情報実験第四「コンパイラ」課題説明資料

萩原茂樹 (hagihara@fmx.cs.titech.ac.jp)
中野瑞樹 (nakano@psg.cs.titech.ac.jp)
高野健太 (takano.k.ai@m.titech.ac.jp)

2016.10.4

1 実験課題の目的

コンパイラの作成を通して、その原理と動作の理解を深めることを目的とする。具体的には、C 言語のコンパイラのコード生成部分を実装し、その原理と動作の理解を深める。

2 注意

- 毎回 9:45 に出席をとる。遅刻は欠席扱いとする。
- 実験の時間終了前の 12:00 より、掃除をする。掃除当番の表は
  `http://www1.csc.titech.ac.jp/class/joho4/compile/`
  にある。各自、自分の当番の日を確認し、その日に掃除をすること。

3 課題を行うに当たっての準備

本課題では、コンパイラ作成の実習に xcc を用いる。xcc とは、権藤克彦先生が作成した教育用 C コンパイラであり、C 言語のサブセットを処理できる。xcc は、以下の機能を持つ C プログラムを構文解析、意味解析できる [1]。

- 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.

本課題で配布する xcc は、これらの構文解析、意味解析に加えて、`1` や `21` 等の即値と、`"Hello world!"`等の文字列定数、`printf` 等の関数呼び出しのみのコードが生成できるバージョンである。それ以外の機能のコードが生成できるように、コンパイラのコード生成部分を拡張することが課題である。xcc はもともと、様々なプラットフォームで利用できるコンパイラであるが、本課題で用いるプラットフォームは、OS は Mac OS X、バイナリ形式は Mach-O、CPU は i386 である。

[1] xcc では関数へのポインタを用いるプログラムも構文解析、意味解析できるが、今回、課題で配布する xcc はその構文解析ができないバージョンである。

3.1 まず最初に実行してみよう。

http://www1.csc.titech.ac.jp/class/joho4/compile/xcc-kadai-2016.tar.gz に xcc のソースがあるので、これを自分の編集できるところで展開する。

```
$ tar zxf xcc-kadai-2016.tar.gz
$ ls
xcc/
$ cd xcc
$ ls
AST.c codegen.c misc.h test/ xcc.h
AST.h codegen.h symbol.c type.c xcc.l
Makefile misc.c symbol.h type.h xcc.y
$ ls test
kadai0.c kadai1.c kadai3.c tmisc-char2.c
kadai1-1.c kadai2.c tmisc-char.c
$
```

xcc ディレクトリの中に xcc のソースがある。xcc/test ディレクトリには、xcc が処理する例プログラムがある。xcc のソースのコンパイルは、次のように行う。

```
$ make
bison -d -v xcc.y
flex xcc.l
gcc -Wall -Wstrict-prototypes ...
$
```

出来上がった C コンパイラ xcc で C プログラムを次のようにコンパイルできる。（test/kadai0.c のコンパイル）この結果、アセンブリコード kadai0.s が生成される。

```
$ ./xcc test/kadai0.c > kadai0.s
$
```

これを、gcc を用いて、次のように実行形式 a.out に変換する [2]。

```
$ gcc -m32 kadai0.s
$
```

a.out は以下のように実行できる。

```
$ ./a.out
hello, world, 10, 20
$
```

3.2 xcc について

xcc は、flex と bison により字句解析と構文解析を行い、それと同時に、記号や式の型の解析、引数や変数の領域の大きさの解析等、意味解析も行い、抽象構文木 (AST) と記号表を生成する。その後、それらを用いて、コード生成を行う。（このコード生成部分を実装してもらうことが課題である。）以下では、コード生成に必要な抽象構文木と記号表について述べる。これらの詳細は各ソースファイルを熟読すること。

[2] この際、ld: warning: PIE disabled. ... という警告が出力される。無視して下さい。

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 *func(int *i);)
declarator	宣言子 (int *func(int *i); のうち,*func(int *i); 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
argument	実引数
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「−」)

