プログラミング言語　8－1
（1986. 10. 17）

コンパイラの代数的仕様記述と
その自動生成

酒井正彦[†]　坂部俊樹[††]　稲垣康善[†]
[†]名古屋大学工学部　[††]三重大学工学部

我々が提案したコンパイラの代数的仕様記述法は，コンパイラをソース言語の構文領域からターゲット言語の構文領域への関数とみなし，これらの二つの構文領域にコンパイリング関数，補助領域，補助関数を加えて拡張した抽象データ型として等式を用いて記述する方法である．
本論文では，この記述法に基づいて設計したコンパイラの仕様記述言語および現在作成中のコンパイラ自動生成系の基本概念および概要について述べる．
このコンパイラ生成系は，Ｕｎｉｘ上のツールであるＬｅｘとＹａｃｃ，および，抽象データ型直接実現システムＣｄｉｍｐｌｅの三つのツールと，仕様記述言語で書かれたコンパイラの仕様を各々のツールの入力に変換するためのプリプロセッサから構成されている．

An Algebraic Specification Method of Compilers and Their Automatic Generation

Masahiko SAKAI[†], Toshiki SAKABE[††] and Yasuyoshi INAGAKI[†]

[†] Department of Electrical Engineering, School of Engineering, Nagoya University, Furo-cho, Nagoya 464 JAPAN
[††] School of Engineering, Mie University, Kamihama-cho, Tsu 514 JAPAN

We have already proposed an algebraic specification method of compilers. In this method, we regard a compiler as the function from the syntactic domain of source language to that of target language and specify it by using equations as an abstract data type which is the two syntactic domains enriched with the compiling function, auxiliary domains and auxiliary functions.
In this paper, we propose the specification language based on the specification method and explain the basic concept and the outline of the automatic compiler generator which we are constructing.
The compiler generator consists of three tools (Lex and Yacc which are tools of Unix, Cdimple which is a direct implementation system of abstract data types) and the preproccessor used for translating to the respective inputs of these three tools from the specification written in our specification language.

1．　はじめに

プログラミング言語の代数的仕様記述法は，コンパイラ自動生成の理論的枠組みとして有望であり，近年，これを用いたコンパイラの仕様記述，仕様の検証，コンパイラの自動生成などの研究が盛んに行われ始めている．[1],[2],[3],[4],[5] これらの多くは，図１に示す図式に基づくアプローチ[1],[2],[3],[5] をとっているが，いずれも，コンパイラをソース言語の構文領域からターゲット言語の構文領域への準同型写像でとらえている．しかし，コンパイラの本質的な部分と考えられる関数呼び出しに伴う各種アドレス計算や変数のアロケーションなどを，準同型写像として記述することはできない．もし，準同型写像で記述したとすると，ターゲット言語の構文領域には，ソース言語の構文領域の各々の関数に対して，対応する関数（または，関数の合成）が一つ存在する．しかし，それでは，ソースプログラム中に現れうる変数は，コンパイラ設計の段階でアロケーション番地が決まっていることになる．

例えば，ＡＤＪグループの方法[3] は，ターゲット言語の構文領域からソース言語の構文領域と準同型な領域を抽出することにより，コンパイラを記述するものである．これは，デライバを用いた記述と実質的に同じであり，コンパイラは，ソース言語の構文領域からターゲット言語の構文領域への準同型写像とみなされている．そのため，上で述べた変数のアロケーション等の記述が困難になっている．

また，Gaudelらによって開発されたコンパイラ生成系PERLUETTE [1],[2],[5] では，ソース言語とターゲット言語の構文領域間の関係を直接記述するのではなく，図１の二つの意味写像と意味領域間の変換を準同型写像として記述している．その際，変数のアロケーション等の記述を可能にするために，メタ記号を導入している．しかし，これらのメタ記号は，ＬＩＳＰで実現された関数名であり，数学的な意味付けがなされておらず，形式性を損なう原因になっている．

そこで，我々は，ソース言語の構文領域とターゲット言語の構文領域の関係を，準同型写像として記述することをやめ，ソース言語とターゲット言語の二つの構文領域の拡張領域上の関数として与えることによって，上に述べた問題を解決し，これに基づいてコンパイラの代数的仕様記述法を考案した．更に，実際にＰＬ／０コンパイラを記述し，その妥当性を示した[6] ．本論文では，コンパイラの代数的仕様記述法と，それに基づいて設計した仕様記述言語，および，現在作成中であるコンパイラ生成系の基本概念と概要を述べる．

