン(株)
Sanden Corporation
†4 筑波大学電子・情報工学系
Institute of Information Science and Electronics, The University of Tsukuba

1. はじめに

文脈自由文法に意味規則を付与した属性文法(attribute grammar)[9]は，構文規則と意味規則を宣言的に記述するものでコンパイラの仕様記述に向いており，また，比較的効率の良い属性評価器（attribute evaluator)が得られることよりコンパイラ生成系への

応用が広く研究されてきた[5]．さらに，属性文法の記述性や，生成されるコンパイラの性能を改善する研究も続けられている．その1つとして，生成規則中に属性評価規則を埋め込み，従来の記述よりも意味規則や文脈条件（context condition）を簡潔に表現できる記法が考案された[20]．また，構文規則表現を簡潔に分かりやすくするために，生成規則の右辺に正規表現（regular expression）を許した正規右辺文法（regular right part grammar）を用いることが考えられており，この正規表現に対応した形で属性評価規則にも正規表現を許した正規右辺属性文法（regular right part attribute grammar）が提案された[6)~8),11),19)]．しかし，その場合，右辺には不定個数現れ得る構文要素が記述できるが，それに対応する意味規則を，構文規則と切り離して独立に記述することが難しいという問題がある．独立に記述するために，属性値の伝播の典型的なパターンに合わせた記述法を導入したり[7]，構文規則における正規表現と対応させた意味規則の正規表現を導入したり[11),19)]されているが，これらは分かりやすさの面から必ずしも成功していないし，パターンにも制限がある．それに対し，生成規則中に属性評価規則を埋め込む方法[6)]は構文と意味との対応が分かりやすい．

1パスで構文解析と同時に属性評価可能な属性文法としては，LL文法を基底文法としたL属性文法やLR文法を基底文法としたLR属性文法が用いられることが多い．LR属性文法はL属性文法に比べ，基底文法を「正規表現を許す文法」に広げた場合，構文解析が複雑になるとともに，もともと継承属性の扱いが複雑であったものが，さらに新たな議論を展開する必要が生じてくる．

基底文法を「正規表現を許す文法」に広げた場合，反復記号の中の属性生起とその外の属性との対応づけが大きな問題となり，これまでもいろいろな提案がなされてきたが，満足なものは得られていない．そこで，我々は，反復記号の中での合成属性はその中でしか参照できないこととすることで，複雑な対応関係を記述することなく意味規則を簡潔に表現できるようにした．その結果，処理系としては，解析した時に属性の処理も行わなければならず，それは，生成規則の右辺の終りまで解析して構文を確定するLR構文解析の手法より，右辺の先頭から順次確定していくLL構文解析の考え方が適している．また，LL文法のLR文法に比べての弱点は，基底文法を「正規表現を許す文法」に広げることと属性値主導構文解析を取り入れることでかなり緩和されており，通常のコンパイラ開発にはほとんど問題がない．そこで，我々は構文解析はLL構文解析を，意味規則はL属性文法を基本とし，これを拡張することとした．

しかし，例えばPascalの変数宣言で

```
var x, y, z : integer
```

の場合，変数xに対してその名前と型の情報(integer)を合わせたものを属性値とする属性評価規則は，xとintegerの間に変数名が不定個数現れ得るので，L属性として表現するのは難しい[6)]．また，

```
    GOTO L;
    ......
L : ......
    GOTO L;
```

の目的コード生成の問題は，L属性文法だけでなく一般の属性文法でも表現が難しい．それは，目的コードにおけるラベルLの値はラベル宣言（位置）までの目的コード（最初のGOTO Lのコードも含む）を生成しないと決まらないからである．この問題は，目的コードをアセンブラ言語のコードとし，コード生成後にアセンブラのパスでLの値を決めることによって解決できるが，それではパスがもう1つ増えてしまう．さらにif文のif C then S1 else S2の場合は，

```
    LOAD C
    JMPF L1
      S1
    JMP L2
L1: S2
L2:
```

の形のコードを生成することになるので，このL1，L2を生成するために，例えば，呼ばれるたびに異なる値を返す（副作用をもった）関数が必要になる．

本論文では，これらの場合でもL属性にとらわれることなく意味規則をより素直に，より宣言的に，より簡潔な属性文法記述となる表現方法を提案する．その記述法としては構文と意味との対応を分かりやすくするため構文要素に属性や意味規則を付記する方式[6)]をとり，さらに，構文要素に付記された属性に条件をつけることによって，構文解析を制御する，すなわち属性主導型構文解析（attribute-directed parsing）をすることも可能にする．

また，この表現方法から，従来，手書きのコンパイラで用いられていた手法であるバックパッチ処理(backpatch)を属性評価器生成系で自動生成する方法を提案する．これにより，従来の1パス型属性文法では，後処理のアセンブラパスが必要であったものを，本当の1パス処理を可能とし，また，if文の属性を表現する場合でも，新しいラベルを生成する手続き的な関

