コンパイラ構成

意味解析

情報工学系
権藤克彦

記号表

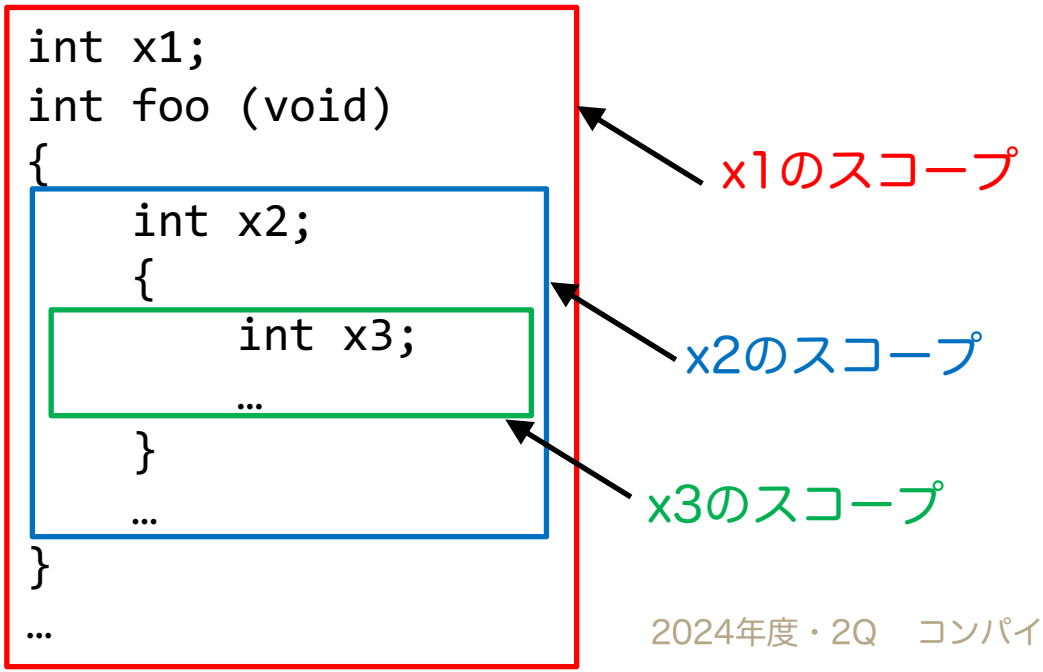
記号表

- 識別子を管理するための表
  - 識別子名，種類，型，名前空間などを管理
  - 変数名，関数名，ラベル，型名，文字列定数などが管理対象
- 記号表の目的
  - 未定義変数や未定義関数をチェック
  - 定義と参照で型があっているかチェック
- 記号表の作成にはスコープとエクステントに要配慮

寿命（生存期間）

スコープ scope（1/3）

- 変数や関数の（プログラムの位置での）可視範囲
  - cf. エクステント（寿命）
- スコープ
  - 外部変数x1は宣言から，そのファイルの最後まで
  - 局所変数は宣言から，そのブロックの最後まで



スコープ scope（2/3）

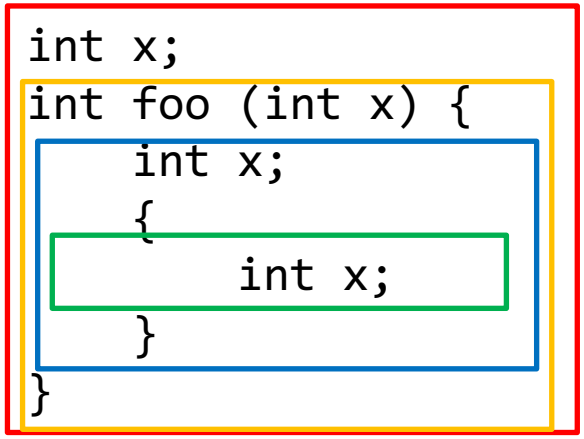
- 同じスコープで同名の変数や関数の宣言はエラー
  - 異なる名前空間ならエラーにならない．例：変数とラベル
- 別スコープで同名の宣言は内側スコープ優先
  - 以下で内側の x は，外側の x を隠蔽する

```
int x;
int x; // 2重定義エラー
int foo (void)
{
    int x;
    {
        int x;
        x = 10;
    }
    …
}
…
```

