ソフトウェア演習V 課題３

（１）課題名：クロスリファレンサの作成
（２）演習期間：11/16 - 12/13
（３）レポート・プログラム提出期間：12/14 - 12/21

（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）がなければ，クロスリファレンス表を出力し，エラーがあれば，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．例えば，作成するプログラム名を cr，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

cr foo.mpl

とすれば(foo.mpl のみが引数)，foo.mpl の内容のプログラムのクロスリファレンス表を出力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，マクロ構文はそれぞれ課題１，２の通りである．MPPL の制約規則を，マクロ構文と共に示す．「//」以降の部分が制約規則またはそれに関連する意味である．ただし，長さが 8 以上の名前は，最初の 8 文字が一致していれば同一の名前としてもよい（もちろん，8 文字より長い先頭部分が一致していれば同一の名前とするようにしてもよい．さらに，全文字列が一致している場合にのみ同一の名前としても良い．どれにしたか，レポートに明記すること)．また，型の一致は構造一致で判定するものとする（型名の宣言がないので，この指定は特に意味はない)．なお，配列型の型が一致するとは，要素型が一致し要素数が同じであることを意味する．

プログラム ::= "program" "名前" ";" ブロック "."
　　// この"名前"はプログラムの名前なので，ブロック中の名前と同じでもかまわない
ブロック ::=｛ 変数宣言部 | 副プログラム宣言 ｝複合文
　　// この複合文中で使われる変数名はこの変数宣言部で宣言されていなければならない
　　// この複合文中で使われる手続き名は副プログラム宣言で宣言されていなければならない
　　// この変数宣言部で宣言されている変数名のスコープはプログラム全体である
　　// すべての名前は，プログラムテキスト中で使用される前に宣言されていなければならない
変数宣言部 ::= "var" 変数名の並び ":" 型 ";"｛ 変数名の並び ":" 型 ";" ｝
　　// 変数名の並びにある変数名の型をその次の":"の次の型とする
変数名の並び ::= 変数名 ｛ "," 変数名 ｝
　　// 同じスコープを持つ同じ名前が複数回宣言されていてはならない
変数名 ::= "名前"
型 ::= 標準型 | 配列型

標準型 ::= "integer" | "boolean" | "char"

// それぞれ integer 型，boolean 型，char 型を表す

配列型 ::= "array" "[" "符号なし整数" "]" "of" 標準型

// 標準型を要素型とする配列型を表す．"符号なし整数"は配列の大きさ(要素数)を表す

// 従って，"符号なし整数"の値は 1 以上でなくてはならない

// 添字の下限は 0 であり，上限は配列の大きさより 1 小さな値である(C と同じ)

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 ";"

// この手続き名のスコープはプログラム全体である

// この手続き名をこの複合文内で使用することはできない．再帰呼び出しはできない

// この仮引数部と変数宣言部で宣言されている名前のスコープはこの複合文である．

// すべての名前は，プログラムテキスト中で使用される前に宣言されていなければならない

// 静的スコープルールを採用する．手続き名，変数名を問わず，同じスコープを持つ複数の同

// じ名前の宣言があった場合には二重定義エラーである

手続き名 ::= "名前"

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

// 仮引数部に現れる型は標準型でなくてはならない

// 変数名の並びにある変数名の型をその次の":"の次の型とする

複合文 ::= "begin" 文 ｛ ";" 文 ｝ "end"

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 ］

// 式の型は boolean 型でなくてはならない

繰り返し文 ::= "while" 式 "do" 文

// 式の型は boolean 型でなくてはならない

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

// 手続き名は副プログラム宣言で手続き名として宣言されていなくてはならない

式の並び ::= 式 ｛ "," 式 ｝

// 式の数はその手続き名の宣言の仮引数部の変数の数と一致していなくてはならない

// 式と仮引数部の変数は順序で対応し，対応する式と変数は同じ標準型でなくてはならない

戻り文 ::= "return"

