教育用コンパイラXCC (x86版)

権藤克彦（東工大）

2009/3/25版

XCCは教育用コンパイラ

- 高性能よりも，高い学習効果が狙い.
- コンパクトで理解しやすい構文規則.
- 本質的な言語機能を保持.
  - 型：int, char, void, 関数型，ポインタ型.
  - 制御構造：if-else, while, goto, return
  - 関数呼出し：引数，ライブラリ使用可.
- C言語の真のサブセット.
  - XCCコードはCコンパイラで処理可能.

XCCの使用例 (x86, 0.3.2版)

- xcc-x86.exe で，*.sファイルを出力.

```
%  xcc-x86.exe foo.c > foo.s
%
```

- gcc で，*.s から a.exe (a.out)を出力・実行.

```
%  gcc foo.s
% ./a.exe
%
```

XCCのコード例：ポインタ(I)

```
int printf ();
int main ()
{
    int i;
    int *p1;
    int **p2;
    int ***p3;
    i = 999;
    p1 = &i;
    p2 = &p1;
    p3 = &p2;
    printf ("%d\n", ***p3);
}
```

ポインタのポインタのポインタも使えます.

XCCのコード例：ポインタ(2)

```
int printf ();
int add (int a, int b)
{
    return a + b;
}
int main (int argc, char **argv)
{
    int (*fp)(int a, int b);
    fp = add;
    printf ("%d\n", fp (10, 20));
}
```

関数へのポインタも使えます.

XCCのコード例：バブルソート

```
void bubble_sort(int *data, int size)
{
    int i; int j;
    i = size - 1;
    while (0 < i) {
        j = 0;
        while (j < i) {
            if (*(data + (j+1)) < *(data + j))
                swap (data + j, data + (j + 1));
            j = j + 1;
        }
        i = i - 1;
    }
}
```

- data[j] ではなく
  *(data + j) と書きます．
  （C言語の式中で，data[j] と
  *(data+j) は同じ意味です．）

- 配列確保は int data[10];
  ではなく，
  data = malloc (4 * 10);
  などとします．

「ポインタ＋整数」はポインタ演算です．

ポインタ演算(pointer arithmetic)

- int *p; int i; のとき，p + i の意味は，
  - pが指している先のオブジェクトi個分，
    p中のアドレスを増やした値.
  - つまり「p中のアドレス + i * sizeof (*p) 」.
  - C言語でも同じ.

```
int printf ();
void *malloc ();
int main (void)
{
    int *p;
    p = malloc (100);
    printf ("%p, %p\n", p, p + 3);
}
```

```
%  ./a.exe
0x6b0240, 0x6b024c
%
```

12増えてるのがポイント.
sizeof(int)=4 だったので，
0x6b0240 + (3 * 4) を計算.

XCCで書けない事（１）

- 「char, int, void, ポインタ型，関数型」以外の型.
  - ○ int x; int *p; int (*fp)();
  - × long, unsigned, const, static, typedef など.
  - × 配列，構造体，共用体，列挙型(enum).

- １つの変数宣言で複数の変数宣言.
- 変数宣言時の初期化.
  - ○ int x; int y; int z; z = 10;
  - × int x, y; int z = 10;

XCCで書けない事（２）

- 一部の制御文
  - ○ if, if-else, while, goto, return
  - × for, do-while, switch, break, continue

- 一部の演算子
  - ○ x < 0 || x == 0
  - × x <= 0
  - × x > 0　（0 < x で代替）

printf の呼出し

- 「int printf (); 」を使用前に宣言しておく.

- int printf (const char *, ...); とは宣言できない.
  - XCCが可変長引数やconstを未サポートのため.
- gccの警告は無視してよい.
  - warning: conflicting types for built-in function 'printf'

```
%  gcc test/t4.c
warning: conflicting types for built-in function 'printf'
%
```

  - 引数を持つプロトタイプ宣言をさぼると「既定の実引数拡張」が起こる.
    - 例：char は int に暗黙に変換.
  - 可変長引数でも「既定の実引数拡張」が起こるので問題なし.

