プログラミング言語 21－2
（１９８９．６．１３）

変換系記述言語ＴＤＬの表現力とその記述例

栗野　俊一

早稲田大学理工学部

変換系を記述する為に必要な表現力について考察を行ない、その表現力を限定した記述形式を提案する。また、この様に表現力が制限された記述形式上で記述された変換系の正当性について述べる。文脈自由文言語から文脈自由文言語からへの変換系を記述するための記述形式として変換文法が知られているが、これは、プログラミング言語の変換系を記述する為には不十分である。そこで、この変換文法に対して必要な拡張を行ない、変換系記述言語ＴＤＬを開発した。このＴＤＬで記述された変換系の検証も変換文法と同程度に容易に示すことができる。

Power of Expression of The Translator Description Language TDL and its Samples

Syun'ich Kurino

School of Science and Engineering, Waseda University, 3-4-1, Okubo Sinjuku-ku, Tokyo 160, Japan

In this paper, the power of expression necessary to describe a translator is considered, and a descriptive language which is restricted in its power of expression is proposed. Next, correctness of a given translator description in this language is shown.

A translator from CFL (Context Free Language) to CFL can be expressed in TG (Transduction Grammar). But TG is too less powerful to express a translator for conventional programming language. Therefore translator description language TDL is developed by extension of TG. Correctness of a translator in TDL is verified almost as easy as in TG.

第１章　はじめに

プログラミング言語の処理系はソフトウェアを作成する為に使われる最も重要なツールの一つである。例えば、新しいパラダイムの下に設計された言語を用いて、プログラムを作成することによって、ソフトウェアの開発が効率よく行なえるようになった例も少なくない。このために、言語の処理系を作成する為の技術もまた重要であり、特にパーサを作成する手法は、形式的言語の研究成果を取入れ、コンパイラ・コンパイラ、あるいは、パーサ・ジェネレータなどの形で、広く利用されている[1]。

我々の作成したＴＤＬ[2]もまた、処理系を作成する為のツールの一つであるが、処理系の記述専用にすることによって、表現能力を押え、その代償として、変換系の正当性など、変換系そのものの性質が検証し易くなっている。

本論文では、はじめに変換系を記述する為に必要な表現力について述べ、次に、これに基づいた変換系記述言語ＴＤＬについて説明する。そして、ＴＤＬで記述された変換系の具体例等を紹介する。

第２章　変換系を記述する為に必要な表現力

２．１　変換系と変換系記述言語

変換系に要求される性質は次の２つである。

①　変換系が意味を保存する。つまり入・出力言語に対して与えられた解釈のもとで、変換系が出力するストリングの意味が、入力されたストリングの意味が等しい。
②　変換系の定義域が入力言語に一致する。

これらの性質を満たすような変換系を記述する為に、変換系記述言語は次の様な要求をみたす必要がある。

①の性質は、変換系の記述者が保証しなければならない。しかし、入・出力言語の解釈が与えられたときに、変換系の記述者が①の性質を検証する為の仕組を用意する必要がある。

②の性質は、変換系の定義域が入力言語を含んでいることと、入力言語が変換系の定義域を含んでいることの二つを示す必要がある。ここで前者の方が変換系の性質としては本質的であるが、工学的な立場からいえば、人間の誤り検査が重視される為に、後者の性質によって、入力言語に含まれないストリングを定義域として含まず、除外できることが重要になる。

また、変換系記述言語で記述可能な変換系の範囲が目的としている変換系を含むほど十分に広い必要がある。

２．２　対象とする変換系の範囲

変換系を記述する為に必要な表現力は、対象とする変換系の範囲によって定まる。ここで、ＴＤＬの目標は現在利用されているようなプログラミング言語間の変換系の記述である。したがってＣＦＬ[3]を真に含むような言語クラスに属する言語間の変換系が対象となる。ここでは、まず初めにＣＦＬからＣＦＬへの変換系の記述言語に関して述べ、さらにそれを拡張した形としてＴＤＬを述べる。

２．３　ＣＦＧ上の変換系

入力言語のクラスがＣＦＧであれば、入力言語の文法を与えることによって、その認識機械をＰＤＡ（Push Down Automaton）の形で実現することができる[3]。ここで、入力と出力の関係を変換系の記述者に把握しやすくする為に、入力の構文規則に対応して、出力言語の出力形式、つまり、出力言語の構文規則を記述する。この様な形式で変換系を記述したものが、ＴＧ（ Transduction Grammer ）である[4]。

