ソフトウェア演習 V 課題４

（１）課題名：コンパイラの作成
（２）演習期間：12/14 - 1/24
（３）レポート・プログラム提出期間：1/25 - 2/1，最終〆切：**2/1(月)**

（４）課題

プログラミング言語 MPPL で書かれたプログラムを読み込み，コンパイルエラー，すなわち，構文エラーもしくは制約エラー（型の不一致や未定義な変数の出現等）があれば，そのエラーの情報（エラーの箇所，内容等）を少なくとも一つ出力するプログラムを作成する．エラーがなければ，オブジェクトプログラムとして, CASLII (別紙参照) のプログラムを出力する. 例えば, 作成するプログラム名を mpplc, MPPL で書かれたプログラムのファイル名を foo.mpl とするとき，コマンドラインからのコマンドを

mpplc foo.mpl

とすれば(foo.mpl のみが引数)，foo.mpl の内容のプログラムをコンパイルした結果（CASL II のプログラム）を持つファイル foo.csl を生成するプログラムを作成する．

入力：MPPL で書かれたプログラムのファイル名．コマンドラインから与える．MPPL のマイクロ構文，マクロ構文，制約規則はそれぞれ課題１，２，３の通りである．MPPL のセマンティクスを，マクロ構文と共に示す．「//」以降の部分がセマンティクスである．

プログラム ::= "program" "名前" ";" ブロック "."
　　// ブロックが実行されるべきプログラムの本体である
ブロック ::= { 変数宣言部 | 副プログラム宣言 } 複合文
　　// 複合文が実行されるべき命令である
変数宣言部 ::= "var" 変数名の並び ":" 型 ";" { 変数名の並び ":" 型 ";" }
　　// 変数の並びにある変数名がその直後の":"の次の型と関連付けられる
変数名の並び ::= 変数名 { "," 変数名 }
変数名 ::= "名前"
型 ::= 標準型 | 配列型
標準型 ::= "integer" | "boolean" | "char"
　　// "integer", "boolean", "char"はそれぞれ整数型，論理型，文字型を表し，
　　// それぞれ，整数値(16bit)，論理値(true, false)，文字(1 バイト文字）を保持できることを表す
配列型 ::= "array" "[" "符号なし整数" "]" "of" 標準型
副プログラム宣言 ::= "procedure" 手続き名 [仮引数部] ";" [変数宣言部] 複合文 ";"
　　// この手続き名が呼ばれた（call された）時は，この複合文が実行される．
　　// 引数の渡し方は参照渡しである
　　// この変数宣言部に宣言されている変数（局所変数）の初期値は不定である
　　// 実行終了後は，この副プログラム（手続き）を呼び出した呼び出し文の次に戻る

//　再帰呼び出しは許さない

手続き名 ::= "名前"

仮引数部 ::= "(" 変数名の並び ":" 型 ｛ ";" 変数名の並び ":" 型 ｝ ")"

複合文 ::= "begin" 文 ｛ ";" 文 ｝ "end"

//　文の並びを前から順に実行する

文 ::= 代入文 | 分岐文 | 繰り返し文 | 手続き呼び出し文 | 戻り文 | 入力文 | 出力文 | 複合文 | 空文

分岐文 ::= "if" 式 "then" 文 ［ "else" 文 ］

//　式を評価し，その式の値が true であれば左の文を，false であれば右の文を実行する

//　式の値が false であって，else 部がなければ何もしない

繰り返し文 ::= "while" 式 "do" 文

//　［式を評価し，その式の値が true であれば文を実行する］ことを繰り返す

//　式の値が false になればこの文の実行を終える

手続き呼び出し文 ::= "call" 手続き名 ［ "(" 式の並び ")" ］

//　手続き名で指定された手続きを呼び出す

//　引数の渡し方は参照渡しである．

式の並び ::= 式 ｛ "," 式 ｝

//　実引数である式と仮引数である仮引数部の変数は並んでいる順番で対応する

//　実引数である式が左辺値を持たないときには，主記憶中に場所を取り，そこに式の値を

//　置いてその場所の番地を渡す．手続きの呼び出しが終わるとその場所は再利用してよい．

戻り文 ::= "return"