数を考えることなく記述を行えるようになる．

本論文では，まず，第2章で，拡張1パス型属性文法の表記法とそれが表す意味を定義し，第3章で提案する属性評価方法を述べる．第4章ではこの属性文法に基づいたコンパイラ生成系EAGLEを用いてPascal S[3]コンパイラを生成した結果とその評価を行う．

2. 拡張1パス型属性文法の記述方法

本論文で提案する記述法は，以下の3層構造からなっている．

1. 構文・意味規則の記述
2. プリミティブを用いた意味関数の記述
3. 目的言語であるCによる意味関数の記述

属性文法による記述は，一般には上の1.と3.からなるが，我々は，L属性に反する記述を許すため，2.の記述を必要とする．2.では論理プログラムに類似した宣言的記述を行い，その記述から自動的にバックパッチを行うプログラムが生成される．

まずは，構文規則の正規表現の記法について述べ，以下，構文・意味規則の記述方法，プリミティブを用いた意味関数の記述方法について述べる．

2.1 構文規則の記法

構文規則の記述法は正規右辺文法を基本とし，それに，記述性向上のための以下のような略記法を付け加えたものである．ただし，この定義は再帰的であり，記号Aは略記法を含めた任意の正規表現を表すものとする．

- 文字列そのものを表す終端記号は，その文字列を一重引用符で囲む．
- 任意記号［A］は，通常の正規表現の（A | ε）を意味する．
- 反復記号{A}は，通常の正規表現のA*，すなわちAの0回以上の繰り返しを意味する．
- 反復記号{A","}のように最後に二重引用符で囲った終端記号がある場合は，A{',' A}，すなわちその終端記号で区切った反復を表す．

2.2 構文・意味規則の記述方法

構文・意味規則は，構文規則の中に意味規則を埋め込んだ形で記述する．構文規則に付随する意味規則は各構文要素近くに記述したほうが分かりやすい．特に正規右辺文法ではプログラム上でその要素に対応するものが複数回出現する場合や，全然出現しない場合もあるが，構文要素に隣接した記述は，その要素の1回の出現に対応するものであるとすることで，対応が明確になる．反復記号中での属性生起の有効範囲は，合成属性については，それを含む一番内側の反復記号の中であり，反復記号の外の属性と対応づけられるのは継承属性としてだけである．たとえば，

idlist（↓type）：{*ident*（↑name）
§TBL（↓name, ↓type）","}．

では，↑nameは反復記号の中だけで使われ，↓typeは同名の属性生起が反復記号中になく，外にあるので，外のそれと対応する．

構文・意味規則の記述に先だって，まず，非終端記号と終端記号（例えば，以下の例の*ident*）および，それらの属性の名前と型の宣言，意味関数や意味関数で使われる記号表などの表の型宣言があり，その後に構文・意味規則が続く形である．構文中に現れる非終端記号/終端記号の有する属性はそれぞれの非終端記号/終端記号の直後に括弧で囲って列挙する（いわゆるPositional Notationを採用している）．意味関数の呼び出しは，構文規則中の任意の場所に，記号"§"に続けて記述する．例えば規則の右辺に

（*a*（↑as）§A（↓as）| *b*（↑bs）§B（↓bs））

という記述が現れたとする．ここで，*a, b*は非終端記号または終端記号であり，"(↑as)"，"(↑bs)"はそれら非終端記号/終端記号の有する属性を表している（この場合は共に一個である）．上矢印↑はその属性が合成属性(synthesized attribute)であること，下矢印↓はその属性が継承属性（inherited attribute）であることを表す．"§A（↓as)"は意味関数§Aを属性値asを引数として評価することを表す．引数に付けられた下矢印は入力引数であることを表し，上矢印は出力引数であることを表す．この意味関数は*a*の直後に記述されており，これは*a*がソースプログラム中に現れたときにその意味関数を評価することを表している．"§B（↓bs)"も同様である．

以下，変数宣言，論理式での目的コードの生成，GOTO文での目的コード生成を例に，本論文で提案する構文・意味規則の記述方法を述べる．さらに，2.2.4項では，構文要素に属性条件を付加した形の構文規則の記述方法について述べる．

2.2.1 変数宣言の記述例

1章で述べた，Pascalにおける変数宣言を例に，その表記法を示す．

```
var x, y, z: integer
```

これに対応する標準的な意味操作は，各変数(の出現)に対して，その名前に型情報integerを組み合わせて記号表に追加（登録）することである．そこで，変数の綴りを表す属性name，型を表す属性typeを用いて，これを記述すると下記のようになる．

decl：'var' *idlist*（↓type）':' *typ*（↑type)．(1)

idlist (↓type) : {*ident* (↑name)
§TBL (↓name, ↓type) ","}. (2)

