データベース・システム 85－4
プログラミング－言語・基礎・実践－ 4－4
（1991. 10. 4）

プログラムジェネレータの自動生成

田中伸明
松下電器産業株式会社 情報通信東京研究所

概要

本稿では、属性文法を用いた仕様記述からプログラム生成系を自動生成する方式について述べる。

従来の構文解析器生成系は、主にコンパイラの生成を目的に開発されているため、C言語などの高級言語のためのプログラム生成系を作成する場合には、必ずしも使いやすいものにはなっていなかった。そこで必要な機能を補い、仕様記述のみからプログラム生成系を自動生成できるようなプログラム生成系の生成系Embryoを開発した。また、仕様記述中のプログラム合成規則を構文解析することにより、プログラム生成系の生成するプログラムに構文的な誤りが含まれていないかどうかを判定する機能も試作した。

A TOOL FOR PRODUCING PROGRAM GENERATORS

Nobuaki Tanaka
Matsushita Electric Industry Co., Ltd. Tokyo Information and Communications Research Laboratory
3-10-1, Higashimita, Tama-ku, Kawasaki 214, Japan

This paper describes a method to generate a program generator from the formal specification described by an attribute grammer. Most of the existing parser generators are developed for generating low-level programing languages, but are not suitable for generating high-level programing languages. We extended some functions to yacc to facilitate generating program generators, and enabled to generate a program generator from the formal specification.

Our specification checker parses program-synthesizing rules in the specification, and detect syntax errors in the program that will be generated by the program generator that is generated from the specification.

1. はじめに

近年、RPC(Remote Procedure Call)用インタフェースプログラムの自動生成の研究が盛んに行なわれている[1]。我々も異機種異言語対応を主目的とした RPCインタフェースの生成方式の研究を行なっている[2]。

我々は,この研究におけるプログラム生成系をyacc[3]とC言語を用いて作成した。しかし、yaccはコンパイラの作成を主目的として設計されており、プログラム生成系で頻繁に行なわれる文字列操作などの機能が不足していた。このため、開発効率や保守性に問題が生じた。このような課題を解決するために、我々は仕様記述からプログラム生成系を自動生成する方式を開発した。この自動生成を行なうツールをEmbryoと呼ぶ。

また、Embryoから生成されるプログラム生成系の信頼性を高めるために、仕様記述に含まれる誤りを検出する検査ツールを作成した。この検査ツールは、仕様記述中のプログラム合成の仕様を記述している部分を解析し、その仕様が文法的に正しいプログラムを生成するか否かを判定する。

本稿では、2章でEmbryoについて、3章で仕様記述検査ツールについて述べる。最後に4章で現状と今後の課題について述べる。

2. Embryo

2.1 課題と解決手段

1章で述べたように、我々はRPC用プログラム生成系をyaccとC言語を用いて開発した。また、設計手法には一般的なウォータフォールデザインの手法を用い、設計書を段階的に詳細化していった。しかし、この手法では仕様変更に大きな労力を要する。また、実際には日本語で書いた設計書よりもyaccの定義ファイルを直接読んだほうが仕様や動作を理解しやすいことがしばしばある。

そこで生成系の開発過程を見直した結果、開発を次のように行なうべきだとの結論に達した。

・プログラム生成系の仕様は、形式的に記述すべきである。
・プログラム生成系は、形式的な記述から自動生成すべきである。

仕様記述の形式は、yaccの定義ファイルの形式を基に設計した。そして、その仕様記述からプログラム生成系を生成するツールEmbryoを作成することにした。

プログラムの生成方法を表現するためには、S属性文法[4]を用いる。文字列が構文木の下から上に合成属性として渡されていき、それぞれの非終端記号の生成規則が還元されるときに下のノードから渡された文字列の合成を行い、合成された文字列を上のノードに渡していく。例えば図1の例は、文法記号列

type_specifier IDENTIFIER ';'