//　副プログラム宣言の複合文中で実行されると，その複合文の実行が終了したときと同様に，

//　その手続きの実行を終了する．

//　それ以外の場所，即ちブロックの複合文内で実行されると，プログラムの実行を終了する

代入文 ::= 左辺部 ":=" 式

//　式を評価し，その式の値を左辺部が表す変数に代入する

左辺部 ::= 変数

変数 ::= 変数名 ［ "[" 式 "]" ］

//　式があれば評価する

//　　変数名が表す配列の，その式の値を添字の値とする配列要素が変数である

//　　式の値が配列の添字の範囲を超えるときは，その旨を表示して実行を停止する

//　式がなければ，変数名が表すものが変数である

式 ::= 単純式 ｛ 関係演算子　単純式 ｝

//　左の単純式の値にその右隣の単純式の値で関係演算子が表す演算を施す

//　　これを左から順に関係演算子と単純式の組がある限り行う

//　　その結果の値が，式の値である

//　関係演算子がなければ，一つある単純式の値が式の値である

単純式 ::= ［ "+" | "-" ］項 ｛ 加法演算子　項 ｝

//　先頭に"-"があるとき，先頭の項の値に-1 を乗じたものをその項の値とする

//　左の項の値にその右隣の項の値で加法演算子が表す演算を施す

//　　これを左から順に加法演算子と項の組がある限り行う

```
        //    その結果の値が，単純式の値である
        //  加法演算子がなければ，一つある項の値が単純式の値である
項 ::= 因子 { 乗法演算子  因子 }
        //  左の因子の値にその右隣の因子の値で乗法演算子が表す演算を施す
        //    これを左から順に乗法演算子と因子の組がある限り行う
        //    その結果の値が，項の値である
        //  乗法演算子がなければ，一つある因子の値が項の値である
因子 ::= 変数 | 定数 | "(" 式 ")" | "not" 因子 | 標準型 "(" 式 ")"
        //  変数については，変数が保持する値が全体の因子の値である
        //  定数については，定数自身の値が全体の因子の値である
        //  "(" 式 ")"については，式の値が全体の因子の値である
        //  "not" 因子については，この因子の値を論理否定した値が全体の因子の値である
        //  標準型 "(" 式 ")"については，次の規則に従う
        //    式の型が integer 型のとき，
        //      標準型が"integer"であれば，式の値が，
        //      標準型が"boolean"であれば，式の値が 0 であれば false が，0 でなければ true が，
        //      標準型が"char"であれば，式の値の 2 の補数表現の下位 7 ビットを文字コードとして
        //        持つ文字が，
        //    全体の因子の値である
        //    式の型が boolean 型のとき，
        //      標準型が"integer"であれば，式の値が false であれば 0 を，true あれば 1 が，
        //      標準型が"boolean"であれば，式の値が，
        //      標準型が"char"であれば，式の値が false であれば 0 の，true であれば 1 の文字コードを
        //        持つ文字が，
        //    全体の因子の値である
        //    式の型が char 型のとき，
        //      標準型が"integer"であれば，式の値の文字コードの整数値が，
        //      標準型が"boolean"であれば，式の値の文字コードの整数値がが 0 であれば false が，
        //        0 でなければ true が，
        //      標準型が"char"であれば，式の値が，
        //    全体の因子の値である
定数 ::= "符号なし整数" | "false" | "true" | "文字列"
        //  "符号なし整数"の値はそれを 10 進数表現として解釈した値が定数の値である
        //  "false"と"true"のときはそれぞれ false，true の論理値が定数の値である
        //  "文字列"のときは，その文字自体が定数の値である．文字列が「"」のときは「'」と解釈する
乗法演算子 ::= "*" | "div" | "and"
        //  それぞれ乗算，除算，論理積の演算を表す
        //  オーバフロー，ゼロ除算が発生すると，その旨を表示して実行を停止する
加法演算子 ::= "+" | "-" | "or"
        //  それぞれ加算，減算，論理和の演算を表す
```

// オーバフローが発生すると，その旨を表示して実行を停止する

関係演算子 ::= "=" | "<>" | "<" | "<=" | ">" | ">="

// 左の被演算子が右の被演算子に対して，それぞれ，等しい，等しくない，小さい，
// 以下である，大きい，以上である，という命題の真偽を判定した結果の論理値を表す
// ただし，文字型については文字コードの大小で結果の真偽が決定する
// 論理型については，false＜ true とする

入力文 ::=（"read" | "readln"）［ "(" 変数 ｛ "," 変数 ｝ ")" ］

// 変数の型は integer 型か char 型でなくてはならない
// 標準入力からデータを読み込み，次のように左の変数から順に代入する
// 変数が integer 型のとき，(1)空白，タブ，改行コードがあればそれらをすべて読み飛ばす，
// (2)数字列があればそれを 10 進数字列とみなして整数に変換して格納する．
// 数字列がなければ 0 を読み込む
// 変数が char 型のとき，行末でなければ 1 文字を読み込んで変数に格納する
// 行末であれば，変数には改行コード('¥n')が格納される
// "readln"の場合は最後にその行の改行コードまで読み飛ばす．"read"の場合は読み飛ばさない

出力文 ::= （"write" | "writeln"）［ "(" 出力指定 ｛ "," 出力指定 ｝ ")" ］

// 出力指定に従って順にデータを標準出力へ出力する
// "writeln"の場合は最後に改行する．"write"の場合はしない

出力指定 ::= 式［ ":" "符号なし整数" ］| "文字列"

// 式であれば，式の値を標準出力へ出力する．
// 式の型が integer 型の時は 10 進数表現で，char 型の時はその文字を，
// boolean 型の時はその値の真偽に従って「true」か「false」を表示する
// "符号なし整数"があればその値の文字数で表示する
// 式の値を表示するのに文字数が余れば，左に空白を補う（右詰めで表示する）
// 式の値を表示するのに文字数が不足すれば，表示される内容は処理系依存である
// "符号なし整数"がなければ，その値を表示できる最小の文字数で表示される
// "文字列"であれば，その文字列自体を標準出力へ表示する
// ただし，その中で「'」が偶数個(2n 個)連続していれば n 個の「'」とみなす

空文 ::= ε

// 何もしない

なお，ここに特に定められていないことは処理系依存であるとする．

出力：入力のプログラム中に構文的な誤りや制約規則違反がなければ，オブジェクトプログラムとして，CASLII（別紙参照）のプログラムを出力せよ．出力ファイル名は入力ファイル名の拡張子を".csl"へ変更したものとせよ．もし構文的な誤りや制約規則違反があれば，それを最初に検出した時点で，誤りまたは違反の内容を標準出力へ出力せよ．

コード生成の方法

コード生成を作成しようとするとき，まず，実行時環境を決めてから，どのようなコードを生成するかを考えます.

１．実行時環境について

1-1 変数の割付

MPPL の仕様では，再帰呼び出しがないので，すべての変数を静的に割り付けることができます．また，integer の大きさを 1 語（2 バイト），boolean と char の大きさを 1 バイトというように決めることができます．しかし，ここでは，簡単のために，integer, boolean, char とも大きさを 1 語とすることにします．従って，たとえば，

```
var aaa : integer ;
```

のように変数が宣言されていたとすれば，目的プログラムには，

```
$aaa    DC    0
```

もしくは，

```
$aaa    DS    1
```

のように，宣言があればよいことになります．ここで，ラベルは変数名の頭に$を付けたものとしています．このようにすると，新しくラベル名を考え出す必要がないこと，頭に$を付けてあるので予約されている文字列と重なることがないことなどの利点があります．もちろん，他の付け方でもかまわないです．例えば，ジャンプ命令用のラベルと同じように各変数のラベル（例えば，L0110 のように）を与える方法もあります．この場合，記号表のその変数のノードにそのラベル番号を覚えておく必要があります.