(1)では *decl* は'var' *idlist*':' *typ* という形をしており，*typ* の合成属性 type を *idlist* の継承属性とすることを示している．すなわち，属性が右から左に渡される形になっている．それを1パスで処理することを考えると'var' を読んだところで *idlist* に type を渡さなければならないが，それはまだ得られていないから，どうするかが問題になる．しかし，記述する人は，そのような1パスコンパイラの動作の制約を意識しないでもよいというのが我々のねらいである．(2)では，*idlist* は *ident* を","で区切って並べたものであり，そのすべての *ident* (の出現)について，その名前(name)と型 (type として渡されるもの) を一緒に記号表に書き込むことを表している．§TBL は，各変数の名前 name と変数の型 type のエントリを記号表に登録する意味関数を表している．

2.2.2 論理式での目的コード生成の記述例

論理式は，論理変数または関係式を要素とし，それらに and, or, not などの論理演算子を作用させたものからなる．ここで，式 A or B が与えられたとき，A が真であることを確認したならば，B を評価しなくてもこの式の値は真であると決めることができる．また，式 C and D が与えられたとき，C が偽であれば D を評価しなくてもこの式の値は偽である．論理式をすべて評価しなければならないかどうかは，プログラム言語の仕様によって規定されるのだが，もし言語の仕様において論理式の一部を評価しないままでおくことを許しているならば，論理式の計算は必要な部分だけを評価してその値を決定すれば良いので，コンパイラは論理式の評価を最適化できる[14]．そのような，IF 文の目的コードを生成するための属性文法の記述の一部は，例えば，次のようになる．

if-stat : 'IF' *condition* (↓nextpl, ↓nextp2) 'THEN'
§CODE (↑nextpl) *statement*
§CODE (↑nextp2).

§CODE は，その呼び出しの場所に対応する目的コードのアドレスを返す意味関数である．従って，↑nextpl の値は *statement* の目的コードの開始番地である．*condition* に渡される属性値の1番目は *condition* の値が真である時の飛び先であり，2番目は偽である時の飛び先である．いくつかの or でつながれた論理式は次のように表現できる．

condition (↓truep, ↓falsep) :
{*bTerm* (↓truep, ↓nextp) 'or'
§CODE (↑nextp)}
bTerm (↓truep, ↓falsep). (3)

LL(1)文法としては，*condition* の構文規則は
condition : *bTerm* {'or' *bTerm*}.
と書かざるをえないが，*bTerm* が偽であるときの飛び先は最後に出現する *bTerm* とそれ以外では異なるからそれを表現するために，構文は，
condition : {*bTerm* 'or'} *bTerm*.
としている．これが構文規則としては
condition : {*bTerm* "or"}.
と同じであることは，処理系が容易に判定することはできるから，構文解析は LL(1) の枠組の中で可能である．この記述の残りは以下のようになる．ここで，§INST は機械命令を順次格納する意味関数である．

bTerm (↓truep, ↓falsep) :
{*bFactor* (↓nextp, ↓falsep) 'and'
§CODE (↑nextp)}
bFactor (↓truep, ↓falsep).

bFactor (↓truep, ↓falsep) :
'not' *bFactor* (↓falsep, ↓truep)
| '(' *condition* (↓truep, ↓falsep) ')'
| *ident* (↑address)
§INST (LOAD, ↓address)
§INST (JMPF, ↓falsep)
§INST (JMP, ↓truep).

2.2.3 GOTO 文での目的コード生成の記述例

前の2項の例は，属性値が1つの構文規則の中で右から左に渡されるものであり，L 属性の範囲を越えるものであった．さらに，1章で述べた GOTO 文の例は，アセンブラのパスを仮定しない限り，属性文法で表現するのは難しく，同じく L 属性文法の範囲を越える．通常考えられる意味規則は，ラベル定義文 LAB : に対しては，その出現位置に対応する機械番地をラベル LAB の値とし，GOTO LAB に対しては，LAB の機械番地へのジャンプ命令をその目的コードとするものである．それを次のように表現できるようにした．この記述では，*label* と *goto* の出現順序については何の規定もしていないので，これも L 属性の範囲を越えるものである．

label : *ident* (↑name) ':' §CODE (↑address) §TBL (↓name, ↓address). (4)
goto : 'GOTO' *ident* (↑name) §TBLGET (↓name, ↑address) §INST (JMP, ↓address). (5)

ここで，§TBLGETはラベル名に対応する機械番地を得る意味関数である．

2.2.4 属性値主導による構文規則の記述例

構文要素に付記された属性に条件をつけることで，属性値による構文解析の制御が可能になる．すなわち，属性値主導構文解析である．例えば，ステートメントが手続き呼び出しか代入文かの文法記述を行う場合，素直に書くと以下のような記述となる．

statement：*call* | *assign*.
call：*ident* ':=' *exp*.
assign：*ident params*.