表 1: 構文の用語

3.2.1 xcc の抽象構文木

xcc が対象とする C 言語のサブセットの構文規則は、xcc.y で定義されているので、xcc.y を参照のこと。ここで用いられている主な非終端記号の直感的な意味は、表 1 の通りである。構文解析後、抽象構文木が構成され、その根へのポインタを変数 ast_root が保持している。抽象構文木の各ノードの型は、構造体 AST で定義されている。詳細は AST.h を参照すること。どのように抽象構文木が生成され、意味解析がなされるかは、xcc.y の中で、各構文規則と共に { と } の中で記述されている。例えば、statement(文) の構文規則のうち、図 1 上に示す while 文の構文規則では、statement のうち while 文は、"while" '(' expression ')' statement から成っていることを表している。{ と } の間では、「expression で生成された抽象構文木の根ノード ($3 のこと、expression が

```
statement
    : .....
    | "while" '(' expression ')' statement
        { $$ = create_AST ("AST_statement_while", 2, $3, $5); }
```

```
function_definition
    : type_specifier declarator
        { $$ = create_AST ("AST_dummy", 2, $1, $2);
          $$->type = type_analyze_declarator ($2, $1->type);
          /* conditonal to cause more natural syntax error */
          if ($$->type->kind == TYPE_KIND_FUNCTION) {
              sym_entry_param ($$->type);
              sym_entry ($$);
              sym_begin_function ();
          } }
      compound_statement
        { $$ = create_AST ("AST_function_definition", 3, $1, $2, $4);
          sym_backpatch ($$, $3->type);
          sym_end_function ($$); }
    ;
```

図 1: while 文と function_definition の構文規則

3 番目に出現するから。) と statement で生成された抽象構文木の根ノード ($5 のこと、statement が 5 番目に出

現するから。) を引数としてもらい、それら二つのノードを子としてもつ、新たなノードを生成し (create_AST)、そのノードの種類を"AST_statement_while"とし、このノードを親 statement で生成した抽象構文木の根ノード ($$のこと) とする」ことを表している。ここで、関数 create_AST は AST.c で定義されているので、そちらを参照すること。

もう少し、複雑な構文規則は図 1 下に示す function_definition の構文規則である。この構文規則は、「function_definition は、type_specifier declarator compound_statement から成っていること」を表している。最初のふたつ type_specifier declarator で、最初の {,} の間で記述された意味解析を実行し (このとき、$1,$2 はそれぞれ、type_specifier と declarator の意味解析で生成された抽象構文木の根ノード)、つぎに、compound_statement も含めて、二つ目の {,} の間で記述された意味解析を実行している。(このとき、$1,$2 は前述のとおり。$4 は、compound_statement の意味解析で生成された抽象構文木の根ノードであり、$3 は、最初の {,} の間の意味解析で得られる$$)

実際に構成された抽象構文木の詳細な構造は、デバッガ gdb を使って確認できる [3] (図 2)。また、xcc のソース中で、関数 dump_AST を使っても、抽象構文木の構造の概略を調べることができる。

```
$ gdb ./xcc
(gdb) break codegen
Breakpoint 1 at 0x10000278d: file codegen.c, line 86.
(gdb) run test/kadai0.c
Breakpoint 1, codegen () at codegen.c:86
86            emit_code (ast_root, "\t.file   \"%s\"\n", filename);
(gdb) print *ast_root
$1 = {ast_type = 0x1000084c7 "AST_translation_unit_pair",
      parent = 0x0, nth = 0, num_child = 2, child = 0x100100dd0,
      type = 0x0, u = {id = 0x0, int_val = 0,
      func = {total_arg_size = 0, total_local_size = 0, global = 0x0,
              arg = 0x0, label = 0x0, string = 0x0},
      local = 0x0,
      arg_size = 0}}
(gdb) print *ast_root->child[0]
$2 = {ast_type = 0x1000084ab "AST_translation_unit_single",
      parent = 0x100100d80, nth = 0, num_child = 1, child = 0x1001004c0,
      ....
      }}
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$
```

図 2: gdb による抽象構文木の解析

3.2.2 xcc の型表現

構文解析中に、式の型の解析が行われ、型の整合性が調べられている。構文解析が終了し、抽象構文木が生成されたときには、型の解析も終了し、宣言と式に対する構文解析木ノードの type メンバにその型がセットされている。型は木構造で表されている。例えば、int *f(int n, int *m) の f の型は図 3 のように表されている。型のデータ構造の詳細は、type.h を参照すること。

xcc では、デバッグ用に型情報をダンプする二つの方法が用意されている。

- xcc のソース中で、関数 type_dump() を使う
- xcc がコンパイルする対象の C プログラム中で、変数として、type_dump_*を宣言する。(図 4)