また，手続き p に局所的な変数 x の場合は，同じ名前の変数が他の手続きにあったり，大域変数であったりするかもしれないので，$x とするのは問題があります（アセンブラで二重定義エラーが出る）．そこで，例えば，$x%p のように，手続き名と変数名を合成したラベルを使うといいかもしれません．変数にラベル番号を与える場合には，全体で別々の番号を与えることにすれば，この問題はありません.

配列の場合は，たとえば，

```
var abc : array［100］of integer;
```

ならば，

```
$abc    DS 100
```

を目的プログラムに生成すればよいわけです.

1-2 実行時スタックについて

上記のように再帰呼び出しがないので，実行時スタックは必要ないのですが，戻り番地を保持するのには，スタックを用いるのが便利です．また，どんな複雑な式でも処理できる必要がある，とすると，式の途中結果を記憶するスタックが必要になります．簡単のため，部分式の計算結果は必ずスタックトップに置くことにします.

もちろん，この決定は，部分式を計算するたびに主記憶への参照が発生するので効率は良くありません．効率を上げようとすると，レジスタを有効に利用する必要があります．これは，各自で考えてください．コンパイラの講義で講義をしませんでしたが，テキストの「８ 目的コード生成」の章が参考になります．また，テキストのサンプルコンパイラのようにレジスタが無数にあるとして，すべてのレジスタをスタッ

クのように利用してコードを生成してしまうという方法もあります．COMET II には汎用レジスタが 8 個しかありませんので，9 個目が必要になった時点で困ったことになります．その場合には，そのときにスタックへ待避するようにすれば良いのかもしれません．普通は，8 個もの部分式の値を覚えておく必要のあるような式は滅多に出てきません．手で無理矢理作ろうとしても，とても長い式になってしまいます．その意味では，9 個以上必要な式の場合にはレジスタを待避するので，効率の悪いコードになるとしても実用上は何の問題もないかもしれません．ただ，9 個以上必要な式は出てこない，と仮定するのはバグの元になりますので，止めてください．

２．コード生成

構文解析ルーチンにコード生成用の命令を挟み込んでいけばできます．

2.1 文のコード生成

たとえば，次のプログラムは if 文の構文解析ルーチンですが，コメントで挟み込んで書いているようにコードを生成するプログラムを書けば，if 文の目的プログラムが生成できます．

```
/* 分岐文 ::= "if" 式 "then" 文 ［ "else" 文 ］ */
        /*  どの"if"に対応するか曖昧な"else"は候補の内で最も近い"if"に対応するとする．*/
int p_ifst(void) {
        if(token != TIF) return(error("Keyword 'if' is not found"));
        token = scan();
        if(p_exp() == ERROR) return ERROR;
                /* p_exp()が条件式のコードを生成してくれていると仮定できるので，  */
                /* 新たなラベル L0001 を確保して，                                */
                /*          POP     gr1                                          */
                /*          CPA     gr1,gr0                                      */
                /*          JZE     L0001                                        */
                /* を生成する．                                                   */
                /* gr0 には常に定数 0 が入っているものとする．最初に 0 をセットして */
                /* それ以後値を変えなければよい．                                 */
        if(token != TTHEN) return(error("Keyword 'then' is not found"));
        token = scan();
        if(p_st() == ERROR) return ERROR;
                /* 同様に p_st()が真の場合の文のコードを生成してくれると仮定できる */
        if(token == TELSE) {
                        /* 新たなラベル L0002 を確保して，          */
                        /*          JUMP    L0002                   */
                        /* L0001                                    */
                        /* を生成する．                             */
                token = scan();
                if(p_st() == ERROR) return(ERROR);
                        /* 同様に p_st()が偽の場合の文のコードを生成してくれると仮定できる */
                        /* L0002                                    */
                        /* を生成する（ラベルを立てる）．           */
        }
        else {
                        /* else がないときは，ラベル                */
                        /* L0001                                    */
                        /* を生成するだけでよい．                       */
```

```
        return NORMAL;
}
```

このように文のコード生成は，条件分岐，無条件分岐，ラベル生成がほとんどです．

2.2 式のコード生成

同様に式の場合も，部分式は部分式の構文解析ルーチンが結果をスタックトップに置くコードを生成してくれると仮定してコード生成ができます．たとえば，次は項に対する構文解析ルーチンですが，コメントのようにコードを生成するプログラムを書けば良いわけです．

```
/* 項 ::= 因子 { 乗法演算子　因子 } */
/* 乗法演算子 ::= "*" | "div" | "and" */
int p_term(void) {
                /* ここに演算子を覚える作業用変数を宣言する．たとえば，    */
                /* int opr;                                                */
        if(p_factor() == ERROR) return ERROR;
                /* p_factor()が部分式のコードを生成してくれる        */
        while(token == TSTAR || token == TDIV || token == TAND) {
                        /* ここで，演算子を覚えておく．つまり      */
                        /* opr = token;                            */
                        /* を加える．                              */
                token = scan();
                if(p_factor() == ERROR) return ERROR;
                        /* p_factor()が部分式のコードを生成してくれる．      */
                        /* ここで被演算子が二つ揃うので，次のようにコード生成する．          */
                        /*         POP     gr2                             */
                        /*         POP     gr1                             */
                        /* もし opr が TSTAR なら，MULA  gr1,gr2           */
                        /* もし opr が TDIV なら，DIVA    gr1,gr2           */
                        /* もし opr が TAND なら，AND     gr1,gr2           */
                        /*         PUSH    0,gr1                           */
                        /* を生成する．                                    */
        }
        return NORMAL;
}
```

この例では，型のチェックは省略してありますが，本来は型のチェックを行うコードも入っている必要があります．

３．参照渡しについて

「参照渡し」とは，手続きを呼んだ側から呼ばれた手続きへ実引数を渡す渡し方の一つである．プログラミング言語によっては実引数が変数（l-value を持っている）のときのみ参照渡しが可能であるが，本演習では，次のように定義して，実引数が式の場合にでも参照渡しができるようにする．