から 非終端記号 declaration が還元されるときに関数list()の引数が連結されて文字列となりdeclarationの属性typedefに入れられ、構文木の上のノードに渡されることを示す。list()の引数の$2.nameと$1.typeは、それぞれ、type_specifierの属性nameとIDENTIFIERの属性typeを表す。

プログラム生成系の読み込むファイルの文法に対してこのようなプログラム合成規則を記述することによってプログラム生成系の仕様を記述する。

また、上の例の"var ", " : ", ";"のような文字列を仕様記述の中に直接書き込むことによって、仕様記述を書きやすく、かつ読みやすくすることができる。このような文字列の操作を効率良く実現するために、yaccに対して文字列管理の機能を追加し、また、シンボルテーブルの扱いのようにプログラム生成系で一般的に用いられる機能は簡単な表記方法で扱えるようにした。

```
declaration
  : type_specifier IDENTIFIER ';'
        { $$.typedef = list("var ",$2.name," : ",$1.type, ";"); }
  ;
```

図1 生成方式の表記法

2.2 仕様記述の例

仕様記述の例を図2に示す。これは、C言語の構造体をPascalのレコードに変換するプログラム生成系の仕様記述である。C言語の構造体の構文は簡略化してあり、配列、ポインタは含んでいない。また、組み込みデータ型は、intとfloatのみとしている。

```
all:    declaration
        {fprint("file",$1.typedef);}
  ;

declaration <<$$.typedef == var_def;>>
  : type_specifier IDENTIFIER ';'
        { $$.typedef = list("var ",$2.name," : ",
                            $1.type, ";"); }
  ;

type_specifier
  : INTEGER
        { $$.type = "integer "; }
  | FLOAT
        { $$.type = "real "; }
  | struct_or_union_specifier
        { $$.type = $1.type; }
  ;

struct_or_union_specifier <symtable>
  : STRUCT '{' struct_decl_list '}'
        { $$.type = list (" record ", $3.field , " end "); }
  ;

struct_decl_list
  : struct_declaration
        { $$.field = $1.field; }
  | struct_declaration struct_decl_list
        { $$.field = list($1.field, " ; ", $2.field); }
  ;

struct_declaration
  : type_specifier IDENTIFIER ';'
        { check_redefinition(#symtable,$2.name);
          register(#symtable,$2.name);
          $$.field = list($2.name, " : ", $1.type); }
  ;
```

図２ 仕様記述の例

終端記号の属性は、字句解析部により与えられるので、この記述中には現れない。また、非終端記号declarationの生成規則の中にある

<<$$.typedef == var_def;>>

という記述は、declarationの属性typedefは出力文法の非終端記号var_defに対応しなければならないことを示す。この指定は後に述べる仕様記述の検査に用いるものでプログラム生成の動作には影響を与えない。

2.3 システム概要

図３に、本論文のプログラム生成方式の概略を示す。

Embryoは、字句解析部、構文解析部、属性評価部、コード出力部、および文字列管理部からなる。字句解析部は仕様記述を読み取って字句に分解し、構文解析部に渡す。構文解析部は字句の列を読み込んで構文解析をし、同時に属性を評価して属性値を求め、その結果に従ってコード出力部が目的とするプログラム生成系のうちの構文解析部、属性評価部、仕様記述中のリテラル文字列、リテラルシンボルを出力する。Embryoの構文解析部、属性評価部はyaccを用いて作成し、構文解析、属性評価、コード出力はyaccの機能を利用し１パスで行なう。文字列管理部は、システム中で用いられる文字列の生成、連結などの機能を提供する。

Embryoによって生成されるプログラム生成系を目的生成系と呼ぶ。目的生成系の構文解析部と属性評価部は、１つのyaccの定義ファイルとして生成される。このファイルをyaccに入力し、C言語