2．コンパイラの代数的仕様記述法

本記述法は，コンパイラをソース言語の構文領域からターゲット言語の構文領域への関数とみなし，その関数を等式によって仕様記述する方法である．ただし，二つの構文領域の他に補助の領域や関数を加えることができる．すなわち，図２に示すように，ソース言語，ターゲット言語の構文領域に補助ソート，補助関数，および，コンパイリング関数f_0を加えて拡張した抽象データ型（コンパイリング領域）としてコンパイラをとらえ，その抽象データ型を等式によって記述するものである．

その仕様は，大きく分けて三つの部分からなり，

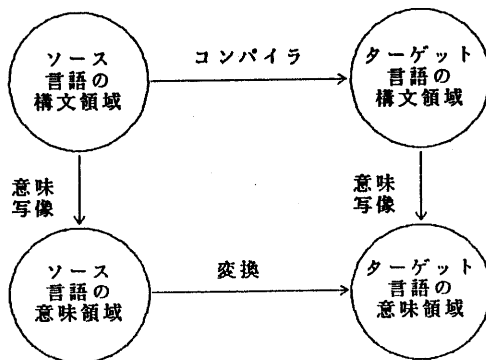

図１　従来の枠組み

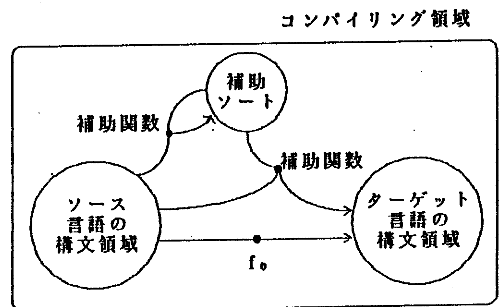

図２　本記述法の枠組み

その内の二つはソース言語とターゲット言語の構文領域の仕様であり，もう一つは，それらの二つの領域の拡張であるコンパイリング領域の仕様である．以下では，それらを順に定義する．　なお，代数的仕様記述法に関する概念および記法については，文献(7),(8)を参照されたい．

【定義1（構文領域の仕様）】
構文領域の仕様は，文脈自由文法

$G=\langle V_T, V, P, S_0\rangle$

である．ここで，

V_Tは終端記号の集合，
Vは非終端記号の集合，
Pは生成規則 $p: N\rightarrow\alpha$ の集合で，
　pは生成規則の名前，$N\in V$，
　$\alpha\in(V_T\cup V)^*$，
S_0はVの要素で開始記号

である．□

文献(9)に述べられているように，文脈自由文法Gをシグニチャ

$G=\langle V, P'\rangle$

とみなすことができる．ここで，

(1)ソートの集合は非終端記号の集合Vである．
(2)演算記号の集合P' は生成規則の集合Pから次のように定める．

$P'=\langle P'_{w,s}\rangle$, $w\in V^*$, $s\in V$
$P'_{w,s}=\{p \mid p: N\rightarrow\alpha\in P, s=N, w=nt(\alpha)\}$

ただし，$nt(\alpha)$は，αからVの要素のみを取り出してできる記号列である．

このようにシグニチャGを定めると，　構文領域$\mathcal{L}(G)$は次のように定義される．

【定義2（構文領域）】
構文領域$\mathcal{L}(G)$は，項代数$\mathcal{T}[G]$である．□

直観的には，構文領域は，生成規則の名前を木の節点とし，終端記号が取り除かれた構文木の集合である．

以上で述べた方法を用いて，ソース言語とターゲット言語の構文領域の仕様

$SG=\langle V_T^{SG}, V^{SG}, P^{SG}, S_0^{SG}\rangle$,
$TG=\langle V_T^{TG}, V^{TG}, P^{TG}, S_0^{TG}\rangle$

を記述すると，それぞれの構文領域$\mathcal{L}(SG)$，$\mathcal{L}(TG)$が定まる．これらの構文領域は，コンパイラの入出力を表す抽象データ型と考えられるので，コンパイラは，これら二つの領域間の関数であるコンパイリング関数としてとらえられる．本記述法では，この関数を記述するために，二つの構文領域を拡張した領域であるコンパイリング領域を導入する．この領域には，二つの構文領域とコンパイリング関数の他に，補助ソート，補助関数が追加されている．

従来の記述法では困難であった変数のアロケーションの問題は，補助ソートとしてのコンパイラの記号表の記述により，ソース言語の変数とターゲット言語の番地が対応づけが可能となるため解決できる．

