

MLTG—マイクロプログラミング・ランゲージ・トランスレータ・ジェネレーター: LALR パーサの一つの応用例†

牧之内 顕 文†† 若 木 利 子††† 毛 利 友 治††

MLTG (A Microprogrammig Language Translator Generator) システムについて論じる。このシステムはマイクロプログラミング用言語のトランスレータ生成システムである。対象となる言語は低レベルだが、

1) プログラマにとって書き易く、2) 保守する人にとって読み易い
自由型式の高水準言語風アセンブラ言語である。このような言語の使用によってプログラミングの生産性が向上し、保守の仕事が容易になると期待される。

このシステムのねらいは、各々のハード開発チームが自分達の言語を設計し、容易にそのトランスレータを開発できるようにすることである。このシステムの使用によって、ハードの設計変更が及ぼす言語の変更にも容易に対応できる。このシステムは機能的に二つのサブシステムに分けられる。一つは LALR(3) 構文解析プログラム生成サブシステム、他はマイクロ命令生成サブシステムである。‡

ユーザが対象言語の構文を柔軟にかつ自然に記述できる種々の機能 (拡張 BNF 記法、アクションの指定、文脈条件指定など) が用意されている。対象言語に独立な構文誤り回復ルーチンも組み込まれている。§

対象言語のレベルに関係づけながら、マイクロ命令生成サブシステムがサポートする諸機能についての議論がなされる。

1. ま え が き

計算機開発の一手段としてマイクロプログラムの概念が発表され、IBM が 360 ファミリーシリーズで大的に使用してから、その重要性、有効性は増大してきた。マイクロプログラムの使用対象が拡大すると共に、ファームウェアで OS のサポートがなされるようになると、マイクロプログラムを書く量は増えてきている。これはとりもなおさず、マイクロプログラムの作成と保守が重要な問題となってきたことを意味する。

現在マイクロプログラムは、主に計算機開発グループが計算機のハードウェア開発と並行して作成し、他の大型、超大型計算機上でシミュレートして、作成したマイクロプログラムが正しく動くことを確認する。

マイクロプログラムを開発し、シミュレートする人は、ハードウェアの設計変更に従うという別の問題を負わされることになる。

開発グループが直面している上記問題の解決の一助となるマイクロプログラミング支援システムの一部として、筆者らは、マイクロプログラミング言語トラン

スレータ生成システム(Microprogramming Language Translator Generator) を作成したので報告する。

マイクロプログラミング言語は、マイクロプログラムの性質上、コンパイラ言語のような高水準化は難しい。しかしシステム記述用言語がオペレーティングシステム作成支援の良好な道具であるように、

1) プログラマにとって書き易く、

2) 保守する人にとって読み易い (すなわち、プログラムが何をしようとしているかが分り易い)

低レベル*だが自由形式の高水準言語風言語 (G. R. Lloyd 等²⁾によればレベル 2 の言語) はマイクロプログラム作成の生産性と保守性を上げると期待される。

一つの標準的マイクロプログラミング言語を決めて、すべてのマイクロプログラム開発グループが使用するとしても、マイクロプログラマブルプロセッサの標準化をしない限りトランスレータは個々に作成しなければならない。一方、対象機械のマイクロ命令形式の異同は、言語の低レベル性の故に言語の構文 (syntax) に反映せざるを得ない。このような考察から筆者らはトランスレータ記述システム (Translator Writing System) 又はコンパイラ-コンパイラ (Compiler-Compiler) の技術を応用することにした。

マイクロプログラムで扱うデータ変数は、通常のプログラムと違って、ユーザが自由に定義して使用するというわけにはいかない。それらは既にハードウェアで定められたレジスタ等である。又それらにはあらか

† MLTG—A Microprogrammig Language Translator Generator by AKIFUMI MAKINOCHI, TOSHIKO WAKAKI, and TOMOHARU MORI (Department of Computer Science, Fujitsu Laboratories Ltd.).

†† (株)富士通研究所電子研究部第二研究室

††† 現在は富士通国際情報科学研究所

* ソース文と対象マイクロ命令が一对一に対応している。

じめ名前が付けられており、マイクロプログラムのプログラムはその名前を予約語のようにして使う。従って、マイクロプログラミング言語では通常のプログラミング言語では許されている一般的なデータ宣言(従ってコンパイラからみればデータの割り付け機能)は不必要である。

このことは、前述した MLTG 対象言語の低レベル性と共に、MLTG のトランスレータ記述システムとしての機能と構造、特に対象コード生成機能*と、その機能を実現するシステムの構造を非常に単純化している。

MLTG は、従って、構文解析プログラム(parser)発生システムのマイクロプログラミング言語トランスレータへの応用として位置づけられる。この応用という面からみると、類似のシステムとして MDS¹⁾がある。一方、構文解析プログラム発生システムとしてみた場合(例えば文献 9)), MLTG は 1) 構文記述上の工夫、2) 構文誤り処理機構という二点で特徴がある。