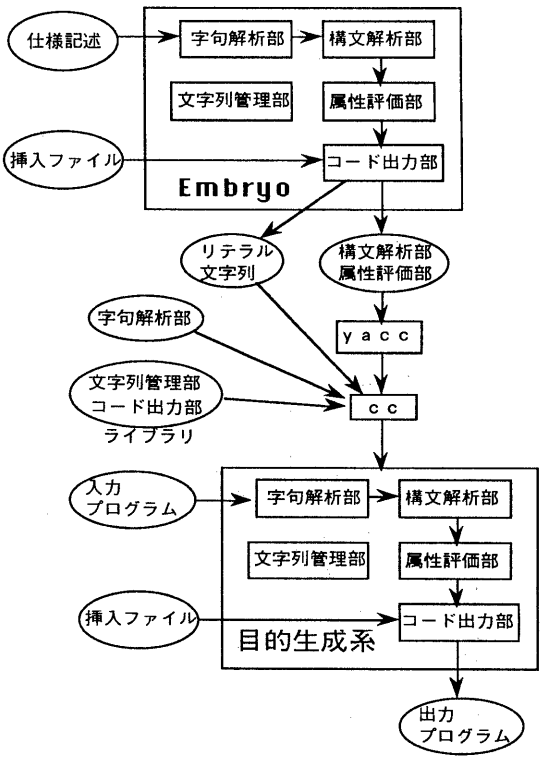

図３ システム構成

で表現された構文解析部と属性評価部を得る。

生成されたプログラム生成系もEmbryoと同様に、字句解析部、構文解析部、属性評価部、コード出力部、および文字列管理部からなる。構文解析部と属性評価部はEmbryoによって生成されたものである。字句解析部は自動生成されないので、ユーザが作成する必要がある。コード出力部と文字列管理部はEmbryoと同じものを使い、ライブラリとして提供される。各部の動作はEmbryoの各部分と同じであり、字句解析、構文解析、属性評価の仕様が違うだけである。

2.4 高級言語生成系のための機能と実装

この節では高級言語の生成系のために必要となった機能と、その実装について述べる。

2.4.1 文字列管理機能

2.1で述べたように、Embryoでは仕様記述の可読性を上げるため、仕様記述の中に合成する文字列を書き込むことを前提としている。また、出力プログラムは、仕様記述中の文字列や字句解析部から得られる文字列を連結していくことによって合成される。そのため、文字列の連結の操作は頻繁に行なわれ、その効率は重要である。そこで、この操作を効率良く行なうための文字列管理の機能を作成した。

Embryoは、仕様記述中の文字列のリストを生成し、目的生成系は起動時に、このリスト中の文字列を生成する。（図４）

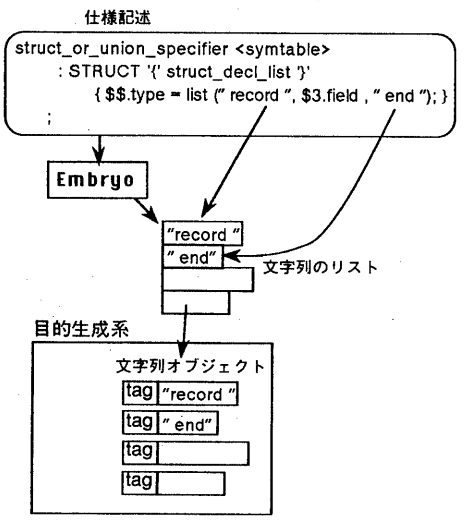

図４　文字列の管理

Embryoと目的生成系の中では、文字列は先頭に文字列であることを示すtagを付けた形で表現される。この形式で表現された文字列を文字列オブジェクトと呼ぶ。この文字列を連結する際には、同様に先頭にtagを付けたコンスセルによってリスト構造を生成して表現する（図５）。この文字列の連結を行なう関数がlist()である。連結された文字列がさらに連結される場合には多段になったリスト構造ができる。連結された文字列を出力するときには、そのリスト構造を左優先で走査しながら出力していく。