【定義3（コンパイリング領域の仕様）】
コンパイリング領域の仕様$\mathcal{C}$は，6項組

$\mathcal{C}=\langle SG, TG, \Sigma, \mathcal{V}, \mathcal{A}, f_0\rangle$

である．ここで，

SGはソース言語の構文領域の仕様，
TGはターゲット言語の構文領域の仕様で，
　$V^{SG}\cap V^{TG}=\phi$，$P'^{SG}\cap P'^{TG}=\phi$，
Σはシグニチャ$\langle\mathcal{S}, \mathcal{F}\rangle$で，
　$V^{SG}\cup V^{TG}\subseteq\mathcal{S}$，$P'^{SG}\cup P'^{TG}\subseteq\mathcal{F}$，
$\mathcal{V}$は変数集合族，
$\mathcal{A}$は等式$\xi==\eta$の集合，
$f_0\in\mathcal{F}_{SS_0, TS_0}$はコンパイリング関数記号

である．□

ここで，仕様$\mathcal{C}$が定めるコンパイリング領域を抽象データ型として定義するのであるが，(1)ソース言語の構文領域およびターゲット言語の構文領域の構造を壊さないという条件（無矛盾性）と(2)これらの領域に余分な要素を加えないという条件（完全性）が必要である．

(1)構文領域$\mathcal{L}(G)$に対する無矛盾性
　$\xi, \eta\in T[G]_s$，$s\in V$に対して，
　$\mathcal{A}\vdash\xi==\eta$ならば$\xi=\eta$
(2)構文領域$\mathcal{L}(G)$に対する完全性
　$\xi\in T[\Sigma]_s$，$s\in V$ならば，
　$\mathcal{A}\vdash\xi==\eta$となる$\eta\in T[G]_s$が存在．

【定義4（コンパイリング領域）】
コンパイリング領域の仕様$\mathcal{C}$から定まるコンパイリング領域SD（$\mathcal{C}$）は仕様$\mathcal{C}$が，ソース構文領域

$\mathcal{L}$（SG），ターゲット構文領域$\mathcal{L}$（TG）に対して，完全性ならびに無矛盾性を満たすとき，商代数$\mathcal{T}[\Sigma]/\equiv$である．ここで，$\equiv$は項代数$\mathcal{T}[\Sigma]$上の合同関係$\langle \equiv_s \rangle$ $s \in \mathcal{S}$である．

$\xi, \eta \in \mathcal{T}[\Sigma]_s$について，

$$\xi \equiv_s \eta \quad \text{iff} \quad \mathcal{A} \vdash \xi == \eta \quad \square$$

直観的には，コンパイリング関数が，仕様記述されたコンパイラそのものを表している．すなわち，ソースプログラムのコンパイルは，ソースプログラムの構文木ξおよびコンパイリング関数記号f_0に対して次式

$$f_0(\xi) \equiv \eta$$

を満たすターゲットプログラムの構文木ηを計算することである．このηの存在は次の命題で保証される．

【命題1（ターゲット項の存在性）】

コンパイリング領域SD（$\mathcal{C}$）が定まるとき，コンパイリング関数記号f_0および任意の$\mathcal{L}$（SG）項ξに対して，

$$f_0(\xi) \equiv \eta$$

を満たす$\mathcal{L}$（TG）項ηが存在する．

<証明>　SD（$\mathcal{C}$）の$\mathcal{L}$（TG）に対する完全性により明らか．□

3．コンパイラの仕様記述言語

本節では，2節で述べたコンパイラの仕様記述法に基づいて設計した仕様記述言語について述べる．ただし，識別子や数字，文字，文字列などの基本的なものについても記述すると，記述が繁雑になるばかりでなく仕様の読解性を損ねる原因となる．そこで，これらについては既知のものと認め，わざわざ記述しないことにする．

この仕様記述言語の構文は，我々が以前に作成した言語開発支援システムLass（LAnguage development Supporting System）[10]の構文に準じている．Lassは言語の代数的仕様からその言語の処理系を自動生成するシステムであり，構文領域の仕様については，本仕様記述言語と互換性がある．

具体的には，specheadkeep部，source部，target部，sort部，oper部，axiom 部の六つの部分から構成されており，以下で順に説明する．　なお，注釈は，/* と */ で囲んで記述する．

3.1　spechead部

この部分は，仕様記述者の名前，作成日付，バージョンなどを記述する．現在のところ，シンタクスは定めておらず，書式は自由である．

3.2　source部

ここは，ソース言語の構文領域の仕様SGを記述する部分であり，　さらに，terminal部，prec部，nonterminal部，rule部の四つの部分からなっている．