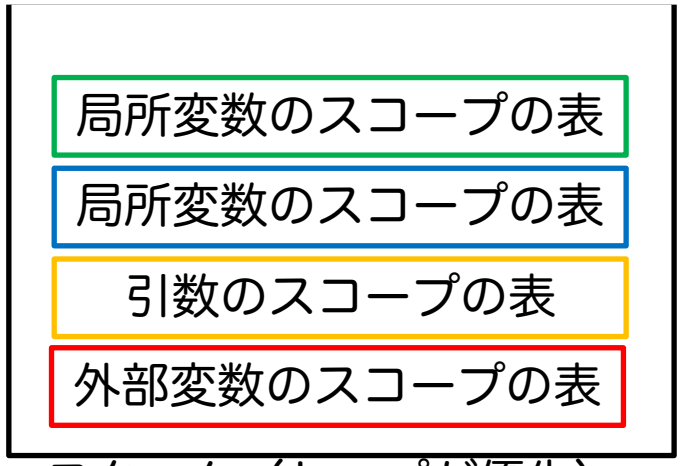
スコープ scope （3/3）

このコードはC言語規格では違法です．
引数xと関数ボディの一番外側のブロックが
同じスコープとなるからです

- スコープのまとめ
  - 同じスコープでは2重定義禁止．異なる識別子は同列に扱う
  - 異なるスコープでは同名の識別子を定義可能．内側が優先
- スコープの概念実装
  - 同じスコープは単純な1次元の表
  - 入れ子スコープはスタックで表現



入れ子のスコープ



スタック（トップが優先）

XCC: 教育用コンパイラ

- XCC=言語XCのコンパイラ
  - 配布資料 src/xcc-bison-dist-64
- 以下を備える
  - GNU Bisonによる構文解析器
  - 意味解析器
- 詳細はソースコードを読んでね
  - 実装方法は他にもいろいろあります.
  - 他人の（汚い）コードを読むのも勉強.
- コード生成器は演習２で作成

XCCでの記号のデータ表現

- 同じスコープ同士の記号は線形リストとして繋ぐ

```
enum NameSpace {                    // 名前空間
    NS_GLOBAL, NS_LOCAL, NS_ARG, NS_LABEL
};

struct Symbol {
  char            *name;        // シンボル名
  struct Type     *type;        // 型
  struct AST      *ast;         // 宣言されたAST上の位置
  int             offset;       // オフセット（局所変数と引数用）
  enum NameSpace name_space;    // 名前空間
  struct Symbol   *next;        // 次のエントリへのポインタ
};
```

注：C言語の名前空間

- C言語では次の4つの名前空間がある
  - ラベル，（構造体，共用体，enumの）タグ，
    （構造体，共用体の）メンバー，
    それ以外（変数名，関数名，typedef名，enum定数）

```
int foo;          // それ以外
struct foo {      // タグ名
    int foo;      // メンバー名
};
int main ()
{
    foo:          // ラベル
    return 0;
}
```