[3] gdb の最初の実行時にパスワードが聞かれます。答えて下さい。

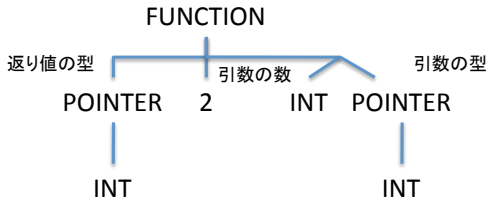


図 3: int *f(int n, int *m) の f の型表現

```
$ cat test-type.c
int *f(int n, int *m);
int type_dump_f;   /* f の型構造を出力 */
...(省略)..
$ ./xcc test-type.c
FUNCTION : -1 f:
=>return
  POINTER  : 4 f:
    PRIMITIVE: 4 f: int
=>arg [0]: n
  PRIMITIVE: 4 n: int
=>arg [1]: m
  POINTER  : 4 m:
    PRIMITIVE: 4 m: int
...
$
```

図 4: 型情報のダンプ

もちろん、より詳細に調べるときには、gdb を使えばよい。

3.2.3 xcc の記号表

構文解析中に記号表を構築済みであり、記号表本体は、大域変数 sym_table である。記号表には、コンパイル対象の C プログラムで定義された大域変数記号や関数記号のリスト、引数の記号のリスト、ラベルの記号のリスト、文字列定数の記号のリスト、（各ネストの深さに対する）局所変数記号のリストがそれぞれ、global, arg, label, string, local メンバにセットされている。記号と記号表のデータ構造は symbol.h で定義されているので熟読すること。また、構造体 AST の、global, arg, label, string, local のメンバにもセットされている。

xcc では、デバッグ用に記号表をダンプする二つの方法が用意されている。

- xcc のソース中で、sym_table_dump() を使う。
- xcc がコンパイルする対象の C プログラム中で、変数として、sym_table_dump を宣言する。(図 5)

もちろん、より詳細に調べるときには、gdb を使えばよい。

3.3 コード生成部分を作成するための注意

- 本実験課題は、構築済みの抽象構文木や記号表を使って、コード生成部分を実装することである。codegen.c 中の関数 codegen() にコード生成部を実装すること。codegen() は、構文解析、意味解析後に xcc.y 中で呼ばれる。

```
$ cat test-symtable.c
int f1;
int f2 (int a1, int a2)
{
  int b1;
  {
    int c1; int c2;
    int sym_table_dump; /* この場所での記号表を出力 */
    L1: goto L2;
  }
  {
    int d1; int d2;
    L2: goto L1;
  }
}

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

$ ./xcc test-symtable.c
global:   f2, f1,
arg:      a2, a1,
label:
string:
local[0]: b1,
local[1]: c2, c1,
...
$
```

図 5: 記号表のダンプ

- コード生成は、抽象構文木をたどりながら行う。各記号にはスコープがあるため、抽象構文木のそれぞれのノードにおいて、有効な記号が異なる。従って、抽象構文木をたどる際、
  - ブロックに入るとき、codegen_begin_block
  - ブロックから出るとき、codegen_end_block
  - 関数定義に入るとき、codegen_begin_function
  - 関数定義から出るとき、codegen_end_function

  を用いて、記号表の修正する必要がある。これらは、配布したソースコードで既になされている。
- アセンブリコード生成には、emit_code 関数を使うこと。
- コード生成時に、記号表を検索するためは、次の関数を用いること。
  - sym_lookup ラベル以外の記号の検索
  - sym_lookup_label ラベル記号の検索
  - string_lookup 文字列定数の検索
- スタックの高さを変えるアセンブリコードを生成するときは、大域変数 frame_height を更新すること。

  Mac OS X のバイナリ形式は Mach-O は、関数呼び出し時に、スタックポインタ%esp が 16 バイト境界を指すことを要求しており、関数呼び出し時に 16 バイト境界を指すように padding を入れる必要がある。このときに、どの大きさの padding を入れればよいか、計算するために、スタックフレームの高さを表す大域

変数 frame_height を使っている。この padding を計算する部分は、関数 codegen_expression_funcall 内で実装済みである。この計算を正しく行うために、スタックの高さを変えるアセンブリコードを生成するときは、大域変数 frame_height を更新すること。