（１）実引数の渡し方（変数の場合）

実引数は主記憶中に値を置く場所を持っているので，その番地を呼ばれた手続きへ渡す．

（2）実引数の渡し方（変数以外の式や定数の場合）

実引数は値のみであり，場所を持っていないので，作業用の領域を確保してその場所にその値を格納し，その場所の番地を呼ばれた手続きへ渡す．再帰呼び出しが許されていないので，その場所は各仮引数毎に一つずつでよい．

（3）仮引数の参照法

呼ばれた手続き内で仮引数が参照されたとき，その参照が値を使用する場合でも代入される場合でも，渡された番地が示す場所を参照する．

出力例：sample11pp.mpl

```
program sample11pp;
procedure kazuyomikomi(n : integer);
begin
        writeln('input the number of data');
        readln(n)
end;
var sum : integer;
procedure wakakidasi;
begin
        writeln('Sum of data = ', sum)
end;
var data : integer;
procedure goukei(n, s : integer);
        var data : integer;
begin
        s := 0;
        while n > 0 do begin
                readln(data);
                s := s + data;
                n := n - 1
        end
end;
var n : integer;
begin
        call kazuyomikomi(n);
        call goukei(n * 2, sum);
        call wakakidasi
end.
---------------------------------------------
; program sample11pp;
$$sample11pp    START
        LAD     gr0,0
        CALL    L0001
        CALL    FLUSH
        SVC     0
; procedure kazuyomikomi(n : integer);
$n%kazuyomikomi DC      0
; begin
;       writeln('input the number of data');
$kazuyomikomi
        POP     gr2
        POP     gr1
        ST      gr1,    $n%kazuyomikomi
        PUSH    0,gr2
        LAD     gr1,    L0002
        LD      gr2,gr0
        CALL    WRITESTR
;       readln(n)
        CALL    WRITELINE
; end;
        LD      gr1,    $n%kazuyomikomi
        CALL    READINT
        CALL    READLINE
; var sum : integer;
        RET
; procedure wakakidasi;
$sum    DC      0
; begin
;       writeln('Sum of data = ', sum)
$wakakidasi
        LAD     gr1,    L0003
        LD      gr2,gr0
        CALL    WRITESTR
; end;
        LD      gr1,    $sum
        LD      gr2,gr0
        CALL    WRITEINT
        CALL    WRITELINE
; var data : integer;
        RET
; procedure goukei(n, s : integer);
$data   DC      0
$s%goukei DC    0
$n%goukei DC    0
;       var data : integer;
; begin
$data%goukei    DC      0
;       s := 0;
$goukei
        POP     gr2
        POP     gr1
        ST      gr1,    $s%goukei
        POP     gr1
        ST      gr1,    $n%goukei
        PUSH    0,gr2
        LD      gr1,    $s%goukei
        PUSH    0,gr1
        LAD     gr1,    0
;       while n > 0 do begin
        POP     gr2
        ST      gr1,0,gr2
L0004
        LD      gr1,    $n%goukei
        LD      gr1,0,gr1
        PUSH    0,gr1
        LAD     gr1,    0
        POP     gr2
        CPA     gr2,gr1
        JPL     L0006
        LD      gr1,gr0
        JUMP    L0007
L0006
        LAD     gr1,1
L0007
        CPA     gr1,gr0
        JZE     L0005
;               readln(data);
        LAD     gr1,    $data%goukei
        CALL    READINT
        CALL    READLINE
;               s := s + data;
        LD      gr1,    $s%goukei
        PUSH    0,gr1
        LD      gr1,    $s%goukei
        LD      gr1,0,gr1
        PUSH    0,gr1
;               n := n - 1
        LD      gr1,    $data%goukei
```

```
            POP     gr2
            ADDA    gr1,gr2
            JOV     EOVF
            POP     gr2
            ST      gr1,0,gr2
            LD      gr1,    $n%goukei
            PUSH    0,gr1
            LD      gr1,    $n%goukei
            LD      gr1,0,gr1
;           end
            PUSH    0,gr1
            LAD     gr1,    1
; end;
            POP     gr2
            SUBA    gr2,gr1
            JOV     EOVF