MLTG は二つのサブシステムからなる。一つは構文解析プログラム発生システム(Parser Generator)、他は対象機械のマイクロプログラム命令列発生システムである。MLTG の入力形式を変形 BNF で示せば次のようになる。

```
MLTG={FACILITY-SECTION}
      {FIELD-SECTION}
      SYNTAX-SECTION
```

但し〔 〕はオプションを表わす。

FACILITY-SECTION 及び FIELD-SECTION は SYNTAX-SECTION の構文記述に埋め込まれたアクション(action)と共に、マイクロ命令列発生のための入力である。これらはオプションなので、MLTG を純粋の構文解析プログラム発生用にも使用できる。以下ではこの二つの機能を別々に論じる。又構文解析プログラム生成部分の説明には読者になじみの深い ALGOL や PL/I 風のコンパイラ言語を例にとって説明する。なお語彙解析(Lexical Parser)についてはこの論文では触れない**。

* MLTG が用意している機能で足りない部分は、通常の(PL/I 等で書かれた)プログラムルーチンで補えるようにしてある。

** MLTG では語の構造は標準化されている。例えば名標(identifier)は英文字で始まる 16 文字以下の英数字列である。以下、\$で始まる記号は語解析ルーチンで認識される一つの単位語である。例えば \$ID は名標である。

*** precedence を満足するように構文を記述するのは骨が折れる。例えば Graham 等の論文⁴⁾に載っている例を見よ。

2. 構文解析プログラム生成システム

2.1 構文解析(parsing)の方法と構文記述

形式文法理論に基づいた表駆動構文解析の方法はトップダウン型とボトムアップ型に大きく分かれるが、我々は解析速度の観点からボトムアップ型をとった。この型の代表例は LR(k) と precedence によるものがあるが、

- 1) 言語設計者が比較的 naturally 構文記述でき***,
- 2) 解析速度を速くできる(例えば文献 3)を参照)

ことを考慮して LR(k) 構文解析法を採用した。

この方法の難点は表が大きくなることだが、Knuth⁵⁾以来、Korenjak⁶⁾, DeRemer⁷⁾ を経過して、一応実用の域に達した。実際、Lalonde⁸⁾, 小島⁹⁾ 等の実現の報告がある。我々が採った実現方法も DeRemer 流儀であり、先き読み記号列(Lookahead string)の計算の仕方を除いて Lalonde のものに似ている。すなわち我々のシステムが受け入れる文法は LALR(3)である。但し後述するように構文規則に条件をつけることができる。

構文解析プログラム生成システムで重要な点は3つあると思われる。

- 1) 対象言語の構文を容易にかつ自然に書き下せること。これは BNF 記号を拡張して、括弧によるくりくりとくり返し記号の導入によりこれを実現している(LR 解析法を選んだ理由もこれにある.)。又構文規則に条件を付与することによって文法の矛盾(そのために LALR(3)に収まらない)、あいまいさを取り除くことが可能になる。これも又構文記述の自然さを助ける。
- 2) 意味付けルーチン(アクション)と構文との対応づけを容易にすること。これは、言語設計者は意味づけを意識しないでまず構文を書き下し、その後、意味づけのためのアクションを構文に対応づけるという一般的過程を易しくするために必要な工夫である。
- 3) 構文誤り処理ルーチンも出力構文解析プログラムに組み込むこと。

2.2 入力文法の記述形式(1)

以下では 2.1 節の 1), 2) で述べたことを MLTG ではどう実現したかについて、主に例をもって説明する。例中の @ は空記号(null string)である。

- 1) 構文規則の基本的記法は BNF である。

例 1

```

SYNTAX SECTION; /*SYNTAX EXAMPLE*/
->BLOCK*,
BLOCK  ->"BEGIN"
        DCLS /*DATA DECL.*/
        STMS /*STATEMENTS*/
        LABELS /*LABELS*/
        "END",
DCLS   ->@ /*NULL PRODUCTION*/
        | DCLSEQ ";",
DCLSEQ ->DECLARE
        | DECLARE ";" DCLSEQ,
STMS   ->@
        | STMSEQ,
LABELS ->@
        | LABELSEQ,
LABELSEQ ->$ID ":"
        | $ID ":" LABELSEQ,
STMSEQ ->STM
        | STM ";" STMSEQ
STM     ->BLOCK
        | goto statement
        | stop statement
        | assignment statement
        | if statement

```

図 1 MLTG による構文記述例

Fig. 1 An example syntax for MICRO-ALGOL-LIKE language.