そこで、ＣＦＧ上の変換系の記述言語として、ＴＧを選び、その記述能力やＴＧで記述された変換系の正当性について述べる。

２．４　ＴＧ

以下、このＴＧに関しては文献[4]に、他の記号に関しては文献[3]に基づいて説明する。

定義　ＴＧ（Transduction Grammar）

ＣＦＧ　Ｇ＝（Ｎ，T_i，Ｐ，Ｓ）に基づいたTransduction Grammar G_tは、３つ組（Ｇ，T_o，ｇ）で、次の性質をみたすものである。

T_oは出力言語のアルファベット

ｇはＧのプロダクション規則から（$T_o \cup N$）*への写像で、ｇ（Ａ→w_i）＝w_oならば、w_oと、w_iに現れる非終端記号（Ｎの要素）の種類と数が一致する。

定義　ＳＤＴ（Syntax Directed Translation）

関数ＦがＴＧ　G_tによって定義される言語Ｌ（Ｇ）からT_o^*への関数であるとき、ＦをG_tによって導かれたＧの Syntax Directed Translation であるという。

ＳＤＴの記述例（算術式から逆ポーランド形式）

Ｓ→Ｅ	{Ｅ}	Ｔ→Ｐ	{Ｐ}
Ｅ→Ｅ+Ｔ	{ＥＴ+}	Ｐ→(Ｅ)	{Ｅ}
Ｅ→Ｔ	{Ｔ}	Ｐ→ｉ	{ｉ}
Ｔ→Ｔ*Ｐ	{ＴＰ*}		

２．５　ＴＧの表現能力

ここでは、簡単な例を用いて、ＴＧの表現能力を述べる。表現能力は、一つのＳＤＴを次の様に文字列の集合を記述していると考え、ＴＧで記述可能な言語のクラスを他のクラスと比較することによっておこなう。

定義　ＳＤＴ　Ｆの表現する言語

ＴＧ　G_t＝（（Ｎ，T_i，Ｐ，Ｓ），T_o，ｇ）によって導かれたＳＤＴ　Ｆを次の様な文字列を認識する述語あるいは、文字列の集合として、解釈する。

$$P_T(x\,\$\,y) \equiv F(x) = y$$

$$S_T = \{x\,\$\,y \mid F(x) = y\}$$

性質　S_Tのクラスは、ＣＦＬのクラスを含んでいる。

なぜなら、任意の入力ｘに対して

$$F(x) = \varepsilon$$

とすれば、

$$S_T = L(G)\ \{\$\}$$

となるからである。

性質　S_Tのクラスは、ＣＦＬのクラスを真に含んでいる。

次のＴＧを考える。

Ｓ→０１　　{２}

Ｓ→０Ｓ１　　{Ｓ２}

このＴＧによって導かれる関数は、次の様な集合を記述していると考えることができる。

$$\{0^N1^N\$\,2^N \mid N>0\}$$

この集合は、ＣＦＬではないことが知られている為、S_TのクラスがＣＦＬのクラスを真に含んでいるといえる。

性質　S_Tのクラスは、ＣＳＬを含んでいない。

まずはじめに、ＴＧの定義から、ＳＤＴ　Ｆの定義域及びに値域は、ともにＣＦＬになることに注意する。そこで、次の様な集合

$$\{0^N1^N\$\,2^N3^N4^N \mid N>0\}$$

を考えると、これは、ＣＳＬであるが、S_Tのクラスには含まれないことが解る。なぜなら、S_Tのクラスの要素は＄の左右が共に、ＣＦＬでなければならず、この例では右側の$2^N3^N4^N$がＣＦＬでないからである。

２．６　ＳＤＴの正当性の検証

ＳＤＴの形で変換系を記述した場合、変換系の記述者は、変換系が意味を保存していることの検証を次の様に行なうことができる。

今、ＣＦＧ　Ｇ＝（Ｎ，T_i，Ｐ，Ｓ）に対して、（Ｎ，T_i，Ｐ）を$G^\#$で表わす。これは、スタートシンボルを与えるとＣＦＧになるので、Ｎからの関数と解釈することができる。例えば、$G^\#(S) = G$である。さらに、G^*を次の様に定義し、Ｇの構文領域と名付ける。

$$G^* \equiv \cup L(G^\#(x))$$

$x \in N$

さらに、入力言語のＣＦＧをG_i、出力言語のＣＦＧをG_oとする。この時、ＴＧ　G_t＝（G，T_o，g）の正当性は、次の手順で示される。

①　G_i^*とG_o^*に共通な意味領域Ｍを設ける。

②　G_i^*とG_o^*に対してそれぞれＭへの解釈M_i、M_oを用意する。ここで、解釈M_iは、α，βがG_i^*の要素で、しかも$\alpha\beta$がG_i^*の要素であれば、$M_i(\alpha\beta) = M_i(\alpha)\,M_i(\beta)$が成立するとし、$M_o$も同様であるとする。

