計算機アーキテクチャ 32－1
（1978. 9. 13）

MPL200/II recursive descent compiler について

重松保弘　飯野秀政　安在弘幸
（九州工業大学）

1. まえがき

MPL200/IIは、高水準マイクロプログラミング言語MPL200[1]の改良版である。MPL200は、エミュレータや特殊目的向き計算機を実現するためのシステム記述言語を目標として開発され、(1) 最適化を行なっている、(2) 構造化プログラミングを取り入れており、記述性に富む、などの特徴を持っている。しかし、問題点がいくつか残っていた。その1つは、言語の拡張に伴なうコンパイラの改造が容易でないという点であり、もう1つは、最適化技法についても検討の余地があるという点である。こうした問題点を解決するため、MPL200/IIでは、前者については、Recursive descent parserと直構文変換（Syntax directed translation）の技法を用い、後者については、MPL200で採用した最適化技法に加えて、ソーステキストから中間形式を生成する時点での最適化技法を導入した。

Recursive descent parserは、入力を識別するために、回帰的（recursive）な手続きの集合を用い、かつ、バックトラックを持たないようなparserのことである。Recursive descent parserは、比較的簡単に書くことができ、もし、手続き呼び出しが効率良く実行されるような言語を用いて書くことができれば、充分、効率的である[2]。また、直構文変換は、semantic actionと呼ばれる手続きを呼び出すための機構を構文規則の中に埋め込むものである。semantic actionは、parserによって起動され、ソーステキストの中間コードを出力する。中間コードの生成を、ソース言語の構文規則の中に直接書くことができるのは、コンパイラの設計者には有益である[2]反面、コンパイラのモジュール構造化を損なう[3]との評価もある。

Recursive descent compilerは、Recursive descent parserに、semantic actionを付け加えたものである。したがって、recursive callを許す言語を用いて作成されることが前提となっている。しかし、筆者らの利用しているFACOM 230-45SのPL/1言語はrecursive callを許さない。そのため、recursive callを許す簡単な（MPL200/II向きの）コンパイラ記述言語ADL（Adapter Language）を作り、ADLを用いてMPL200/IIのコンパイラを記述した。ADLで記述されたMPL-200/IIのコンパイラは、ADLトランスレータによってPL/1言語のソーステキストに変換される。ADLを用いた結果、極めて記述性良く、MPL200/IIのコンパイラを作成することができた。

マイクロプログラム・コンパイラの処理過程をFig. 1のようにとらえるとき、マイクロプログラム（μP）の最適化は②の過程で行なわれることが多い。MPL200のコンパイラにおいても②の過程で最適化を行なっていた。これに対し、MPL200/IIでは、①の過程でも最適化を行なっている。すなわち、利用者によるspace optimization（μPのオブジェクト語数を最小にする）または、speed optimization（μPの実行時間を最小にする）の指定

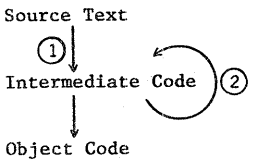

Fig.1 Compiling process of a microprogram compiler.

に従って、コンパイラは、生成すべき中間コードを選択する。この方法によって②の段階では困難な最適化を①の段階で容易に行なうことができる。

2. MPL200/IIの言語上の特徴

MPL200/IIは、MPL200を改良して開発されたものであり、MPL200の次のような特徴を受け継いでいる。(1) エミュレータ等を実現するためのシステム記述言語である。(2) 高級言語であるため記述性が良い。(3) マイクロ命令(μI)の先取りや、並列実行によるハードウェアリソースの競合を考えなくてよい。(4) 制御の流れ(control flow)の構造的な記述が可能である。(5) アセンブリ言語形式で記述されたμPをソーステキストに埋め込むことができる。(6) U-200Lのショートアドレス・ページング(条件分岐μIでは、256語単位のページを越えて分岐することができない)を考えなくてよい。