代入文 ::= 左辺部 ":=" 式

// 左辺部と式の型は同じ型で標準型でなくてはならない

左辺部 ::= 変数

// 左辺部の型は変数の型である

変数 ::= 変数名 ［ "[" 式 "]" ］

// 式が付いているときの変数名の型は array 型，式の型は integer 型でなくてはならない

// そのときの変数の型は array 型の要素型である．

// 式が付いていない時の変数の型は変数名の型である

式 ::= 単純式 ｛ 関係演算子　単純式 ｝

// 一つの単純式だけの式の型はその単純式の型である

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

// 関係演算子の被演算子の型は同じ標準型でなくてはならない．結果の型は boolean 型である
単純式 ::= ["+" | "-"] 項 { 加法演算子　項 }
// 一つの項だけの単純式の型はその項の型である
// "+"か"-"があるとき左の項の型は integer 型でなくてはならない．結果の型は integer 型である
加法演算子 ::= "+" | "-" | "or"
// "+"と"-"の被演算子の型は integer 型でなくてはならない．結果の型は integer 型である
// "or"の被演算子の型は boolean 型でなくてはならない．結果の型は boolean 型である
項 ::= 因子 { 乗法演算子　因子 }
// 一つの因子だけの項の型はその因子の型である
乗法演算子 ::= "*" | "div" | "and"
// "*"と"div"の被演算子の型は integer 型でなくてはならない．結果の型は integer 型である
// "and"の被演算子の型は boolean 型でなくてはならない．結果の型は boolean 型である
因子 ::= 変数 | 定数 | "(" 式 ")" | "not" 因子 | 標準型 "(" 式 ")"
// 変数や定数のとき，因子の型はそれぞれの型である
// "(" 式 ")"のとき，因子の型は式の型である．
// "not"の被演算子の型は boolean 型でなくてはならない．因子の型は boolean 型である
// 標準型 "(" 式 ")"のとき，因子の型はその標準型である．
// この式の型は標準型でなくてはならない
定数 ::= "符号なし整数" | "false" | "true" | "文字列"
// "符号なし整数"のとき，定数の型は integer 型，"false"と"true"のときは boolean 型である
// 文字列は長さが 1 でなくてはならず，そのときの定数の型は char 型である
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
出力文 ::= ("write" | "writeln") ["(" 出力指定 { "," 出力指定 } ")"]
出力指定 ::= 式 [":" "符号なし整数"] | "文字列"
// 式の型は標準型でなくてはならない
// 文字列は長さが 0 か 2 以上である．
空文 ::= ε

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，そのプログラムのクロスリファレンス表（下記参照）を標準出力へ出力せよ．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，検出した行の番号と，誤りまたは違反の内容（配列の添字の型が integer ではない，など）を標準出力へ出力せよ．

［クロスリファレンス表］

プログラム中に現れた名前の型，その名前が定義された行の番号，参照された（定義以外で現れた）行の番号を表形式で表したもの．定義の場合でも参照の場合でも，その名前が出現している行の番号を表示する．名前でソートされていることが望ましい（名前でソートすることは拡張仕様である）．たとえば，次のプログラム，

```
01:     program sample11;
02:     var n, sum, data : integer;
03:     begin
04:             writeln('input the number of data');
05:             readln(n);
06:             sum := 0;
07:             while n > 0 do begin
08:                     readln(data);
09:                     sum := sum + data;
10:                     n := n - 1
11:             end;
12:             writeln('Sum of data = ', sum)
13:     end.
```

のクロスリファレンス表は,

Name	Type	Def.	Ref.
data	integer	2	8,9
n	integer	2	5,7,10,10
sum	integer	2	6,9,9,12

のようなものが期待されている．これは一例である．Name，Type，Def，Ref はそれぞれ，名前，型，定義行，参照行を示している．プログラム名は含めなくてよい．型が配列型の時の型の表示は，例えば，「array［100］of integer」のようにせよ．手続き名の型の表示は引数がなければ「procedure」とし，例えば integer 型の引数が二つあれば「procedure(integer, integer)」のようにせよ．もし，名前が副プログラム宣言内で宣言されている変数名であれば，Name の欄は,

data : proc　　//　手続き proc の変数 data の意味

のように区別すること．また，仮引数名はその副プログラム宣言内で宣言されている変数名と同様に扱ってよい.

