１パス型属性文法におけるバックパッチ処理の自動生成

2N－4　平見 知久　山下 義行　中田 育男
筑波大学

1　はじめに

コンパイラ生成系では文脈自由文法や属性文法に基づき構文解析器、意味解析器の自動生成を行う。その中でも特に、１パス型の属性文法が使用されることが多い。

しかし１パス型の属性文法においては右依存的な属性評価は許されない。したがってこれに基づく生成系では右依存的な属性評価は使用できず、記述において右依存的な属性評価としたほうが素直な場合についても、コンパイラ作成者が右依存のない記述に修正しなければならなかった。このような処理系の制約による記述の変更を行なった場合、得られた記述は本来作成者が意図していたものとは別のものであるため、記述の読解性が低下してしまうという問題があった。

本発表では、右依存的な属性評価に対してバックパッチと呼ばれる処理を自動生成することにより、記述を変更することなく右依存的な属性評価を行えるような方法を提案する。また、現在我々が開発中である属性評価器生成系 EAGLE[1] に実現を行い、その有効性について検討を行なった。

2　L 属性文法の制限とバックパッチ処理

１パス型の属性文法は左から右へと属性評価が行えるものであり、L 属性文法は１パス型である。右依存的な属性は L 属性文法では持つことは許されず、一般にそれを１パスで評価することはできない。しかし実際の言語の記述を考えてみた場合、その多くは L 属性で解析できるものの、右依存的な属性評価が必要となる場合は少なくない。例えば、Pascal のような言語では変数宣言は変数名の後に型名を記述する。

```
var a, b:integer;
```

このような変数宣言に対応する記述は以下の通りである。

```
decl:  'var'
       { IDENT(↑name)
       記号表へ登録 (↓name, ↓type_name) // ','}
         ':' type(↑type_name) ';'.
```

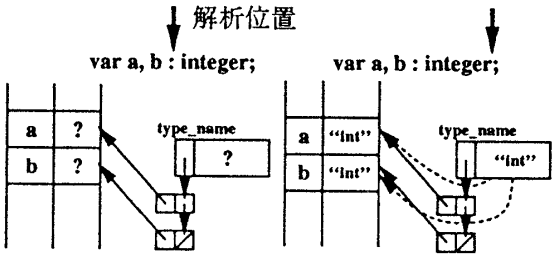

図 1: バックパッチ処理の概念図

この記述では各 IDENT について記号表へ登録をしているが、登録の段階では type_name の値は決定していない。そのため、従来の L 属性文法では評価できない。

これに対してバックパッチと呼ばれる手法を用いることで記述をほとんど変更することなく、かつ１パスで解析を行うことができる。バックパッチとは、歴史的には１パスのアセンブラで用いられていた手法であり、値が決定できないものについてはとりあえずその値が必要とされる場所を覚えておき、値が決定した段階で覚えておいた場所に値を埋め込む方法である (図 1)。

EAGLE では右依存的な属性についてはバックパッチ処理を自動生成することによって、右依存的な属性の記述も許す。また、実現にあたっては目的言語として C++ を使用し、属性をバックパッチ機能を持ったクラスとすることによってバックパッチ処理を実現している [2]。

バックパッチ処理をクラス内で行うことにより、生成されるコードは非常に簡潔なものとなる。そのため、記述のデバッグ時などにおいて生成されるコードの動作を既存のデバッガなどで追った場合にも、ユーザーはバックパッチを意識することなく解析を行うことができる。一方、すべての属性をバックパッチ機能つきのクラスとすることは簡単であるが、バックパッチリストの管理などのオーバーヘッドを考えた場合バックパッチクラスの使用は最小限に留めるべきである。そのためにはバックパッチが必要な属性の検出が必須となる。

3　バックパッチ処理が必要な属性の検出

バックパッチ属性の検出において問題となるのは、依存関係が伝播することである。そのため、一見すると右

Generation of back-patch process on one-pass type attribute grammar
Tomohisa HIRAMI, Yoshiyuki YAMASHITA, Ikuo NAKATA (Univ. of Tsukuba)

依存的でないような属性の使用関係であっても実際には右依存的であるといったことが発生する。特に、継承属性、相続属性との依存関係が存在する場合、バックパッチの検出は複数の規則にまたがって行わなければならない。

例を挙げて説明する。

E : A($\downarrow a$) B($\uparrow a$).
A($\downarrow a$) : %{ $\uparrow b = \downarrow a$;%} C($\downarrow b$).