既定の実引数拡張 default argument promotion

構文の用語

用語	意味
translation unit	翻訳単位（ファイルのこと）
declaration	宣言（宣言全体．例：int *(*fp)();）
declarator	宣言子（int *(*fp)(); のうち，*(*fp)()の部分）
compound statement	ブロック構文（ブレース {} で囲まれる部分）
parameter	仮引数 （formal parameter とも言う）
argument	実引数 （actual argument とも言う）
statement	文
expression	式
identifier	識別子（名前のこと）
binary operator	二項演算子（例：4-3の「-」）
unary operator	単項演算子（例：-3 の「-」）

XCCの構文(1/3)

GNU Bison (xcc.y)と
GNU Flex (xcc.l)で実装.

```
translation_unit: external_declaration
    | translation_unit external_declaration ;
external_declaration: function_definition
    | declaration ;
function_definition: type_specifier declarator
compound_statement ;
declaration_list: /* empty */
    | declaration_list declaration ;
declaration: type_specifier declarator ';' ;
type_specifier: "void" | "char" | "int" ;
declarator: identifier
    | '*' declarator
    | '(' declarator ')'
    | declarator '(' parameter_list ')'
    | declarator '(' ')' ;
parameter_list : parameter_declaration
    | parameter_list ',' parameter_declaration ;
parameter_declaration: type_specifier declarator ;
```

XCCの構文(2/3)

```
statement_list: statement
    | statement_list statement ;
statement: expression_opt ';'
    | compound_statement
    | "if" '(' expression ')' statement
    | "if" '(' expression ')' statement "else" statement
    | "while" '(' expression ')' statement
    | "goto" identifier ';'
    | identifier ':' statement
    | "return" expression_opt ';' ;
compound_statement: '{' declaration_list
statement_list '}' ;
expression_opt: /* empty */
    | expression ;
expression: identifier
    | INTEGER_CONSTANT
    | CHARACTER_CONSTANT
    | STRING
```

XCCの構文(3/3)

```
    | expression '='  expression
    | expression "||" expression
    | expression "&&" expression
    | expression "==" expression
    | expression '<'  expression
    | expression '+'  expression
    | expression '-'  expression
    | expression '*'  expression
    | expression '/'  expression
    | unary_operator  expression
    | expression '(' argument_expression_list ')'
    | expression '(' ')'
    | '(' expression ')' ;
unary_operator : '&' | '*' | '+' | '-' | '!' ;
argument_expression_list: expression
    | argument_expression_list ',' expression
    ;
identifier : IDENTIFIER ;
```

XCCの抽象構文木(AST) (1/4)

- 構文解析後，変数 ast_root が抽象構文木の根へのポインタを保持.

xcc.y

```
struct AST *ast_root;
```

XCCの抽象構文木(AST) (2/4)

- AST構造体のメンバー（共通部分）

ASTノードの種類.
例："AST_statement_while"
(xcc.y の create_ASTの第一引数を参照)

AST.h

```
char        *ast_type;
struct AST  *parent;
int         nth;
int         num_child;
struct AST  **child;
```

親ノードへのポインタ
親から見て自分が何番目の子か
子ノードの数
子ノードへのポインタの配列

XCCの抽象構文木(AST) (3/4)

- 構文木の各ノードへの訪問例

```
void visit_AST (struct AST *ast)
{
    int i;
    printf ("%s\n", ast->ast_type);
    for (i = 0; i < ast->num_child; i++) {
        visit_AST (ast->child [i]);
    }
}
```

XCCの抽象構文木(AST) (4/4)

特定のノード用メンバー

AST.h

```
struct Type *type;
union {
  char   *id;
  int    int_val;
  struct {
    int    total_arg_size;
    int    total_local_size;
    struct Symbol *global;
    struct Symbol *arg;
    struct Symbol *label;
    struct String *string;
  } func;
  struct Symbol *local;
  int    arg_size;
} u;
```

