

拡張1パス型属性文法に基づくコンパイラ生成系の実現

大木 康幸^{*1} 平見 知久^{*2} 中川 裕之^{*3} 山下 義行^{*4} 中田 育男^{*4}

^{*1}(株)日立製作所 ^{*2}筑波大学 理工学研究科

^{*3}キャノンソフトウェア(株) ^{*4}筑波大学 電子・情報工学系

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行なわれている。本稿では、簡潔な記述より効率の良いコンパイラを自動生成することを目指した拡張1パス型属性文法を提案し、この文法を元に開発したコンパイラ生成系 EAGLE を紹介する。拡張1パス型属性文法は、簡潔な文法記述のために正規右辺文法を採用している。また、文法記述からバックパッチ処理を自動生成することにより、1パスで評価可能か否かを考慮せずに文法を記述できる。さらに、本文法では、属性値主導構文解析を可能にするための記法も用意している。本稿では、拡張1パス型属性文法にしたがった属性評価の方法、および、与えられた文法が属性値主導の LL(1) 構文解析により解析可能か否かを判定する方法も述べる。コンパイラ生成系 EAGLE は、従来の属性文法に基づくコンパイラ生成系に比べ、0.78 倍の記述量で Pascal S コンパイラを記述できた。生成されたコンパイラの実行時間は手書きに比べ 1.6 倍となり、自動生成のコンパイラとしては良い結果が得られた。

An Implementation of a Compiler Generator based on Extended One-pass Attribute Grammar

Yasuyuki OOKI^{*1}, Tomohisa HIRAMI^{*2}, Hiroyuki NAKAGAWA^{*3}, Yoshiyuki YAMASHITA^{*4}, Ikuo NAKATA^{*4}

^{*1}Hitachi, Ltd. ^{*2}Master Program of Science and Engineering, Univ. of Tsukuba

^{*3}Canon Software, Inc. ^{*4}Institute of Information Sciences and Electronics, Univ. of Tsukuba

Attribute grammar is suitable for a specification of compilers, and has been applied to compiler generators. In this paper, we propose an extended one-pass attribute grammar that aims to generate an efficient compiler from a terse description; and explain our compiler generator EAGLE that is based on this grammar. The grammar adopts a regular right-part grammar to make a grammar description terse. To generate a back-patch process from the grammar description, we don't need to consider a one-pass attribute evaluation. The grammar, furthermore, prepares a notation for attribute-directed parsing. Also in this paper, we introduce the evaluation method using back-patching, and describe how to check which grammar can be analyzed by attribute-directed LL(1) parsing. The description of Pascal S compiler in EAGLE was 0.78 times of that in the compiler generator based on conventional attribute grammar. The compilation time of the compiler generated by EAGLE was 1.6 times of a hand-written compiler, and it was relatively short in automatically generated compilers.

1 はじめに

文脈自由文法に意味規則を付加した属性文法 (attribute grammar)[1] は、コンパイラの仕様記述に向いており、比較的効率の良い属性評価器が得られるため、コンパイラ生成系への応用が広く研究されてきた。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。本稿では、簡潔な文法記述より、効率の良いコンパイラの生成が可能な属性文法として、拡張 1 パス型属性文法を提案する。また、この文法を元に開発したコンパイラ生成系 EAGLE(Extended Attribute Grammar based on L-attributE)を紹介する。

本稿で述べる拡張 1 パス型属性文法は、以下の 3 つの特徴を持っている。

文法記述を簡潔にする方法としては、言語の意味を記述する属性評価規則 (意味規則ともいう) を、言語の構文要素となる非終端 / 終端記号に付記させる記法が提案されている [2]。また、構文要素の繰り返しなどを正規表現を用いて簡潔に記述できる正規右辺文法を、属性文法の構文記述に用いる正規右辺属性文法も提案されている [3][4][5][6]。この場合、構文要素の繰り返しの記述に、どのように意味規則を付記するかが問題となる。本稿で提案する文法は、正規右辺文法を基にし、各構文要素のすぐ隣に意味規則を記述する方法をとる。これにより、構文と意味の対応が分かり易く、直観的な文法記述ができる。

効率の良いコンパイラを生成する方法としては、1 パスで属性評価が可能な L 属性文法を用いることが多い。しかし、実際のコンパイラ記述では、L 属性文法として記述しにくい規則がよくある。たとえば、

```
goto L
...
L: ...
```

といった goto 文とラベル定義の規則の場合、goto 文で使用するラベル L のアドレスを 1 パスで決定することは難しく、L 属性文法では記述できない。このような規則は、値が決定した後にその値を埋め込むというバックパッチ (back-patch) の技法を用いると、解決できることが多い。そこで我々は、文法記述から自動的にバックパッチ処理を生成することにより、1 パスで評価可能かを考慮せずに文法を記述できるようにした。