- gcc が出力するアセンブリコードも大変参考になる。アセンブリコードへは以下のように変換できる。（hogehoge.c を hogehoge.s へ変換。）

```
$ gcc -m32 -static -S hogehoge.c
$
```

4 コード生成のポイント

以下に、どのようなコードを生成すればよいか、幾つかポイントを述べる。

4.1 関数定義のコード生成

関数の定義部分のコードは図 6 の通り。

文字列定数のコード	
.text	(以降を.text セクションに)
.globl _funcname	(関数名を外部から利用可)
_funcname:	(関数名のラベル)
pushl %ebp	(新しいスタックフレームの作成)
movl %esp, %ebp	
subl $局所変数のサイズ, %esp	(局所変数の領域確保)
ブロック文のコード	
label:	(return 文のジャンプ先)
movl %ebp, %esp	(スタックフレームの破棄)
popl %ebp	
ret	(制御を戻り番地へ)

図 6: 関数定義のコード

4.2 文字列定数のためのコード

文字列定数の領域確保するコードは、図 7 のとおりである。文字列定数にアクセスするときは、そのメモリアドレスを表すラベルをスタックに積めばよい。配布したバージョンでは実現済み。

4.3 式に対するコード生成のポイント

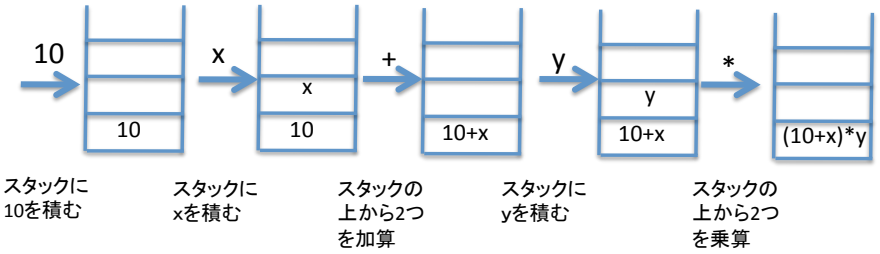
後順序に抽象構文木を訪問する。式の途中結果はスタックに積む。

```
   .cstring
L1:                    ("abcdef\0"が格納されているメモリアドレスを表すラベル)
   .ascii "abcdef\0"   (最後に \0 を忘れない。)
L2:
   .ascii "1234566\0"
   ⋮
```

図 7: 文字列定数のコード

```
/* 後順序で訪問する */
void
postorder (struct AST *ast)
{
  int i;
  for (i = 0; i < ast->num_child; i++) {
    postorder (ast->child [i]);
  };
  (このノードの対する処理)
}
```

式 (10+x)*y を例に、式のコード生成を説明する。この式を抽象構文木を後順序で訪問すると、(10, x, +, y, *) という順序でノードが訪問される。このとき、次の動作をするコードを出力すればよい。



つまり、定数や変数は プッシュ命令でその値をスタックに積むコード、演算子は、演算子の引数をポップして演算して結果をプッシュするコードに変換する。具体的には、次のようなコードに変換すればよい。

10 に対するコード生成	pushl $10	(定数 10 をスタックに積む)
x に対するコード生成	pushl _x	(変数 x の値をスタックに積む)
+ に対するコード生成	popl %ecx	(スタックからポップして x の値をレジスタ%ecx に格納)
	popl %eax	(スタックからポップして値 10 をレジスタ%eax に格納)
	addl %ecx, %eax	(%ecx と %eax の和を%eax に格納)
	pushl %eax	(%eax の値 (=10+x) をスタックに積む)
y に対するコード生成	pushl _y	(変数 y の値をスタックに積む)
* に対するコード生成	popl %ecx	(スタックからポップして y の値をレジスタ%ecx に格納)
	popl %eax	(スタックからポップして 10+x の値をレジスタ%eax に格納)
	imull %ecx,%eax	(%ecx と%eax の積 (=(10+x)*y) を%eax に格納)
	pushl %eax	(%eax の値 (=(10+x)*y) をスタックに積む)

これら以外にも、例えば、== 演算子に対しては、次のようなコードを生成すればよい。

popl %ecx	(スタックからポップして第 1 引数の値をレジスタ%ecx に格納)
popl %eax	(スタックからポップして第 2 引数の値をレジスタ%eax に格納)
cmpl %ecx,%eax	(%ecx と%eax を比較 (%eflags が変化))
sete %al	(等しいかの結果 (0 か 1) を%al に代入)
movzbl %al,%eax	(%al の値をゼロ拡張して %eax に)
pushl %eax	(この式全体の結果 (%eax) をスタックに積む)