また、MPL200/IIは、MPL200に比べて、次のような改良がなされている。(1) recursive descent parserで構文解析を行なうとき、直構文変換を効率よく行なえるよう構文規則を一部改めた。(2) 機能分岐に関する文を追加し、エミュレータの記述を容易にした。(3) 構文上のあいまいさをなくすよう構文規則を一部改めた。(4) その他、記述性を増すための改良を行なった。以上の改良点について、以下に、具体例で説明する。

(1) の例を次に示す。Fig. 2は、ラベル付き文と内部手続きを宣言する文の構文規則であり、(a)はMPL200、(b)はMPL200/IIにおける構文規則を、それぞれ示す。

```
<labeled statement>::=<label>:<statement>;   <labeled statement>::=<label>:<statement>;
<dcl. int. proc.>::=<label>:PROC;            <dcl. int. proc.>::=PROC <label>;
```

(a) MPL200　　(b) MPL200/II

Fig.2 The syntax rule of labeled statement and declaration of internal procedure.

Fig. 2の(a)では、PROCを発見するまで、<label>が内部手続き名であることがわからない。そのため、手続き名の処理が後に延ばされてしまう。また、そのため、<label>を一時的に記憶しておく必要がある。これに対して、(b)では、こうした余分な操作を必要としない。

(2)は、具体的にはSSA文の追加がこれに相当する。U-200Lの機能分岐の機構はFig. 3に示すようになっており、次のような手順で機能分岐が起こる。まず、機械コードをOPR(Operation Register)に格納し、機能分岐を指定するμI(MPL200/IIでは、SA指定をしたSET文)を実行する。すると、①SSA(Set Start Address)領域選択機構により OPRの内容がデコードされ、②ローカルメモリのSSA領域(64語)のうち1語が選択され、③MAR(Micro instruction Address Register)に格納される。したがって、機能分岐を行なわせるためには、その飛先番地をSSA領域に、あらかじめ格納しておかなくてはならない。MPL200では、プログラマが、あらかじめ飛先番地を計算し、アセンブラ文を使って、直接的にSSA領域に値を格納しなくてはならなかった。これに対して、MPL200/IIでは、次に示すように、飛先のラベルを指定するだけでSSA領域には、そのラベルの制御記憶上の番地が格納される。

SSA <SSA領域の場所><飛先のラベル>;

(3)は1F文の改良がこれにあたる。すなわち、1F文の終りに必ずFIを付けることにしたため、1F文のネスティングにおける構文上の曖昧さが解消された。

(4)は、具体的には次のような点の改良がこれにあたる。すなわち(a)READ文やWRITE文のアドレス指定などに式が使用できるようになった。(b)名標の長さの制限がなくなった。ただし、先頭の8文字のみが有効。(c)名標に定数が割り付けられるようになった。(d)算術演算子"++"(Add with carry)が付加された。(e)DO文の代りに、FOR文が加えられた。

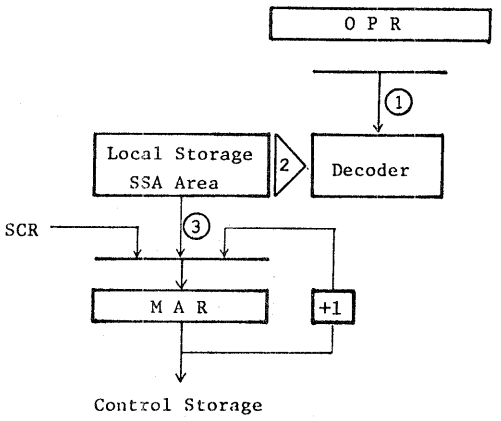

Fig.3 The mechanism of functional branch of U-200L.