本文法では、さらに、プログラムの意味情報を表す属性値を構文解析の制御に用いる、すなわち属性値主導構文解析 (attribute-directed parsing)[7] を可能にしている。このため、構文規則中に属性値の条件を記述できるようにした。これにより、従来単純に構文解析ができなかった文法も記述が可能である。また、意味情報を表す属性値の条件を構文規則中に記述できるため、構文と意

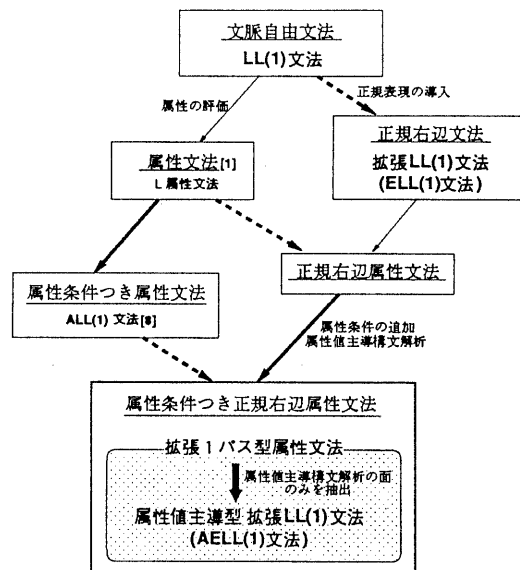


図 1: 本稿で用いる文法の関係図

味が複雑に関係する規則も、より柔軟に記述できる。なお本文法では、LL(1) 文法に属性値主導の概念を採り入れている。

本稿では、与えられた文法が属性値主導の LL(1) 構文解析により解析可能かを判定する方法を定式化する。定式化は、LL(1) 文法であるか否かの判定に用いる First, Follow 集合を、属性値の条件を考慮したものに拡張することにより行う。このような文法を、本稿では、属性値主導型 拡張 LL(1) 文法 (Attribute-directed Extended LL(1) grammar: AELL(1) 文法) と呼んでいる。本稿で用いる各文法の関係図を図 1 に示す。

我々が開発したコンパイラ生成系 EAGLE は、従来の属性文法を用いて記述されるコード生成機能なしの Pascal S 意味チェッカに比べ、0.78 倍の記述量でコード生成機能ありの Pascal S コンパイラを記述できる。生成されたコンパイラの実行時間は、手書きのものに比べ 1.6 倍程度の長さになった。これは、EAGLE の改良により、より短くすることも可能である。

2 拡張 1 パス型属性文法の特徴

我々が提案する拡張 1 パス型属性文法は、簡潔な文法記述から効率の良いコンパイラを生成する、という目的を達成するため、以下の特徴を持っている。

2.1 正規右辺属性文法

構文規則を簡潔に記述するために、正規右辺文法を属性文法の構文記述に用いる正規右辺属性文法を採用している。正規右辺属性文法での意味規則の記述方法はいくつか提案されている [3][4][5][6] が、我々は意味規則を構文規則の中に埋め込む記述方法を採用した。これにより、複数回現れうる構文要素に対応した意味規則を、直観的に記述できる。

なお本文法は、正規右辺文法上での LL(1) 文法である拡張 LL(1) 文法 (Extended LL(1) grammar: ELL(1) 文法) に基づいている。ELL(1) 文法から生成される構文解析器は、元の文法との対応が良く、改良が容易である。LL 文法の弱点であるクラスの狭さは、後に述べる属性値主導の構文規則の記法と共に用いることにより、補うことができると思う。

以下に、本文法の記述方法を述べる。

構文規則の記述方法: 本文法では、以下に示す記法を用いて構文規則を記述する。

- $[A]$... 省略可。 $\varepsilon \mid A$
- $\{A\}$... 反復。 $\varepsilon \mid A \mid AA \mid \dots$
- $\{A/s\}$... 区切り付き反復。 $A \mid AsA \mid AsAsA \mid \dots$

たとえば、`var x,y,z : integer;` のような Pascal の変数宣言の構文規則は、次のように記述する。

```
decl: 'var' idlist ':' typ .
      idlist: { idlist // ' , ' } .
```

意味規則の記述方法: 構文中に現れる非終端 / 終端記号の有する属性は、その名前の直後に括弧で囲って列挙し、また、合成属性 / 継承属性の種別を矢印 $\uparrow/\downarrow$ で区別する。意味規則は記号 $\$$ に続けて関数呼び出し形式で構文規則中に記述する¹。先述の Pascal 変数宣言の構文意味規則は次のようになる。

```
decl: 'var' idlist( $\uparrow$ type) ':' typ( $\uparrow$ type) .
      idlist( $\downarrow$ type):
      { idlist( $\downarrow$ name) $TBLSET( $\downarrow$ name, $\downarrow$ type) // ' , ' } .
```

ここでは、型名 typ の合成属性 $type$ と、識別子 $ident$ の合成属性 $name$ を、意味関数 $\$TBLSET$ により、記号表へ登録している。