③　G_tのｇによって定められる対応

$$A \rightarrow w_i \qquad \{w_o\}$$

にたいして、それぞれ、

$$M_i(w_i) = M_o(w_o)$$

を示す。ただし、w_iに含まれる非終端記号Ｂに対しては、

$$M_i(B) = M_o(B)$$
$$\updownarrow$$
$$g(B \rightarrow v_i) = v_o$$
$$M_i(v_i) = M_o(v_o)$$

とする。

この様にＴＧあるいはＳＤＴは、ＣＦＬ上の変換系を記述するのに適した性質をもつが、現在利用されているプログラミング言語は、幾つかの点で、ＣＦＬのクラスに入らないことが知られている。例えば定義機能がそれである。定義を用いているため、未定義な記述と定義されている記述の間には区別が必要になるが、文脈を考慮に入れなければ、それを区別することができない。そこで、この様な問題を解決する為に、ＴＧに対して幾つかの拡張を行なうと共に、上記の様な性質を保存することを目的として考えた言語が変換系記述言語ＴＤＬである。

第３章　変換系記述言語ＴＤＬ

３．１　ＴＧとＴＤＬ

ＴＧとＴＤＬの関係を示す為に、ＴＧで紹介した例をＴＤＬで記述しなおし、以下に示す。

```
s(⟨S⟩) = s(⟨E⟩) = e(⟨E⟩) .
e(⟨E0⟩) = e(⟨E⟩ '+' ⟨T⟩) = e(⟨E⟩) t(⟨T⟩) '+'.
e(⟨E⟩) = e(⟨T⟩) = t(⟨T⟩).
t(⟨T0⟩) = t(⟨T⟩ '*' ⟨P⟩) = t(⟨T⟩) p(⟨P⟩) '*'.
t(⟨T⟩) = t(⟨P⟩) = p(⟨P⟩).
p(⟨P⟩) = p('(' ⟨E⟩ ')') = e(⟨E⟩).
p(⟨P⟩) = p(⟨I⟩) = integer(⟨I⟩).
```

ＴＧからＴＤＬへの変換規則を一般的に述べると次の様になる。

①　非終端記号に対してそれぞれ一意な関数名を用意する。

②　ＴＧに現れる非終端記号は、“（＜”と、“＞）”で挟み、さらに、①で定めた関数名を前に付加する。

③　ＴＧに現れる終端記号は、“’ ”と、“’ ”で挟む。

④　ＴＧのメタ記号“：：＝”，“｛”，“｝”はそれぞれＴＤＬのメタ記号“＝”，“＝”，“．”に書換える。

この規則にしたがって作成されたＴＤＬの記述は書換える前のＴＧの記述と同じものである、したがって、ＴＧで記述可能な変換系はＴＤＬを用いても記述することが可能である。

３．２　ＴＤＬのＴＧに対する拡張機能

ＴＤＬでは、ＴＧに対して次の様な二つの機能を拡張している。

①　入力言語のクラス拡大するために、シンボルテーブル処理記述を可能にし、入力に対する、より厳しいチェックを行なえるようにした。

②　出力言語のクラス拡大するために、出力文字列の位置の指定のを可能にし、出力に対する、より柔軟性のある制御記述を行なえるようにした。

①の機能は、あらかじめ用意された組込の関数（ここでは、“put_table”と“check_table”）を、形式的にダミーの構文要素として挿入することによって実現する。

例　変数宣言とその確認

```
var_decl(⟨var_decl⟩)
=       var_decl( ⟨type⟩ ⟨name⟩ )
=       type( ⟨type⟩ )
```

```
        name( 〈name〉 )
        put_table( 〈name〉, 'variable' ) .
        ....
variable( 〈variable〉 )
=       variable( 〈name〉 )
=       name( 〈name〉 )
        check_table( 〈name〉, 'variable' ) .
```