MPL 200/IIの構文規則を付録に示す。構文規則は、正規表現形式の超言語式を変形したもので記述している。

また、MPL 200/IIによるプログラミング例をFig.4に示す。この例は、U-200L. PASCALマシンの一部である。

```
DCL   CCA_DSR              EQU   X'0600' ,
      CCA_BCR              EQU   X'0602' ,
      CCA_MAR              EQU   X'0604' ,
      CCA_CMR              EQU   X'0606' ;
DCL   MEM_ADDRESS          EQU   X'1100' ,
      ATTENTION            EQU   X'0200' ,
      C_COMMAND            EQU   X'0007' ,
      R_COMMAND            EQU   X'0200' ,
      START                EQU   X'0800' ,
      CONTROL              EQU   X'0700' ,
      ASW                  EQU   X'0010' ,
      PSTR                 EQU   X'0202' ;
DCL   STATUS               EQU   MB ,
      COMMAND              EQU   MB ,
      ENTRY_KEY            EQU   ENT ;

/* TRANSFER AND TEST */
      WRITE MEM_ADDRESS INTO CCA_MAR WORD ;
      LOOP
           REPEAT
                READ STATUS FROM CCA_DSR WORD ;
                STATUS = STATUS AND X'0200' ;
           UNTIL STATUS = ATTENTION ;
           READ COMMAND FROM CCA_CMR WORD ;
           COMMAND = COMMAND AND X'00FF' ;
      EXIT IF COMMAND = C_COMMAND ;
           WRITE 256 INTO CCA_BCR WORD ;
           WRITE R_COMMAND INTO CCA_CMR WORD ;
      POOL
```

Fig.4 An example program written in MPL200/II.

3. Recursive descent Compiler

Recursive descent Compilerについては、文献2, 3において詳細に述べられている。

Fig.5, Fig.6に、代入文の構文規則例および、そのrecursive descent compilerの一部を、それぞれ示す。これは文献2から引用したものである。

この例からもわかるように、原始言語の構文規則の定義とほぼ同じ形式でコンパイラを記述できるため、極めて記述性が良い。ただ、構文解析と意味解釈が混在するため、コンパイラの構造を、構文解析モジュールと意味解釈モジュールのように分けることができない。

```
Assignment = Variable '=' Expression;
Expression = (|'-') Term (('+' | '-') Term)*;
Term = Factor (('x' | '/') Factor)*;
Factor = Constant | Variable | '(' Expression ')';
Variable = Identifier;
Identifier = 'A' | 'B'| ... |'Z';
Constant = '0' | '1'| ... |'9';
```

Fig.5 A syntax rule of Assignment Statement.

4. Adapter Language

コンパイラ等の開発において、汎用高級言語（FORTRAN, PL/1等）を用いることは、開発労力の低減や記述性の向上に効果がある。そのため、MPL200の開発においてもPL/1言語を用いた。しかし、汎用高級言語は、その汎用性のために、CDL[3]等のようなコンパイラ記述言語に比べると、充分コンパイラの記述に向いているとは言えない。そうかといって、汎用のコンパイラ記述言語を新たに設計、開発するのは、コンパイラの開発以上のコストを要する。そこで、ある特定のコンパイラの記述に適した簡単なコンパイラ記述言語を設計することにすれば、コンパイラの開発に要する全体的なコストを低減することができると考えられる。このように、強い制限はあるが、ある特定のアプリケーションに向いている言語を、ここではADL（Adapter Language）と呼ぶことにする。MPL200/IIのコンパイラの開発においても、MPL200/IIのコンパイラ向きのADL（以後、単にADLと記す）を設計し、ADLでコンパイラを記述した。ADLはFig. 6 a recursive descent compilerの記述形式をPL/1言語に取り入れたものである。ADLは、FACOM 230-45SのPL/1言語に対して、次のような特徴を持っている。すなわち、①MPL200/IIのコンパイラの記述に適している。②回帰呼び出しを許すので、recursive descent compilerの記述に適している。

```
proc Expression = void:
begin string t;
      co First check for unary minus.  co
      if token = "-"
      then token := scan;
           Term;
           emit ("NEG")
      else Term
      fi;
      co Now process a sequence of adding operators.  co
      while token = "-" ∨ token = "+"
      do t := token;
         token := scan;
         Term;
         if t = "-" then emit ("NEG") fi;
         emit ("ADD")
      od
end;
```

Fig.6 A Recursive Decent Compiler of Fig.5. (part)

```
<main ext-proc> ::= PROC MAIN <proc-name> ; <proc-block><int-proc>* END <proc-name> ;
```

(a) The syntax rule of main external procedure.

```
<main ext-proc> ::= PROC MAIN <proc-name> ; <proc-block><after block>
<after block> ::= END <proc-name> ; | <int-proc><after block>
```

(b) BNF representation of (a).