(1)　terminal部では，終端記号の集合V_T^{SG}の要素を

　　正規表現　＝　終端記号　；

の形式で定義する．正規表現の記法はLex[11]に準じている．終端記号は，英字で始まる英数字，または，引用符で囲まれた一文字である．ここで，正規表現と終端記号が同じ場合には，終端記号を省略し，次の形式で記述する．

　　正規表現　；

また，注釈文は，その始まりと終わりを示す記号列を combegin, comend を用いて記述する．下に記述例を示す．

```
terminal{{
  begin ;   end ;
  '-' ;     '+' ;
  ":=" = ASSIGN ;
  "(*" : combegin ;
  "*)" : comend ;
}}
```

(2)　prec部では，終端記号の結合性と優先順位が指定できる．この指定は，数式に現れる演算子に対して行われるのが普通である．　この指定により，rule部で記述する生成規則数を減らすことができる．prec部の下方に書かれた演算子は，上方の演算子より優先順位が高く，コンマで区切られて並べて書かれた演算子は，等しい優先順位を持つ．コロンの後には，結合性（left, right, nonassoc のいずれか）が記述される．下に例を示す．

```
prec{{
  '+', '-' : left ;
  '*', '/' : left ;
  UNARY    : left ;
}}
```

ここで，UNARY が終端記号でない場合には，"%prec" という機能を用いて，同一の演算子が異なる順位を持つときの記述が行える．例えば，rule部において生成規則を次のように記述すると，演算子 '－' の

優先順位を局所的に変更することができる．

```
UMINUS: expr ::= '-' expr %prec(UNARY)
```

(3)　nonterminal部では，非終端記号の集合 V^{SG} の要素を次のように並べて定義する．

```
nonterminal{{
  program, statement, expr, ...
}}
```

(4)　rule部では，生成規則の集合 P^{SG} を記述する．生成規則は次の形式である．

　　生成規則名 ：　非終端記号
　　　　　　::= 終端記号または非終端記号の並び

ここで，rule部で最初に記述される生成規則の左辺の非終端記号を開始記号とする．また，非終端記号 int, id, char, string については，これらの構文は既知のものとして与えられているものとする．

3.3　target部

この部分では，source部と全く同様の形式でターゲット言語の構文領域の仕様TGを記述する．

3.4　sort部，oper部，axiom部

これらの部分では，コンパイリング領域の仕様の残りの部分，すなわち，シグニチャ Σ，変数集合族 $\mathcal{V}$，等式集合 $\mathcal{A}$，コンパイリング関数記号 f_0 を記述する．

sort部，oper部では，シグニチャ Σ のソートの内で，ソース，ターゲット両言語から定まるシグニチャ以外のシグニチャを次のように記述する．

```
sort{{
  symtab  , sym_attr
}}
oper{{
  COMP     : prog         -> t_prog
  COMP_PROG: prog, symtab -> t_prog
  ...
}}
```

ここで，oper部で最初に記述される関数記号をコンパイリング関数記号 f_0 とする．

axiom部では，等式集合 $\mathcal{A}$ を記述する．等式は次の形式とする．

$$\xi \;=\; \eta \;;$$

ここで，ξ と η は関数記号と変数で構成される項である．ただし，関数記号は，source部とtarget部中のrule部で記述された生成規則の名前と，oper部で記述された関数記号，および，システムでサポートされている関数名に限る．また，それ以外の名前は変数とみなされる．システムが用意する関数を次にあげる．

```
TRUE, FALSE, NOT, AND, OR, IF
ZERO, SUCC, PRED, ADD, DIFF, MULT, DIV,
MOD, EQ, NE, GT, LT, GE, LE
...
```