            LD      gr1,gr2
            POP     gr2
            ST      gr1,0,gr2
            JUMP    L0004
L0005
; var n : integer;
            RET
; begin
$n          DC      0
;           call kazuyomikomi(n);
L0001
            LAD     gr1,    $n
            PUSH    0,gr1
;           call goukei(n * 2, sum);
            CALL    $kazuyomikomi
            LAD     gr1,    $n
            LD      gr1, 0,gr1
            PUSH    0,gr1
            LAD     gr1,    2
            POP     gr2
            MULA    gr1,gr2
            JOV     EOVF
            LAD     gr2,    L0008
            ST      gr1, 0,gr2
            PUSH    0,gr2
            LAD     gr1,    $sum
            PUSH    0,gr1
;           call wakakidasi
            CALL    $goukei
; end.
            CALL    $wakakidasi
            RET
L0002       DC      'input the number of data'
L0003       DC      'Sum of data = '
L0008       DC      0
EOVF
  CALL  WRITELINE
  LAD   gr1, EOVF1
  LD    gr2, gr0
  CALL  WRITESTR
  CALL  WRITELINE
  SVC   1   ;   overflow error stop
EOVF1     DC  '***** Run-Time Error : Overflow *****'
E0DIV
  JNZ   EOVF
  CALL  WRITELINE
  LAD   gr1, E0DIV1
  LD    gr2, gr0
  CALL  WRITESTR
  CALL  WRITELINE
  SVC   2   ;   0-divide error stop
E0DIV1    DC  '***** Run-Time Error : Zero-Divide *****'
EROV
  CALL  WRITELINE
  LAD   gr1, EROV1
  LD    gr2, gr0
  CALL  WRITESTR
  CALL  WRITELINE
  SVC   3   ;   range-over error stop
EROV1     DC  '***** Run-Time Error : Range-Over in Array
Index *****'
WRITECHAR
; gr1 の値（文字）を gr2 のけた数で出力する.
; gr2 が 0 なら必要最小限の桁数で出力する
  RPUSH
  LD    gr6, SPACE
  LD    gr7, OBUFSIZE
WC1
  SUBA  gr2, ONE    ; while(--c > 0) {
  JZE   WC2
  JMI   WC2
  ST    gr6, OBUF,gr7   ;   *p++ = ' ';
  CALL  BOVFCHECK
  JUMP  WC1   ; }
WC2
  ST    gr1, OBUF,gr7   ; *p++ = gr1;
  CALL  BOVFCHECK
  ST    gr7, OBUFSIZE
  RPOP
  RET
WRITESTR
; gr1 が指す文字列を gr2 のけた数で出力する.
; gr2 が 0 なら必要最小限の桁数で出力する
  RPUSH
  LD    gr6, gr1    ; p = gr1;
WS1
  LD    gr4, 0,gr6    ; while(*p != '¥0') {
  JZE   WS2
  ADDA  gr6, ONE    ;   p++;
  SUBA  gr2, ONE    ;   c--;
  JUMP  WS1   ; }
WS2
  LD    gr7, OBUFSIZE   ; q = OBUFSIZE;
  LD    gr5, SPACE
WS3
  SUBA  gr2, ONE    ; while(--c >= 0) {
  JMI   WS4
  ST    gr5, OBUF,gr7   ;   *q++ = ' ';
  CALL  BOVFCHECK
  JUMP  WS3   ; }
WS4
  LD    gr4, 0,gr1    ; while(*gr1 != '¥0') {
  JZE   WS5
  ST    gr4, OBUF,gr7   ;   *q++ = *gr1++;
```

```
   ADDA   gr1, ONE
   CALL   BOVFCHECK
   JUMP   WS4  ; }
WS5
   ST   gr7, OBUFSIZE  ; OBUFSIZE = q;
   RPOP
   RET
BOVFCHECK
      ADDA   gr7, ONE
      CPA    gr7, BOVFLEVEL
      JMI   BOVF1
      CALL   WRITELINE
      LD gr7, OBUFSIZE
BOVF1
      RET
BOVFLEVEL   DC 256
WRITEINT
; gr1 の値（整数）を gr2 のけた数で出力する.
; gr2 が 0 なら必要最小限の桁数で出力する
   RPUSH
   LD   gr7, gr0   ; flag = 0;
   CPA   gr1, gr0   ; if(gr1>=0) goto WI1;
   JPL   WI1
   JZE   WI1
   LD   gr4, gr0   ; gr1= - gr1;
   SUBA   gr4, gr1
   CPA   gr4, gr1
   JZE   WI6
   LD   gr1, gr4
   LD   gr7, ONE   ; flag = 1;
WI1
   LD   gr6, SIX   ; p = INTBUF+6;
   ST   gr0, INTBUF,gr6   ; *p = '¥0';
   SUBA   gr6, ONE   ; p--;
   CPA   gr1, gr0   ; if(gr1 == 0)
   JNZ   WI2
   LD   gr4, ZERO   ;   *p = '0';
   ST   gr4, INTBUF,gr6
   JUMP   WI5  ; }
WI2         ; else {
   CPA   gr1, gr0   ;   while(gr1 != 0) {
   JZE   WI3
   LD   gr5, gr1   ;    gr5 = gr1 - (gr1 / 10) * 10;
   DIVA   gr1, TEN   ;    gr1 /= 10;
   LD   gr4, gr1
   MULA   gr4, TEN