Fig.7 An example of the syntax rule of MPL200/II.

Fig. 7に、MPL200/IIの構文規則の例と、これを*を含まない形に変形したものを示す。これは、主外部手続きの構文規則である。また、Fig. 9に、Fig. 7 (b)の構文規則に対するrecursive descent compilerを、ADLで記述したものの一部を示す。

ADLで記述されたMPL200/IIコンパイラはADLトランスレータによってPL/1言語に変換される。Fig. 8に、

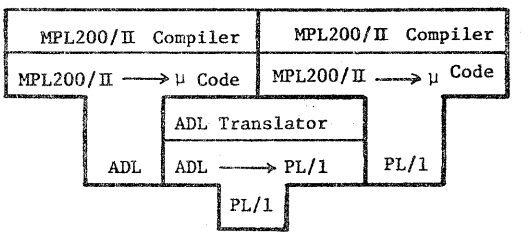

Fig. 8 T diagram for generation of MPL200/II compiler.

```
  MEXTPROC :                              /* MAIN EXTERNAL PROCEDURE */
P     '<MEXTPROC>'
T     IF TOKEN='PROC' THEN
         DO ;
T          IF TOKEN='MAIN' THEN
              DO ;
                LAB-DEF=TRUE ;
N               PROCNAME ;
A               MAINSTRT ;
A               GETTOKEN ;
T               IF TOKEN=';' THEN
                   DO ;
                     ST_NO=ST_NO+1 ;
                   END ;
                ELSE CALL WARN ;
N               PROCBLOC ;
N               AFT_BLOC ;
R               *
              END ;
A          ERRPRINT('MAIN') ;
R          *
         END ;
A     ERRPRINT('PROC') ;
A     GETTOKEN ;
R     *
```

Fig.9 A part of the recursive descent compiler, written in ADL, according to the syntax rule of Fig.7(b) .

```
    DCL *S(1000)  LABEL, * FIXED(4) STATIC INIT(1);
 GO TO *START; *RET:
* PUT EDIT('@') (A);
 *=*-1; GO TO *S(*); *START:
```

(a) The declaration of stack and its operation part.

```
  MEXTPROC :                              /* MAIN EXTERNAL PROCEDURE */
*     PUT EDIT('<MEXTPROC>'                                  ) (A);/* PRT */
      IF TOKEN='T049' THEN
         DO ;                                                 CALL GETTOKEN;
           IF TOKEN='T044' THEN
              DO ;                                            CALL GETTOKEN;
                LAB_DEF=TRUE ;
                *S(*)=*004; *=*+1; GO TO PROCNAME ;                    *004:
                CALL MAINSTRT ;                                  /* ACT */
                CALL GETTOKEN ;                                  /* ACT */
                IF TOKEN='T001' THEN
                   DO ;                                        CALL GETTOKEN;
                     ST_NO=ST_NO+1 ;
                   END ;
                ELSE CALL WARN ;
                *S(*)=*005; *=*+1; GO TO PROCBLOC ;                    *005:
                *S(*)=*006; *=*+1; GO TO AFT_BLOC ;                    *006:
                GO TO *RET;                                      /* RET */
              END ;
           CALL ERRPRINT('MAIN') ;                               /* ACT */
           GO TO *RET;                                           /* RET */
         END ;
      CALL ERRPRINT('PROC') ;                                  /* ACT */
      CALL GETTOKEN ;                                          /* ACT */
      GO TO *RET;                                              /* RET */
```

(b) The recursive descent compiler translated from the program of Fig.9 .

Fig.10 The output program, written in PL/1, of ADL translator .