矢印 $\uparrow/\downarrow$ の後には属性名だけではなく、定数値や、属性を使用した式を記述することができる。次の記述は、型名が `integer` なら 0 を、`real` なら 1 を合成属性として返す構文意味記述である。

```
typ( $\uparrow$ 0): 'integer' .
typ( $\uparrow$ 1): 'real' .
```

¹ 本文法に基づく生成系 EAGLE では、便宜上の記法として、`C++` で記述された意味規則を `%{...}` で括って記述できる。

2.2 拡張した L 属性文法

拡張 1 パス型属性文法では、1 パスで属性評価が可能な L 属性文法を基に、一部 L 属性的でない属性の評価も可能にしている。L 属性文法では、ある属性の値は、必ずそれより左側で値が決定する属性に依存する。しかし、本文法では、バックパッチの技法を用いて評価可能なものについては、右側で値が決定する属性についても、その属性の使用を許すことにした。これにより、L 属性的でない文法も文法の書き直しをせずに、簡潔に記述できる。

たとえば、前節で示した Pascal 変数宣言の構文意味規則の記述では、`decl` の規則で使われている属性 `type` が L 属性文法に反しているが、記述可能である。

より複雑な例として、1 章で示した `goto` 文とラベルについての構文意味規則の記述がある。この規則は、通常の属性文法でも記述が難しく、L 属性文法の範囲を超えている。本文法ではこれを次のように記述できる。

```
label: ident( $\uparrow$ name) ':' $GETADDR( $\uparrow$ address)
      $TBLSET( $\downarrow$ name, $\downarrow$ address) .
goto: 'goto' ident( $\uparrow$ name) $TBLGET( $\downarrow$ name, $\uparrow$ address)
      $INST('JMP', $\downarrow$ address) .
```

この規則では、ラベルと `goto` の出現順序を規定していない。単純にラベルとそのアドレスをテーブルに登録・参照しているだけで、記述として分かり易い。

この場合、 $\$TBLSET$ 、 $\$TBLGET$ 、 $\$INST$ といったテーブル操作の意味規則でバックパッチを行えば、この規則全体を評価できる。我々は、テーブル操作の意味規則の記述から、テーブルを介したバックパッチ処理を自動生成することにした。これにより、L 属性文法の制約を気にすることなく、分かり易い意味規則を記述できる。

本文法に基づくコンパイラ生成系 EAGLE では、テーブルへの参照を行なう `member`、追加を行なう `append` というプリミティブを用意し、これを用いてテーブル操作の意味規則 ($\$TBLSET$ など) を記述する。これらのプリミティブからは、テーブルを介したバックパッチ処理が自動生成される (§3.2)。

プリミティブによる意味規則は、Prolog に似せた表記法により、テーブルへの追加 / 参照を行う。次にその構文を示す。上矢印 $\uparrow$ のついたメンバ名は、メンバの取り出しを表している。

意味関数: 関数名 ' $\langle$ ' パラメタ ' $\rangle$ ' ' $\rightarrow$ ' { プリミティブ // ' $\langle$ ' , ' $\rangle$ ' } .

プリミティブ: `member` 関数 | `append` 関数 .

`member` 関数: '`member`' ' $\langle$ ' エントリ ' $\rangle$ ' テーブル名 ' $\rangle$ ' .

`append` 関数: '`append`' ' $\langle$ ' テーブル名 ' $\rangle$ ' , エントリ ' $\rangle$ ' .

エントリ: ' $\langle$ ' { ' $\uparrow$ ' { ' $\uparrow$ ' メンバ名 // ' $\langle$ ' , ' $\rangle$ ' } ' $\rangle$ ' } .

先の文法で用いた意味規則は、次のように記述する。

`symbol` は記号表の名称、`program` は目的コードを格納

するテーブルの名称である。

```
$TBLSET(name,address):-not(member([name],symbol)),
                        append(symbol,[name,address]).
$TBLGET(label,[address]):-member([label,[address],symbol]).
$INST(opcode,operand):-append(program,[opcode,operand]).
```

2.3 属性値主導による構文規則の記述

構文要素に付記された属性に条件 (属性条件) を付けることで、属性値を用いた構文解析の制御、つまり属性値主導構文解析が可能である。属性条件を記述することにより、ELL(1) 構文解析ができなかった構文規則も、書き換えることなく簡潔に記述できる。本文法では、属性条件の記述方法を、その用途毎にいくつか用意している。

終端記号に付随する属性条件: 次の構文規則は、代入文と手続き呼び出しからなる文を表している。

```
statement: call | assign.
assign: ident ':=' expr.
call: ident '(' params ')'.

```

これは LL(1) 文法ではない。しかし、構文解析で *ident* を読み込んだ時に、その種別 (変数名 / 手続き名) を表す合成属性 *kind* を参照できれば、どちらの構文規則かが決められる。このように先読み記号 (終端記号) の合成属性を用いた属性条件は、その終端記号の直後に $\langle \dots \rangle$ で括って次のように記述する。

```
statement: call | assign.
assign: ident(kind=var) ':=' expr.
call: ident(kind=proc) '(' params ')'.