ここで、記述“put_table(〈name〉,'variable')”は、入力が“name(〈name〉)”の関数“name”と同じで、定義域はT_1^*全体、出力はε、で副作用として、“〈name〉”対応する文字列を“'variable'”として登録する事を意味する。一方、記述“check_table(〈name〉, 'variable')”は、入力が“name(〈name〉)”の関数“name”と同じで、定義域は“'variable'”としてテーブルに登録された文字列の集合、出力はε、で副作用はない事を意味する。この例から解るように、“〈name〉”は二つの非終端記号を表わす関数“name”と“put_table”あるいは“check_table”とが同じ値を引数とすることを表現するに用いられている。この二つの関数を用いることによって、例えば宣言されていない識別子の参照を含むような記述を除外することができる。

また、この目的の為に、これらとは他に、テーブルに対する取消しや、ブロック構造を認識するための組込み関数が用意されている。

②の機能を実現するため、まずメタ記号“,”を用意し、関数の値を文字列の二つ組の形に拡張する。次に、一方をこれまでの関数の値として扱い、他方を特別なキーワード“define”を付加することによって取り出すことにする。

例　拡張ＢＮＦにおける繰返しの除去

```
repeat(〈repeat〉)
=       repeat( '{' 〈var〉 '}' )
=       new_name( 〈name〉 )
        name(〈name〉)
,       /* 二つ組の区切り子 */
        name(〈name〉) '::='
                text(〈var〉) name(〈name〉)
                        '|' 'ε' .
```

ここで、関数“repeat”に対して入力が“{A}”であったとするとこの関数の値は“V”と“V ::= AV | ε”の二つ組である。ここで、前者の“V”はもとの拡張ＢＮＦには含まれていない変数名で、組込みの関数“new_name”によって作成される。後者はその変数を導出する為のプロダクション規則である。もし、後者が出力されるＢＮＦに含まれるならば、“V”と“{A}”は同じ意味になるので、“{A}”を“V”に置き換えることによって拡張ＢＮＦから繰返しが除去できる。しかし、そのためには、後者の“V ::= AV | ε”を適切な位置に挿入する必要がある。そこで、関数“repeat”の記述では、“V”と“V ::= AV | ε”別々に求めている。この後半の部分を出力するには、“define”を関数の前に付加する。

例　キーワード“define”の使用法

```
line( 〈line〉 )
=       line( 〈var〉 '::='
                〈var1〉 〈repeat〉 〈var2〉 )
=       define:repeat( 〈repeat〉 )
        var( 〈var〉 ) '::='
                var( 〈var1〉 )
                repeat( 〈repeat〉 )
                var( 〈var2〉 )
```

これによって、関数“line”に対する入力が“X ::= Y {A} Z”であれば、出力は、“V ::= AV | ε　X ::= YVZ”となる。

この二つの拡張はいずれも言語における定義あるいは宣言の記述および、それらの参照に関連している。①の拡張は識別子に対する参照が行なわれた場合に、それがすでに定義されているかどうかを確認する為に導入された機能である。一方、②の拡張は出力言語の定義機能を変換系が利用する場合に必要になる機能である。

3．3　ＴＤＬの表現能力

ＴＧと同様にしてＴＤＬの表現能力を定義する。これによって次のことがいえる。

性質　ＴＤＬのクラスは、ＴＧのクラスを含んでいる。（定義より明らか）

性質　ＴＤＬのクラスは、ＴＧのクラスを真に含んでいる。

入力言語のクラスあるいは出力言語のクラスが広いことを示せばよいので、ここでは、入力言語のクラスが広いことを示す。

入力言語を次の言語であるとする。

$\{\alpha \% \alpha \mid \alpha$は英字列$\}$

ＴＤＬでは次の様に記述することができる。
ただし、出力言語は入力言語と同じで、変換系は恒等関数を記述している。

```
s((S))
=       s( ⟨Alpha⟩ '%' ⟨Beta⟩ )
=       alpha( ⟨Alpha⟩ ) '%' beta( ⟨Beta⟩ ).
alpha( ⟨Alpha⟩ )
=       alpha(⟨Letters⟩)
=       letters( ⟨Letters⟩ )
        put_table( ⟨Letters⟩, 'occuer' ).
beta( ⟨Beta⟩ )
=       beta(⟨Letters⟩)
=       letters( ⟨Letters⟩ )
        check_table( ⟨Letters⟩, 'occuer' ).
```