この変換過程をT diagramで示す。また、Fig.10に、Fig.9の例をADLトランスレータを用いてPL/1言語に変換した結果を示す。

ADLトランスレータは、ADLで記述された文の行における第1桁目を調べ、これによって、変換方法を決定する。第1桁目の文字と、その意味について、以下に説明する。

P (Print)	:	"で囲まれた文字列を印字する。(デバッグに使用)
T (Terminal)	:	端記号のチェックを行なう行であることを示す。
N (Non-terminal)	:	非端記号の構文解析ルーチンを呼び出す行であることを示す。
A (Action Routin)	:	中間コードの生成、エラーメッセージの印刷等のsemantic actionを呼び出す行であることを示す。
R (Return)	:	該当する非端記号の構文解析の終了を示す。
I (Initialize)	:	スタックの宣言と操作部を生成する。具体的には、Fig.10a (a)が生成される。

これらのうち、スタック操作に関係があるのはNとRである。Nでは、戻り番地(ラベル)をスタックし、指定された非端記号の構文解析ルーチンへ飛ぶ。Rでは、スタックの先頭に格納された番地をポップ・アップし、そこへ飛ぶ。また、Tでは、"記号で囲まれた端記号の文字列を、端記号テーブルへの固定長のポインタに変換する。これは、コンパイラの単語解析フェイズにおいて、ソーステキストは、すでに、トークンに分解され、各種テーブルへのポインタに変換されているからである。したがって、構文解析フェイズでは、端記号をチェックする代りに、その端記号を指すポインタをチェックすることになる。

Fig.10の例では、非端記号の構文解析ルーチンの入口で非端記号名(<MEXTPROC>)が印字され、出口(Fig.10の(b))で@記号が印字される。これは、デバッグに用いられている。また、不必要なら、PL/1コンパイラへの指示によって消去することもできる。

また、ADLトランスレータは、利用者の指示によって、MPL200/IIコンパイラの中間コードの格納領域の大きさ等を変更する。現在、この大きさは3種類が指定できる。すなわち、①Sサイズ(500セル)、②Mサイズ(3000セル)、③Lサイズ(5000セル)である。セルは、中間コードをインプリメントするときの単位であり、1セルが4語(64ビット)を占める。

5. Optimization Technique について

MPL200/IIaコンパイラの処理過程をFig.11に示す。このコンパイラは6フェイズに分かれている。すなわち、(1)単語解析(lexical analysis)、(2)構文解析(syntax analysis)および意味解釈(semantic analysis)、(3)最適化(optimization)、(4)ページチェック(page boundary check)、(5)コード生成(code gencration)、(6)アセンブル(assemble)である。

このうち、MPL200のコンパイラと異なるのは、単語解析、構文解析、意味解釈の各フェイズである。したがって、MPL200/IIaコンパイラは、最適化段階においてMPL200の次のような特徴を受け継いでいる。すなわち、(1)タイミング

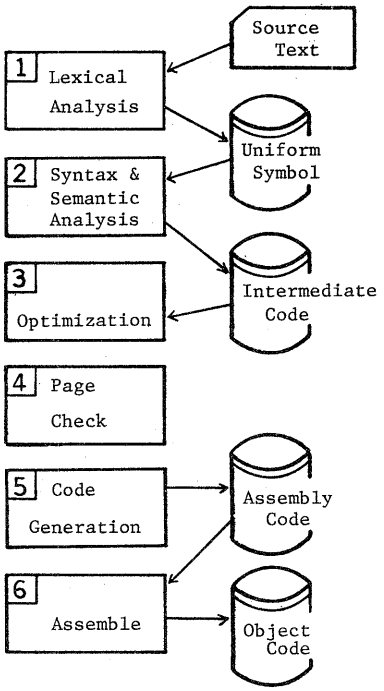

Fig.11 The compiling process of MPL200/II COMPILER.

Operation	Operand 1	Operand 2	Clocks
LP	11	CT	4 T
MV	GR 1	GR 0	2 T
BC	*	K	2 T
SR	GR 0	GR 0	2 T

(a) The case using BC μI.

