6G－9

拡張LL(1)パーサ生成系の提案*

金谷英信　中川裕之　中田育男
筑波大学

1　はじめに

近年、種々多様な言語処理系（コンパイラ）が登場し、また、コンパイラを作るためのシステムであるコンパイラ生成系（コンパイラ・コンパイラ）の開発も盛んに行なわれている。
その主たる構文解析手法には、例えばyaccに代表されるLALR(1)文法とCoCo/R[3]に代表されるLL(1)文法を挙げることが出来るが、前者は後者に比べてエラー処理において、また、後者は前者に比べて扱える文法クラスの範囲において、それぞれ見劣りがある。
そこで、本論文では、下記の項目を作成目標としたコンパイラ生成系の基本方式を提案し、その有効性を考察する。

- 効率の良い1パスコンパイラで適切なエラー処理を容易に行なえるコンパイラを生成する。
- 新しい記法を導入することで言語クラスの範囲を広げ、また、解り易い記述にする。

2　本コンパイラ生成系の特徴と従来のコンパイラ生成系の問題点

上記作成目標を達成するため、文法と構文解析器との対応が良く、効率の良い解析器を作ることが容易であり、エラー処理も見通しよく行なえるLL(1)を基本として、それを拡張することにした。

2.1　従来のLL(1)の問題点

LL(1)では扱える言語のクラスが比較的小さく、左再帰性除去や左括りだしを行なうと下記のような問題点が発生する。

- 生成規則の数が増え、効率が低下する。
- 構文規則が複雑になり、意味解析との対応が取りにくい。

左再帰性の問題は正規右辺文法でほとんど解決できるが、その他にも以下のような問題もある。

2.1.1　構文規則の書き換えによる複雑な文法記述

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なう場合、直観的には、下記のような記述を行なう。

statement ::= *call*|*assign*
assign ::= *ident* ':=' *exp*
call ::= *identparam*

しかし、これはLL(1)文法でなくなるため、左括りだしの技法を用い下記のように文法を書き換えなくてはならず[2]、解りにくくなる。

statement ::= *ident*(*rest_call*|*rest_assign*)
rest_assign ::= ':=' *exp*
rest_call ::= *param*

2.1.2　左から右へ属性評価が出来ない場合の文法記述

pascalで変数の型宣言をする場合、プログラムの記述は下記のようになる。

```
var v1,v2,v3,v4:INTEGER ;
```

L属性文法では、左から右へ属性評価する時点で依存する属性の値がすでに決まっていなくてはならない。今この例で各変数の属性値として変数の型を与えるとき、各変数を読んだ時点では、型(INTEGER)は、まだ読んでいないからL属性文法ではない。このような場合にも1パスで構文解析と一緒に属性評価をするためには、何らかの工夫をしなくてはならない。その1つの方法に下記のような再帰型の表現[2]があるが、文法記述が複雑になり意味解析との対応が取りにくくなる。

variable(↑ *list*, ↑ [*name* → *type*], ↑ *type*) ::=
　↑ *ident*(↑ *name*),
　　(↑ *comma*, *variable*(↑ *list*, ↑< *list* >, ↑ *type*)
　　| ↑ *colon*, *typename*(↑ *list*, ↑ *type*)
　　)
typename(↑ [], ↑ *type*) ::=↑ *ident*(↑ *type*)

2.2　本生成系の特徴と提案機能

上記の問題を解決するための提案機能と本生成系の特徴を以下に記す。

(1) 構文解析手法としてLL法を用いたコンパイラを生成する。

(2) 文法記述は、左再帰性を大体除去できる正規右辺文法を基本とする。

(3) 意味規則の簡潔な表現を可能とする。

(4) 2.1.1に挙げた問題点を解決するために、attribute directed parsingが出来るような表現を導入し文法の書換えを行なわなくても良いようにする。

*A practical ELL(1) parser generator
Hidenobu KANAYA, Hiroyuki NAKAGAWA, Ikuo NAKATA
(Univ. of Tsukuba)

(5) back patch できるものに限り、L属性でない表現も許す。

3 実現方式

上述の問題点の解決方法を中心に本生成系の特徴的な記述及び、その実現方式を以下に記す。

3.1 attribute directed parsing が出来るような表現の導入

上述の 2.1.1 のように LL(1) 文法に反する記述ではあるが、文法を書き換えることにより生成規則の数が増え、解りにくくなったり効率の低下を招く恐れのある場合の解決方法として attribute directed parsing が出来るような表現を導入する。これは、構文規則を書き換えるのではなく、属性値によって識別子の区別が出来るようにするものである。下記にその文法記述を示す。