（“letters”は英文字列上の恒等関数）

しかし、この言語はＣＦＧではないので、ＳＤＴでは表現できない。

3．4　ＴＤＬによる変換系の正当性

ＴＤＬで変換系を記述した場合も基本的にはＳＤＴと同様に行なうことができる。しかし、ＴＤＬのＳＤＴから拡張された部分に関しては、注意する必要がある。これは、ＴＤＬで記述された変換関数の定義域を入力言語の意味関数の定義域が、あるいは、変換関数の値域を出力言語の意味関数の定義域が含むことをそれぞれ調べなければならないからである。

①入力言語に関する拡張機能が使用されている場合。

ここで、問題となるのは、組込関数“check_table”が利用されている場合である。この関数が表われた場合、それは通常、この関数を用いて定義した変換関数の定義域を制限している。ここで、入力言語の構文領域に対する意味関数の定義域がこの変換関数の定義域を覆っているかどうかを調べる必要がある。

②出力言語に関する拡張機能が使用されている場合。

もし、この拡張機能が出力言語の定義機能を利用する為に用いられているならば、つまり、変換関数が返す値が常に出力言語の定義記述と、その定義された識別子の参照の二つ組の形であれば、それぞれ、出力言語の意味関数の定義域にあると考えることができる。しかし、それ以外の使用法が行なわれた場合は検証が難しい。したがって、この出力言語に関する拡張機能は、出力言語の定義機能を利用する為にだけに利用すべきで、将来的には、それ以外の使用を何等かの形で制限しなければならないと思われる。

第４章　変換系の記述例

ＴＤＬで記述された変換系は、変換系生成系を用いて prolog のプログラムに変換される。このプログラムは、あらかじめ用意されたレキシカルアナライザや、組込関数を表現している prolog のプログラムと結合して用いる。このＴＤＬで記述した変換系は、現在のところ、ＴＤＬ、Ｚ[5]、そしてＷＳＮ[6]からそれぞれ prolog への変換系の３点であるが、ここではＴＤＬから prolog への変換系（変換系生成系）について述べる。

ＴＤＬから prolog の変換は、生成された変換系の速度効率を考慮しなければ、次の様な簡単な規則で行なうことができる。

１）　ＴＤＬの関数名に対して、 prolog の述語名を一意に対応させる。

２）　ＴＤＬの非終端記号に対して、 prolog の

変数名を一意に対応させる。

3）　ＴＤＬの関数定義に対して prolog の述語定義を次の様に対応させる。

3－1）先頭の部分は①，②にしたがって書換える。

3－2）最初の“＝”は、“：－”に書換える。

3－3）次の部分は、関数名を２引数の述語“append_list”に書換える。この述語の一つ目の引数は先頭部分に現れた変数で、二つ目の引数は、関数の引数を、 prolog のリストの形にしたものである。

3－4）２番目の“＝”は“＆”に書換える。

3－5）その次の部分は、各要素の間に“＆”を追加しながら、次の様に書換える。

3－5－1）非終端記号の部分は①，②にしたがって関数名を述語名に、非終端記号は変数名に書換える。

3－5－2）終端記号はの部分は“（”と、“）”で挟み、組込の述語名“echo”を前に付加する。

3－6）“．”はそのままに出力する。

ただし、この書換え規則を用いて、作成した prolog の述語に対しては、入力の文字列をリストの形で与える必要がある。

この変換規則はＴＤＬを用いて次の様に記述できる。

```
tdl((tdl))
= tdl( { (define) } )
= { define( (define) ) } .
define((define))
= define( (head) '=' (inp) '=' (out) '.' )
= head((head)) ':-'
        inp((inp)) '&'
        out((out)) '.'.
head( (head) )
= head( (func) '(' '(' (var) ')' ')' )
= name( (func) ) '(' variable( (var) )  ')'
        put_table( (var), 'head_var' ).
inp((inp))
= inp( (func) '(' (term_seq) ')' )
= void:name( (func) )
  'append_list' '(' get_table( 'head_var' )
',' '{' term_seq( (term_seq) ) '}' ')'
term_seq( (term_seq0) )
= term_seq( (term0) | (term1) (term_seq1) )
= term( (term0) ) |
  term( (term1) ) ',' term_seq( (term_seq1) ).
                ...
```