Operation	Operand 1	Operand 2	Clocks
SR	GR 1	GR 0	2 T
SRD	GR 0	GR 0	4 T
SRD	GR 0	GR 0	4 T
SRD	GR 0	GR 0	4 T
SRD	GR 0	GR 0	4 T
SRD	GR 0	GR 0	4 T

(b) The case using SR,SRD μI.

Fig.12 Intermediate code of GR0 = GR1 SHR 11; .

	Storage Size	Execution Time
(a)	4 words	50 T
(b)	6	22

Fig.13 The storage size and the execution time in each case of Fig.12.

の同期用に挿入した無効μIの削除、(2) 冗長なμIの削除、(3) 主記憶のアクセス時間の有効利用、を主とした最適化を行なっている。

MPL200では、最適化は、意味解析フェイズによって出力された中間コードを対象としている。これに対して、MPL200/IIでは、意味解析フェイズにおいて、利用者の意図に添った中間コードを生成することができる。すなわち、利用者はMPL200/IIのコンパイラに対して、① speed optimization、② memory optimization、のいずれかを指定することができる。前者は、オブジェクトμPの実行時間を最少にするとき、また、後者は、オブジェクトμPの語数を最少にしたいとき、それぞれ指定する。

次の文の中間コードを例にあげて説明する

GR0 = GR1 SHR 11;

Fig.12の(a)の中間コードは、条件分岐(BC)μIを利用したものである。しかし、この文に対しては、Fig.12の(b)の中間コードも考えられる。(b)の中間コードはシフト(SR, SRD)μIを並べたものである。この2通りの中間コードから生成されるオブジェクトコードの語数と実行時間を比較したものをFig.13に示す。この例では、オブジェクトμPの語数を最少にするためには(a)が、また、実行時間を最少にするためには(b)が、それぞれ望ましいことがわかる。

次の文は、あるリソース<R>の内容をnで示すビット数だけ右へシフトすることを指定する文である。

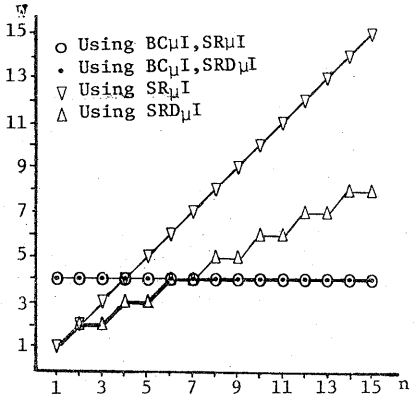

Fig.14 The relation between the shift number n and the word size W for the object μP of the statement <R> = <R> SHR n;.

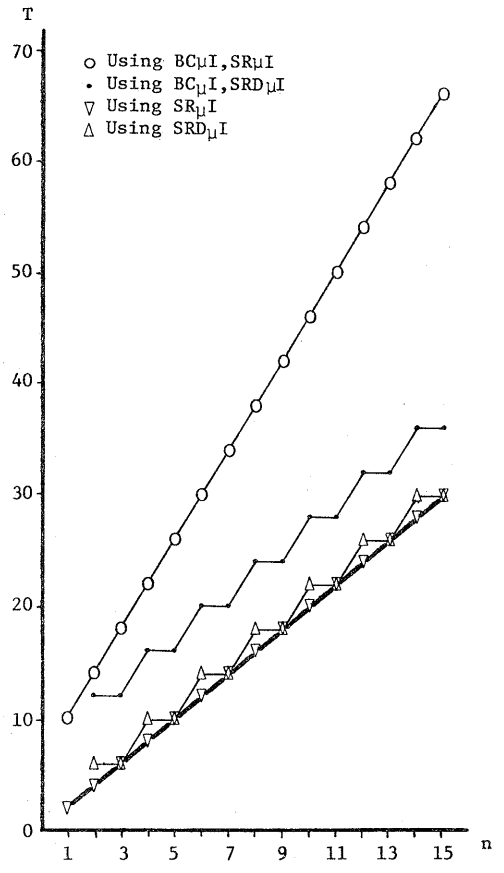

Fig.15 The relation between the shift number n and the execution time T for the μP of Fig.14.