また，この言語による仕様記述例として，代入文の系列からなる言語をスタックマシンの命令系列に変換するコンパイラの仕様を図3に示しておく．

```
spechead{{
  compiler of simple language,
  target is stack machine,
  sakai,  ver-1.1,  19/Sep/86
}}

source{{
  terminal{{
    var           =  VAR ;
    ":="          =  ASSIGN ;
    '+';   ';';   ',';   '.';
    "(*"          : combegin ;
    "*)"          : comend ;
  }}

  prec{{
    '+'           :left;
  }}

  nonterminal{{
    prog, var_part, var_list, var_name,
    stm, stm_list, expr, id, int
  }}

  rule{{
    PROG:
       prog ::= var_part stm_list '.' ;
    VAR_PART:
       var_part ::= VAR var_list ';' ;
    VAR_LIST:  var_list ::= var_name ;
    VAR_LIST2:
       var_list ::= var_list ',' var_name ;
    VAR_NAME:  var_name ::= id ;
    STM:       stm ::= id ASSIGN expr ;
    STM_LIST:  stm_list ::= stm ;
    STM_LIST2:
       stm_list ::= stm_list ';' stm ;
    E_VAR:     expr ::= id ;
    E_CONST:   expr ::= int ;
    E_ADD:     expr ::= expr '+' expr ;
  }}
}}

target{{
  terminal{{
    lit          =  LIT ;
    opr          =  OPR ;
    lod          =  LOD ;
    sto          =  STO ;
    int          =  IINT ;
  }}

  nonterminal{{
    t-prog, inst_list, inst, int
  }}
```

図3　コンパイラの仕様記述例（その1）

```
rule((
  T-PROG:  t-prog ::= inst_list ;
  INST_LIST:
          inst_list ::= inst ;
  INST_LIST2:
          inst_list ::= inst_list inst ;
  I_LIT:  inst ::= LIT int ;
  I_OPR:  inst ::= OPR int ;
  I_LOD:  inst ::= LOD int ;
  I_STO:  inst ::= STO int int ;
  I_INT:  inst ::= IINT int int ;
 ))
))

sort(( tab ))

oper((
 CMP          : prog           -> t-prog ;
 CMP_PROG     : prog,tab       -> t-prog ;
 CMP_VAR_PART: var_part,tab -> inst_list ;
 CMP_VAR_LIST: var_list,tab -> inst_list ;
 CMP_VAR_NAME: var_name,tab -> inst_list ;
 CMP_STM      : stm,tab        -> inst_list ;
 CMP_STM_LIST: stm_list,tab  -> inst_list ;
 CMP_EXPR     : expr,tab       -> inst_list ;
 AUX_VAR_PART: var_part,tab -> tab ;
 AUX_VAR_LIST: var_list,tab -> tab ;
 AUX_VAR_NAME: var_name,tab -> tab ;
 APPEND_INST : inst_list,inst_list
                               -> inst_list ;
 INIT_TAB     :                -> tab ;
 SYM_OFF      : id,int,tab     -> tab ;
 INS_SYM      : id,tab         -> tab ;
 RET_OFF      : id,tab         -> int ;
 ONE, TWO     :                -> int ;
 UNDEF_INT    :                -> int ;
))

axiom((
 INS_SYM(id0,INIT_TAB())
 == SYM_OFF(id0,ZERO(),INIT_TAB());

 INS_SYM(id0,SYM_OFF(id1,int,tab))
 == SYM_OFF(id0,
            ADD(int,ONE()),
            SYM_OFF(id1,int,tab));

 RET_OFF(id0,SYM_OFF(id1,int,tab))
 == IF(EQ_ID(id0,id1),
       int,
       RET_OFF(id0,tab));

 RET_OFF(id0,INIT_TAB()) == UNDEF_INT();

 APPEND_INST(inst_list,INST_LIST())
 == inst_list;

 APPEND_INST(inst_list,
             INST_LIST2(inst_list1,inst))
 == INST_LIST2(
      APPEND_INST(inst_list,inst_list1),
      inst);

 CMP(prog) == CMP_PROG(prog,INIT_TAB());

 CMP_PROG(PROG(var_part,stm_list),tab)
 == T-PROG(
      APPEND_INST(
        CMP_VAR_PART(var_part,tab),
        CMP_STM_LIST(
          stm_list,
          AUX_VAR_PART(var_part,tab))));

 CMP_VAR_PART(VAR_PART(var_list),tab)
 == CMP_VAR_LIST(var_list,tab);

 CMP_VAR_LIST(VAR_LIST(var_name),tab)
 == CMP_VAR_NAME(var_name,tab);

 CMP_VAR_LIST(
   VAR_LIST2(var_list,var_name),
   tab)
 == APPEND_INST(
      CMP_VAR_LIST(var_list,tab),
      CMP_VAR_NAME(
        var_name,
        AUX_VAR_LIST(var_list,tab)));

 CMP_VAR_NAME(VAR_NAME(id),tab)
 == INST_LIST2(
      INST_LIST2(
        INST_LIST2(INST_LIST(),
                   I_INT(ONE())),
        I_LOD(ZERO())),
      I_STO(RET_OFF(id,INS_SYM(id,tab))));

 CMP_STM(STM(id,expr),tab)
 == INST_LIST2(CMP_EXPR(expr,tab),
               I_STO(RET_OFF(id,tab)));

 CMP_STM_LIST(STM_LIST(stm),tab)
 == CMP_STM(stm,tab);

 CMP_STM_LIST(STM_LIST2(stm_list,stm),tab)
 == APPEND_INST(
      CMP_STM_LIST(stm_list,tab),
      CMP_STM(stm,tab));

 CMP_EXPR(E_VAR(id),tab)
 == INST_LIST2(INST_LIST(),
               I_LOD(RET_OFF(id,tab)));

 CMP_EXPR(E_CONST(int),tab)
 == INST_LIST2(INST_LIST(),I_LIT(int));

 CMP_EXPR(E_ADD(expr0,expr1),tab)
 == INST_LIST2(
      APPEND_INST(CMP_EXPR(expr0,tab),
                  CMP_EXPR(expr1,tab)),
      I_OPR(TWO()));

 AUX_VAR_PART(VAR_PART(var_list),tab)
 == AUX_VAR_LIST(var_list,tab);

 AUX_VAR_LIST(VAR_LIST(var_name),tab)
 == AUX_VAR_NAME(var_name,tab);

 AUX_VAR_LIST(VAR_LIST2(var_list,var_name),
              tab)
 == AUX_VAR_NAME(
      var_name,
      AUX_VAR_LIST(var_list,tab));

 AUX_VAR_NAME(VAR_NAME(id),tab)
 == INS_SYM(id,tab);

 ONE() == SUCC(ZERO());

 TWO() == SUCC(ONE());
))
```