&& と || の式は、まず、左オペランドを先に計算し、式全体の値が確定せず、右オペランドの計算が必要な場合のみ、右オペランドを計算する。生成するコードもこのようなコードである必要がある。注意すること。

4.4 変数の取り扱いのポイント

- 大域変数については、コンパイル時に.bss セクションにその変数領域を確保し、実行時にそのラベル名でアクセスする。
- 局所変数については、実行時にスタック上にその変数領域を確保し、ベースポインタのオフセットでアクセスする。
- 引数については、実行時に実引数をスタックに積むことでその領域を確保し、ベースポインタのオフセットでアクセスする。

4.4.1 変数領域の確保

大域変数のメモリ領域は、コンパイル時に.bss セクションに確保する。このためには、アセンブラ命令.comm を使えばよい。ここで、アラインメントは 2 のべき乗である。

.comm シンボル, サイズ, アラインメント

例えば、大域変数定義 `int x;` に対して、

```
.comm _x,4,2
```

により、大域変数領域を.bss セクションに 4 バイト確保すればよい。

一方、局所変数の領域は、実行時にスタック上に確保する。新しいスタックフレームが作られた後に、局所変数の領域を確保すればよい。スタック上に領域を確保するには、スタックポインタ%esp の値を、その領域の大きさの分だけ減らせばよい。例えば、C プログラムの局所変数定義 `int i; int k;` に対して、

```
subl $8, %esp
```

により、スタック上に 8 バイト (4 バイト ×2 つ) の局所変数領域を確保すればよい。

関数の引数は、関数が呼び出される際、スタックに逆順に実引数を積むことにより、領域を確保する。

このように、大域変数、局所変数、引数の領域を、図 8 のように確保すること。

4.4.2 変数の利用

変数の「値」は、左辺値と右辺値の二種類ある。左辺値はその変数の値が格納されているメモリアドレスであり、右辺値はその値自身である。

例えば、int 型の変数 x の左辺値と右辺値をスタックに積むコードは、図 2 のとおりである。ここで、_x は、大域変数 x の格納場所のラベルであり、offset はベースポインタからの相対オフセットである。

変数へのアクセスの目的に応じて、どちらか適当な値を計算し、その値がスタックに積まれる。例えば、代入式のコードは次のとおりである。

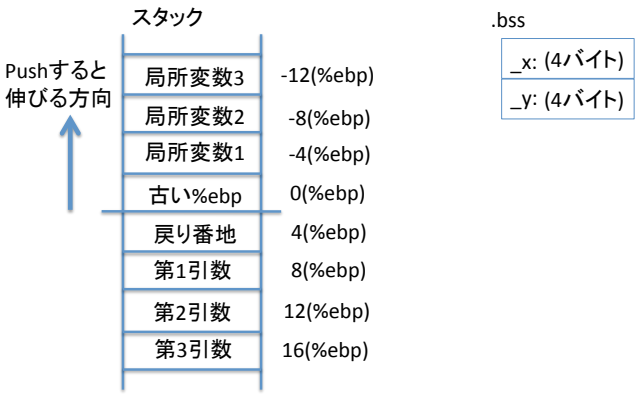


図 8: 大域変数、局所変数、関数の引数の領域

	左辺値	右辺値
大域変数	pushl $_x	pushl _x
引数	leal offset(%ebp), %eax pushl %eax	pushl offset(%ebp)
局所変数	leal offset(%ebp), %eax pushl %eax	pushl offset(%ebp)

表 2: 大域変数、局所変数、引数の左辺値と右辺値

4.4.3 代入式のコード生成

代入式は、式 1 = 式 2 という形をした式であり、式 2 の値が式 1 に代入される。これを実現するコードは、図 9 のとおりである。代入式自体の値は、代入した値そのものであるため、代入する値をスタックトップに残して

```
式 2 の右辺値をスタックに積むコード     (代入する値をスタックへ)
式 1 の左辺値をスタックに積むコード     (代入先のアドレスをスタックへ)
popl %eax                               (アドレスを%eax に代入)
movl 0(%esp),%ecx                       (スタックトップの値 (=代入する値) を%ecx に代入)
movl %ecx,0(%eax)                       (%ecx の値を%eax が指すメモリ中に代入)
```