この記述はLL(1)文法ではないので，LL(1)構文解析ができるようにするためには，構文規則を書き換えなくてはならないが，属性値によって区別ができるような記法を導入することで，構文規則を書き換えなくてもよいようにした．

属性値によって区別するための属性条件は，終端記号(terminal symbol)にのみ付随して書くことができる．属性評価規則ではなく，属性条件であることを明示するため，特に“＜＞”で括って記述する．属性条件は，“＜↑属性名＝＝属性値＞”の形で記述する．上の例は，この記法を用いて記述すると下記のようになる．

statement：*call* | *assign*. (6)
assign：*ident*＜↑mode＝＝var＞(↑name) ':=' *exp*. (7)
call：*ident*＜↑mode＝＝proc＞(↑name) *param*. (8)

2.3 プリミティブを使用した意味関数の記述方法

本論文でここまで述べてきた記述法による記述からコンパイラを生成し，その記述の中でL属性の範囲を越えるもののうち，比較的簡単に実現でき，1パスにする効果が期待できる範囲において，通常の手書きのコンパイラで行っているバックパッチの方法を適用する，というのが我々の方針である．しかし，記述する人にそのことを意識させないで実現するためには，意味関数の記述に適当な制限をつける必要がある．本システムではmemberとappendという2つのプリミティブを用意し，このプリミティブを用いて記述された意味関数については，必要ならばバックパッチのプログラムを自動的に生成することにした．

コンパイラの意味処理で基本的なものは記号表の操作とコード生成である．記号表に対する操作としては，記号表に新しい項目を書き込む操作と，ある項目が記号表に入っているかどうかを探す操作とがある．これらの操作を計算機内部での表現とは独立に表現するためには，データ（項目）を関係として扱うことが最も適している[4]．そこで，本システムにおける意味関数の記述は，Prologの表記法を基本とし，コンパイラの意味処理に適合した形で記述できるようにした．意味関数は，一般に次の形で記述する．

B：－$A_1 \cdots A_n$.

Bは意味関数，$A_1 \cdots A_n$はそれぞれプリミティブの呼び出しを表しており，$A_1, \cdots, A_n$の順に評価される．各A_iはtrueかfalseの値を返す．falseの値が返されるかA_nの評価が終了したとき，Bの評価が終了する．

プリミティブとして記号表に新しい項目を書き込む操作を行うappendと，ある項目が記号表にあるかどうかを判定する操作を行うmemberの2つを用意する．appendは記号表に書き込めた時trueを返す．memberはtrue/falseを返すだけでなく，記号表から値を取り出すのにも使われる．また，例えば，二重定義の検査を記述できるようにするために，memberの結果にはnot演算子が適用できるものとする．

appendとmemberはともに2つのパラメータをもち，appendは，第1パラメータには，テーブル名称を記述し，第2パラメータには，登録エントリの要素名称を[]内に','で区切って列挙する．また，memberは，第1パラメータには検索キーの名称，または，出力パラメータの名称を[]内に','で区切って列挙し，第2パラメータにはテーブル名称を記述する．ここで，テーブル名称（記号表やコード格納領域の名称）とテーブルを構成している要素名称，ならびにその型は，終端記号や非終端記号の宣言と同様に構文・意味規則の記述前に宣言する．パラメータは，この宣言されたテーブル名称とその構成要素名称を用いて記述を行う．

これを使って，各ラベルの名前nameとその位置に対応する目的コードのアドレスaddressのエントリを記号表symbolに登録する，(4)の意味関数§TBLを記述すると次のようになる．なお，nameとaddressの型，および，これらが記号表symbolの構成要素であることは，事前に宣言されている．

```
§TBL (name, address) : －
        not (member ([name], symbol)),
    append (symbol, [name, address]).
                                    (9)
```

意味関数の記述では，出力パラメータ（変数）に↑を付加し，入力パラメータ（定数）には矢印を付加しないで表現することにした．

§TBL(name, address)は意味関数名が§TBLで，入力パラメータがnameとaddressであることを表

している．not (member ([name], symbol)) は記号表 symbol の中に入力パラメータ name をもつエントリが存在しないこと，つまり，二重定義のチェックを行っている．そして二重定義でないときに append により入力パラメータ name と address を記号表 symbol に追加登録している．

コード生成については append を用いてコード格納領域に順次コードの格納を行うのが通常である．(5)の意味関数§INST を記述すると次のようになる．

```
§INST (opcode, operand) :-
  append (program, [opcode, operand]).
```
(10)

member の使用例として，入力パラメータ name をキーに記号表 symbol の検索を行い，合致したエントリの構成要素の1つである address を得る(5)の意味関数§TBLGET の記述は，次のようになる．

```
§TBLGET (label, ↑address) :-
  member ([label, ↑address], symbol).
```
(11)

3. 拡張1パス型属性文法の評価方法

我々は，提案する拡張1パス型属性文法に基づいてコンパイラ生成系 EAGLE (Extended Attribute Grammar based on L-attributE) の開発を行った．EAGLE により生成された属性評価器は，LL 構文解析と同時に解析木を作らずに1パスで L 属性文法を基本とした属性評価を行う．さらに，2章で述べたように L 属性の範囲を越えた記述も許している．そこで，ここではその属性評価方法を述べる．