図３　コンパイラの仕様記述例（その２）

４．コンパイラ生成系

4.1　基本概念

２節の議論から，本記述法で記述されたコンパイラの仕様から生成されるコンパイラは次の三つのフェーズで構成されるのが自然である．

第一フェーズ：　ソースプログラムからソース言語の構文木ξを抽出し，項の形で出力する．

第二フェーズ：　第一フェーズで得られたξにコンパイリング関数f_0を適用してターゲット言語の構文木ηを得る．

第三フェーズ：　第二フェーズで得られたηからターゲットプログラムを構成する．

ここで，第一フェーズおよび第三フェーズは，構文解析と構文解析の逆変換であり，どちらも直構文変換ＳＤＴ（Syntax Directed Translation)として実現できる．

例えば，次のソース言語の生成規則

```
ASSIGN:  stm  ::=  id  ':='  expr
```

に対して，第一フェーズを行うＳＤＴの規則は次に示す生成規則の対になる．

```
stm  ::=  id  ':='  expr
stm  ::=  'ASSIGN('  id  ','  expr  ')'
```

すなわち，はじめの規則を用いて構文解析をすると同時に，二番目の規則で文字列を生成してゆくと，ソースプログラムの構文解析が終わった時点で構文木を表す項を得ることができる．

第三フェーズについても，第一フェーズと同様なＳＤＴ規則になる．ただ，ＳＤＴ規則の順番が逆になっている点のみが異なっている．例えば，ターゲット言語の生成規則

```
LODE:  inst  ::= 'lod'  int  ','  int
```

に対して，次のＳＤＴ規則を用いればよい．

```
inst  ::=  'LODE('  int  ','  int  ')'
inst  ::=  'lod'  int ','  int
```

以下では，残りの第二フェーズとその生成について議論する．

２節より，第二フェーズが行うべき仕事は，ソースプログラムの構文木である$\mathcal{L}$（ＳＧ）項ξを受け取り，コンパイリング関数記号f_0に対して，$f_0(\xi) \equiv \eta$をみたす$\mathcal{L}$（ＴＧ）項ηを計算することである．すなわち，等式集合$\mathcal{A}$を用いて項$f_0(\xi)$を書き換えてターゲット言語の構文木である$\mathcal{L}$（ＴＧ）項ηを求めればよい．

我々は，このフェーズを項書き換えシステム（ＴＲＳ）として実現する．すなわち，等式集合$\mathcal{A}$の等式を左辺から右辺への項書き換え（右向きの書き換え）規則とみなして，項ξを書き換えて得られる正規形をξが属する同値類の代表元と考える．ここで，正規形とは，それ以上，右向きに書き換えることのできない項である．