2) 括弧によるくり出しの導入

例 1 の BLOCK を括弧を使って書き直すと図 2 の例 2 のようになる。

例 2

```

BLOCK ->"BEGIN"
        (DCLSEQ "; " | @)
        (STMSEQ | @)
        (LABELSEQ | @)
        "END"

```

図 2 括弧によるくり出し

Fig. 2 Syntax description using parentheses.

3) くり返し記法の導入

$R(n[:m])(\alpha)$ (但し n, m は整数, α は文法要素) は α が n 回以上 m 回 (m が省略されたら無限回) くり返されることを意味する。例 1 の DCLSEQ, LABELSEQ をこの記法で書き直すと例 3 のようになる。

例 3

```

DCLSEQ ->DECLARE R(0) (" ;" DECLARE)
LABELSEQ ->R(1) ($ID ":" )

```

図 3 くり返し記法の導入

Fig. 3 DCLSEQ and LABELSEQ in Fig. 1 are rewritten with repeat notation.

* $\rightarrow$ の左の空白は出発記号 (start symbol) とみなされる。

** この理論的考察は文献 14) 参照。

*** この例は 15), 16) の文献で研究された LALR(k) パーサであまいな文法を解析する方法では扱えない。以下の例は両文献で述べられている方法でも可能。なお文脈条件で文法を規定する試みは、文献 17) にもある。著者等の方法は LALR データベース作成時に簡単に組み込めるのが特徴である。(詳細は文献 14))

4) アクションの位置は構文規則の右辺中任意である。これは 2.1 節の 2) での考察の結果でもあり、又 BNF の拡張の自然の結果でもある。今アクションを “ $\langle$ ” と “ $\rangle$ ” でくくったものとして表現すれば、例 3 は、例えば次のように書ける。

例 4

```

DCLSEQ ->DECLARE <act 1>
        R(0) (" ;" DECLARE <act 2>): <act 3>
LABELSEQ ->R(1) ($ID <act 4> " :"): <act 5>

```

図 4 アクションの挿入

Fig. 4 Embedding actions in the production value showed in Fig. 3.

上記の一連のアクション act 1, ..., act 5 の起動の契機は明らかであろう。

2.3 入力文法の記述形式 (2)

構文生成規則 (production rule) の左辺に、その規則が文生成時に適用される時に満足されねばならない条件を書くことができる。例**によってこれを示そう。

例 1 にその一部を示した MICRO-ALGOL-LIKE 言語の文法は、実は LALR(3) ではないのである。それは、例えば “BEGIN FIN 1: FIN 2: FIN 3: FIN 4: END” という語列が入力されたとき BEGIN を走査した後で、STMS $\rightarrow$ @ を認識すべきか、STMS $\rightarrow$ STMSEQ の認識を目標として入力語列の走査を続けるべきかの決定が、次に続く語列からは判断できないことになる。そこでこの矛盾 (これは先読み記号列の衝突 (conflict) として現われる) を解決するために STMS $\rightarrow$ @ の替りに STMS $\downarrow$ (\$ID) $\rightarrow$ @ とする。これは規則 STMS $\rightarrow$ @ が適用されるためには END の前にラベルがあってはならないことを意味する。従って、もし MICRO-ALGOL-LIKE 言語にこの構文規則を導入すれば、上記の入力語列は構文規則違反になる。これは、実行文のない (従って飛び越し文がない) ブロックにはラベルは要らないという判断から正当化されるだろう。

以上は、2.1 節の 1) の後半で述べた文法の矛盾を簡単にとり除くために文脈条件を使用した例である***。この条件は文法のあいまいさを除くためにも効果を発揮する。

例 5 は + と * の演算記号をもった算術式を表現する文法であるが生成のあいまいさがある。そのあいまいさを除くために文脈条件をつけると例 6 のようになる。

例 5

```

->EXPR,
EXPR ->EXPR "+" EXPR,
EXPR ->EXPR "*" EXPR,
EXPR ->$ID ,/*IDENTIFIER*/
EXPR ->("EXPR");

```

図 5 あいまいな文法

Fig. 5 An ambiguous grammar for the arithmetic expression with operators + and *.

例 6

```

->EXPR,
["+", "*"] EXPR["*"]
->EXPR "+" EXPR,
["*"] EXPR
->EXPR "*" EXPR,
EXPR ->$ID,
EXPR ->("EXPR");

```

図 6 文脈条件によってあいまいさの消えた文法

Fig. 6 An unambiguous grammar for the same arithmetic expression as the one in Fig. 5.