図 9: 代入式 (式 1=式 2) のコード

いることに注意する。これは、x = y = z = 999 という式を評価するために必要である。

4.5 制御文に対するコード生成のポイント

制御文に対しては、(条件) ジャンプ命令を使い、実行する文を制御するようなコードを生成する。

4.5.1 式文

式文とは、式にセミコロンがついた文であり、式を計算した後、その式の値をスタックからを捨てる文である。式文 式 1; のコードは図 10 のとおりである。

```
式 1 のコード
addl $4,%esp        (スタック上の式 1 の値を捨てる)
```

図 10: 式文 (式;) のコード

4.5.2 goto 文とラベル

C 言語では、goto 文はラベルと共に用いられる。C 言語のラベル (ラベル 1:) と、goto 文 (goto ラベル 2) のコードは、図 11 のとおりである。

```
label:      (label はラベル 1 に対応するラベルで、
             他のラベルと重複しない)
```

```
jmp label   (label はラベル 2 に対応するラベル)
```

図 11: ラベル (ラベル 1:) と、goto 文 (goto ラベル 2) のコード

4.5.3 return 文

return 文 (return 式;) のコードは、図 12 のとおりである。返り値を伴わない場合 (return;) もあるので、注意。

```
式のコード
popl %eax           (関数の返り値を%eax に格納)
movl %ebp,%esp      (スタックフレームを廃棄)
popl %ebp
ret                 (ret 命令で戻り番地に制御を戻す)
```

または

```
式のコード
popl %eax
jmp label           (関数定義の最後へ jmp)
⋮
label:              (関数定義の最後の部分)
movl %ebp,%esp
popl %ebp
ret
```

図 12: return 文 (return 式;) のコード

4.5.4 while 文, if 文、if-else 文

while 文 (while (式) 文) のコードは、図 13 の通りである。if 文、if-else 文も、while 文と同様に、条件によって、実行する文を制御するコードを生成すればよい。

4.6 関数呼び出しに対するコード生成のポイント

図 14 のように、実引数をスタックに積んで call 命令を呼ぶ。ここでは、実引数を逆順に積んでいる。また、関数呼び出し時にスタックポインタが 16 バイト境界を差す必要があるため、適当な padding を入れている。配布した xcc のバージョンで、既に実装されている。

```
L1:
    式のコード
    popl %eax          (式の値を%eax へ)
    cmpl $0,%eax       (%eax の値と 0 と比較)
    je L2              (そうならば、ループから脱出)
    文のコード
    jmp L1             (再び条件判定へ)
L2:
```

図 13: while 文 (while (式) 文) のコード

```
subl $padding の大きさ, %esp    (関数呼び出し時 16 バイト境界条件を満たすように、適当な padding を入れる)
式 n のコード                   (逆順に実引数を積む)
    ⋮
式 1 のコード
call _func                      (関数呼び出し、返り値は%eax にセットされる)
addl $引数の大きさ, %esp        (引数を捨てる)
addl $padding の大きさ, %esp    (padding を捨てる)
pushl %eax                      (返り値をスタックに乗せる)
```

図 14: 関数呼び出し (func(式 1,. . ., 式 n)) のコード

4.7 ポインタについて

ポインタで用いる単項演算子 &,∗ の左辺値と右辺値は、表 3 のように取り扱う。

加算、減算の 2 項演算子 + と − は、オペランドがポインタの時、表 4 のように意味が変わることに注意。4 倍するには、算術左シフト命令 sall を使えばよい。

4.8 char 型について

言語仕様上,char 型のデータは式中では int 型に暗黙に変換することになっているため、char 型の変数の右辺値は int 型に変換してプッシュする。例えば、char 型の局所変数の右辺値は、図 15 のように int 型に変換し、スタックに積む。

```
movsbl offset(%ebp),%eax    (1 バイトを 4 バイトに符号拡張し%eax へ)
pushl %eax                  (その 4 バイトデータをスタックへ)
```

図 15: char 型の変数の右辺値をスタックに積むコード

逆に、char 型の変数に値を格納するときは、下位 1 バイトを格納すればよい。

また、char 型変数を引数に持つ関数を呼び出す際は、実引数を 4 バイト境界におこう。1 バイトのデータを置く場合は、3 バイトの隙間を入れよう。

	左辺値	右辺値
& A	コンパイルエラー	A の左辺値
* A	A の右辺値	A の右辺値 popl %eax (A の右辺値をレジスタ%eax に格納) movl 0(%eax),%eax (%eax が指すメモリ中の値を%eax に格納) pushl %eax (%eax 値をプッシュ)

