自己拡張可能な構文解析器生成系*

舞田純一 (学籍番号 200621335)
研究指導教員:長谷川秀彦
副研究指導教員:中井央

1. はじめに

コンパイラの研究では、新たな機能を考案し、それを実装して、既存の機能との比較や検討などを行う必要がある。しかし既存の生成系を用いてコンパイラを実装する場合、生成系もしくは生成物の改造が必要となるため、新たな機能の実現は容易ではなかった。

そこで本研究では、生成系または生成物に一切の改造を行うことなく、新たな機能を容易に実現するために、用途に合わせて機能を動的に拡張できる生成系について考察し開発を行った。

コンパイラへの機能追加に関する既存の研究として、Decorator パターンによって構文解析器の各機能を独立したモジュール(以下、**付加機能**と呼ぶ)として扱う構文解析器の構成法[1]がある。この手法によって任意の機能を**付加機能**とすることができるが、一方で文献[1]で示された生成系では対応できる**付加機能**が限定されていた。本研究では、生成系を用いたコンパイラの開発において、コンパイラに目的に合わせてさまざまな**付加機能**を導入できるようにするために、生成系にプラグインの機構を導入することを考え、そのためのシステムを開発した。

2. 設計の方針

生成系がアクション機能に対応するには、例えば以下のような記述を扱える必要がある。

```
expr : expr '+' term { $1 + $3 }
```

また、木の構築機能に対応するには、例えば以下のような記述を扱える必要がある。

```
expr : expr '+' term => add(expr, term)
```

* "A Self Extensible Parser Generator"
by Jun'ichi MAITA

```
 1 %class TinyCalc
 2 %extend Lexer ('depager/lex.rb')
 3 %extend Action ('depager/action.rb')
 4 %decorate @Action
 5 %%
 6
 7 %LEX{
 8   /¥s+/, /¥#.*/ { }
 9   /[1-9][0-9]*/ { yield :NUM, $&.to_i }
10   /./           { yield $&, $& }
11 %}
12 #begin-rule
13   expr :
14       expr '+' NUM { val[0] + val[2] }
15     | NUM          { val[0] }
16   ;
17 #end-rule
18 %%
```

図 1 入力記述例

このように目的とする機能により生成系が扱うべき記述は異なるものとなる。そのため、多様な機能に対応するには入力記述の仕様自体が動的に変更できなければならない。この実現には、記述をプリプロセッサによって外部的に処理する方法とプラグインと生成系本体が協調的に処理する方法が考えられる。プリプロセッサ方式では対応できる機能や組み合わせに制限が多いが、協調方式ではそのような制限が少ない。そのため、本システムではプラグインによる協調方式を採用した。

3. システムの概要

3.1 入力記述

本システムでは、生成系本体とプラグイン(以下、**拡張**と呼ぶ)が協調して記述を処理する。入力記述は、使用する**拡張**を指定する部分(宣言部)、文法部分(文法部)、**拡張**により処理される**付加機能**のための記述(拡張部)から構成される。

図1は入力記述の例である。6 行目までが宣言部、それ以降が文法部と拡張部であり、字句解析

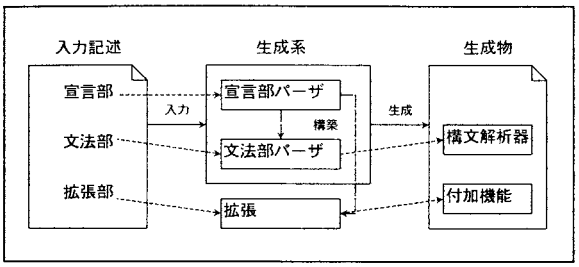

図 2 システムの構成

(7～11 行目の%LEX{…%}の部分)とアクション(14、15 行目の{…}の部分)のための**拡張**を用いている。

3.2 生成系本体と拡張の協調

このような入力記述を解析するために、本システムは宣言部パーザ、文法部パーザ、そして**拡張**によって構成される。宣言部パーザは宣言部を解析し、文法部パーザと**拡張**のインスタンスを作成する。そして、文法部パーザと**拡張**が協調して文法部、拡張部を解析する(図 2)。

生成系本体と**拡張**が協調できるようにするために、入力記述中の以下の部分(ポジション)に拡張部を設けて**拡張**により解析できるようにした。

- 文法全体の前後
- 各規則の前後
- 各左辺の後
- 各右辺の並びの前後
- 各右辺の前後
- 各右辺の各記号の後

さらに、1 つの**拡張**が複数のポジションに対し構文解析を行えるように、**拡張**には**付加機能**のコード生成を担うマスターパーザと入力記述の解析を担う複数のスレーブパーザを持たせた。

全体の処理の流れは、以下のようになる。

1. 宣言部パーザが宣言部を解析し、文法部パーザと指定された拡張のマスターパーザのインスタンスを作成する
2. 拡張のマスターパーザがスレーブパーザを解析すべきポジションに登録する
3. 文法部パーザが文法部を解析する。スレーブパーザが登録されたポジションに達したら、それに制御を移し拡張部を解析する
4. すべての記述の解析が完了したら、生成したコードを出力する

3.3 拡張の作成

拡張は一種の構文解析器であるため、本システムの生成系と**拡張**によって生成できる。また、本システムは、入力記述ファイルの情報取得、文法部の情報取得、コード生成補助などの**拡張**の実装を支援する API を提供する。

本システムで拡張を作成する場合、拡張部のポジションと文法、処理内容を指定する。以下に簡単な例を示す。ここでは、ポジションとして右辺の後(postrhs)を指定している。

```
%hook postrhs
%%
 start: '{' '}' { warn 'HIT' } ;
%%
```

4. まとめ

コンパイラの新たな機能を容易に実現するには、生成系がその**付加機能**のための記述に対応できる必要がある。このため、本研究では生成系本体と**拡張**が協調して入力記述を処理するための仕組みを実装した。また**拡張**自体も本システムによって生成できるようにした。そして、実際に本システムを用いてアクション、木の構築、字句解析などの機能の実装を行った。

拡張の実装において、その作成者は生成系本体の改造を行う必要がなく、また本システムによって処理の本質的な部分の実装に専念できるため、比較的容易に新たな機能を実現できる。

本システムの問題点としては、複数の**拡張**を組み合わせたときの衝突が挙げられる。現状では運用により衝突を回避するようにしているが、これに関しては今後の課題である。

文献

[1] 佐竹力, 中井央. オブジェクト指向に基づいた構文解析器構成法の提案. 情報処理学会論文誌:プログラミング, Vol. 45, No. SIG.12 (PRO 23), pp. 25-38, 2004.