型情報．宣言と式の全てに．

識別子．
AST_IDENTIFIER, AST_expression_string

整数定数の値．
AST_expression_int

引数の合計サイズ，
局所変数の合計サイズ，
関数・大域変数情報のリスト，
引数情報のリスト，
ラベル情報のリスト，
文字列定数のリスト．
AST_function_definition

局所変数情報のリスト
AST_compound_statement

引数のサイズ．
AST_argument_expression_list_{single, pair}

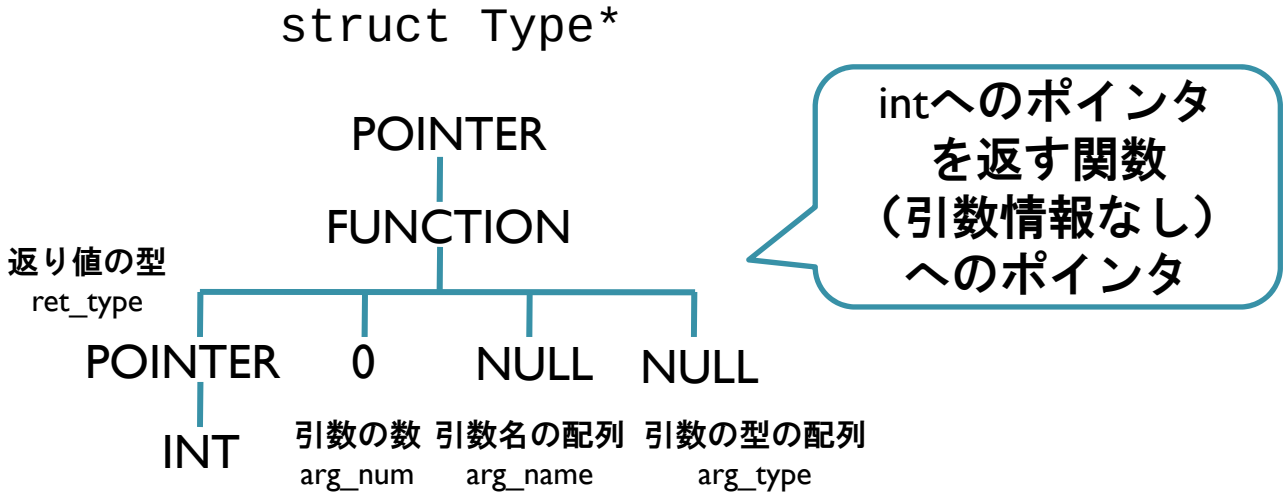
デバッガ gdb を使う

- **複雑な構造体はデバッガで中身を確認.**
  - gcc の -g オプションでコンパイルしておくこと.

```
% gdb xcc-x86.exe
(gdb) break codegen     ブレークポイントの設定
(gdb) run test/t1.c     実行開始
Breakpoint 1, codegen () at codegen-x86.c:677   ブレークポイント
677    ast = search_AST_bottom (ast_root,        で一時停止した
678           "AST_translation_unit_single", NULL);
(gdb) print *ast_root   データの表示
$1 = {ast_type = 0x40aa01 "AST_translation_unit_pair",
      parent = 0x0, nth = 0, num_child = 2, child = 0x6d7f70,
      type = 0x0, u = {id = 0x0, int_val = 0,
      func = {total_arg_size = 0, total_local_size = 0,
               arg = 0x0, label = 0x0,string = 0x0},
      local = 0x0, arg_size = 0}}
(gdb) print *ast_root->child [0]
$2 = {ast_type = 0x40a9e5 "AST_translation_unit_single",
      parent = 0x6d7f40, ...（省略）...}
(gdb) quit    gdbの終了
%
```

XCCの型表現：概要

- type.h を熟読して下さい.
- 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）グラフになる．XCCでは木で十分.
  - 例：int *(*f)(); の型の木表現.