3.1 属性文法の記述から，属性が右から左に渡されることが分かる場合の評価方法

Pascal の変数宣言を例 (2.2.1項) として説明する．構文規則(1)では，*idlist* (↓type) の ↓type が右側の *typ* (↑type) から渡されていることから，(2)の§TBL (↓name, ↓type) の ↓type はバックパッチされる属性であることが分かる．そこで，§TBL の処理では，(9)の記述に従って，記号表 symbol に変数名の登録を行い，型を格納すべき場所をバックパッチリストに登録する．それ以後，型が現れるまでの変数については，先に登録したバックパッチリストにつないでいく．そして，型が現れた時点で，リストをたどってバックパッチを行うような処理方式にする．

3.2 属性の依存/出現の関係に応じた属性評価方法の拡張

(4)と(5)のように，記述には直接現れてない属性の依存関係があるのに，その出現順序が規定できない場合の属性評価方法を考える．この場合，ラベルが実際のプログラムの中で依存関係と同じ順序で出現するならば問題なく属性評価が可能であるが，その逆の順序で出現した場合は，バックパッチが必要になる．その方法を(4)と(5)の例で説明する．(4)の§TBL (↓name, ↓address) では，↓address が入力パラメータであり，その値は (5) の§TBLGET (↓name, ↑address)の出力パラメータとして決まる．以下，入力パラメータと出力パラメータ，それぞれの評価方法を述べる．

3.2.1 出力パラメータの評価方法

プリミティブに出力パラメータをもつ(11)の member を評価するとき，GOTO 文の出現位置に応じて，出力パラメータ ↑address は次の3種類の状態になることを考えなくてはならない．

1. 既にラベル定義がなされており，ラベル名 LAB は記号表にある．そのため機械番地である ↑address の値は確定している．
2. GOTO LAB のコード生成において，ラベル名 LAB が初めて出現し，LAB は記号表に登録されていない．そこで，記号表にエントリを登録するが，↑address の値は未定状態である．
3. GOTO LAB のコード生成において，ラベル名 LAB は初めての出現ではなく，記号表にあるが，↑address の値が決定の状態ではない．

そこで上述の各状態に対処する機能が member に求められる．member による出力パラメータの処理方法をまとめると下記のようになる．

1. 1の場合には，取り出す項目の値を，そのまま出力パラメータの値とする．
2. 2の場合は，検索項目を記号表に書き込み，取り出す項目は未確定の状態にしてバックパッチリストに登録する．
3. 3の場合には，バックパッチリストにつなぐ．

3.2.2 入力パラメータの評価方法

プリミティブ append を用いた2種類の意味関数を例に，入力パラメータの評価方法を検討する．append は入力パラメータを記号表やコード領域など，指定された領域に順次登録を行う関数である．最初に，(5)から呼ばれる(10)を考える．GOTO 文の出現位置により入力パラメータ operand の状態はそれぞれ異なる．ここでは append を評価するときには次の3種類の状態に応じた評価方法を考慮しなければならない．

1. 機械番地 operand の値が確定している場合：

programの格納領域にopcodeとoperandを格納する．

2. 初めてGOTO LABに出会った場合：このときはoperandの値は未定である．そこで，opcodeをprogramの格納領域に格納し，operandを格納する場所をバックパッチリストにつなぐ．
3. 再びGOTO LABに出会った場合：このときはoperandの値は未定である．そこで，上記2と同様にopcodeをprogramの格納領域に格納し，operandを格納する場所をバックパッチリストにつなぐ．

次に，(4)から呼ばれる(9)を考える．ラベル定義の出現位置により入力パラメータaddressの状態はそれぞれ異なる．そこでappendを評価するときには次の2種類の状態に応じた機能を考慮しなければならない．

1. 記号表に，初めて登録するとき(ラベルが，まだ登録されていない場合)：ラベル名labelと機械番地addressを記号表へ登録する．
2. 記号表に，ラベルが既に登録されているが，その値は定義されていないときに実際の定義が現れた場合：リストを辿ってバックパッチを行う．
3. 記号表に，ラベルが既に登録されていて，その値が定義されているときは，エラーの処理を行う．

以上のような考察から，appendにおける入力パラメータに関しては次の2点に留意した処理方法を行う．

- 値が決まっていない入力パラメータはバックパッチリストにつなぐ．
- 入力パラメータの値が確定していて，入力パラメータがバックパッチリストにつながれている場合は，入力パラメータを表に書き込むと同時にその値でバックパッチを行う．

3.3 論理式の目的コード生成における属性評価の方法

(3)の属性評価を1パスで行うためには，↓nextpや↓falsepが通常の継承属性であれば，*bTerm*の構文解析をするときに，それらを渡さなければならない．'or'でつながれた*bTerm*の中の最後の*bTerm*に渡すべきものは，↓truepと↓falsepであり，それ以外の*bTerm*に渡すべきものは↓truepと↓nextpである．それが*bTerm*の最後の出現であるかどうかは次にくるものがorでないかどうか確認しないと分からないから，それを区別して渡すのは不可能である．しかし，本論文で提案した記述方法を用いた場合には，これらは後で決まる属性であることが分かっている．我々の処理方式ではバックパッチによって値を渡すものであるから，実際には，*bTerm*の解析をするとき値を渡す必要はなく，バックパッチするものであるという情報だけ渡しておけば良い．そこで，まず*bTerm*には，↓truepと↓nextpを(バックパッチするものとして)渡しておき，*bTerm*の最後の出現であることが分かった時点で，その*bTerm*に渡してあった↓nextpを↓falsepに変えればよい．実際に，その値がバックパッチされるのは，そのさらに後である．このように，一般に構文が{A "c"}の形でAに渡される属性が，バックパッチされるものであることが分かっているときは，最後のAだけ意味規則が違っていても，以上の処理方法により1パスで処理可能である．