<R>=<R> SHR n;

この文において、nとオブジェクトμPの語数の関係をFig.14に，nとオブジェクトμPの実行時間の関係をFig.15に、それぞれ示す。また、Fig.14とFig.15における太実線部は、それぞれ、語数最少、実行時間最少となるオブジェクトμPである。この例では、nが1，2の場合は、SR μIのみからなるオブジェクトμPが語数、実行時間ともに最少であることがわかる。また、nが4の場合は、SR μIを用いると実行時間が最少となり、SRD μIを用いると語数が最少となるので、この選択は利用者に任されることになる。

6. あとがき

ADLの使用、recursive descent compilerの技法の採用によって，極めて効率的にMPL 200/IIのコンパイラを作ることができた。これによって、コンパイラの記述性も向上し、言語の拡張に伴なうコンパイラの改造も容易になった。なお現在、800ステップのソーステキストで、lexical analysisに約140秒、syntax analysisに約35秒かかる。

また、最適化の方法の選択については、通常は実行時間を最少にする方法を指定し、制御記憶量を少し上回ったときのみ、語数を最少にする方法を指定することにしている。ASSEMBLER文と外部手続きの処理については、現在インプリメント中である。

参考文献

1) 重松，有川，安在：マイクロプログラミング言語MPL200とその最適化技法，情報処理，vol.19，No.8（1978）

2) A.V.Aho，J.D.Ullman：Principles of compiler Design，Addison-Wesley Publishing Company pp.604（1977）

3) F.L.Bauer et al.：Compiler Construction，Springer-Verlag，pp.638（1976）

4) 松崎：実用化の試みが始まったマイクロプログラムの最適化，日経エレクトロニクス，第185号，pp.76～91（1978）

5) P.Lucas：Die Structuranalyse von Formeluberseten，Electron，Rechnanl，3，pp.159～167（1961）

6) M.E.Conway：Design of a separable transition diagram compiler，C.ACM vol.6，No.7，PP398～408（1963）

<付録> MPL200/IIの構文規則

BASIC SYMBOL

```
<basic symbol> ::= <alphabet> | <digit> | <delimiter> | <special symbol>
<alphabet> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|#
<digit> ::= 0|1|2|3|4|5|6|7|8|9
<delimiter> ::= ;|:|,|(|)|=|+|-|>|<|¬| /|*|space
<special symbol> ::= _|.|'
```

IDENTIFIER

```
<identifier> ::= <alphabet><letter>*[[.|_]<letter>+]*
<letter> ::= <alphabet> | <digit>
```

CONSTANT

```
<constant> ::= <binary constant> | <decimal constant> | <hexadecimal constant>
<binary constant> ::= B'<bit>+'
<bit> ::= 0|1
<decimal constant> ::= <digit>+
<hexadecimal constant> ::= X'<hexadecimal digit>+'
<hexadecimal digit> ::= <digit>|A|B|C|D|E|F
```

RESERVED WORD

```
<reserved word> ::= AND|ASSEMBLER|BY|BEGIN|BRANCH|CALL|CASE|CARRY|CC|DO|DCL|DSTOP|END|
                    EQU|EOR|ELSE|ESAC|EXIT|ENTRY|EX|FI|FOR|FROM|FOREVER|GO|IF|INTO|LOOP|
                    MAIN|OF|OR|ORG|OPREG|PROC|POOL|READ|REPEAT|SHL|SHR|SET|SSA|SA|TO|
                    THEN|TIMES|UNTIL|WRITE|WHILE|WORD|WCS| <resource>
```

RESOURCE

```
<resource> ::= <r/w-able resource> | <r-only resource> | <w-only resource>
<r/w-able resource> ::= GR0|GR1|GR2|GR3|GR4|GR5|GR6|GR7|GRD|GRD0|GRS|AB|WB|OPR|CT|MB|
                        SR|STR|IRC|FL2|ER|FRD|FRD0|FR0|FR1|FR2|FR3|FW0|FW1|FW2|FW3
<r-only resource> ::= DISP|DEC|C0000|C0001|C0002|C0004|C0007|C00F0|C2000|CFFFF|ENT|
                      KEY1|KEY2
<w-only resource> ::= SCR
```