この文法では普通の算術式の文法と違って単純生成構文規則 (single production) がないので、式の解析速度の向上が実現できる。

2.4 LR 表の生成と構造

図 7 に MLTG システムの全体像を示した。実線は制御の流れを、二重線はデータの流れを示している。意味付け用情報と構文解析用情報とは BNF&ACTGEN (BNF & ACTION GENERATOR) で分離される。構文規則はそこで基本 BNF 型式に変換される。基本 BNF で表現された文法から、CFSMGEN (Characteristic Finite State Machine GENERATOR) によって CFSM に変換される。この時、構文規則に左文脈条件が付与されていれば適当な処理を行う。この条件の存在は出力される CFSM の形態には何ら影響がない (左文脈条件は、それが付与されている構文規則が CFSM の適当な部分の形成に参加できるか否かの条件を与えているに過ぎない)。次いで、DPDAGEN (Deterministic Pushdown Automator GENERATOR) がその CFSM

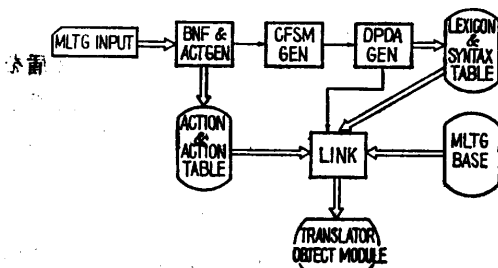


図 7 MLTG システムの全体像

Fig. 7 Overall structure of MLTG.

を DPDA に変換しファイルに出力する。この時、DPDAGEN は主に二つの処理を行う。一つは、前方参照記号列の計算であり、他は、右文脈条件 (もしあれば) の処理である。右文脈条件は不適当状態 (Inadequate state) に付与された前方参照記号列の集合からとり除くべき記号列を指定していると解釈される。我々が MLTG で採用した DPDA (LR 表) のデータ構造、CFSM から DPDA への変換方式は Lalonde が用いたもの¹⁰⁾と類似のものである (もちろん、文脈条件の処理は Lalonde にはない)。但し CFSM から DPDA への変換方式の内、前方参照記号列の計算の仕方は少しく異なる。文献 10) で記述された方式は、

- 1) CFSM の不適当状態の前方参照記号列集合を SLR(*k*) (Simple LR(*k*)) の方法で計算する。
- 2) 前方参照記号列集合が排反 (disjoint) にならない場合に、それを局所化する (局所化前方参照記号列の計算)。

我々は 1) を省いて最初から局所化前方参照記号列を計算している。一般に、文法が SLR(*k*) に収まる場合でも、局所化した方がしない場合より計算の結果得られる記号列の数は少ないので LR 表を小さくできる利点と、前方参照記号列計算アルゴリズムの一貫性が保てる利点がある。以上定義なしで使用した用語については文献 3), 9) 又は 13) 等を参照されたい。なお付録 I に LR 表の例を示しておいた。

2.5 構文誤りの処理

形式文法に基づく構文解析における構文誤りの訂正 (correction) と回復 (recovery) の研究は、理論と実際との両面から行われている (例えば文献 3), 4), 11) など) が、構文誤り処理の方法を考えるにあたって我々がとった立場は、日頃 PL/I 等のコンパイラ言語を使用しプログラミングしているプログラミング実務家 (practitioner) としての経験から導かれたものである。それは、構文誤り処理の目標はプログラム中に存在する構文誤りをできるだけ多く検出することであって、誤りを訂正 (挿入、置換、削除) するのも、それ以後の構文解析をよりよく続行するためということである。どれだけでもっともらしく訂正したかは余り問題にしない。

次に、我々の誤り処理方式は LR 構文解析法 (LR parsing) に依存したものであって、より一般的方法の LR 構文解析法への応用といった性質のものではない。

第 3 に強調すべきことは、誤り処理のコストをでき

るだけ小さくすること。これは、誤り処理の追加が誤りのないプログラムの解析速度に何ら影響を及ぼさないことはもちろんのこと、そのために LR 表の作成コスト(時間及び大きさ)が余り高くなって困るということである。MLTG では第3の点の始めの条件は満足できたが、二番目の条件はさけられなかった。しかしこの点も実際的には問題はないと考えられる。

以下、方法と実験結果について述べる。

2.5.1 方法

始めに LALR 構文解析用 DPDA の動き方を説明する。DPDA は DPDA 遷移表 (LR 表) とプッシュダウン・スタックからなる。DPDA の現在の状態は $(S_0S_1\cdots S_n, S_m, \underline{ax}, \alpha)$ で表現される。 $S_0S_1\cdots S_n$ はスタックの内容であり、 S_n が先頭である。 S_m は DPDA の次の動きの決定に参加する LR 表の現在の状態である。 $\underline{ax}$ は入力語列であり (a は一つの語、 x は語列)、下線は a が入力語列の現在の走査語であることを示す。 α は出力語列であり、普通は認識された構文規則の番号の列である。

MLTG の LR 表には四つの状態—R, L, A 及び E—があり、さらに R, L, A の三状態はプッシュダウンされるべき状態—PR, PL 及び PA—とプッシュダウンされない状態—R, L 及び A—に分かれる。R は Read, L は Lookahead, A は Apply の頭文字をとった短縮形である。E は End 状態で、この状態で DPDA の動作は終了する。