XCCの型表現：Cでの型定義（１）

type.h

```
enum PrimType {
    PRIM_TYPE_VOID,
    PRIM_TYPE_CHAR,
    PRIM_TYPE_INT
};
```

原始型（基本型） primitive type

- void型
- char型
- int型

type.h

```
enum TypeKind {
    TYPE_KIND_PRIM,
    TYPE_KIND_FUNCTION,
    TYPE_KIND_POINTER
};
```

型の種類

- 原始型
- 関数型
- ポインタ型

XCCの型表現：Cでの型定義（2）

type.h

```
struct Type {
  enum TypeKind   kind;
  int             size;
  char            *id;
  union {
    struct { enum PrimType  ptype; } t_prim;
    struct { struct Type    *type; } t_pointer;
    struct {
      struct Type *ret_type;
      int          arg_num;
      char         **arg_name;
      struct Type **arg_type;
    } t_function;
  } u;
};
```

ポインタが指す先の型

- 型の種類
- 型のサイズ
- 関連する識別子
（引数名などで使用）
- 原始型
- ポインタ型
- 関数型
  - 返り値の型
  - 引数の数
  - 引数の名前の配列
  - 引数の型の配列

XCCの型表現：デバッグ

- type_dump () を使う.

```
type_dump (type);
```

XCCコンパイラ中で
型情報をダンプ.

- 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-x86.exe 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の記号表：概要

- symbol.h を熟読して下さい.
- 構文解析中に記号表を構築済み.
  - AST構造体の以下のメンバにセット.
    - global, arg, label, string, local
- コード生成時に次の関数で記号を検索.
  - sym_lookup, sym_lookup_label, string_lookup
- 大域変数 sym_table が記号表本体.
- コード生成時に次の関数で記号表を要修正.
  - codegen_begin_function, codegen_end_function
  - codegen_begin_block, codegen_end_block

XCCの記号表：記号 struct Symbol

```
enum NameSpace {
  NS_GLOBAL,
  NS_LOCAL,
  NS_ARG,
  NS_LABEL
};
```

名前空間
- 大域（関数の外）
- 局所（関数の中）
- 関数引数
- ラベル

```
struct Symbol {
  char              *name;
  struct Type       *type;
  struct AST        *ast;
  int               offset;
  enum NameSpace    name_space;
  struct Symbol     *next;
};
```

記号の情報
- 記号名
- 記号の型
- 記号が宣言されたASTノード
- オフセット（局所変数と引数のみ）
- 名前空間
- 次の記号へのポインタ

XCCの記号表：記号表 struct SymTable

```
struct SymTable {
  struct Symbol  *global;
  struct Symbol  *arg;
  struct Symbol  *label;
  struct String  *string;
  int            local_index;
  struct Symbol  *local [MAX_BLOCK_DEPTH];
};

extern struct SymTable sym_table;
```