- 前ページの NameSpace はC言語の名前空間とは別（実装上の都合）

XCCでの記号表のデータ表現

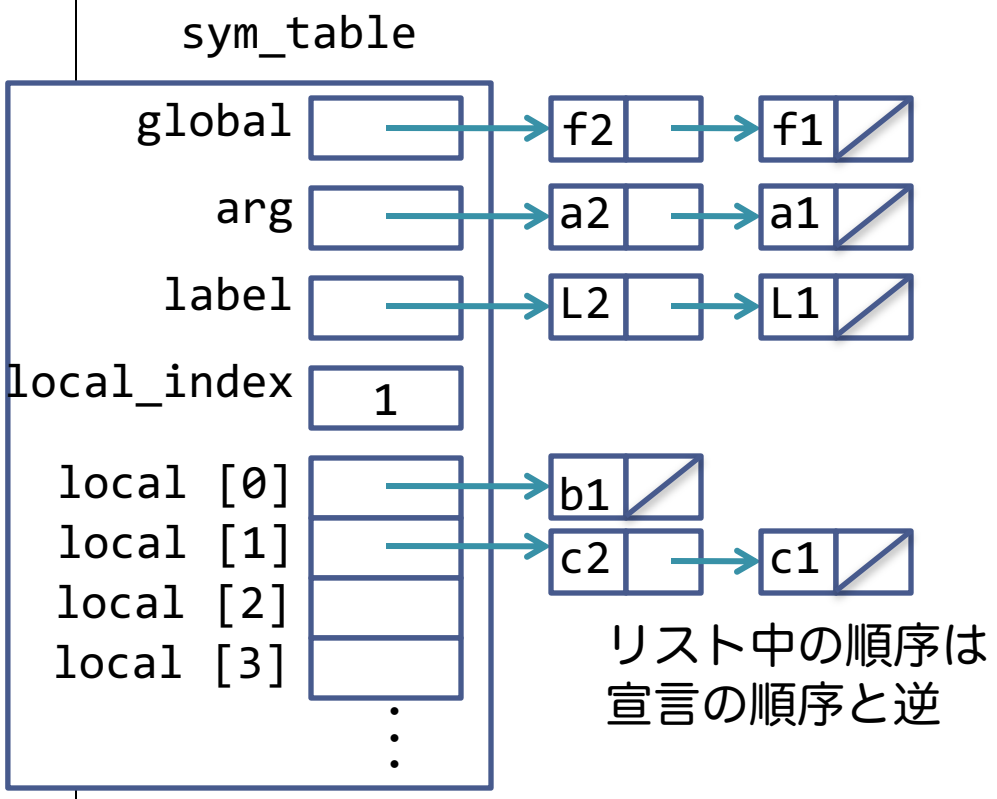
- 全体で SymTable構造体は1つだけ

```
enum { MAX_BLOCK_DEPTH = 128 };
struct SymTable {
    struct Symbol  *global;
    struct Symbol  *arg;
    struct Symbol  *label;
    struct String  *string;
    int            local_index;
    struct Symbol  *local [MAX_BLOCK_DEPTH]; // 記号表スタッ
ク
};
struct SymTable sym_table = {NULL, NULL, NULL, NULL, -1};
```

XCCの記号表：sym_tableの例

```
int f1;
int f2 (int a1, int a2)
{
  int b1;
  {
    int c1; int c2;
    /* (A) */
    L1: goto L2;
  }
  {
    int d1; int d2;
    L2: goto L1;
  }
}

int f3 (int a3)
{ int e1;; }
```



(A)地点のコード生成時の
記号表 sym_table の内容
(ラベルは構文解析時に解析済み)

XCCの記号表のダンプ（デバッグ用）

- XCCの実装コード中で，sym_table_dump() を呼ぶ
- XCコード中で sym_table_dumpを宣言する

foo.c

```
int sym_table_dump; //
(A)
```

```
% xcc foo.c
global:    f2, f1,
arg:       a2, a1,
label:
string:
local[0]: b1,
local[1]: c2, c1,
```

XCCコード中で記号表の
内容のダンプを指示

コード生成時ではなく
構文解析時の内容なので，
ラベルの前方参照は未表示.

XCCの記号表の処理の概要

- 構文解析時に記号表を構築
  - AST構造体の以下のメンバーにセット
    - global, arg, label, string, local
  - 同じスコープに同名あり→二重定義エラー
- コード生成時に以下の関数で記号を検索
  - sym_lookup, sym_lookup_label, string_lookup
  - 検索失敗→未定義エラー
- 大域変数 sym_table が記号表本体（1つのみ）
- コード生成時，ASTを移動する際に記号表を要修正
  - 関数を訪問・離脱する際に次の関数を呼ぶ
    - codegen_begin_function, codegen_end_function
  - ブロックを訪問・離脱する際に次の関数を呼ぶ
    - codegen_begin_block, codegen_end_block