この方式を用いたときの利点は、次の２点である。

(1) 文字列オブジェクトへのポインタのリストを作成するだけで文字列の連結をすることができるので、文字列操作の実行効率とメモリ効率が向上する。

(2) 文字列オブジェクト、コンスセルの両方の先頭にtagが付いているので、その２つのデータを簡単に識別でき、連結された文字列を出力することが簡単にできる。

2.4.2 ファイル挿入機能

プログラム生成系では、大量の固定文字列を生成することがある。このような文字列を仕様記述中に書き込むと仕様記述の可読性を劣化させるので、大量の文字列は仕様記述とは別の挿入ファイルに入れておいて、そのファイルを仕様記述中に挿入するような機能を加えた。

ファイルの挿入を行なうタイミングは、Embryoの動作時、目的生成系の動作時の２つから選べる。目的生成系の動作時にファイルの挿入を行なうとファイルの読み込みのため実行が遅くなるが、挿入ファイルの修正が、再生成、再コンパイルしなくても目的生成系に反映される。このようにこの方式は、修正が頻繁に行なわれる場合などに適している。

また、挿入ファイルの一部をプログラム生成時

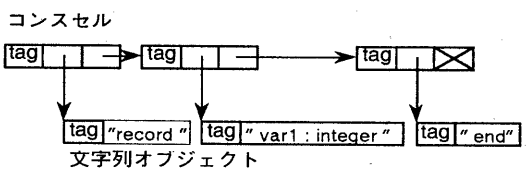

図５　文字列の連結

に得られる文字列と置換する機能もあり、生成するプログラムのテンプレートを挿入ファイルとして作っておき、プログラムの生成時に必要な文字列を当てはめるような使い方もできる。

2.5 仕様記述言語

仕様記述言語はyaccの定義言語をもとに設計した。yaccとの主な違いは、次のようなものである。

(1) 属性の宣言

たとえば下図のように表記するだけで、非終端記号struct_decl_listとstruct_declarationの持つ属性を宣言しておかなくても、struct_decl_listの属性fieldと、struct_declarationの属性field を使うことができる。

```
struct_decl_list
        : struct_declaration
                { $$.field = $1.field; }
        ;
```

(2) シンボルテーブルの操作を示す記法

文法の生成規則の後に、その文法記号の中でネストが深くなるシンボルテーブルの名前を書く。（シンボルテーブルを複数持つこともでき、それぞれに名前が付いている。）たとえば、次のような記述があったとする。

```
a :
        BEGIN def_list END <symtable>
          { $$.val = $2.val ; }
        ;
```

このとき、def_listとENDの中では、シンボルテーブルsymtableのネストが深くなる。（実装上、この例でのBEGINのような、先頭のシンボルについては、ネストが深くならないが、一般的な言語の設計では、これはあまり問題にならない。）これ以外のシンボルテーブルを操作する機能、たとえばシンボル、値の登録や参照はライブラリ関数として提供されている。

(3) 文字列操作の表記

文字列の連結は、2.1節でも述べたように、仕様記述の中ではlist()という関数を用いて表される。この関数は、可変個の引数を持つC言語の関数として実装されており、ライブラリとして提供されている。

連結された文字列の出力は、仕様記述の中でfprint()というライブラリを用いて出力する。

3. 仕様記述の検査

3.1 開発のねらい

2章で述べたEmbryoを使ってプログラム生成系を開発することによって、開発効率を向上させることができる。しかし、実際の開発においては、出来上がったプログラム生成系のテスト／デバッグに多くの時間が費やされる。そのため、開発サイクル全体を考えると、テスト／デバッグの効率向上が望まれる。

そこで、仕様記述中の誤りを直接発見するツールを開発し、そのツールによって開発工程の早い段階で誤りを発見することによって、開発工程全体の効率向上を図った。

3.2 検査ツールの概要

作成した検査ツールは、目的生成系が出力するプログラムが文法的に誤りを含む可能性を判定する。なお、目的生成系が読み込む入力の文法を入力文法、出力するプログラムの文法を出力文法と呼ぶことにする。