```

構文規則の任意の位置に埋め込める属性条件: 終端記号に付随する属性条件の他にも、 $\langle \dots \rangle$ で括った属性条件を構文規則中の任意の位置に記述できる。これは、先読み記号と関係のない属性条件を記述するものである。次の例は、Pascal の実引数の構文意味規則である。実引数には、値渡し / 参照渡しの区別があるが、ここでは記号表から得た実引数の情報を意味する継承属性 *param* の値により、規則の場合分けをしている。

```
actual-param(lparam):
    (param = value) expr
    | (param = refer) ident.
expr: ident | number.

```

パラメタ位置で指定する属性条件: 最後に、非終端 / 終端記号のパラメタ位置での、属性の受渡しで指定する属性条件について述べる。次の記述は、代入文に対応する (スタックマシンの) 機械語命令を整数 / 実数型で出力する構文意味規則である。ここでは、型を表す継承属性 *type* の値 (0: 整数 / 1: 実数) により、規則の場合分けをしている。

```
assign: ident(!type)(kind=var) ':=' expr set_op(!type).
set_op(!0): $CODE(!"store.int").
set_op(!1): $CODE(!"store.real").

```

この記述は、次の記述と同等である。

```
assign: ident(!type)(kind=var) ':=' expr
    ( (!type = 0) $CODE(!"store.int")
      | (!type = 1) $CODE(!"store.real") ).

```

このように拡張 1 パス型属性文法では、属性条件を用いて意味規則の場合分けも記述できる。従来の属性文法を用いたコンパイラ生成系では、これらの意味規則は C 言語などで記述するため冗長であったが、本文法ではこれを文法記述の一部として簡潔に記述できる。

従来の属性条件の記法 [8][9] では先読み記号を属性条件に記述させていた。たとえば、*ident(kind=var)* という我々の記法による条件は、 $\langle \text{ident.kind} = \text{var} \rangle$ *ident* となる。我々記法では、属性条件を終端記号に付随させることにより、先読み記号の明示的な記述をなくしている。また、終端記号以外にも属性条件を記述可能にすることにより、属性条件を用いて非終端記号や意味規則までも選択できるようにしている。

3 拡張した L 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たコンパイラ生成系 EAGLE を開発した。EAGLE が生成する構文解析器は、ELL 構文解析と同時に L 属性文法を基本とした属性評価を行う。

3.1 右依存的な属性の評価方法

Pascal 変数宣言の記述 (§2.1) のように、構文解析中に値が決定していない属性 (*ident* の属性 *type*) を評価する場合、バックパッチの技法を用いる。まず属性値を「未定義」とし、未定義属性をバックパッチすべきリストに登録する。属性値が決定した場合、その未定義リストを辿りながら値を埋め込んでいく。これが基本的な評価方法である。

EAGLE では、属性のバックパッチ処理を C++ のクラスを用いて実現している。属性の参照・定義を全てこのクラスに任せることができるため、バックパッチを考慮することなく、構文解析器の自動生成を実現できる。また、このクラスは、ユーザが C++ で記述した意味規則の中でも使用可能であり、汎用性が高い。以下に整数型属性についてのバックパッチクラスを示す。

```
class Int {
    List<Int>* bplist; // バックパッチをしなれば
                      // ならない属性のリスト
    int value;         // 実際の属性の値
}

```

```

    int isAssigned;    // 値が定義済みかを示すフラグ
public:
    // オブジェクトの初期設定 (コンストラクタ)
    Int() { isAssigned=0; bplist=空リスト; }
    // 属性値の参照
    operator int() {
        if ( !isAssigned ) { 未定義エラー処理; }
        return value;
    }
    // バックパッチクラス間の代入演算
    Int& operator=(Int& bv) {
        if ( bv.isAssigned )
            *this = bv.value;
        else
            bv.bplist に自分 (this) を繋ぐ;
        return *this;
    }
    // 属性値の定義。バックパッチ処理。
    Int& operator=(int v) {
        value = v;
        isAssigned = 1;
        リストに繋がっている各属性 bv について、
            bv = v; を行う。
        return *this;
    }
}

```

3.2 テーブルを使用したバックパッチ属性の評価方法

EAGLEでは、テーブルを用いたバックパッチ属性の評価を、バックパッチクラス (§3.1) と、プリミティブ (§2.2) から自動生成されるテーブル操作により行う。ここでは、2章の goto 文とラベルの構文意味規則を例にとり、概要を述べる。

プリミティブ member の実現方法: 意味規則 §TBLGET で使用しているプリミティブ member では、記号表 symbol をラベル名 label で検索し、ラベル名が既に登録されていれば、ラベルに対応するアドレス address を返す。