XCCのAST.h （1/2）

```
struct AST {
    /* 共通メンバー */
    char            *ast_type;
    struct AST      *parent;
    int             nth;
    int             num_child;
    struct AST      **child;
```

XCCのAST.h （1/2）

```
    /* 特定の ASTノード用メンバー */
    struct Type    *type; // 宣言と式の型
    union {
       char   *id; // for AST_IDENTIFIER, AST_expression_string
       int    int_val;   // for AST_expression_int
       struct {
            int    total_local_size;
            struct Symbol *global;
            struct Symbol *arg;
            struct Symbol *label;
            struct String *string;
       } func;                    // for AST_function_definition
       struct Symbol *local;  // for AST_compound_statement
       int    arg_size; // for AST_argument_expression_list_*
    } u;
};
```

前方参照 (forward reference)

- 前方参照＝定義より先に参照が出現すること
- C言語やXC言語ではラベルのみ前方参照が起こる

```
int foo (void)
{
    goto bar; // ラベルの参照
    …
bar: // ラベルの定義
}
```

- 定義より先に参照が出現→仮に記号表に未定義としてエントリ作成．後で定義が出現→登録＆OKとする
- XCCでは2パス（構文解析とコード生成）なので問題なし

文字列定数の処理

- 要求
  - 文字列定数には，ユニークなラベルを割り振りたい
  - 文字列定数は，複数箇所に同じ文字列定数が出現しても，以下のアセンブリコードとして，1つにまとめたい
    - C言語の文字列定数が書き替え不可なのは，1つにまとめるから

```
int main (void)
{
    char *s1 = "hello\n";
    char *s2 = "hello\n";
}
```

```
L_str_23:
    .asciz  "hello\n"
```

- 解決法：文字列定数も記号表で管理

XCCの文字列定数の処理

- データ表現

```
struct String {
    char *data;  // 文字列定数（例："Hello"）
    char *label; // ユニークなラベル（例："L_str_23"）
    struct String  *next;
};
```

- ユニークなラベルの生成

```
static char *
create_string_label (void)
{
    static int num = 0; // オーバーフローしないことを仮定
    char *label = emalloc (32);
    snprintf (label, 32, "STR%d", num++);
    return label;
}
```

cf. Linuxの list_head 構造体

リスト構造：外付け v.s. 内付け

- nextメンバーを別構造体にする実装も当然あり
  - C言語では汎用コンテナの実装難（パラメタ多相が無い
    → 中身ごとにコンテナを別々に要定義（無駄））

```
struct String {
    char *data;
    char *label;
};

struct StringList {
    struct String string;
    struct String *next;
};
```

パラメタ多相
=ジェネリクス

多相=polymorphism

- データ構造とモジュール分割の良い設計は難しい
  - 作って変更してみないと，正解は分からないから

型解析

型 (type)

- 型=プログラミング言語が扱うデータの種類（集合）
  - 例：int型は，32ビット長の場合，-2^{31}～$2^{31}-1$の範囲の整数
- 型検査（型付け）＝変数や関数や式や宣言の型を決める（確認する）ことで，データエラーを検出

型の種類

- 原始型 (primitive type)
  - 最初から型としてプログラミング言語に備わっている型
  - 例：整数型，浮動小数点数型，文字型，文字列型，
- ユーザ定義型 (user-defined type, derived type)
  - 他の型を組み合わせて定義して作り出す新しい型
  - 例：直積型 (struct)，直和型 (union)，関数型，
    多相型（多態型）
  - 多相型（ポリモルフィズム polymorphism．複数の型を持つこと）
    - パラメタ型多相 （Javaのジェネリクス．Hash<String> ）
    - 包含多相 （オブジェクト指向言語の継承．人間は動物でもある）

静的型付けと動的型付け

- 静的型付け＝コンパイル時に型検査を行う
  - C, Javaなど手続き型言語に多い
- 動的型付け＝実行時に型検査を行う
  - Ruby, JavaScriptなどスクリプト言語に多い