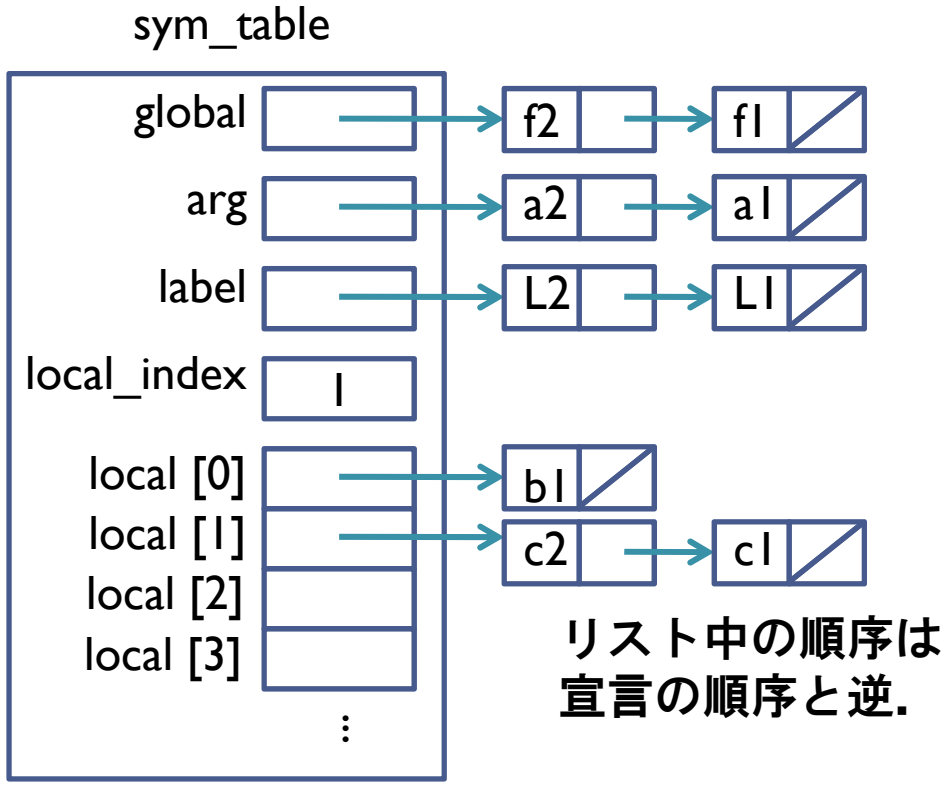
記号表
- 大域の記号リスト
- 引数の記号リスト
- ラベルの記号リスト
- 文字列定数のリスト
- 局所記号のネスト数
- 局所の記号リストの配列

記号表を保持する大域変数

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

XCCの記号表：デバッグ

- sym_table_dump () を使う.

```
sym_table_dump ();
```

XCCコンパイラ中で
記号表の内容をダンプ.

- sym_table_dump を宣言する.

foo.c

```
int sym_table_dump;
```

XCCコード中で記号表の
内容をダンプ.

```
% xcc-x86.exe foo.c
global:    f2, f1,
arg:       a2, a1,
label:
string:
local[0]: b1,
local[1]: c2, c1,
```

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：記号表の修正

- 関数やブロックの出入り毎に記号表(sym_table)を要修正.
- 以下の関数を使って下さい.

codegen-x86.c

```
static void codegen_begin_block    (struct AST *ast);
static void codegen_end_block      (void);
static void codegen_begin_function (struct AST *ast);
static void codegen_end_function   (void);
```

それぞれ以下の時に呼び出します.
・ブロックに入る時
・ブロックを出る時
・関数に入る時
・関数を出る時

XCCの記号表：記号表の検索

- 以下の関数を使って下さい.

string.h

```
struct Symbol *sym_lookup       (char *name);
struct Symbol *sym_lookup_label (char *name);
struct String *string_lookup    (char *data);
```

それぞれ，以下を検索します.
・（ラベル以外の）記号の検索.
・ラベル記号の検索.
・文字列定数の検索.

codegen () と emit_code ()

- codegen-x86.c 中の codegen 関数にコード生成部を実装してください.
  - 当然，分かりやすくモジュール分割して下さい.
    - 例：codegen_add (), codegen_while (), ...

- アセンブリコード生成にはemit_code 関数を使ってください.
  - ○ emit_code (ast, "\t jmp %s\n", label);
    - astは関連するASTノードへのポインタを指定.
    - emit_codeはfprintfのラッパ関数． 可視化のため.
  - × fprintf (xcc_out, "\t jmp %s\n", label);

XCC_VIS って何？

- 無視してください.
  - コンパイラ可視化のためのコードです.

xcc.y

```
int
main (int argc, char *argv [])
{
    init (argc, argv);
    yyparse ();
#ifdef XCC_VIS
    xcc_vis_main (ast_root, 0);
#endif
    codegen ();
    return 0;
}
```

この部分は
無視してよい.