各状態には特有のリストが付随している。R(PR)には RNL (Read and Next List) で、これはこの状態が期待する語とそれを読み込んだ後に遷移すべき次の状態の対のリストである。L(PL)には LNL (Look and Next List) で、これは期待する (k 個の) 語列と、その語列を見て次に遷移すべき状態の対のリストである。A(PA)には、ANL (Apply and Next List) で、これはプッシュダウン・スタックからポップアップすべき内容の個数、ポップアップした後のスタックの先

頭として期待する状態、その後に遷移すべき次の状態及び出力規則番号の四項対リストである。

現在の DPDA の状態を $(S_0S_1\cdots S_n, S_m, \underline{a_1a_2}\cdots a_kx, \alpha)$ とする。次の状態は下のようなアルゴリズムで決まる。(付録 I を参照)

- 1) S_m が R(PR) のとき、 $(a_1, S_p) \in \text{RNL}^*$ ならば $(S_0S_1\cdots S_n(S_m), S_p, \underline{a_2}\cdots a_kx, \alpha)$ 。(但し (S_n) は S_m が PR ならばプッシュダウンされていることを示す。以下同様) そうでなかったら、誤り。
- 2) S_m が L(PL) のとき、 $(a_1a_2\cdots a_k, S_p) \in \text{LNL}^*$ ならば、 $(S_0S_1\cdots S_n(S_m), S_p, \underline{a_1a_2}\cdots a_kx, \alpha)$ 。そうでないなら誤り。
- 3) S_m が A(PA) のとき、もし $(q, S_{n-q}, S_p, \#k) \in \text{ANL}^*$ ならば $(S_0S_1\cdots S_{n-q}(S_m), S_p, \underline{a_1a_2}\cdots a_kx, \#k\alpha)$ 。そうでないなら誤り。

上記 1), 2) 及び 3) で誤り状態になった時、誤り処理ルーチンは次のことを行う。

挿入: 1) 及び 2) の誤り状態のとき。

挿入可能な語 b を求め、 a_1 の前に挿入して状態 $(S_0S_1\cdots S_n, S_m, \underline{ba_1}\cdots a_kx, \alpha)$ から正常解析を再開する。但し挿入可能な語 b とは $(S_0S_1\cdots S_n, S_m, \underline{ba_1}\cdots a_kx, \alpha) \Rightarrow (S_0S_1\cdots S_n(S_m), S_p, \underline{(b)a_1}\cdots a_kx, \alpha)$ で $(S_0S_1\cdots S_n(S_m), S_p, \underline{(b)a_1}\cdots a_kx, \alpha)$ が上記 1), 2) ** あるいは 3) で誤り状態とならないような S_p が存在するような語***である。但し上記状態記述中 S_m, b の存在は S_m の状態の種類による。

削除****: 挿入に失敗した場合又は上記 3) で誤りが検出されたとき。

スタック中にある LR 表状態 S_i , まだ走査されていない入力語列中の語 b で次の条件を満たすものを見つける: $S_i, ***** \in \{S_i | \text{LR 表の中に、ある A 又は PA 状態 } S_j \text{ が存在して、} S_j \text{ に付随する ANL に対して } (*, *, S_i, *) \in \text{ANL}\}$ でかつ $(S_0S_1\cdots S_i, S_p, \underline{by}, \alpha)$ が上記 1), 2) あるいは 3) で誤りにならないような S_p が存在する。但し $*$ は don't care を意味するものとする。

そのような三項対*****が見つかったら状態 $(S_0S_1\cdots S_i, S_p, \underline{by}, \alpha)$ から正常解析を再開する。但し by は $a_1\cdots a_kx$ の右端から b の左までの語を削除した後の入力語列である。

* このような対は存在するとすればただ一対に限られる。

** 実用上、誤り状態がどうかの条件は「 $(a_1y, S_p) \in \text{LNL}$ なる S_p が存在する」にしてある。但し y は $k-1$ 個の任意の語列とする。

*** このような語 b は一般には複数個ある。そのような場合、最初に見つかった語をとる。

**** 挿入、削除のくり返して無限ループにおち入る可能性があるため、ループ検出ルーチンがこの他に必要である。

***** CFMSM で非終端記号遷移が出ている状態。

***** このような三項対は一般に複数個ある。実際上は入力語列の先頭から一番近い b とそれに対応する S_i を探している。なおそのような三項対は必ず存在する。

この構文誤り処理法のコストは、A及びPA状態に、これがなければ不必要な前方参照記号列表を加えたこと（これは挿入可能な語を見つけるために必要）である。これはLR表の作成の時間と作成されたLR表の大きさの両方に悪影響を及ぼす。なお構文解析プログラムそのものも、誤り処理のない場合の約2.5倍のステップ数になった。但し誤り処理ルーチンの存在は誤りのない入力の場合の構文解析速度には何ら影響がないことは当然である。