表 3: ポインタ用の単項演算子 &,* の右辺値、左辺値

`int *p, *q, i;` のとき、

ポインタ演算の式	その意味
`p+i`	`p + i*sizeof(*p)` を計算
`p-i`	`p - i*sizeof(*p)` を計算
`p-q`	`(p - q)/sizeof(*p)` を計算
`p+q`	コンパイルエラー

表 4: ポインタ演算

4.9 その他

権藤先生による「アセンブリ言語」の講義において、「i386 機械語命令 (1)」、「i386 機械語命令 (2)」、「アセンブラ命令」のスライドに、アセンブラ命令や機械語命令が説明されている。参考にすること。

5 課題

課題に取り組むに当たっての注意

- 提出するプログラムの採点は csc で行うので、csc で動くようにしておくこと。必要な動作確認は各自で行なっておくこと。
- 拡張した内容が明らかになるような C による例プログラムを必ず用意すること。例プログラムが指定されている課題については、必ずその例プログラムも含めること。簡単でも実際に利用可能なものを記述してみるとよい。

課題 1

int 型の大域変数の代入式 (= 式) を使ったプログラムをコンパイルできるように、コンパイラを拡張せよ。さらに、以下の式と文をコンパイルできるように拡張せよ。

- while 文
- < 式, + 式, − 式 (2 項演算子)

そして、`test/kadai1.c` をコンパイルし、正しく実行出来ることを確認せよ。(gcc でコンパイル・実行した結果と同じであることを確認せよ。)

課題 2

int 型の局所変数と関数引数を使ったプログラムをコンパイルできるように、コンパイラを拡張せよ。さらに、以下の式と文をコンパイルできるように拡張せよ。

- if 文, if-else 文, return 文
- == 式, || 式, && 式, * 式, / 式

そして、`test/kadai2.c` をコンパイルし、正しく実行出来ることを確認せよ。また、`test/kadai2.c` では、if 文, && 式, / 式を使っていない。これらの文と式を使ったサンプルプログラムを自作し、コンパイル・実行し、動作を確認せよ。

課題 3

ポインタ型（大域変数、局所変数、関数引数、関数の返り値）を使ったプログラムをコンパイルできるように、コンパイラを拡張せよ。単項演算子 *, & の演算や、ポインタ演算 (ポインタ型の式に対する + や − の 2 項演算) を伴うプログラムをコンパイルできるように拡張せよ。そして、`test/kadai3.c` をコンパイルし、正しく実行出来ることを確認せよ。なお、`test/kadai3.c` を実行するには、`10` などの引数が必要である。（例: `./a.out 10`）

小課題

課題 1 から課題 3 のレポートの点数と出席点を合わせて 100 点満点（本課題の分のみ）であるが、課題レポートの点数が足りないと考えた方は、小課題をやると、課題レポート点の範囲で加点する。自分で課題を設定すること。例えば、

- 課題では未実装だったコード生成
  - char 型の導入 `tmisc-char.c`, `tmisc-char2.c` を動作させる。
  - 未実装だった文、式のコード生成など
- 今回配布した xcc の構文解析と意味解析が対応していなかった機能の実現
  - 最適化など

6 日程とレポート締め切り

以下の 12 回を予定。

- 10/4 (火) 課題説明
- 10/11(火), 10/14(金)
- 10/18(火), 10/21(金)
- 10/25(火), 10/28(金)
- 11/1(火), 11/4(金)
- 11/8(火), 11/11(金)
- 11/15(火)

また、以下は予備日なので、出席を取らないが、優先的に実験室が使える。

- 11/18(金), 11/22(火), 11/25(金)

締め切り

課題 1 10/24(月)

課題 2 11/7(月)

課題 3、小課題 11/28(月)

7 レポート提出方法

メールで提出すること。

- 宛先: report@fmx.cs.titech.ac.jp
- サブジェクト: report 学籍番号 課題名 (例えば、report 01_12345 kadai1)
- 自分にも cc で送信して、無事の送信できたことを確認すること。
- 提出物は以下の 2 つである。メールに 2 つのファイルを添付すること。