しかし、ラベル名が初めて出現したものであった場合、そのラベルに対応するアドレスは存在しない。この場合、member では新たに [label の値, "未定義"] といったエントリを生成し、あたかもそのラベル名が登録されていたかのようにふるまわせる。結果として address は、「未定義」となり、バックパッチリストに追加される。この属性、および記号表のエントリの要素は、先述のバックパッチクラスの変数により実現されており、変数自身でバックパッチ処理を行っている。

プリミティブ append の実現方法: 先述のように、プリミティブ member では仮想的なエントリを作成する場合がある。したがって、プリミティブ append は、登録に先だってテーブルを検索し、仮想的なエントリが存在しない場合にのみ新たなエントリを生成すればよい。

もし仮想的なエントリが存在していた場合、このエントリのうち、「未定義」の要素について値を定義する。エントリ中の各要素は、バックパッチクラスの変数とし

て実現されてるため、値が定義されれば自動的にバックパッチ処理が行われる。

4 属性値主導構文解析

拡張 1 パス型属性文法のうち、属性値主導構文解析の面に注目し、一般化したものを、属性値主導型拡張 LL(1) 文法 (AELL(1) 文法) と呼ぶことにする。

AELL(1) 文法は、属性値主導の ELL(1) 構文解析が可能な文法である。与えられた文法が AELL(1) 文法である条件、AELL(1) 条件は、First, Follow 集合を用いた ELL(1) 条件を、属性値主導を考慮して拡張したものとなっている。

4.1 条件つき終端記号列

まず初めに、(終端記号の列, 属性条件) の組を考え、これを条件つき終端記号列と呼ぶことにする。属性値主導構文解析では先読み記号と属性条件を同時に扱うので、これらを組にすると定式化が容易となる。次に、条件つき終端記号列について (長さ 1 の) 接続 $conc_1$ を定義する。以降では $conc_1$ の定義を集合にまで拡張して用いることもある。

[定義] 条件つき終端記号列 t_1, t_2 の接続 $conc_1(t_1, t_2)$:

$$conc_1((x, p), (y, q)) = \begin{cases} (x, p) & x \neq \varepsilon \text{ の時} \\ (y, p \wedge q) & x = \varepsilon \text{ の時} \end{cases}$$

4.2 CFirst, CFollow, CDirector 集合

次に ELL(1) 文法で用いる構文規則 e の先頭記号集合 $First(e)$ を条件つき終端記号列を扱うものへと拡張し、これを $CFirst(e)$ と呼ぶことにする。 $CFirst(e)$ は e を導出した時に先頭に来る終端記号と、その時点で満足する属性条件を、条件つき終端記号列の集合として表している。例として次の構文規則を考える。

$$\begin{aligned} A &: \langle a = 1 \rangle B. \\ B &: \langle b = 2 \rangle xy \langle c = 3 \rangle z. \end{aligned}$$

この場合、 $CFirst(A) = \{ (x, a = 1 \wedge b = 2) \}$ となる。条件 $\langle c = 3 \rangle$ は先頭記号 x を導出する際には必要ない条件であるため、 $CFirst(A)$ に含めない。

ここで注意しなければならないのは、先頭記号を導出するまでに現れる属性評価式 (意味規則) も、 $CFirst$ 集合に含めることである。何故なら、属性条件 $\langle \dots \rangle$ 中の属性は、それより左側の属性評価で得られるかもしれないからである。

次の例を考える。構文規則中の §PLUS($\uparrow c, \downarrow b, \downarrow v$) は、 $c = b + v$ という属性評価を行う意味規則である。

$$\begin{aligned}
S(\downarrow a) &: A \ (e_1(\downarrow a) \mid e_2(\downarrow a)) \ x. \\
e_1(\downarrow b) &: \$PLUS(\uparrow c, \downarrow b, \downarrow 1) \ (c=3) \ x. \\
e_2(\downarrow b) &: \$PLUS(\uparrow c, \downarrow b, \downarrow 2) \ (c=3) \ .
\end{aligned}$$

一見、 $CFirst(e_1) = \{x, c=3\}$ である。しかし、非終端記号 A の直後において値が決定している属性は a であり、条件 $\{c=3\}$ を評価できない。このため、 $CFirst$ 集合には、パラメタの受渡し（ここでは $b=a$ ）、意味規則 ($\$PLUS(...)$) も属性条件に含める必要がある。次にこの例の $CFirst$ 集合を示す²。

$$\begin{aligned}
CFirst(e_1) &= \{ (x, b=a \wedge PLUS(\uparrow c, \downarrow b, \downarrow 1) \wedge c=3) \} \\
&= \{ (x, b=a \wedge c=b+1 \wedge c=3) \} \\
&= \{ (x, a=2) \} \\
CFirst(e_2) &= \dots \text{同様} \dots = \{ (e, a=1) \}
\end{aligned}$$