- 境界が微妙な場合も
  - 例：Javaは静的型付けだが，ダウンキャストやリフレクションを使う部分は動的型付けになる

ダウンキャストの例：親クラスのインスタンスを子クラスに型変換

静的

- 静的型付けのメリット
  - コンパイルエラーが出ること（実行前に確認可能）
    - スペルミス，存在しない変数や関数の検出
  - プログラマの意図が（多少）分かる
  - 実行速度が速い（実行時に型検査不要だから）
  - IDEによる支援が可能（型チェックや入力補完など）
- 動的型付けのメリット
  - 型宣言が不要
    - 最近の静的型言語は推論可能なら型宣言を省略可
  - ダックタイピングが可能
  - プログラムの記述が簡潔・柔軟になる

cf. Javaのインタフェース

ダックタイピング (duck typing)

- 「アヒルのように歩き、アヒルのように鳴くのなら、それはアヒルである。」
- 同じメソッド名を持つ別クラスを同じものとして扱う
  - 静的型付け言語だと，型ごとに場合分けや直和型の定義が必要
- 例：

```
def test(foo)
   puts foo.sound
end
class Duck
   def sound
     'quack'
   end
end
```

```
class Cat
   def sound
     'myaa'
   end
end
test(Duck.new)
test(Cat.new)
```

Ruby言語での例

注：同じ名前，同じ引数でも
中身の動作が違うとまずいことにも

明示的な型付けと暗黙的な型付け

- C言語などでは明示的な型宣言が必要
  - 例： int x;
- 型宣言が無くても型推論やデフォルト規則で型付けすること
  - 例：C89以前のCではプロトタイプ宣言が無い関数を，int foo (); という型と見なして処理していた
  - 例：ML言語では型推論で（なるべく）型を決める

```
- fun twice x = x * 2;
val twice = fn: int -> int
```

1行目を入力すると，言語処理系が自動的に型推論して２行目を出力.
関数twiceの引数や返り値の型を宣言していないが整数2のかけ算を手がかりに型推論して，静的に型付け

キャスト（明示的な型変換）

- キャストの例
  - double d = 2.0;
    int x = (int)d;　// 実はこのキャストは不要
- 許されないキャストもある
  - 例：異なる構造体同士のキャスト
- キャストの乱用に注意
  - 不要（冗長）なキャストはコードを読みにくくする
  - キャストはコンパイルエラーを抑止（悪い場合あり）

```
#include <stdio.h>
int main (void)
{
    const char *s1 = "hello\n";
    char *s2 = (char *)s1;
    s2[0] = 'H'; // 文字列リテラルを不正書き替え
    printf ("%s\n", s2);
}
```

コンパイル時ではなく
実行時にエラー

暗黙の型変換 (coercion)