ここで問題となるのは，我々は計算結果として$\mathcal{L}$（ＴＧ）項ηが必要だということである．項ηの存在は命題１で保証されてはいるが，これがＴＲＳを用いて得られる正規形になるとは限らない．このため，仕様がある十分条件を満たすときηがＴＲＳの正規形になることを示す．

【定理１（ηの計算可能性）】　コンパイリング領域ＳＤ（$\mathcal{C}$）が定まるとき，等式集合$\mathcal{A}$が合流性を満たし，かつ，等式集合$\mathcal{A}$の全ての等式の右辺にのみターゲット構文領域の関数記号が現れるならば，コンパイリング関数記号f_0，$\mathcal{L}$（ＳＧ）項ξに対して，項$f_0(\xi)$を右向きに書き換えて得られる正規形は$\mathcal{L}$（ＴＧ）項である．

＜証明＞　右向きに書き換えて得られる正規形が$\mathcal{L}$（ＴＧ）項でないような項$\xi' = f_0(\xi)$が存在するとする．命題１により，ξ'を両方向に書き換えて得られる$\mathcal{L}$（ＴＧ）項ηが存在する．すなわち，両方向の書き換え系列$\xi', \xi_1, \xi_2, \ldots, \xi_n, \eta$が存在する．ここで，$\eta$を構成する関数記号は等式の左辺に現れないので，$\eta$は正規形である．　いま，$\xi_k$から$\xi_{k+1}$への書き換えが最後に用いられた左向きの書き換えとすると，ξ_{k+1}を右向きに書き換えるとξ_kとξ_{k+2}が得られるので，合流性により，ξ_kとξ_{k+2}から右向きの書き換えのみを用いて同一の項η'が得られる．　ところが，正規形ηは，ξ_{k+2}から右方向の書き換えのみを用いて得られるため，再び合流性を用いると，η'を右向きに書き換えるとηが得られる．したがって，左向きの書き換え回数が一つ少ない書き換え系列を作ることができる．よって，帰納法により，右向きのみの書き換え系列$\xi', \ldots, \eta$が存在し，$\mathcal{L}$（ＴＧ）項ηは正規形なので，矛盾が生ずる．□

4.2　コンパイラ生成系の実現

第二フェーズの生成は，抽象データ型直接実現シ

ステムＤｉｍｐｌｅ[12]のＣ言語版Ｃｄｉｍｐｌｅ[13]を用いて実現できる．これは，抽象データ型の仕様，すなわち，等式集合をＴＲＳとみなして実現するシステムである．ただし，Ｃｄｉｍｐｌｅは，その入力となる等式集合が次の条件をみたすことを要求している．したがって，コンパイラ生成系の入力であるコンパイラの仕様もこれらの条件をみたさなければならない．

(1)等式 $f(\xi_1, \ldots, \xi_n) == \eta$ の左辺の引数 $\xi_1, \ldots, \xi_n$ が正規形である．

(2)合流性

(3)有限停止性

(4)線形性

以下では　第一，第三フェーズの生成について述べる．

4.1節の議論より，第一フェーズと第三フェーズの生成には，ＳＤＴプログラムを生成すればよい．これには，Ｕｎｉｘ上のツールであるＬｅｘ[11]とＹａｃｃ[14]が利用できる．例えば，4.1節で例として示したＳＤＴ規則

```
stm  ::=  id  ':='  expr
stm  ::=  'ASSIGN('  id  ','  expr  ')'
```

は，次のような形式で，それぞれＬｅｘとＹａｃｃに入力することによって実現できる．

```
Lex:
  ":="      return(ASGN);

Yacc:
  stm  :  id  ASGN  expr
    = $$ = concstr("ASSIGN(",$1,",",$3,")");
```

ここで，$$，$1，$3はそれぞれ生成規則の左辺，右辺の一番目，右辺の三番目の非終端記号の属性変数である．また，concstr は文字列の連接を行う関数である．

我々は，以上で述べた考え方に基づいて，現在，コンパイラ生成系を作成中である．その構成は図４に示すように，Ｌｅｘ，Ｙａｃｃ，Ｃｄｉｍｐｌｅの三つのツールとプリプロセッサからなっている．ＬｅｘとＹａｃｃは，上で述べたようにＳＤＴプログラムの生成に用いている．Ｃｄｉｍｐｌｅは，抽象データ型直接実現システムで，抽象データ型の仕様をＣプログラムとして実現する．プリプロセッサは，　コンパイラの仕様を，　Ｌｅｘ，Ｙａｃｃ，Ｃｄｉｍｐｌｅの各々のツールの入力形式への変換を行っている．