規則Eでは、構文解析はAからBへと進んでゆく。ところが、属性の依存関係に着目すると、aの値はBを解析しないとわからない。したがって、属性aは右依存である。

一方、規則Aにおける属性bも右依存的である。一見するとCの解析の前にbはaによって定義されているため、右依存的でないように見えるが、実際にはaが右依存的な属性であるため、依存関係が伝播し、bは右依存の属性となる。もし、aが右依存でなかったならば、bも右依存でないことになる。aが右依存かどうかについてを知るためには、入力記述中から規則Eのような規則Aを使用しているすべての規則について、Aに渡す属性が右依存的であるかを調べる必要がある。

EAGLEでは入力記述から中間木を作成する際に属性の依存グラフを作成してゆき、全規則の解析後にその依存グラフ上で右依存の属性のチェックを行う。

4 実験

右依存的な属性の検出を行い、バックパッチクラスの使用を限定することによる実行効率の向上について評価を行うために、検出ルーチンをEAGLE上に実装し、生成したコンパイラについてその実行速度の比較を行った。

バックパッチ処理の発生頻度は入力の記述によって異なる。そこで、何種類かの例を用意し、バックパッチ処理の発生頻度を変えて実験を行うこととした。実際には、簡易電卓、Pascal風言語の変数宣言、Pascal S[3]の3つを使用した。

簡易電卓はバックパッチ処理の全く必要のない記述であり、バックパッチ処理の選択を行うにあたって、最も効果が得られると考えられるような例である。一方、Pascal風の変数宣言はその記法上、バックパッチ処理が非常に多くなる例であり、右依存的な属性の検出による最適化のききにくい例であるといえる。最後のPascal Sは一般的な言語程度でのバックパッチの使用量での比較である。

実際の評価はIndy(SGI, CPU:R4600 100MHz)上で行ない、測定にはpixieコマンドを使用した。。計測結果を表1に示す。

eagle(opt)がEAGLEの生成したコンパイラの実行時間である。eagle(allbp)は全属性についてバックパッチクラスを使用した場合の実行時間を示している。また、比較対象として、pascalによるもの(pas)、字句解析を含めて完全にC++で書かれたもの(手書き(full))、字句解析器についてはflexを使用したもの(手書き(flex))を挙げる。表より、eagle(opt)はpasに比べて1.3倍から2.3倍程度の実行時間で解析を行っていることがわかる。。また、eagle(opt)とeagle(allbp)を比較すると、バックパッチ処理が多い変数宣言では実行時間はそれほど変わらないものの、他の例では実行時間が大幅に減少していることから、バックパッチ属性の使用を必要最小限で抑えることにより実行効率が向上しているといえる。

処理系	簡易電卓	変数宣言	Pascal S
eagle(opt)	13.7	34.9	44.4
eagle(allbp)	18.4	38.9	80.3
手書き(pas)	8.6	15.1	33.6
bison	22.1	35.9	—
手書き(full)	8.0	17.1	—
手書き(flex)	12.1	24.4	—

表1: 実行時間(単位:ms)

5 まとめと今後の課題

本発表ではバックパッチ属性の検出方法について述べた。また、そのような方法によって検出された属性のみをバックパッチ属性にすることによって、手書のものにかなり近い実行時間で解析ができることを示した。

現在のリストを用いたバックパッチ処理では、値が未決定の時にはその場所だけを覚えておき、実際に値が定まった時に属性値の代入処理を行っているため、コピールールにしか対応できていない。しかし実際には、属性評価式中に右依存的な属性を書き、属性値が定まった段階でその演算を行いたいと行った場合も存在する。今後の課題としては、このようなバックパッチ動作に伴って計算が発生するような場合の処理を自動生成することが挙げられる。

参考文献

[1] 大木, 平見, 中川, 山下, 中田:
"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", 情報処理学会研究報告 94-PRG-17, 1994

[2] Damian Conway:
"Parsing with C++ Classes", ACM SIGPLAN Notices, Vol.29, No.1, 1994

[3] ロバートE.ベリー著, 武市正人訳:
"プログラム言語の処理系 Pascal Sの翻訳系・通訳系", 近代科学社 1983