3.4 属性値主導構文解析の処理方法

2章で述べたような属性条件がある場合は，属性値主導構文解析を行う．この場合は，属性値を何らかの方法で得て，それによって構文規則の選択をすれば良いので，特に問題はない．属性条件が終端記号に付加されている場合，EAGLEは，属性値を得るための意味関数§get_attribを呼び出すコードを生成することにしているので，その関数を定義しておく必要がある．例えば，(7)に従って属性値主導構文解析を行う場合，意味関数§get_attribの記述は次のようにすれば良い．

```
§get_attrib (name, ↑mode) :-
        member ([name, ↑mode], symbol).
```

(7)に従って構文解析をする場合は，先頭に識別子*ident*が現れたとき，この意味関数§get_attribで記号表の検索を行い，合成属性modeの属性値を得て，属性modeの値により*assign*か*call*のいずれに構文解析を進めていくかが決定される．

4. システムの評価と考察

拡張1パス型属性文法に基づくコンパイラ生成系EAGLEを実現し，EAGLEを用いてPascal S[1),2)]コンパイラの自動生成を行った．ここでは，Pascal Sコンパイラの記述および，生成されたコンパイラについて評価考察を行う．

表1 Pascal Sコンパイラの記述量の比較
Table 1 Comparison of attribute grammar for Pascal S compiler.

	EAGLEによる記述	Rie[17)]による記述
記述量（行）	730	808
記述量（語）	1090	2572

```
vardecl : VAR var_decl_list SEMI
        {%transfer I_env,S_env;}
| /* empty */
        {var_dec.S_env = var_dec.I_env ;}
;
var_decl_list : var_decl_list SEMI var_decl
        {%thread I_env S_env;}
| var_decl
        {%transfer I_env,S_env;}
;
var_decl : ident_list COLON type
        {%transfer I_env ;
          var_decl.S_env =
            EnterVar(ident_list.S_nlist,type.S_type,var_decl.I_env);
        };
ident_list : ident_list COMMA ident
        {ident_list[2].I_env = ident + list[1].I_env ;
         ident_list[1].S_nlist =
                MakeNameList(ident.S_name,ident_list[2].S_nlist);
        }
| ident
        { ident_list.S_nlist =
                MakeNameList(ident.S_name,Null_Name);
        }
        ;
```

図1 Rieによる変数宣言の記述
Fig. 1 Description for variable declarations by Rie.

4.1 Pascal Sコンパイラの記述例とその評価

Pascal Sは，Pascal[1]のサブセットである．Pascal SにはGOTO文がない．そこで，本論文で提案したバックパッチ処理の確認のため，簡単なGOTO文とGOTO文のためのラベル定義を追加した．

拡張1パス型属性文法と正規表現を許さない表記法の属性文法[17]でPascal Sコンパイラの記述を行った[10]．そこで，両者を記述量および記述の可読性において比較する（**表1**）．ただし，後者の属性文法ではコード生成を行っていないが，そのまま比較を行った．実際には，後者は，この数字より大きくなるはずである．

拡張1パス型属性文法による記述では，インデントを多用して読解性を向上させているため行数は正規表現を許さない表記法の属性文法に比べ約0.90倍であ

```
vardecl(↓ scop ) :
        VAR { var_body(↓ scop ) '','' }
        §writesize(↓ scop ).
var_body(↓ scop ) :
        { ident(↑ name )  Enter_var(
          ↓ scop, ↓ name, ↓ typ, ↓ param) '','' }
        COLON type(↓ scop, ↑ typ, ↑ param) S_COLON.
Enter_var(↓ scop, ↓ name, ↓ typ, ↓ param) :
        §writesize(↓ scop )
        §enterv(↓ scop, ↓ name,
                        ↓ typ, ↓ param).
```

図2 EAGLEによる変数宣言の記述
Fig. 2 Description for variable declarations by EAGLE.

表 2 Pascal S コンパイラの自動生成と手書きの比較
Table 2 Comparison of generated with hand-written compiler in Pascal S.