$Follow$ 集合を条件つき終端記号列に拡張した $CFollow$ 集合も同様である。 $CFollow$ 集合の計算方法を表 1,2 に示す。先の文法の $CFollow$ 集合を次に示す。

$$CFollow(e_1) = CFollow(e_2) = \{ (x, true) \}$$

表 1, 2 に示した $CFirst$, $CFollow$ 集合の計算方法は、 $First$, $Follow$ 集合の計算方法をそのまま拡張したものであり、これらと同時に計算することができる。

同様に $Director$ 集合を拡張した $CDirector$ 集合を定義する。 $CFollow$ 集合に $\{e, \dots\}$ という要素がないため、 $CDirector$ 集合にもこのような要素は含まれない。

[定義] 条件つき $Director$ 集合 $CDirector$:

$$CDirector(e) = conc_1(CFirst(e), CFollow(e))$$

先の文法の $CDirector$ 集合を次に示す。

$$\begin{aligned}
CDirector(e_1) &= conc_1(\{ (x, a=2) \}, \{ (x, true) \}) \\
&= \{ (x, a=2) \} \\
CDirector(e_2) &= \dots \text{同様} \dots = \{ (x, a=1) \}
\end{aligned}$$

4.3 AELL(1) 条件

与えられた文法が AELL(1) 文法である条件、AELL(1) 条件を次に示す。

[定義] AELL(1) 条件:

文法中の各構文規則 e において、次が成立する。

1. $e = e_1 \mid e_2$ の場合、
$$(u, p) \in CDirector(e_1) \wedge (u, q) \in CDirector(e_2) \\ \Rightarrow p \wedge q = false$$
2. $e = \{e_1\}$ の場合、
$$(u, p) \in CDirector(e_1) \wedge (u, q) \in CFollow(e) \\ \Rightarrow p \wedge q = false$$

²ここでは説明のために $b = a \wedge c = b+1 \wedge c=3 = a=2$ にまとめているが、AELL(1) 文法か否かの判定 (§4.3) には、この必要はない。

表 1: CFirst 集合の計算方法

構文規則 e	$CFirst(e)$
ϵ	$\{ (\epsilon, true) \}$
$\langle p \rangle$	$\{ (\epsilon, p) \}$
x	$\{ (x, true) \}$, ただし $x \in V_T$
$x(p)$	$\{ (x, p) \}$, ただし $x \in V_T$
$\$R(\text{式}1, \dots, \text{式}n)$	$\{ (\epsilon, R(\text{式}1, \dots, \text{式}n)) \}$, R は意味関数
$A(\text{式}1, \dots, \text{式}n)$ ただし $A \in V_N$	$CFirst(\langle p \rangle e_1 \langle q \rangle)$ $(A(exp_1, \dots, exp_n) : e_1.) \in P$ $p = \bigwedge_{exp_i = \uparrow \dots} (exp_i = \text{式}_i)$ $q = \bigwedge_{exp_i = \downarrow \dots} (exp_i = \text{式}_i)$
$e_1 e_2$	$conc_1(CFirst(e_1), CFirst(e_2))$
$e_1 \mid e_2$	$CFirst(e_1) \cup CFirst(e_2)$
$\{e_1\}$	$\{ (\epsilon, true) \} \cup CFirst(e_1)$

表 2: CFollow 集合の計算方法

構文規則 e	$CFollow$ 集合の計算方法
$e_1 e_2$	$CFollow(e_2) = CFollow(e)$ $CFollow(e_1) = conc_1(CFirst(e_2), CFollow(e_2))$
$e_1 \mid e_2$	$CFollow(e_1) = CFollow(e)$ $CFollow(e_2) = CFollow(e)$
$\{e_1\}$	$CFollow(e_1) = CFollow(e)$
$A(\dots)$	生成規則右辺に現れる全ての A に対し、部分的に $CFollow$ 集合を求め、その集合和を取ったものが、 $CFollow(A)$ となる。 ($A = S$ の時、更に $\{ (\$, true) \}$ と和を取る)

先の文法は、1. で $u = x, p \wedge q = (a=2) \wedge (a=1) = false$ となり、AELL(1) 条件を満足する。

4.4 AELL(1) 文法から構文解析器へ

AELL(1) 文法に従った構文解析器を生成する方法は、ELL(1) 文法のそれに、意味規則と属性条件の評価を加えたものとなっている。部分正規表現 e から構文解析器のコードを得る写像 $Code(e)$ を表 3 に示す。

4.5 EAGLE における AELL(1) 文法の制限

AELL(1) 文法では、意味規則 $\$PLUS$ やパラメタの受渡し $b = a$ といった属性評価規則も属性条件として考慮することにより、属性条件全体の評価が可能となる。しかし、これをコンパイラ生成系でそのまま実現した場合、属性評価のバックトラックが発生する。先の文法では、 e_1 と e_2 を選択するとき、まず e_1 のために、 $b = a$ と $\$PLUS(\uparrow c, \downarrow b, \downarrow 1)$ が評価される。しかし、 e_1 が選択されない場合、この属性評価は無効としなければならない