ただし、出力されるプログラムは目的生成系中で1つの属性として表現されるとは限らず、一般的には属性の出力用ライブラリ関数を使って、何回かに分けて出力される。そこで、ユーザに図2における非終端記号declarationの属性typedefのように<<>>の中に属性に対応する出力文法中の非終端記号を書いてもらい、その属性が常に指定された非終端記号として構文解析できるか否かを判定する。

なお、この検査アルゴリズムを考えるに当たって、問題を単純にするため、次の2つの条件を置いた。

条件（1） 出力文法はLL文法に属する。
条件（2） 仕様記述中の属性の値は、常に出力文法の1つの非終端記号に相当する。

検査は2つのフェーズに分けて行なわれる。

第1のフェーズでは仕様記述を構文解析して、構文木を作る。

第２のフェーズでは、構文木の中から検査すべき属性を探しだし、その属性を合成する合成規則を構文解析していく。この構文解析のことを検査用構文解析と呼び、次節でその手法について述べる。

3.3 検査用構文解析手法

図２に示した仕様記述を例にとって、第２のフェーズで行なわれる検査用構文解析の方式について説明する。図６は、この仕様記述の検査用構文解析の進む様子を示している。また、図７に、この仕様記述から生成されるプログラムの文法（出力文法）を示す。

まず、フェーズ１で、仕様記述は構文木に展開される。図６では、説明のため、仕様記述の上で検査の進めかたを示しているが、実際には、フェーズ２では構文木の中を解析していく。

次にフェーズ２の検査の手順を説明する。

(1) 最初に仕様記述中で、属性に対する出力文法の中の非終端記号の指定があるところを見つける（図６の(1)）。

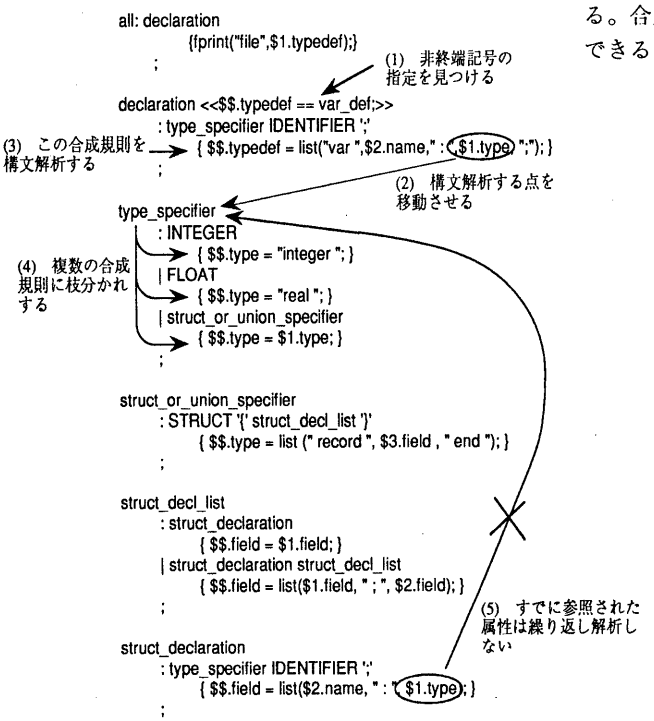

図６. 検査用構文解析

(2) 指定された属性の合成規則の内容をLL構文解析手法で構文解析する。（図６の(2)）この例では "var " , $2.name （IDENTIFIERの属性name）, " : " を順に読み込んで構文解析する。IDENTIFIERは終端記号である。終端記号の属性がどのような値を取るかは、この仕様記述からは知ることはできない。そのため、終端記号の属性は、あらかじめ出力文法中のどの文法記号に相当するかが分かっているものとする。ここではIDENTIFIERの属性nameは出力文法におけるIDENTIFIERに対応するものとする。

この合成規則を、出力文法のvar_defの生成規則と比べてみると、先頭の"var " は出力文法中のVARに、$2.nameはIDENTIFIERに、" : " は ' : ' に対応し、ここまでは正しく構文解析できることが分かる。