	EAGLE で生成されたコンパイラ	手書きのコンパイラ
コンパイラソース行	5253 行	3734 行
259 行コンパイル時間	0.68 秒	0.45 秒

まり差がないが，語数で比較すると約 0.42 倍となっている．これらの値から，拡張 1 パス型属性文法による記述量は，正規表現を許さない表記法の属性文法の記述量に比較して大幅に少なくなることが分かる．特に意味規則を生成規則に埋め込んだ記述をするため，正規右辺文法により生成規則が減ることは意味規則の記述量を減らすことにつながっている．量的に少なくなることは，記述性と可読性の両面でよい性質である．

次に記述の可読性の点から比較する．上述のように記述量が軽減することは記述の読みやすさにつながるが，さらに，正規表現による直観的な記述で読みやすさが増す．例として，Pascal S 言語における変数宣言を比較する．**図 1** と**図 2** の 2 つの記述を比較して，拡張 1 パス型属性文法による記述の方が見通しが良く変数宣言全体の構文を読み取りやすいことが分かる．また，拡張 1 パス型属性文法による記述では，構文規則に意味規則を埋め込んだことで，コンパイラの構造が直観的に分かりやすいものとなっている．

4.2 生成された Pascal S コンパイラに関する評価

評価に用いた Pascal S プログラムの例は参考文献[21]にあるシフトと加算を用いて乗算を行うプログラム multiply である．EAGLE の評価は，EAGLE が生成した Pascal S コンパイラと手書きの Pascal S コンパイラ[3]を，コンパイラの大きさとコンパイル時間の比較により行った．手書きのコンパイラは Pascal で記述されていたため，C 言語プログラムに変換した．また，測定の誤差を減らすため，ソースプログラム中の手続き multiply を 10 個複写してソースプログラムの行数を増やした．測定の結果を**表 2** に示す．

コンパイラの大きさに関しての考察を述べる．手書きのコンパイラよりも EAGLE で生成したコンパイラの方が約 1.4 倍の大きさとなっている．これは，生成系によって生成されたものであることが最大の要因であるが，他に，拡張 1 パス型属性文法による Pascal S の記述が，手書きのコンパイラをもとにした記述であるため，拡張 1 パス型属性文法にそぐわない部分もあり，記述が冗長となったのが原因と考えられる．今後の課題として，拡張 1 パス型属性文法に最も適した記述を行うことが挙げられる．

コンパイル時間は，EAGLE で生成したコンパイラの方が手書きのコンパイラに比べて約 1.5 倍になっている．今回は実現の早さを目指したため，まだ効率向上の余地は多々ある．例えば，構文解析については，通常の再帰的下向き構文解析法に従って演算子の優先順位のレベル数だけの手続きを生成しているため，再帰的呼び出しのネストが深くなり効率が落ちている．そこで今後，再帰的下向き構文解析に演算子の優先順位を宣言する表現[15]を取り入れることにより，手続き呼出の回数の減少を目指し実行時間の効率化を図りたい．

最後に，本論文で提案したバックパッチの効果について述べる．EAGLE で生成したコンパイラは，変数宣言，IF 文，GOTO 文など，各所で，バックパッチを適用できるよう，その処理方法を一般化している．一方，手書きのコンパイラでは，IF 文など制御文におけるコード生成に適用を限定して，簡潔な処理方法でバックパッチを行っている．

今回，このバックパッチを取り入れたことで，手書きのコンパイラとほぼ同じ 1 パスのコンパイラを実現することができた．これがなければ，さらにアセンブラのパスを追加せねばならず，さらに効率が悪くなったはずであるが，それがどの程度であるかは，今回は検討していない．

5. おわりに

正規右辺文法の中に簡潔な表現の意味規則を埋め込み，分かりやすい記述で効率の良い 1 パスコンパイラを生成する生成系の構文・意味規則の記述方法と，その意味規則の評価方法を提案した．その拡張 1 パス型属性文法を用いて Pascal の部分集合である Pascal S を記述した結果，従来の 1 パス型属性文法での記述に比べ語数で約 60%減となり，記述量の軽減が図れた．このように，記述量が軽減することは記述の読みやすさにつながるが，さらに正規表現による直観的な記述で可読性が増した．また，基本的には右側の意味規則で値が定まるような属性値を左側で参照してはならないが，本方式ではそのような参照関係を気にせず書けるようにし，可読性を高めた．さらに処理時間については，提案する手法に基づいて Pascal S コンパイラを生成し，その評価を行った結果から，手書きのコンパイラと比較して約 1.5 倍であることが分かった．今後

の課題として，より効率の良いコンパイラを生成するため，生成系での最適化の方法や，演算子の優先順位を宣言する表現を取り入れた再帰的下向き構文解析法の検討を行っている．