表 3: 構文規則 e と構文解析器のコード $Code(e)$ の対応

e	$Code(e)$	e	$Code(e)$
ϵ	空文;	$e_1 \mid \dots \mid e_n$	if Condition(next, e_1) then Code(e_1) elseif ... elseif Condition(next, e_n) then Code(e_n) else error;
$\langle p \rangle$	if not p then error;	$\{e_1\}$	while Condition(next, e_1) do Code(e_1);
$x \in V_T$	if next = x then gettoken else error;	$A(\text{式}1, \dots, \text{式}n) : \alpha$	procedure A($\boxed{1}, \dots, \boxed{n}$); begin Code($\langle p \rangle$); ここで、 式 $i = \downarrow \dots$ の時、 $\boxed{i} = \text{式}i$ 式 $i = \uparrow \dots$ の時、 $\boxed{i} = \text{work}_i$ $p = \bigwedge_{\text{式}i = \uparrow \dots} (\text{work}_i = \text{式}i)$ ※ work_i は一時変数 end ここで、式 $i = \downarrow \dots$ の時、 $\boxed{i} = \text{work}_i$ 式 $i = \uparrow \dots$ の時、 $\boxed{i} = \text{var work}_i$ $p = \bigwedge_{\text{式}i = \downarrow \dots} (\text{work}_i = \text{式}i)$ (A の生成規則が複数ある場合、 等価な1つの生成規則にまとめておく)
$x_{\langle p \rangle}$	Code(x) Code($\langle p \rangle$)		
$A(\text{式}1, \dots, \text{式}n)$	A($\boxed{1}, \dots, \boxed{n}$); Code($\langle p \rangle$) ここで、 式 $i = \downarrow \dots$ の時、 $\boxed{i} = \text{式}i$ 式 $i = \uparrow \dots$ の時、 $\boxed{i} = \text{work}_i$ $p = \bigwedge_{\text{式}i = \uparrow \dots} (\text{work}_i = \text{式}i)$ ※ work_i は一時変数		
$\$r(\text{式}1, \dots, \text{式}n)$	r($\boxed{1}, \dots, \boxed{n}$); Code($\langle p \rangle$) ここで、 $\boxed{i}$ は $A(\text{式}1, \dots, \text{式}n)$ に同じ		
$e_1 \dots e_n$	Code(e_1)...Code(e_n)		

next は先読み記号を保持する変数であり、トークンの読み込みを行う手続き gettoken により値が変化する。条件評価関数 Condition(next, e) は、先読み記号が next の時、規則 e を導出可能ならば true になる関数である。

$$\text{Condition}(\text{next}, e) = \bigvee_{x \in \text{Director}(e)} ((\text{next} = x) \text{ and } p), \quad \text{ここで、} p = \bigvee_{(x, p_i) \in C\text{Director}(e)} p_i$$

い。

このため、EAGLE では、意味規則やパラメタの受渡しを、属性条件に加えないという制限を設けた。また、これにより、評価不可能な属性条件ができる場合がある。先の例では、属性条件 ($c = 3$) がそうである。したがって、このような評価不可能な属性条件を排除する必要がある。

以上の理由および生成系の早期実現のために、現在の EAGLE では、この制約をさらに強めている。AELL(1) 条件の判定で用いる属性条件は、規則の選択が行われる位置に記述されたものだけを扱う。すなわち、 $\langle p \rangle e_1 \mid \langle q \rangle e_2$ といった規則の場合、属性条件は $\langle p \rangle, \langle q \rangle$ だけを扱い、 e_1 や e_2 を導出した先にある属性条件は排除する。先の文法の生成規則 S は、現在の EAGLE では次のように記述する必要がある。

$$S(\downarrow a) : A \quad (\langle a = 2 \rangle e_1(\downarrow a) \mid \langle a = 1 \rangle e_2(\downarrow a)) x.$$

5 システムの評価と考察

我々は EAGLE を用いて Pascal のサブセットである Pascal S[10] コンパイラの構文意味規則を記述し、評価を行った。

表 4: 記述量の比較 (行数)

記述言語	字句規則	構文意味規則	C/C++ 関数	計
拡張 1 パス型属性文法	10	884	168	1062
通常の属性文法 *1	98	719	547	1364

*1: 構文解析、意味解析の記述のみ。コード生成はしない。

5.1 記述量の評価

拡張 1 パス型属性文法を用いて Pascal S コンパイラと、正規表現を許さない通常の属性文法を用いて記述された Pascal S コンパイラ [11] の記述量を表 4 に示す。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。

拡張 1 パス型属性文法の記述量 (字句規則の記述量を除いたもの) は、通常の属性文法によるものに比べ、0.78 倍である。これは、正規右辺文法により簡潔な構文記述が可能になったこと、および属性条件の記法により従来 C 言語で記述していたような意味規則も我々の提案する文法で記述可能になったこと (§2.3 を参照) が大きい。また、プリミティブ (§2.2) を用いて記号表の操作が簡潔に記述できたことも挙げられる。