(3) 次に、仕様記述中から、$1.typeを読み込む。これは非終端記号type_specifierの属性typeの値を参照している。そのため、type_specifierに移動し（図６の(3)）、その属性typeの合成規則の解析を始める。

ここでは、この属性typeは出力文法のtype_denoterに相当するものと仮定して解析を進める。合成規則中の属性が対応するものとして仮定できる出力文法中の非終端記号は、次に読み込む

```
all:    var_def
var_def
        : VAR IDENTIFIER ':' type_denoter ';'
        ;

type_denoter
        : INTEGER
        | REAL
        | record_type
        ;

record_type
        : RECORD field_list END
        ;

field_list
        : record_section field_list_B
        ;

field_list_B
        :
        | ';' field_list
        ;
record_section
        : IDENTIFIER ':' type_denoter
        ;
```

図７　出力文法

可能性のある非終端記号である。この例では、type_denoterの他にrecord_typeを読み込む可能性もあるので、どちらを選んでもよい。ただし、record_typeを最初に選んだ場合には(5)で述べるように途中で構文解析が失敗してバックトラックが起き、type_denoterが次の候補として選ばれて再度、解析が行なわれる。

(4) type_specifierの属性typeの合成規則は３つある。これらの合成規則のうち、どれが実行される場合でも、(3)で仮定したとおりに、出力文法のtype_denoterとして構文解析できなければならない。

また、一番下の合成規則はさらにstruct_or_union_specifierの属性typeを参照しているので、さらに、その属性へ解析する点を移動させることが必要となる。

(5) type_specifierの一番下の合成規則からは、struct_or_union_specifierの属性typeが参照されている。さらに、合成規則の中の属性の参照をたどっていくと、struct_declarationの属性fieldの合成規則の中で、type_specifierの属性typeが参照されている。この属性は(3)で、出力文法のtype_denoterに対応すると仮定しているので、この属性を再度解析することはせず、type_denoterを読み込んだとして解析を続けていく。

この場合には、struct_declarationの解析は正しく行なえるが、最初に仮定した文法記号が間違っており、他の文法記号と仮定すると正しく構文解析が行なえない場合がある。よって、すでに出力文法の文法記号が仮定されている属性を読み込んだ時に、出力文法と仕様記述が合致しない場合には、その仮定をしたところまでバックトラックして、他の文法記号の候補を仮定してみて、再度構文解析を行なう必要がある。この例では、(3)で、type_denoterではなく、record_typeを先に仮定すると、ここで解析に失敗し、(3)までバックトラックする。

3.6 検査ツールの動作例

この検査ツールを用いた際の動作例を示す。図8は前に述べた仕様記述に間違いが入った例である。（変更された部分のみを抜き出して示している。）この例では、レコードの要素の間の区切り子がセミコロンではなく、コロンになっている。

```
struct_decl_list
        : struct_declaration
                { $$.field = $1.field; }
        | struct_declaration struct_decl_list
                { $$.field = list($1.field," : ", $2.field); }
        ;
                        /* ここが間違っている */
```

図8 間違いを含む仕様

この仕様記述から生成された生成系が間違ったプログラムを出力する場合を図9に示す。

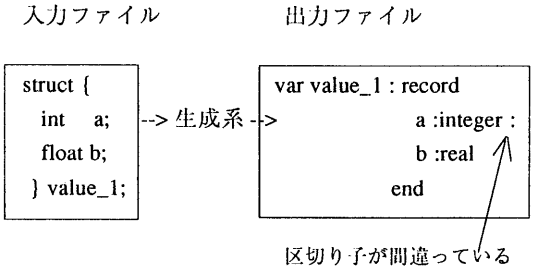

図9 間違いを含む出力

従来は、このように仕様記述が間違っていることは実際に生成された生成系を動作させないと見つけられなかった。しかしこの検査ツールを使うことによって、生成系の生成、コンパイルの前に間違いを発見することができる。間違いを発見したときの検査ツールの表示を10図に示す。仕様記述中の非終端記号struct_decl_listの属性fieldを解析中に、トークン「:」のところでエラーを発見したことが分かる。また、出力文法中の非終端記号field_list_Bの１番目の文法記号として解析中にエラーが検出されたものであることも分かる。