SYNTAX

```
<program> ::= <main ext-proc><ext-proc>*
<main ext-proc> ::= PROC MAIN <proc-name> ; <proc-block><int-proc>* END <proc-name> ;
<ext-proc> ::= <procedure>
<int-proc> ::= <procedure>
<procedure> ::= PROC <proc-name> ; <proc-block> END <proc-name> ;
<proc-block> ::= <dcl-st>*[[<label> : ]* <statement> ]*
<proc-name> ::= <identifier>
<label> ::= <identifier>

<dcl-st> ::= DCL <declaration> [ , <declaration> ]* ;
<declaration> ::= <identifier> EQU <attribute> |
                  <identifier> ENTRY |
                   ( <identifier> [ , <identifier> ]* ) EQU <attribute> |
                   ( <identifier> [ , <identifier> ]* ) ENTRY
<attribute> ::= <resource> | <constant>

<statement> ::= <assignment> | <control> | <io> | <set> | <compound> | <special> | <empty>
<assignment> ::= <identifier> = <expression> ; |
                 <identifier><identifier> = <double shift> ;
<expression> ::= <arith-express> | <bit operation> | <single shift>
<arith-express> ::= [-] <operand> [ <arith-op><operand> ]*
<arith-op> ::= +|-|++
<operand> ::= <constant> | <identifier>
<bit operation> ::= <operand><bit-op><operand >
<bit-op> ::= AND|OR|EOR
<single shift> ::= <identifier><shift-op> [ <shift number> ]
<shift-op> ::= SHR|SHL
<shift number> ::= <constant> | <identifier>
<double shift> ::= <identifier><identifier><shift-op> [ <shift number> ]

<control> ::= <go to> | <if> | <case> | <loop> | <while> | <repeat> | <for> | <call> | <dstop>
<go to> ::= GO TO <label> ;
<if> ::= IF <condition> THEN <statement> [ ELSE <statement> ] FI
<condition> ::= <operand><relational op><operand> | <status test>
<relational op> ::= >|<|=|>=|<=|¬=
<status test> ::= CARRY | BRANCH

<case> ::= CASE <parameter> OF <correspond>+ [ ELSE : <statement> ] ESAC
<parameter> ::= <identifier>
<correspond> ::= <constant> : <statement>
<loop> ::= LOOP <statement>* EXIT IF <condition> ; <statement>* POOL

<while> ::= WHILE <condition> DO <statement>
<for> ::= FOR <identifier> = <initial> TO <final> [ BY <increment> ] DO <statement>|
          FOR <number> TIMES DO <statement>
<initial> ::= <constant> | <identifier>
<final> ::= <constant> | <identifier>
<increment> ::= <constant> | <identifier>
<number> ::= <expression>
<repeat> ::= REPEAT <statement>+ UNTIL <condition> ; |
             REPEAT <statement>+ FOREVER
<call> ::= CALL <proc-name> ;
<dstop> ::= DSTOP
<io> ::= <read> | <write> | <wcs>
<read> ::= READ [ <identifier> ][ FROM <address> ][ WORD ] ;
<write> ::= WRITE [ <data> ][ INTO <address> ][ WORD ] ;
<wcs> ::= WCS [<data> ][ INTO <address> ] ;
<data> ::= <expression>
<address> ::= <expression>
<set> ::= SET <set designation> ;
<set designation> ::= CC|OPREG|SA|SA EX
<compound> ::= BEGIN <statement>+ END
<empty> ::= ;
<special> ::= <origin> | <ssa> | <assembler>
<origin> ::= ORG <origin designation> ;
<origin designation> ::= <constant>
<ssa> ::= SSA <ssa address><start address> ;
<ssa address> ::= <constant>
<start address> ::=<label>
<assembler> ::= ASSEMBLER <assembler-like uinstruction>+ END
```