$statement ::= call|assign$

$assign ::= ident <\uparrow mode = var >\text{':='}exp$

$call ::= ident <\uparrow mode = proc > param$

構文解析では、識別子 ident が現れても $<\uparrow mode = var >$、$<\uparrow mode = proc >$ といった属性値を用いることにより assign か call のいづれに解析を進めていけば良いかが解る。このように、属性値を解析情報として用いながら構文解析を LL(1) のごとくに行なうことが出来る。

3.2 back patch で処理できる範囲での表現の拡張

back patch で処理できるものに限り、L属性でない表現を許すことにより、次のような記述ができる。

3.2.1 右から左への属性評価を可能にした表現

上述の 2.1.2 のような右から左に属性が決まるL属性文法に反する記述は、再帰的な表現に変更することなく、そのまま記述できるようにする。そのためには、一連の生成規則内において右から左への属性受渡しを許すようにする。この表現の導入により構文規則が簡潔になり、容易に意味解析との対応がつく。その実現のためには、back patch を用いてその属性評価を行なえるようにする。下記に、その記述を示す。

$variable ::= \{ident(\uparrow name)$

$\S TABLE(\downarrow name, \downarrow types, \downarrow mode = var)\ ^{\triangledown}, ^{\triangledown}\}$

$\text{':'}type(\uparrow types)$

$\S TABLE$ は、意味記述を表し、宣言された変数の名前と型を登録する関数を示すものであり、$^{\triangledown}$,$^{\triangledown}$は読み込んだトークンが「,」である間、{} 内の生成規則を繰り返すことを意味する記法である。$\S TABLE$ 実行時に渡されるパラメータの内、$\downarrow name$ と $\downarrow mode = var$ は決定しているが、$\downarrow types$ は決定していない。しかし、$\downarrow types$ は、$type$ を構文解析した時点で決まるものであるから、その時点で back patch する必要がある。そのために back patch リストに登録しておく。「:」が現れたところで 内の解析を終了し $type$ を読み込んだ時点で back patch 処理を行なう。

3.2.2 目的コードに属性を用いた表現

条件文のように、条件評価の結果によって飛び先が異なるような文法を記述する場合、LL(1) 文法で記述を行なうと非常に複雑になる。しかし、目的コードに属性を用いることにより、下記のような簡潔な記述とすることが可能となる。

下記に、その文法記述を示す [1]。

$if\_stat ::= \text{'IF'}condition(\downarrow nextp1, \downarrow nextp2)\text{'THEN'}$

$\S CODE(\uparrow nextp1)statement$

$\S CODE(\uparrow nextp2)$

$condition(\downarrow truep, \downarrow falsep) ::= \{bterm(\downarrow truep, \downarrow nextp)$

$^{\triangledown}\vee^{\triangledown}\S CODE(\uparrow nextp)\}$

$bterm(\downarrow truep, \downarrow falsep)$

ここで、$\S CODE$ はその時点での目的コードのアドレス（次の命令の番地）を返す意味記述である。$condition$ に渡される属性 $truep$ は $condition$ が真である場合の飛び先、属性 $falsep$ は偽である場合の飛び先を表す。$condition$ は LL(1) 文法としては、

$condition ::= bterm\{^{\triangledown}\vee^{\triangledown}bterm\}$

のように書くべきところではあるが、最後の $bterm$ の分岐飛び先のアドレスだけが異なるために上記のような記述となる。

$condition$ の記述は LL(1) に反するものではあるが、$bterm$ の構文解析を行ない、次のトークンが$^{\triangledown}\vee^{\triangledown}$でなければ最後に解析された「$bterm(\downarrow truep, \downarrow nextp)$」を「$bterm(\downarrow truep, \downarrow falsep)$」とみなして $condition$ を終了させるようにすれば、LL(1) の手法のなかで処理可能である。$condition$ の属性 $truep$ は if_stat の$\S CODE(\uparrow nextp1)$ で backpatch されるものである。同様に、$condition$ の属性 $falsep$ は if_stat の$\S CODE(\uparrow nextp2)$ で、$condition$ の属性 $nextp$ は $condition$ の$\S CODE(\uparrow nextp)$ で back patch される。

一見、属性値の書き替えをしなくてはならないように見えるが、実際に構文解析して得られるものは後で back patch すべきもののリストであるから、次のトークンが'$\vee$' であるかどうかによって、backpatch リストの連結を変更してやれば実現が可能である。

4 まとめ

attribute directed parsing ができるような表現と back patch で処理できる範囲内での表現の拡張を行なうことによる簡潔でわかりやすい文法記述法を提案した。

今後は、生成系の完成度を向上させ、従来のコンパイラ生成系との比較、ならびに、手書きのコンパイラとの性能比較を行ないたい。

参考文献

[1] 中田育男: コンパイラ, 産業図書, 1981.

[2] 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).

[3] H Mössenböck:CoCo/R–A Generator for Fast Compiler Front-Ends, ETH, 1990.