EAGLE には字句解析器の自動生成機能が用意されているため、字句規則の記述量も減少している。EAGLE

表 5: Pascal S コンパイラの実行時間の比較

使用したコンパイラ	EAGLE で生成	手書き
コンパイル時間	0.79 秒	0.48 秒

では 'goto' のような記述がある場合、対応する字句解析器の記述 (lex 記述) を自動生成する。表中の EAGLE の字句規則の記述量は、'...' の形で記述できない字句規則によるものである。次に識別子を表す終端記号 IDENT とコメントの字句規則を示す。

```
%TERMINAL ident({name): /[A-Za-z][A-Za-z0-9]*/
                        $getString({name}).
%TERMINAL %SPACE: /"[{}]*"/
```

5.2 生成されたコンパイラの実行速度の評価

評価に用いた Pascal S プログラムは [12] にある PL/0 プログラム multiply (を 30 回並べたもの。492 行) である。このプログラムを、EAGLE が生成した Pascal S コンパイラと、手書きのコンパイラ [10] によりコンパイルし、実行時間を測定した (表 5)。

EAGLE で生成したコンパイラの実行時間は、手書きのコンパイラに比べて 1.6 倍になった。今回は実現の早さを目指したため、効率改善の余地は多々ある。たとえば、現在の全ての属性がバックパッチクラスの変数となっているため、属性の定義 / 参照の度に余分なオーバーヘッドが生じている。効率改善策としては、バックパッチ変数の使用数の減少が考えられる。また、現在配列で実現されている記号表のハッシュ化も挙げられる。

6 おわりに

本稿では、簡潔な記述、および効率の良いコンパイラの生成を目指した拡張 1 パス型属性文法を提案した。拡張 1 パス型属性文法は、正規右辺属性文法を記述の基本とし、バックパッチ処理の自動生成や、属性値主導構文解析を可能にするような記法を採り入れた文法である。

さらに、拡張 1 パス型属性文法にしたがった属性評価の方法、特にバックパッチを用いた属性評価の方法も述べた。また、与えられた文法が属性値主導の ELL(1) 構文解析により解析可能かを判定する方法も述べ、これを AEEL(1) 文法という新しい文法クラスとして定義した。

我々は、拡張 1 パス型属性文法に基づくコンパイラ生成系 EAGLE を開発し、実際に Pascal S コンパイラを記述してみた。この結果、記述量は 0.78 倍と、通常の属性文法に比べ簡潔な記述ができた。生成されたコンパイラ

の実行時間は手書きに比べ 1.6 倍となり、自動生成のコンパイラとしては良い結果が得られた。

今後は、生成されるコンパイラの実行速度の向上、および記述量の減少のために、再帰的下向き構文解析における演算子順位構文解析 [13] の手法を我々のコンパイラ生成系 EAGLE に採り入れたい。

謝辞

本研究において、EAGLE の初期バージョンの開発を行った元本研究室の金谷英信氏 (現ソニー)、星野秀之氏 (現サンデン) に感謝します。また、現在のバージョンへの変更を手伝って頂いた本研究室の山下敬治氏にも感謝します。

参考文献

- [1] Knuth.D.E: Semantics of Context-Free Languages (Math. Syst. Th., Vol.2, No.2, pp.127-145, 1968).
- [2] Ole Lehrmann Madsen: On Defining Semantics by Means of Extended Attribute Grammars (Lecture Notes in Computer Science(LNCS) Vol.94, pp.258-299, 1980).
- [3] Julling.R.K, DeRemer.F: Regular Right-Part Attribute Grammars (SIGPLAN NOTICES pp.171-178, Vol.19, No.6, 1984).
- [4] E.F.Elsworth, M.A.B.Parkes: Automated Compiler Construction based on Top-down Syntax Analysis and Attribute Evaluation (SIGPLAN NOTICES, pp.37-42, Vol.25, No.8, 1990).
- [5] Kastens,U., Hutt,B. and Zimmermann,E.: GAG: A Practical Compiler Generator, (LNCS Vol.141, 1982)
- [6] 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 pp.204-212, Vol.30, No.2, 1989).
- [7] David A.Watt: Rule splitting and attribute-directed parsing (LNCS Vol.94, pp.363-392, 1980).
- [8] D.R.Milton, C.N.Fischer: LL(k) Parsing for Attribute Grammars (LNCS Vol.71, pp.422-430, 1971).
- [9] Terence J.Parr, Russell W.Quong: Addig Semantic and Syntactic Predicates To LL(k): pred-LL(k) (Int'l Conf. on Compiler Construction, 1994).
- [10] R.E.Berry: Programming Language Translation (Ellis Horowood, 1981).
- [11] 萩原 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 (筑波大学大学院 理工学研究科 修士論文, 1993).
- [12] Wirth.N: Algorithms+Data structure=Programs (Preitice-Hall, 1976).
- [13] 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, Feb. 1993).