謝辞 本論文の内容に対して貴重なご意見を頂いた，東京工業大学佐々政孝教授に感謝する．

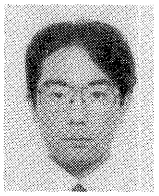
参考文献

1) Barron, D. W. (ed.): *Pascal—The Language and Its Implementation,* John Wiley (1981).
2) Berry, R. E.: Experience with the Pascal P Compiler, *Softw. Parct. Exper.,* Vol. 8, No. 5, pp. 617-627 (1978).
3) Berry, R. E.: *Programing Language Translation,* Ellis Horwood (1981).
4) Codd, E. F.: A Relational Model for Large Shared Data Bases, *Comm. ACM,* Vol. 13, No. 6, pp. 377-387 (1970).
5) Deransart, P., Jourdan, M. and Lorho, B.: Attribute Grammars Definitions, Systems and Bibliography, *LNCS,* Vol. 323, Springer-Verlag (1988).
6) Elsworth, E. F. and Parkers, M. A. B.: Automated Compiler Construction based on Top-down Syntax Analysis and Attribute Evaluation, *SIGPLAN NOTICES,* Vol. 25, No. 8, pp. 37-42 (1990).
7) Jullig, R. K. and DeRemer, F.: Regular Right-Part Attribute Grammars, *SIGPLAN NOTICES,* Vol. 19, No. 6, pp. 171-178 (1984).
8) Kastens, U., Hutt, B. and Zimmermann, E.: GAG: A Practical Compiler Generator, *LNCS,* Vol. 141, Springer (1982).
9) Knuth, D. E.: Semantics of Context-Free Language, *Mathematical Systems Theory,* Vol. 2, No. 2 (1968), pp. 127-145. 訂正, ibid, Vol. 5, No. 1 (1971), pp. 95-96.
10) 荻原一隆：ECLR属性文法に基づくインクリメンタルな構文・意味解析の研究，筑波大学理工学研究科，修士論文 (1993).
11) 丁 亜希，渡辺美樹，中田育男，佐々政孝：正規右辺属性文法と1パス再帰降下属性評価器の生成，情報処理学会論文誌，Vol. 30, No. 2, pp. 204-212 (1989).
12) 星野秀之：属性評価器生成系EAGLEによるPascal Sコンパイラの実現，筑波大学第三学群情報学類卒業論文 (1993).
13) 中川裕之，金谷英信，中田育男：拡張1パス型属性文法の提案，第45回情報処理学会全国大会論文集，分冊5, pp. 65-66 (1992).
14) 中田育男：コンパイラ，産業図書 (1981).
15) 中田育男，山下義行：再帰的下向き構文解析における演算子順位構文解析，情報処理学会論文誌，Vol. 34, No. 2, pp. 239-245 (1993).
16) 佐々政孝：属性文法，コンピュータソフトウェア，Vol. 3, No. 4, pp. 73-91 (1986).
17) Sassa, M. et al.: Rie Introduction and User's Manual, Tech. Memo PL-14, Inst. of Inf. Sc., Univ. of Tsukuba (1988).
18) 佐々政孝：プログラミング言語処理系，岩波書店 (1989).
19) 渡辺美樹：正規右辺属性文法とそのLR型属性評価器の自動生成法，筑波大学理工学研究科，修士論文 (1988).
20) Watt, D. A. and Madsen, O. L.: Extended Attribute Grammars, *Comp. J.,* Vol. 14, No. 2, pp. 142-153 (1983).
21) Wirth, N.: *Algorithms + Data structure = Programs,* Pretice-Hall (1976).
22) Wirth, N.: *Programing in Modula-2,* Springer (1982).

(平成5年7月30日受付)
(平成7年2月10日採録)

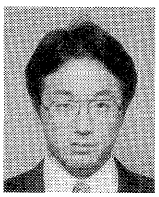
中川 裕之（正会員）

1958年生．1981年広島工業大学工学部電子工学科卒業．1981年キヤノンソフトウェア（株）入社．1991年筑波大学大学院修士課程入学．1993年修了．現在，キヤノンソフトウェア（株）システム研究所勤務．言語処理系，ソフトウェア工学などに興味を持っている．ACM会員．



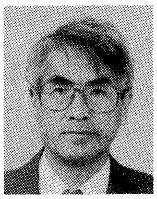
金谷 英信（正会員）

1968年生．1991年筑波大学第3学群情報学類卒業．1993年同大学院修士課程修了．現在ソニー（株）勤務．コンパイラ記述言語などに興味をもつ．



星野 秀之

1970年生．1993年筑波大学第3学群情報学類情報工学科卒業．現在サンデン(株)勤務．プログラミング言語処理系に興味を持っている．



中田　育男（正会員）

1935年生．1958年東京大学理学部数学科卒業．1960年同大学院修士課程修了．1960～79年（株）日立製作所中央研究所，同システム開発研究所勤務．1979年4月より筑波大学電子・情報工学系教授．理学博士．プログラム言語，言語処理系，ソフトウェア工学などに興味を持っている．著書「コンパイラ」（産業図書），「基礎FORTRAN」（岩波書店）．ソフトウェア科学会，電子情報通信学会，ACM，IEEE各会員．

山下　義行（正会員）

1959年生．1982年大阪大学理学部物理学科卒業，日立マイクロコンピュータ・エンジニアリング（株）入社．1986年退社．1987年筑波大学大学院博士課程電子・情報工学系入学．1989年退学，東京大学大型計算機センター助手．1992年より筑波大学電子・情報工学系講師．プログラミング言語，コンピュータ・グラフィックスの研究に従事．日本ソフトウェア科学会会員．