ここで、この記述から得られる変換系は、拡張部分に関する記述がないため、必ずしも、入力言語であるＴＤＬの全てを覆っているわけでもない。逆に、文脈依存な条件を守っていないような正しくない記述も何等かの無意味なものに変換してしまう。このことは、この変換系の記述からＴＤＬの構文規則の部分を取り出し検査して得られる。しかし、この変換系に対して正しいＴＤＬの記述を入力することによって正しく動作する変換系を得ることができる。これは、次の様に確かめられる。

今、ＴＤＬで記述された関数Ｆ（Ｓ→ｗ｛ｖ｝）と、それから変換された述語Ｐに対して、

Ｆ（ｘ）＝ｙ　　≡　　Ｐ（ｘ）
　　｛副作用としてｙを出力｝

を示す。

①　ｗが終端記号であるとき、

```
f( (S) )
=       f( 'w' )
=       'v' .
p( S ) :-
        apppend_list( S, {'w'} ) &
        echo( 'v' ).
```

この時は、述語“apppend_list”によって、変数“S”の値と“{'w'}”が一致しなければならない為、述語“p”が真になるのは、入力が“{'w'}”の時だけである。そしてこの時、出力として“'v'”が得られる。

②　ｗに非終端記号が現れるとき

```
f( (S) )
=       f( (A)(B) )
```

```
=       fb(⟨B⟩) fa(⟨A⟩) .
p( S ) :- apppend_list( S, {A,B} ) &
          pb( B ) &
          pa( A ).
```

この時は、述語“pa”，“pb”と関数“fa”，“fb”の間は、求める関係を満たしていると仮定する。すると、述語“apppend_list”によって、変数“S”の値と変数“A”と“B”の値を結合したものが一致し、変数“A”と“B”の値が述語“pa”、“pb”によってチェックされらる為、述語“p”が真になるのは、入力が“f”の定義域になる時だけである。そしてこの時、出力として述語“pa”，“pb”の出力、つまり、関数“fa”，“fb”の値の結合が出力として得られる。

我々が現在使用しているＴＤＬの変換系は、上記の記述をさらに拡張する一方、効率の点から、重リストを用いる等の改良を加えたものである。ＴＤＬで変換系を記述する場合には、この様にプロトタイプを作成し、その正当性を検証しつつ、機能の拡張を進め、目的とする変換系を完成させることができる。

第５章　おわりに

問題の対象となる領域が限られていれば、その領域専用のプログラミング言語を作成することによって、プログラムの記述量や、その言語に対する処理系が容易に実現できる。

ＣＦＬ上の変換系の記述では、ＴＧが有効であると考えられるが、これは現在のプログラミング言語の変換系を記述するに不十分である。ＴＤＬは、その様な点を補おうとして考えられた。

その結果として、幾つかのプログラミング言語間の変換系が記述可能であることが解った。また、変換系の正当性の検証における容易さも、幾らかは劣化したものの同様な手法が適用できることが確認できた。これは、ＴＤＬを利用することの利点であると考えられる。

しかし、このＴＤＬが現在、あるいは将来的に要求される能力をもっているかどうか、あるいは逆に、不用意に用いた場合に、余りにも能力がありすぎて、対象を限ることによって得られる利点が失われている可能性もある。

したがって今後、ＴＤＬに記述力に関する考察を進め、その限界を明らかにすると共に、変換系記述に必要な能力の拡張と不必要な能力の制限を行なう予定である。

謝辞

本論文は、早稲田大学情報科学研究教育センター「ソフトウェア開発における仕様記述あるいは検証の実用性に関する研究」部会の研究成果の１つであり、この研究活動に参加あるいは支援してくたさった方々に心より感謝致します。

参考文献

[1] S.C.Johnson, "Yacc:Yet Another Compiler Compiler", Computing Science Technical Report No.32, Murray Hill, 1975.

[2] 尹　他：仕様記述言語向きの変換記述言語ＴＤＬ，情報処理学会第３７回全国大会，1988.

[3] John E. Hopcroft and Jeffrey D. Ullman, "Introduction to Automata Theory, Languages and Computation", Addison-Wesley Publishing Co. Inc. Reading, Massachusetts, U.S.A., 1979.

[4] Lewis, P. M., II, and R.E.Stearms, "Syntax-Directed Transduction", J. ACM 15:3, 465-488, 1968.

[5] J.M.Spivey, "Understanding Z", Cambridge Tracts in Theoretical Computer Science 3., 1988.

[6] 河野他：ＷＳＮによるスケジューラの詳細仕様記述とその経験，Bulletin of Centre for Informatics, Waseda Univ., Vol.5, 1987.