   SUBA   gr5, gr4
   ADDA   gr5, ZERO   ;    gr5 += '0';
   ST   gr5, INTBUF,gr6   ;    *p = gr5;
   SUBA   gr6, ONE   ;    p--;
   JUMP   WI2   ;   }
WI3
   CPA   gr7, gr0   ;   if(flag != 0) {
   JZE   WI4
   LD   gr4, MINUS   ;    *p = '-';
   ST   gr4, INTBUF,gr6
   JUMP   WI5   ;   }
WI4
   ADDA   gr6, ONE   ;   else p++;
     ; }
WI5
   LAD   gr1, INTBUF,gr6   ; gr1 = p;
   CALL   WRITESTR   ; WRITESTR();
   RPOP
   RET
WI6
   LAD   gr1, MMINT
   CALL   WRITESTR   ; WRITESTR();
   RPOP
   RET
MMINT      DC   '-32768'
WRITEBOOL
; gr1 の値（真理値）が 0 なら'FALSE'を
; 0 以外なら'TRUE'を gr2 のけた数で出力する.
; gr2 が 0 なら必要最小限の桁数で出力する
   RPUSH
   CPA   gr1, gr0   ; if(gr1 != 0)
   JZE   WB1
   LAD   gr1, WBTRUE   ;   gr1 = "TRUE";
   JUMP   WB2
WB1         ; else
   LAD   gr1, WBFALSE   ;   gr1 = "FALSE";
WB2
   CALL   WRITESTR   ; WRITESTR();
   RPOP
   RET
WBTRUE      DC   'TRUE'
WBFALSE      DC   'FALSE'
WRITELINE
; 改行を出力する
   RPUSH
   LD   gr7, OBUFSIZE
   LD   gr6, NEWLINE
   ST   gr6, OBUF,gr7
   ADDA   gr7, ONE
   ST   gr7, OBUFSIZE
   OUT   OBUF, OBUFSIZE
   ST   gr0, OBUFSIZE
   RPOP
   RET
FLUSH
   RPUSH
   LD gr7, OBUFSIZE
   JZE FL1
   CALL WRITELINE
FL1
   RPOP
   RET
READCHAR
; gr1 が指す番地に文字一つを読み込む
   RPUSH
   LD   gr5, RPBBUF   ; if(RPBBUF != '¥0') {
   JZE   RC0
   ST   gr5, 0,gr1   ;   *gr1 = RPBBUF;
   ST   gr0, RPBBUF   ;   RPBBUF = '¥0'
   JUMP   RC3   ;   return; }
RC0
   LD   gr7, INP   ; inp = INP;
   LD   gr6, IBUFSIZE   ; if(IBUFSIZE == 0) {
   JNZ   RC1
```

```
   IN   IBUF, IBUFSIZE   ;   IN();
   LD   gr7, gr0   ;   inp = 0;
      ; }
RC1
   CPA   gr7, IBUFSIZE   ; if(inp == IBUFSIZE) {
   JNZ   RC2
   LD   gr5, NEWLINE   ;   *gr1 = '¥n';
   ST   gr5, 0,gr1
   ST   gr0, IBUFSIZE   ;   IBUFSIZE = INP = 0;
   ST   gr0, INP
   JUMP   RC3   ; }
RC2          ; else {
   LD   gr5, IBUF,gr7   ;   *gr1 = *inp++;
   ADDA   gr7, ONE
   ST   gr5, 0,gr1
   ST   gr7, INP   ;   INP = inp;
RC3          ; }
   RPOP
   RET
READINT
; gr1 が指す番地に整数値一つを読み込む
   RPUSH
RI1          ; do {
   CALL   READCHAR   ;   ch = READCHAR();
   LD   gr7, 0,gr1
   CPA   gr7, SPACE   ; } while(ch==' ' || ch=='¥t' || ch=='¥n');
   JZE   RI1
   CPA   gr7, TAB
   JZE   RI1
   CPA   gr7, NEWLINE
   JZE   RI1
   LD   gr5, ONE   ; flag = 1
   CPA   gr7, MINUS   ; if(ch == '-') {
   JNZ   RI4
   LD   gr5, gr0   ;   flag = 0;
   CALL   READCHAR   ;   ch = READCHAR();
   LD   gr7, 0,gr1
RI4          ; }
   LD   gr6, gr0   ; v = 0;
RI2
   CPA   gr7, ZERO   ; while('0' <= ch && ch <= '9') {
   JMI   RI3
   CPA   gr7, NINE
   JPL   RI3
   MULA   gr6, TEN   ;   v = v*10+ch-'0';
   ADDA   gr6, gr7
   SUBA   gr6, ZERO
   CALL   READCHAR   ;   ch = READSCHAR();
   LD   gr7, 0,gr1
   JUMP   RI2   ; }
RI3
   ST   gr7, RPBBUF   ; ReadPushBack();
   ST   gr6, 0,gr1   ; *gr1 = v;
   CPA   gr5, gr0   ; if(flag == 0) {
   JNZ   RI5
   SUBA   gr5, gr6   ;   *gr1 = -v;
   ST   gr5, 0,gr1
RI5          ; }
   RPOP
   RET
READLINE
;   入力を改行コードまで（改行コードも含む）読み飛ばす
   ST   gr0, IBUFSIZE
   ST   gr0, INP
   ST   gr0, RPBBUF
   RET
ONE      DC   1
SIX      DC   6
TEN      DC   10
SPACE      DC   #0020   ; ' '
MINUS      DC   #002D   ; '-'
TAB      DC   #0009   ; '¥t'
ZERO      DC   #0030   ; '0'
NINE      DC   #0039   ; '9'
NEWLINE      DC   #000A   ; '¥n'
INTBUF      DS   8
OBUFSIZE   DC   0
IBUFSIZE   DC   0
INP      DC   0
OBUF      DS   257
IBUF      DS   257
RPBBUF      DC   0
            END
```