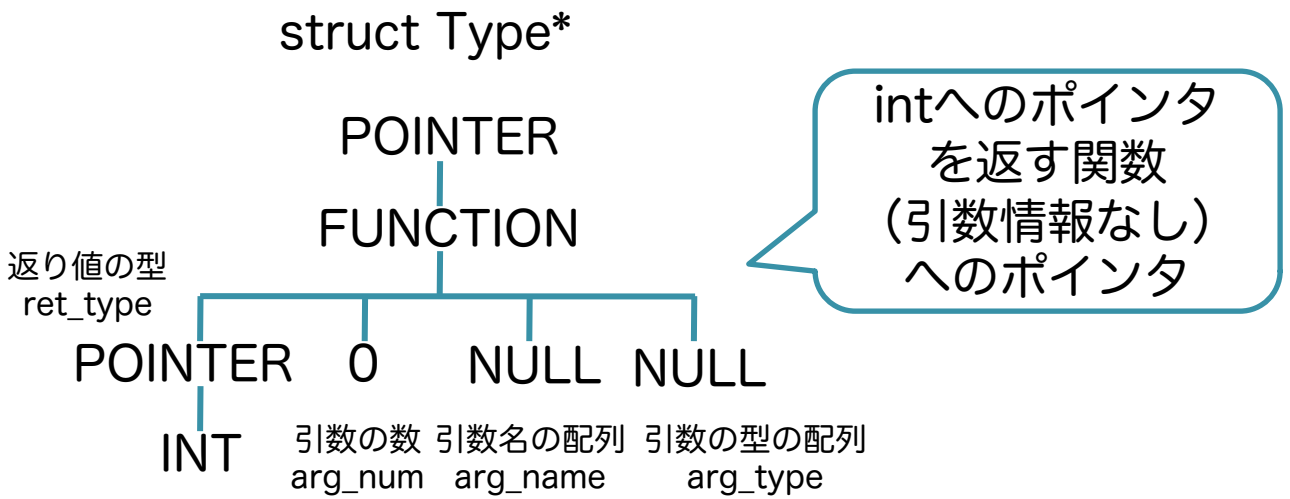
- 異なる型への代入
  - double d = 2.0;
    int x = d;　// 暗黙に double→intに変換
- 異なる型同士の演算
  - オペランドの型を同じにする暗黙の型変換が起こる
    - 例： double と int の乗算では，int型のデータをdouble型に変換後に乗算
  - int x = 10;
    double pi = 3.14;
    int y = pi * x;　// これは int y = (int)(pi*(double)x); と同じ

型チェック

- 変数の型チェック
  - 未定義ではなく，定義が存在するか
  - 型は合致しているか
  - スコープはあっているか
- 関数の型チェック
  - 上記に加え，定義と呼び出し側で，引数の型（個数を含む）と返り値の型は合致しているか

XCCの型表現（概要）

- type.h を良く読む
- 構文解析中に型解析済み
  - 宣言と式のAST構造体の type メンバにセット
- 型を木構造で表現
  - 注：一般的には型再帰のためグラフ．XCCでは木で十分
  - 例：int *(*f)(); の型の木表現



XCCの型のデータ表現（1/2）

```
enum PrimType { // 原始型
    PRIM_TYPE_VOID, // void型
    PRIM_TYPE_CHAR, // char型
    PRIM_TYPE_INT,  // int型
    PRIM_TYPE_LONG  // long型
};

enum TypeKind { // 型の種類
    TYPE_KIND_PRIM,     // 原始型
    TYPE_KIND_FUNCTION, // 関数型
    TYPE_KIND_POINTER   // ポインタ型
};
```

XCCの型のデータ表現（2/2）

```
struct Type {
    enum TypeKind   kind;
    int             size;  // この型の全サイズ
    char            *id;   // 識別子が無ければNULLをセット
    union {
        struct { enum PrimType  ptype; } t_prim;
        struct { struct Type    *type; } t_pointer;
        struct {                // ↑ポインタが指す先の型（へのポインタ）
            struct Type *ret_type;   // 返り値の型（へのポインタ）
            int          arg_num;    // 引数の個数
            char        **arg_name;  // 引数名の配列
            struct Type **arg_type;  // 引数の型（へのポインタ）の配列
        } t_function;
    } u;
};
```

arg_nameの気持ちは *arg_name[] （arg_typeも同様）

XCCの型情報のダンプ（デバッグ用）

- XCCの実装コード中で，type_dump() を呼ぶ
- XCコード中で type_dump_* を宣言する

foo.c

```
int *(*f)();
int type_dump_f;
```

XCCコード中で変数 fの
型情報のダンプを指示.

```
% xcc foo.c
POINTER  : 4 f:
  FUNCTION : -1 f:
  =>return
    POINTER  : 4 f:
      PRIMITIVE: 4 f: int
```

関数型にサイズは
無いので，-1になっている.