2.5.2 実験例

付録Ⅱの図Ⅱ-1に構文誤り処理の能力を試すために使った極めて単純なPL/Iのサブセットの文法を提示してある。これは商用化されたPL/Iコンパイラの構文誤り処理結果との比較の便宜のためである。図Ⅱ-2は構文誤りのあるPL/Iプログラムの例である。メッセージERROR1は“DCL ID1 BIN FIXED”の後に“;”を挿入したことを示している。そのために次の“ID2 BIN FIXED;”は誤りとされ削除されている（メッセージERROR2からERROR4まで）。図Ⅱ-1の文法から明らかなように“;”の代りに“,”を挿入語として選べば、この語列は誤りとはされなかっただろう。これは挿入、削除に当たって「虫眼鏡」しか参考にしない構文誤り処理法の欠点である。

この例に関しては、我々が使用した商用PL/Iコンパイラの誤り処理より「より多くの誤りを検出する」点で優れていたことを書きそえておく（そのPL/Iコンパイラは誤りがあると大抵は次の“;”まで削除してしまう）。

3. マイクロ命令列生成用機能

MLTGが対象とするマイクロプログラミング言語のレベルは、ソース言語の一個の文(a sourcestatement)が一個のマイクロ命令に置き換わるレベルである。この制限はMLTGの意味付け言語機能の設計を非常に簡便化した。すなわち、MLTGでマイクロ命令のビット列を生成するために必要な機能としては、

- 1) マイクロ命令のフィールド分け*の機能
- 2) 言語の構文に依存したアクションで、フィールドへビット列の代入が行える機能

があればよい。

マイクロプログラムでは、対象機械のファシリティ

の名前、例えばREGISTER 1, REGISTER 2, 等々、がちょうど普通のプログラムでの変数と同様に使われるが、両者の違いは、前者ではトランスレータごと（機械ごと）に定まっていることである。このことは、それらファシリティ名がマイクロプログラミング言語でのキー語(key word)としての役割をはたしていることを意味する。一方それらは、マイクロ命令内では特定のビット列によって表現される。これから、三番目の機能として

- 3) ファシリティ名とそのビット列表現との結合機能

が便利である。

MLTGでは1)はFIELD-SECTIONで実現される。ここではマイクロ命令の名前、長さ、フィールドの名前、長さ、位置等をPL/I様の構造体で宣言できる。2)はSYNTAX-SECTIONのアクションで実現される。アクションでは、

- 2-1) ビットの代入文（フィールドにビット値を代入）
- 2-2) 条件文(IF.....THEN.....[ELSE]....., これはフィールド値のテストの結果によって他のフィールドに値を選択的に代入するのに使用)
- 2-3) ブロック(DO;END;., これは文のグループ化のため)
- 2-4) メッセージ出力文

を書くことが可能である。最後に、3)はFACILITY-SECTIONに用意されている。

なお、MLTGで用意されている意味記述機能だけでは不足な場合、アクション内に外部手続きの呼び出しが書ける。図8に実用例の一部を掲げた。

4. むすび

MLTGはマイクロプログラミング言語用トランスレータ記述システムとして設計された。その主要部分は構文解析プログラム生成システムである。一方モジュール化をはかることにより、このシステムとしてのみMLTGを使うことも可能にしてある。MLTGをその見地から見ると、ユーザが構文規則を柔軟にかつ自然に書くための色々の工夫がなされている。それはBNFを拡張して括弧づけを許していること、くり返し記号の導入、アクションの場所の任意性、文脈条件の付与に集約される。一方、実際の構文誤り処理ルーチンをくみ込み、一応満足できる結果を得たことはLALR構文解析法の一層の現実性をうらづけたも

* マイクロ命令は通常複数のフィールドに分けられ、ビット値の意味はそれが格納されているフィールドに依存する。

```

MLTG      FACILITY SECTION ;
          SOURCE LISTING
          STMT RUL.NO
          FACILITY SECTION ;
          2      WR0      (B*000*) ;
          3      WR1      (B*001*) ;
          4      WR2      (B*010*) ;
          5      WR3      (B*011*) ;
          6      WR4      (B*100*) ;
          .
          .
          .
          238     BYTEARITHOR (B*11*)
          FIELD SECTION ;
          240     1      COMMON .
          241     2      OPCODE      (1:4) ;
          242     2      DUMMY1      (5:5) ;
          243     2      WRI         (6:8) ;
          .
          .
          .
          344     2      WRJ      (14:16) ;
          345     SYNTAX SECTION ;
          346     1      -> MICROPROGRAM : <CALL MPROG ;>
          347     2      MICROPROGRAM -> R(0) (NOTENDSTMT ";" ) ENDSTMT ";"
          348     6      ENDSTMT      -> "END" : <CALL ENDQI ;>
          349     7      NOTENDSTMT   -> MICROSTMT
          : <CALL MICRO :
          COMMON.OPCODE = B*0* ; /*
          COMMON.DUMMY1 = B*0* ; /* CLEAR /*
          COMMON.WRI = B*0* ; /* MICROINST /*
          COMMON.DUMMY2 = B*0* ; /* FIELD. /*
          COMMON.WRJ = B*0* ; /*
          | QUASISTMT : <CALL QUASI ;>
          350     9      MICROSTMT     -> MICROLABEL NLBLMICROSTMT

```

図 8 MLTG を使った実用マイクロプログラミング言語記述例の一部

Fig. 8 An example of language description using MLTG.