  1. レポート pdf 形式とする。ファイル名は、学籍番号-課題名.pdf (例えば、01_12345-kadai1.pdf)

  2. 実装したソースコードとサンプルコード 提出するコードは、

  – AST.c, codegen.c, misc.h, xcc.h, AST.h, codegen.h, symbol.c, type.c, xcc.l, Makefile, misc.c, symbol.h, type.h, xcc.y, test/*.c (サンプルコード)

  である。これらを、以下の手順でまとめる。

  1. Makefile 中の以下の GAKUSEKI_BANGO(学籍番号) KADAI(課題名) をそれぞれ、適切に変更する。

```
  .....
GAKUSEKI_BANGO = 01_12345
KADAI = kadai1
  .....
```

  2. 以下のように、make pack する。すると、必要ファイルをまとめたファイル 01_12345-kadai1.tgz ができるので、それをレポートと共にメールに添付して送ること。

```
$ make pack
if [ -e 01_12345-kadai1 ]; then rm -rf 01_12345-kadai1; fi
mkdir 01_12345-kadai1
cp xcc.h  AST.h type.h symbol.h misc.h codegen.h AST.c type.c symbol.c misc.c
 Makefile xcc.y xcc.l codegen.c 01_12345-kadai1
if [ ! -e 01_12345-kadai1/test ]; then mkdir 01_12345-kadai1/test; fi
cp -r test 01_12345-kadai1/
tar cvzf 01_12345-kadai1.tgz 01_12345-kadai1
a 01_12345-kadai1
a 01_12345-kadai1/AST.c
a 01_12345-kadai1/AST.h
a 01_12345-kadai1/codegen.c
   ..(省略)..
a 01_12345-kadai1/test/kadai3.c
a 01_12345-kadai1/test/Makefile
a 01_12345-kadai1/test/tmisc-char.c
a 01_12345-kadai1/test/tmisc-char2.c
$ ls *tgz
01_12345-kadai1.tgz
$
```

8 レポートの形式

最低限以下の内容を含むこと。

- 表紙（課題名、学籍番号、ログイン名、名前、メールアドレス）
- 目的
  新たな機能などを実装した場合の利点を簡潔に述べておく。
- 変更点
  拡張の仕様、拡張のために変更すべき点について説明する。ソースコードのどこをどう直すという説明ではなく（それはソースを見ればわかる）、自分の変更で「コンパイラが現在こういう動作をしているものが、こういう風に動作するようになる」こと、それで「目的が達成される」こと（つまり、変更の意図）を説明すること。拡張によりコンパイラが生成可能となったアセンブリコードの意味と意図も説明すること。
- 例プログラムとその実行結果
  実行結果が長い場合は、わかるように適当に省略すること。また、実行結果にコンパイラが出力するアセンブリコードは必要ない。
- 評価
  目的に対してどの程度実現できたかを自分で評価しておく。
- 考察
  さらに発展として考えられることや、他の方法などを整理しておく。拡張に当たって自分で調べたことがあれば、それもここで整理しておく。
- まとめ

9 提出されたレポートの採点

レポートは「レポートの形式」のページに指示されている形式になっているものについてのみレポートの採点を行ない、自分か何をやっているかきちんとわかっていると判断される場合にのみプログラムの採点を行う。なお、採点に関してはレポートを 7 割、プログラム 3 割程度で行う予定である。

10 掃除当番

`http://www1.csc.titech.ac.jp/class/joho4/compile/`

11 参考文献

- i386 機械語命令のマニュアル
  IA-32 インテル アーキテクチャー・ソフトウェア・デベロッパーズ・マニュアル
  http://www.intel.co.jp/content/www/jp/ja/processors/architectures-software-developer-manuals.html
  以下で、google で検索をすると、以前の日本語で書かれたマニュアルも見つかりました。
  アーキテクチャー ソフトウェア デベロッパーズ マニュアル ”IA-32”
- GNU アセンブラのマニュアル
  OS X Assembler Reference
  https://developer.apple.com/library/content/documentation/DeveloperTools/Reference/Assembler/000-Introduction/introduction.html

- ABI for Mac OS X
  Introduction to Mac OS X ABI Function Call Guide
  https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/LowLevelABI/000-Introduction/introduction.html
  Mach-O Programming Topics
  https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/MachOTopics/0-Introduction/introduction.html