出力例：sample35.mpl

```
program typeconv;
 var i : integer; b : boolean; c : char;
begin
   i := integer(false);
   while i <= integer(true) do begin
      writeln( boolean(i), ' : ', i);
      i := i + 1;
   end;
   writeln;
   i := integer(' ');
   while i < 127 do begin
      if i div 16 * 16 = i then writeln;
      write(char(i), ' ');
      i := i + 1;
   end;
   writeln
end.
```

```
; program typeconv;
$$typeconvSTART
            LAD         gr0,0
            CALL        L0001
            CALL        FLUSH
            SVC         0
;   var i : integer; b : boolean; c : char;
$i          DC          0
$b          DC          0
; begin
$c          DC          0
;    i := integer(false);
L0001
            LD          gr1,gr0
;    while i <= integer(true) do begin
            ST          gr1,        $i
```

```
L0002
        LD      gr1,    $i
        PUSH    0,gr1
        LAD     gr1,1
        POP     gr2
        CPA     gr2,gr1
        JPL     L0004
        LAD     gr1,1
        JUMP    L0005
L0004
        LD      gr1,gr0
L0005
        CPA     gr1,gr0
        JZE     L0003
;       writeln( boolean(i), ' : ', i);
        LD      gr1,    $i
        CPA     gr1,gr0
        JZE     L0006
        LAD     gr1,1
L0006
        LD      gr2,gr0
        CALL    WRITEBOOL
        LAD     gr1,    L0007
        LD      gr2,gr0
        CALL    WRITESTR
        LD      gr1,    $i
        LD      gr2,gr0
        CALL    WRITEINT
;       i := i + 1;
        CALL    WRITELINE
        LD      gr1,    $i
        PUSH    0,gr1
        LAD     gr1,    1
;   end;
        POP     gr2
        ADDA    gr1,gr2
        JOV     EOVF
        ST      gr1,    $i
;   writeln;
        JUMP    L0002
L0003
;   i := integer(' ');
        CALL    WRITELINE
        LAD     gr1,    32
;   while i < 127 do begin
        ST      gr1,    $i
L0008
        LD      gr1,    $i
        PUSH    0,gr1
        LAD     gr1,    127
        POP     gr2
        CPA     gr2,gr1
        JMI     L0010
        LD      gr1,gr0
        JUMP    L0011
L0010
        LAD     gr1,1
L0011
        CPA     gr1,gr0
        JZE     L0009
;       if i div 16 * 16 = i then writeln;
        LD      gr1,    $i
        PUSH    0,gr1
        LAD     gr1,    16
        POP     gr2
        DIVA    gr2,gr1
        JOV     E0DIV
        LD      gr1,gr2
        PUSH    0,gr1
        LAD     gr1,    16
        POP     gr2
        MULA    gr1,gr2
        JOV     EOVF
        PUSH    0,gr1
        LD      gr1,    $i
        POP     gr2
        CPA     gr2,gr1
        JZE     L0014
        LD      gr1,gr0
        JUMP    L0015
L0014
        LAD     gr1,1
L0015
        CPA     gr1,gr0
        JZE     L0012
;       write(char(i), ' ');
        CALL    WRITELINE
L0012
        LD      gr1,    $i
        LAD     gr2,127
        AND     gr1,gr2
        LD      gr2,gr0
        CALL    WRITECHAR
        LAD     gr1,    32
        LD      gr2,gr0
        CALL    WRITECHAR
;       i := i + 1;
        LD      gr1,    $i
        PUSH    0,gr1
        LAD     gr1,    1
;   end;
        POP     gr2
        ADDA    gr1,gr2
        JOV     EOVF
        ST      gr1,    $i
;   writeln
        JUMP    L0008
L0009
; end.
        CALL    WRITELINE
;
        RET
L0007   DC      ' : '
EOVF
  CALL  WRITELINE
  LAD   gr1, EOVF1
  LD    gr2, gr0
  CALL  WRITESTR
  CALL  WRITELINE
  SVC   1   ;   overflow error stop
```

```
EOVF1     DC  '***** Run-Time Error : Overflow *****'
E0DIV
  JNZ  EOVF
  CALL  WRITELINE
  LAD  gr1, E0DIV1
  LD  gr2, gr0
  CALL  WRITESTR
  CALL  WRITELINE
  SVC  2  ;  0-divide error stop
E0DIV1     DC  '***** Run-Time Error : Zero-Divide *****'
EROV
  CALL  WRITELINE
  LAD  gr1, EROV1
  LD  gr2, gr0
  CALL  WRITESTR
  CALL  WRITELINE
  SVC  3  ;  range-over error stop
EROV1     DC  '***** Run-Time Error : Range-Over in Array
Index *****'
WRITECHAR
; gr1 の値（文字）を gr2 のけた数で出力する.
; gr2 が 0 なら必要最小限の桁数で出力する
  RPUSH
  LD  gr6, SPACE
  LD  gr7, OBUFSIZE
WC1
  SUBA  gr2, ONE  ; while(--c > 0) {
  JZE  WC2
  JMI  WC2
  ST  gr6, OBUF,gr7  ;  *p++ = ' ';
  CALL  BOVFCHECK
  JUMP  WC1  ; }
WC2
  ST  gr1, OBUF,gr7  ; *p++ = gr1;
  CALL  BOVFCHECK
  ST  gr7, OBUFSIZE
  RPOP
  RET
WRITESTR
; gr1 が指す文字列を gr2 のけた数で出力する.
; gr2 が 0 なら必要最小限の桁数で出力する
  RPUSH
  LD  gr6, gr1  ; p = gr1;
WS1
  LD  gr4, 0,gr6  ; while(*p != '¥0') {
  JZE  WS2
  ADDA  gr6, ONE  ;  p++;
  SUBA  gr2, ONE  ;  c--;
  JUMP  WS1  ; }
WS2
  LD  gr7, OBUFSIZE  ; q = OBUFSIZE;
  LD  gr5, SPACE
WS3
  SUBA  gr2, ONE  ; while(--c >= 0) {
  JMI  WS4
  ST  gr5, OBUF,gr7  ;  *q++ = ' ';