のといえる。

他方, MLTG を当初の目的のシステムとしてみた場合対象言語レベルを低くおさえたことが意味付け言語の簡単化*に寄与して, 上述した諸特徴とあいまって MLTG を実際に使える (operational) ものにしている。

なお, 構文解析用テーブルのきめ細かな最適化は MLTG の第一版では考慮していないが, それをくみ込めば出力テーブルの大きさは現在のものよりかなり小さくできる見通しを得ている。これが今後の課題である。

謝辞 電子研究部長宮川達夫氏, 同部第二研究室長林達也氏, 東工大教授井上謙蔵氏の方々には, この研究上, 色々とお世話になりました。ここに記して深謝の意を表します。

* マイクロプログラムの番地付けは, マイクロプログラムの翻訳の後のステップで行われるのが現実的であるという立場をとっている。それは番地付けは機械ごとに大きく変化があること, グローバルな情報が必要とされることからである。これも又この簡単化に大きく寄与している。

参 考 文 献

- 1) E. W. Dubbs et al.: A Microprogram Design System Translator—An Introduction—, COMPCON '71.
- 2) G. R. Lloyd et al.: Design Consideration for Microprogramming Language, SIGMICRO Newsletter, Vol. 5, No. 1 (1974).
- 3) A. V. Aho and J. D. Ullman: The Theory of Parsing, Translating and Compiling, Prentice-Hall, Englewood, N. J. (1972).
- 4) S. L. Graham and S. P. Rhodes: Practical Syntactic Error Recovery, Commun. ACM, Vol. 18, No. 11 (1973).
- 5) D. E. Knuth: On the Translation of Language from Left to Right, Inf. Control, Vol. 8 (1965).
- 6) A. J. Korenjak: A Practical Method for Constructing LR(k) Processors, Commun. ACM, Vol. 12, No. 11 (1969).
- 7) F. L. DeRemer: Practical Translation for LR(k) Languages, Project MAC MIT (1969).
- 8) W. R. Lalonde: An Efficient LALR Parser Generator, Technical Report CSRG-2, Univer-

sity of Tronto (1971).

- 9) 小島富彦, ほか: LR(k)-parser 生成システムとその Fortran コンパイラへの応用, 情報処理, Vol. 15, No. 2 (1974).
- 10) 牧之内顕文: LR(k) 言語アナライザのもう一つの実際の構成法—SLR(k) の拡張, 情報処理, Vol. 17, No. 8 (1975).
- 11) G. Lyon: Syntax-directed Least-error Analysis for Context-free Languages; A Practical Approach, Commun. ACM, Vol. 17, No. 1 (1974).
- 12) 牧之内顕文, 沢井利子, 毛利友治: MLTG; マイクロプログラミング言語用 TWS, 昭和 51 年度情報処理学会全国大会.
- 13) 牧之内顕文, 沢井利子, 林 達也: マイクロプログラミング言語 FML とそのトランスレータについて, 昭和 49 年度情報処理学会全国大会.
- 14) 牧之内顕文: On Single Production Elimination in Simple LR(k) Environment, JIP, Vol. 1, No. 2 (1978).
- 15) A. V. Aho and S. C. Johnson: Deterministic Parsing of Ambiguous Grammars, Commun. ACM, Vol. 18, No. 8 (1975).
- 16) J. Earley: Ambiguity and Precedence in Syntax Description, Acta Inf. 4, (1975).
- 17) 井上謙蔵: 右順位文法と包含関係, 情報処理, Vol. 13, No. 10 (1972).