XCCの型付け（2/2）

- 宣言を解析して型付け

```
declaration
    : type_specifier declarator ';'
        { $$ = create_AST ("AST_declaration", 2, $1, $2);
          $$->type = type_analyze_declarator ($2, $1->type);
          sym_entry ($$); }
    ;
```

```
type_analyze_declarator (struct AST *ast_decr, struct Type *type)
{
    struct Type *type1, *type2;
    …
    } else if (!strcmp (ast_decr->ast_type, "AST_declarator_pointer")) {
        type1 = create_pointer_type (type); // ポインタ型を作って
        type2 = type_analyze_declarator (ast_decr->child [0], type1);
        type->id = type2->id; // ↑ 残りは再帰的に処理
        return type2;
    } else if …
}
```

例：int *(*f()); に対して
asc_decrに *(*f()) ,
type に int が渡される

ポインタ演算（引き算）

- ポインタ - ポインタ // OK
- ポインタ - 整数　　// OK
- 整数 - ポインタ　　// NG

```
void *malloc ();
int main ()
{
    int i1; int i2;
    int *p1; int *p2; int *p3;
    i1 = 10; i2 = 20;
    p1 = malloc (4 * 10);
    p2 = p1 + 3;  // OK
    p2 = 3  + p1; // OK
    p3 = p2 - 3;  // OK
    i2 = p2 - p2; // OK
    p3 = 3  - p2; // NG
}
```

XCCの型付け（1/2）

type1やtype2が関数型だった場合のエラー処理はサボり

- 式を解析して型付け（例：ポインタ演算の引き算）

```
expression …
    | expression '-'  expression
        { $$ = create_AST ("AST_expression_sub", 2, $1, $3);
          $$->type = type_sub ($1->type, $3->type); }
```

```
struct Type *
type_sub (struct Type *type1, struct Type *type2)
{
    if ((type1->kind == TYPE_KIND_POINTER)
        && (type2->kind == TYPE_KIND_POINTER)) {
        return types.t_int; // ポインタ同士の引き算の型は int型
    }
    if (type1->kind == TYPE_KIND_POINTER)
         return type1; // 第1引数がポインタ，第2引数が整数なら，
                       // 結果の型はtype1（つまりポインタ型）
    if (type2->kind == TYPE_KIND_POINTER) // 第2引数だけがポインタならエラー
        yyerror ("scalar - pointer prohibited\n");
    return types.t_int; // それ以外は結果の型は int型
}
```

XCCの型検査（1/2）

- 未定義変数の検出
  - 例：左辺値のコード生成で，識別子を検索．
    存在しない→エラー

```
static void
codegen_lvalue (struct AST *ast)
{
    struct Symbol *sym;
    if (!strcmp (ast->ast_type, "AST_expression_id")) {
        sym = sym_lookup (ast->child [0]->u.id);
        assert (sym != NULL);
    …
```

注：本来はassertをエラー処理に使うべきでは無い．
ここではエラー処理の手抜きで assertを使用．
assertは「起きないはず」のことを書く．

XCCの型検査（2/2）

- type.cの型の整合性チェックは必要最低限
- 例：「*式」で，式がポインタ型以外ならエラー

```
expression …
    | unary_operator  expression %prec UNARY_OP
        { $$ = create_AST ("AST_expression_unary",  2, $1, $2);
          $$->type = type_uop ($1, $2->type); }
```

```
struct Type *
type_uop (struct AST *ast, struct Type *type1)
{
    …
    } else if (!strcmp (ast->ast_type, "AST_unary_operator_deref"))
{
        if (type1->kind == TYPE_KIND_POINTER) {
            return type1->u.t_pointer.type;
        } else { // ポインタ型以外に * を付けたらエラー
            yyerror ("no pointer type dereferenced\n");
        }
    …
}
```

C言語の暗黙の型変換（一部）

- 整数拡張 integer promotion
  - 例：式中では char型やshort型は int型に変換
- 通常の算術型変換 usual arithmetic conversion
  - 2項演算子のオペランドの型が異なる時，演算を行う前にオペランドの型を変換して同じ型にする
  - 例：unsigned int と int 型の演算は unsigned int に揃える

```
unsigned int x = 0;
int y = -1;
printf ("%d\n", x < y); // 1
```

unsigned と signed は混ぜちゃダメ

- 既定の実引数拡張 default argument promotion
  - プロトタイプ宣言が無い関数の引数や，可変長引数では，例えば char は int に変換される