4. 現状と今後の課題

Embryoの評価のために、RPCのインタフェースの定義からFortranの構造体（処理系の独自仕様のもの）を生成するプログラムを、Embryoを用いて作成した。このプログラムはFortranでRPCを使用する際に用いられる。このときに作成した仕様記述は約250行であった。

Embryoについては今後、より詳細な評価を行なう必要がある。具体的には、インタフェースプログラム生成系をすべて、Embryoを使って生成する

```
Syntax Error::field_list_B No.1
 Current parsing attribute = (struct_decl_list field)
 token = : (token No.= 58)
 Current_object = :
 Dumping stack state
 stack[0] = nil
 stack[1] = ((var (IDENTIFIER name) : (type_specifier type) ;))
 stack[2] = ((type_specifier type) ;)
 stack[3] = (type_specifier type)
 stack[4] = nil
 stack[5] = ((struct_or_union_specifier type))
 stack[6] = (struct_or_union_specifier type)
 stack[7] = nil
 stack[8] = ((struct_decl_list field) end )
 stack[9] = (struct_decl_list field)
 stack[10] = nil
 stack[11] = ( : (struct_decl_list field))
```

図10 仕様記述検査ツールの出力

予定である。

仕様記述検査ツールについては試作を行ない、簡単な仕様記述の検査が行なえるようになった段階である。

仕様記述検査ツールについては、現在の検査方法では仕様記述に関する制限が厳しすぎるので、この制限を緩和することが当面の目標である。特に、3.2節で述べた条件(2)は非常に厳しく、実際にこの検査ツールを用いる際には大きな障害になると予想される。また、この制限のもとでは、仕様記述を書く場合に、出力文法、入力文法に関する知識をかなり要求される。また、合成規則中に関数呼び出しが現れたり、条件分岐によって合成される文字列が変わるといった場合は扱えず、文字列と属性を連結するだけの簡単な合成規則のみについての検査が行なえるだけである。現状では、誤りがある時に、それを検出できる例を作ることができたという段階であり、任意の仕様記述の正当性を検査できるという段階ではない。本稿で述べた方式はLL構文解析をベースにして構文解析方法を構築しているが、LR構文解析をベースにした解析方法を開発することによって、仕様記述に対する制限を緩和することができると考えている。

5. まとめ

プログラム生成系の形式的仕様記述から、プログラム生成系を生成するツールを作成した。このツールを利用して開発を行なうと、従来に比べて仕様の記述性と理解しやすさを向上させることができた。

さらに、プログラム生成系の仕様記述を検査し、その仕様から生成される生成系の生成するプログラムが文法的な誤りを含む可能性の有無を検査するツールを試作し、仕様記述がある制限を満たせば、今回提案したアルゴリズムによって仕様記述の検査が行なえることを確認した。

今後、Embryoの実用性の評価と、仕様の検査を行なう際の仕様記述への制限を緩和し、実用レベルの仕様記述を検査できる方式の開発に取り組む予定である。

参考文献

[1] Micheal B. Jones, Richard F. Rashid, Mary R. Thompson, "Matchmaker: An Interface Specification Language for Distributed Processing", Procceding of the 12th ACM SIGACT-SIGPLAN Symposium on Princeples of Programming Languages, ACM, January 1985.

[2] 末広 他, "異機種分散処理環境におけるインタフェース自動生成処理方式（1）～（3）", 情報処理学会第41回全国大会講演論文集.

[3] Stephen C.Johnson,"Yacc: Yet another Compiler Compiler", ULTRIX-32 Supplementary Documents Volume 2 Programmers, Degital Equipment Corporation, pp 3-79 - 3-111.

[4] 佐々政孝, "プログラミング言語処理系", 岩波書店, pp136-144, 1989.