付 録 I

```

→S      , /*STARTING SYMBOL*/
S →"a"   , /*#1*/
S →E "c" , /*#2*/
E →"a" E "b", /*#3*/
E →"a" "b" ; /*#4*/

```

図 I-1 LALR (1) 文法例

Fig. I-1 An LALR(1) grammar G

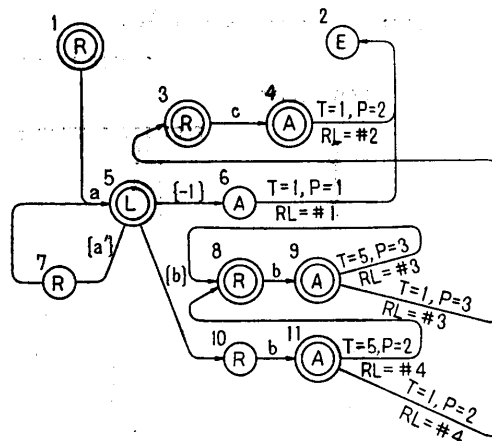


図 I-2 図 I-1 の文法に対応する状態遷移図

Fig. I-2 DPDA transition diagram for the grammar G; R, L, A and E are the abbreviations of Read, Lookahead, Apply and End state respectively. States represented by double circle are to be pushed down into the push-down stack associated with DPDA. T means top state, P pop-up no. and RL production rule no; these information are used as follows: When an apply state reached, it pops the push-down stack P times and sees if the top is T. If so, production rule RL is recognized and makes a transition to the state pointed to by the arrow from the apply state. "1" is a special symbol to indicate input end.

```

(1, 1, aabb1, 1, 1) ⇒ (15, 5, aabb1, 1, 1)
                      ⇒ (15, 7, aabb1, 1, 1)
                      ⇒ (155, 5, bbb1, 1, 1)
                      ⇒ (155, 10, bbb1, 1, 1)
                      ⇒ (15511, 11, bbb1, 1, 1)
                      ⇒ (158, 8, bbb1, 1, 1)
                      ⇒ (1589, 9, c1, 1, 1)
                      ⇒ (13, 3, c1, 1, 1)
                      ⇒ (134, 4, 1, 1, 1)
                      ⇒ (1, 2, 1, 1, 1)

```

図 I-3 解析遷移図

Fig. I-3 State transition of the DPDA with input "aabb1".

付 録 II

MLTG

SOURCE LISTING

```

SYMT  RUL,NO
1      1      SYNTAX SECTION:
2      1      -> EXTPROC
3      2      EXTPROC  -> #ID ":" "PROC" ";"
                        PROCBODY
                        "END" ";"
4      3      PROCBODY -> R(0) ( STATEMENT )
5      7      STATEMENT -> /* DCL, STM, */
                        "DCL" #ID ATTRIBUTE
                        R(0) ( " " #ID ATTRIBUTE ) ";"
                        /* PROC, CALL STM, */
                        "CALL" #ID ";"
                        /* ENTRY STM, */
                        #ID ":" "ENTRY" ";"
                        /* ASSIGNMENT STM, */
                        #ID
                        R(0) ( " " #ID )
                        " " EXPR ";"
6      17      -( "+" , "*" ) EXPR -> EXPR "+" EXPR
7      18      -( "*" ) EXPR -> EXPR "*" EXPR
8      19      EXPR      -> "(" EXPR ")"
                        ; #ID
9      21      ATTRIBUTE -> "BIN" "FIXED"
                        ; "DECIMAL" "FLOAT"
                        ;

```

END OF SOURCE LISTING.

図 II-1 構文誤り処理能力検定のための簡単な文法

Fig. II-1 Grammar for a PL/I like language. This is used to test the ability of our error correction/recovery method.

SOURCE & ERROR MESSAGE

```

CARD 1*****10*****20*****30*****40*****50*****60*****70*****
1      TEST4: PROC;
2      DCL ID1 BIN FIXED
3      ID2 BIN FIXED;
          ERROR1 ***> ; IS INSERTED BEFORE COLUMN0014 ON CARD 0003.
          ERROR2 ***> AN ID./KEY WORD IS NEGLECTED AT COLUMN 0018 ON CARD 0003.
          ERROR3 ***> AN ID./KEY WORD IS NEGLECTED AT COLUMN 0022 ON CARD 0003.
          ERROR4 ***> ; IS NEGLECTED AT COLUMN 0027 ON CARD 0003.
4      DCL ID3 DECIMAL FLOAT
5      ID1=ID2*ID3 (ID3+ID1);
          ERROR5 ***> ; IS INSERTED BEFORE COLUMN0010 ON CARD 0005.
          ERROR6 ***> + IS INSERTED BEFORE COLUMN0022 ON CARD 0005.
6      CALL TEST2;
7      IN ENTRY;
          ERROR7 ***> ; IS INSERTED BEFORE COLUMN0008 ON CARD 0007.
8      ID2 ID3*(ID2+ID3 ;
          ERROR8 ***> = IS INSERTED BEFORE COLUMN0015 ON CARD 0008.
          ERROR9 ***> ) IS INSERTED BEFORE COLUMN0029 ON CARD 0008.
9      END;

```

END OF SOURCE & ERROR MESSAGE LISTING. THERE ARE 9 ERRORS.

図 II-2 誤り検出例

Fig. II-2 A program with syntax errors and error message.

(昭和52年3月22日受付)

(昭和53年8月15日採録)