５．まとめ

本報告では，コンパイリングをソース言語の構文領域からターゲットの構文領域への関数とみなして，自然かつ単純な枠組みを用いたコンパイラの代数的仕様記述法を考案した．また，具体的な仕様記述言語を定め，それに基づくコンパイラ生成系の基本概念を示した．

さて，本方法における仕様の正当性は，従来の図式（図１）の可換性として定式化できる．我々は，ＰＬ／０サブセットのコンパイラに対して，既に，この証明を行っている．[15]

<謝辞>

日頃御指導賜る豊橋技術科学大学本多波雄学長，名古屋大学福村晃夫教授，並びに御討論下さる阿曽弘具助教授，北英彦さん，直井徹さんをはじめとする研究室の皆様に感謝致します．なお本研究は，一

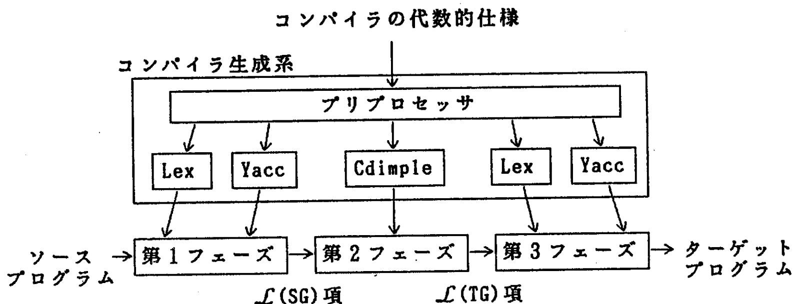

図４　コンパイラ生成系の構成

部，文部省科研費（一般(c) 課題番号60550263，特定研究(1)多元知識情報課題番号61102003）および，倉田奨励金の援助による．

＜文献＞

(1) Gaudel M. C. : Specification of Compilers as Abstract Data Types Representations, Proc. Workshop on Semantics-Directes Compilers in Aarhus, LNCS94, pp140-164, (1980)
(2) Deschamp P. : PERLUETTE : A Compiler Producing System using Abstract Data Types, Proc. 5th Coll in Programming, LNCS137, pp63-77, (1982)
(3) Thatcher J. W., Wagner E. G., Wright J. B. : More on Advice on Structuring Compilers and Proving them Correct, Proc. Workshop on Semantics-Directed Compilers in Aarhus,LNCS94, pp165-188, (1980)
(4) Mosses P. : A Constructive Approach to Compiler Correctness, Proc. Workshop on Semantics-Directed Compilers in Aarhus,LNCS94, pp165-188, (1980)
(5) Despeyroux J. : An Algebraic Specification of PASCAL Compiler, SIGPLAN Notices, Vol.18, No.12, pp34-48, (1983)
(6) 酒井，北，坂部，稲垣：コンパイラの代数的仕様記述法，電子通信学会，技術報告SS86-9, (1986)
(7) 稲垣，坂部：抽象データタイプの代数的仕様記述の基礎(1) ー多ソート代数と等式論理，情報処理，Vol.25, No.1, pp47-53, (1984,1)
(8) 稲垣，坂部：抽象データタイプの代数的仕様記述の基礎(2) ー抽象データタイプ， 情報処理，Vol.25, No.5, pp491-501, (1984,5)
(9) Thatcher J. W., Wagner E. G. Wright J.B.: Initial Algebra Semantics and Continuous Algebra, JACM, Vol.24, No.1, pp68-95, (1977)
(10)酒井，北，坂部，稲垣：プログラミング言語の代数的仕様記述からのコンパイラ自動生成，電子通信学会，技術報告AL85-10, (1985)
(11)Lesk M.E. : LEX - A Lexical Analyzer Generator, Comp. Sci. Tech. Rep., No.39, Bell-Lab., (1975)
(12)川辺，坂部，稲垣，本多：抽象データ型の直接実現システム， 電子通信学会，技術報告AL83-65, (1984)
(13)酒井，坂部，稲垣：抽象データ型直接実現システム Cdimple，日本ソフトウエア科学会，関数的プログラミング研究会，(1986,11発表予定)
(14)Johnson S. C. : Yacc - Yet Another Compiler-Compiler, Comp. Sci. Tech. Rep., No.39, Bell-Lab., (1975)
(15)酒井，坂部，稲垣：コンパイラの代数的仕様の検証，電気関係学会東海支部連合大会，(1986,10)