  CALL  BOVFCHECK
  JUMP  WS3  ; }
WS4
  LD  gr4, 0,gr1  ; while(*gr1 != '¥0') {
  JZE  WS5
  ST  gr4, OBUF,gr7  ;  *q++ = *gr1++;
  ADDA  gr1, ONE
  CALL  BOVFCHECK
  JUMP  WS4  ; }
WS5
  ST  gr7, OBUFSIZE  ; OBUFSIZE = q;
  RPOP
  RET
BOVFCHECK
    ADDA  gr7, ONE
    CPA  gr7, BOVFLEVEL
    JMI  BOVF1
    CALL  WRITELINE
    LD gr7, OBUFSIZE
BOVF1
    RET
BOVFLEVEL  DC 256
WRITEINT
; gr1 の値（整数）を gr2 のけた数で出力する.
; gr2 が 0 なら必要最小限の桁数で出力する
  RPUSH
  LD  gr7, gr0  ; flag = 0;
  CPA  gr1, gr0  ; if(gr1>=0) goto WI1;
  JPL  WI1
  JZE  WI1
  LD  gr4, gr0  ; gr1= - gr1;
  SUBA  gr4, gr1
  CPA  gr4, gr1
  JZE  WI6
  LD  gr1, gr4
  LD  gr7, ONE  ; flag = 1;
WI1
  LD  gr6, SIX  ; p = INTBUF+6;
  ST  gr0, INTBUF,gr6  ; *p = '¥0';
  SUBA  gr6, ONE  ; p--;
  CPA  gr1, gr0  ; if(gr1 == 0)
  JNZ  WI2
  LD  gr4, ZERO  ;  *p = '0';
  ST  gr4, INTBUF,gr6
  JUMP  WI5  ; }
WI2       ; else {
  CPA  gr1, gr0  ;  while(gr1 != 0) {
  JZE  WI3
  LD  gr5, gr1  ;    gr5 = gr1 - (gr1 / 10) * 10;
  DIVA  gr1, TEN  ;    gr1 /= 10;
  LD  gr4, gr1
  MULA  gr4, TEN
  SUBA  gr5, gr4
  ADDA  gr5, ZERO  ;    gr5 += '0';
  ST  gr5, INTBUF,gr6  ;    *p = gr5;
  SUBA  gr6, ONE  ;    p--;
  JUMP  WI2  ;  }
WI3
  CPA  gr7, gr0  ;  if(flag != 0) {
  JZE  WI4
  LD  gr4, MINUS  ;    *p = '-';
  ST  gr4, INTBUF,gr6
  JUMP  WI5  ;  }
WI4
```

```
    ADDA   gr6, ONE   ;   else p++;
      ; }
WI5
    LAD   gr1, INTBUF,gr6   ; gr1 = p;
    CALL   WRITESTR   ; WRITESTR();
    RPOP
    RET
WI6
    LAD   gr1, MMINT
    CALL   WRITESTR   ; WRITESTR();
    RPOP
    RET
MMINT       DC   '-32768'
WRITEBOOL
; gr1 の値（真理値）が 0 なら'FALSE'を
; 0 以外なら'TRUE'を gr2 のけた数で出力する.
; gr2 が 0 なら必要最小限の桁数で出力する
    RPUSH
    CPA   gr1, gr0   ; if(gr1 != 0)
    JZE   WB1
    LAD   gr1, WBTRUE   ;   gr1 = "TRUE";
    JUMP   WB2
WB1          ; else
    LAD   gr1, WBFALSE   ;   gr1 = "FALSE";
WB2
    CALL   WRITESTR   ; WRITESTR();
    RPOP
    RET
WBTRUE       DC   'TRUE'
WBFALSE      DC   'FALSE'
WRITELINE
; 改行を出力する
    RPUSH
    LD   gr7, OBUFSIZE
    LD   gr6, NEWLINE
    ST   gr6, OBUF,gr7
    ADDA   gr7, ONE
    ST   gr7, OBUFSIZE
    OUT   OBUF, OBUFSIZE
    ST   gr0, OBUFSIZE
    RPOP
    RET
FLUSH
    RPUSH
    LD gr7, OBUFSIZE
    JZE FL1
    CALL WRITELINE
FL1
    RPOP
    RET
READCHAR
; gr1 が指す番地に文字一つを読み込む
    RPUSH
    LD   gr5, RPBBUF   ; if(RPBBUF != '¥0') {
    JZE   RC0
    ST   gr5, 0,gr1   ;   *gr1 = RPBBUF;
    ST   gr0, RPBBUF   ;   RPBBUF = '¥0'
    JUMP   RC3   ;   return; }
RC0
    LD   gr7, INP   ; inp = INP;
    LD   gr6, IBUFSIZE   ; if(IBUFSIZE == 0) {
    JNZ   RC1
    IN   IBUF, IBUFSIZE   ;   IN();
    LD   gr7, gr0   ;   inp = 0;
      ; }
RC1
    CPA   gr7, IBUFSIZE   ; if(inp == IBUFSIZE) {
    JNZ   RC2
    LD   gr5, NEWLINE   ;   *gr1 = '¥n';
    ST   gr5, 0,gr1
    ST   gr0, IBUFSIZE   ;   IBUFSIZE = INP = 0;
    ST   gr0, INP
    JUMP   RC3   ; }
RC2          ; else {
    LD   gr5, IBUF,gr7   ;   *gr1 = *inp++;
    ADDA   gr7, ONE
    ST   gr5, 0,gr1
    ST   gr7, INP   ;   INP = inp;
RC3          ; }
    RPOP
    RET
READINT
; gr1 が指す番地に整数値一つを読み込む
    RPUSH
RI1          ; do {
    CALL   READCHAR   ;   ch = READCHAR();
    LD   gr7, 0,gr1
    CPA   gr7, SPACE   ; } while(ch==' ' || ch=='¥t' || ch=='¥n');
    JZE   RI1
    CPA   gr7, TAB
    JZE   RI1
    CPA   gr7, NEWLINE
    JZE   RI1
    LD   gr5, ONE   ; flag = 1
    CPA   gr7, MINUS   ; if(ch == '-') {
    JNZ   RI4
    LD   gr5, gr0   ;   flag = 0;
    CALL   READCHAR   ;   ch = READCHAR();
    LD   gr7, 0,gr1
RI4          ; }
    LD   gr6, gr0   ; v = 0;
RI2
    CPA   gr7, ZERO   ; while('0' <= ch && ch <= '9') {
    JMI   RI3
    CPA   gr7, NINE
    JPL   RI3
    MULA   gr6, TEN   ;   v = v*10+ch-'0';
    ADDA   gr6, gr7
    SUBA   gr6, ZERO
    CALL   READCHAR   ;   ch = READSCHAR();
    LD   gr7, 0,gr1
    JUMP   RI2   ; }
RI3
    ST   gr7, RPBBUF   ; ReadPushBack();
    ST   gr6, 0,gr1   ; *gr1 = v;
    CPA   gr5, gr0   ; if(flag == 0) {
    JNZ   RI5
    SUBA   gr5, gr6   ;   *gr1 = -v;
    ST   gr5, 0,gr1
RI5          ; }
```

```
   RPOP
   RET
READLINE
;  入力を改行コードまで（改行コードも含む）読み飛ばす
   ST   gr0, IBUFSIZE
   ST   gr0, INP
   ST   gr0, RPBBUF
   RET
ONE      DC   1
SIX      DC   6
TEN      DC   10
SPACE      DC   #0020   ; ' '
MINUS      DC   #002D   ; '-'
TAB      DC   #0009   ; '¥t'
ZERO      DC   #0030   ; '0'
NINE      DC   #0039   ; '9'
NEWLINE      DC   #000A   ; '¥n'
INTBUF      DS   8
OBUFSIZE   DC   0
IBUFSIZE   DC   0
INP      DC   0
OBUF      DS   257
IBUF      DS   257
RPBBUF      DC   0
               END
```