［課題 3 の実行例］

/* 入力 */

```
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.
```

/* 出力 */

```
-------------------------------------------------------------------------
Name                 Type                          Def.   Ref.
data                 integer                         12 |
data:goukei          integer                         14 | 18, 19
goukei               procedure(integer,integer)      13 | 26
kazuyomikomi         procedure(integer)               2 | 25
n                    integer                         23 | 25, 26
n:goukei             integer                         13 | 17, 20, 20
n:kazuyomikomi       integer                          2 | 5
s:goukei             integer                         13 | 16, 19, 19
sum                  integer                          7 | 10, 26
wakakidasi           procedure                        8 | 27
-------------------------------------------------------------------------
```

<u>クロスリファレンサの作り方</u>

課題２の結果を利用して作ります．コンパイルエラーとして新たにチェックする点は次の通りです．

(a) 変数の二重定義（同じスコープを持つ同じ名前が二度以上宣言されている）があれば，エラーとする．
(b) 変数の未定義（宣言されていない名前が式中などで使われている）があれば，エラーとする．
(c) 演算子には被演算子の型として許されるものが指定されている．それ以外の型であればエラーとする．
(d) 同様に if 文，while 文の条件式の型は boolean しか認められていないので，それ以外の型であれば，エラーとする．
(e) そのほか，型が指定されている部分の型が，それ以外の型であれば，エラーとする．

以上のチェックでエラーがなければ，クロスリファレンス表を出力します．
以上の処理は，以下のような手順で作ることができます．

（1）課題 2 のプリティプリント機能は不要なので，出力部分はすべて削除します．これは最後に行ったほうがよいかもしれません．

（２）宣言された変数の型を記憶しておく必要があります．つまり，記号表が必要です．そのためのモジュールとして，id-list.c が参考になるかもしれません（ただし，id-list.c は線形リストで記録する記号表なので，動くアルゴリズムではありますが，推奨できるアルゴリズムではありません）．struct　ID に型情報を記憶するメンバを追加すればいいわけです．また，クロスリファレンス表を出力するために，宣言された行番号や，出現した行番号を記録するメンバも必要です．課題１で必要だった出現個数は不要なので，関数 id_countup は不要ですが，代わりに，型情報を記憶するための関数，行番号を記憶する関数などが必要かもしれません．それらは関数 id_countup を参考にして作成することができます．なお，記号表には，名前の文字列と型情報以外に，種類（特に仮引数名か否か）も記録しておくとよいでしょう．

また，記号表は大域名用と副プログラム宣言内用の最大二つが同時に必要です．後者は副プログラム宣言が終わるごとに空にする必要があります．

以上を考慮すると，データ構造の宣言は次のようにするといいかもしれません．

```
struct TYPE {
        int ttype;              /* TPINT TPCHAR TPBOOL TPARRAY TPARRAYINT TPARRAYCHAR
                                   TPARRAYBOOL TPPROC */
        int arraysize;          /* size of array, if TPARRAY */
        struct TYPE *etp; /* pointer to element type if TPARRAY */
        struct TYPE *paratp; /* pointer to parameter's type list if ttype is TPPROC */
};

struct LINE {
        int reflinenum;
        struct LINE *nextlinep;
};

struct ID {
        char *name;
        char *procname; /* procedure name within which this name is defined */ /* NULL if global name */
        struct TYPE *itp;
        int ispara;             /* 1:formal parameter, 0:else（variable） */
        int deflinenum;
        struct LINE *irefp;
        struct ID *nextp;
}   *globalidroot, *localidroot;            /* Pointers to root of global & local symbol tables */
```

（３）構文解析用に作った関数で，値を持つもの（単純式とか項などに対応するもの）は，ERROR や NORMAL だけを返すだけではすみません．その型も返す必要があります．これは，関数値として返してもいいし，引数で返してもいいし，大域変数を経由して返してもかまいません．

（４）以上の型情報を利用して，型のチェックルーチンを必要な関数に組み込みます．

（５）最後に，記号表の中身を出力例のように出力するルーチンを書きます．副プログラム宣言用は，その宣言が終わるたびに出力すると楽ですが，それでは，名前の辞書式順に出力できません（副プログラムごとに出力されます）．全体を辞書式順に出力しようとすると工夫が必要です．