



2016年はXCC全部を作ってもらいました。
2017年以降は小さなサブセットにします。
2019年, long型(64ビット)を追加。



コンパイラ構成

課題 1 : XCC-small 再帰下降型構文解析器

情報工学系
権藤克彦



課題の概要

- XC-small の再帰下降型構文解析器を実装する
- すること
 - 次ページ以降の文法に対して、与えるfirst/followを用いて、ラベル文と式文、および、ぶらぶらelseが衝突するので、LL(1)文法では無いことを示せ（要提出）
 - xcc-small.c を修正・拡張して、parse_translation_unit 関数とunparse_AST 関数を実装
 - テスト用入力ファイル(test1/t*.c, test2/t*.c)でテストする
- 締切：7/12（金）17:00
 - 遅刻レポートも受け取るが秘密の半減期で減点。
遅刻レポート受け取りは2番目の課題の〆きり時間まで。



XC-small のLL(1)文法 (1/2)

```
translation_unit
  : ( type_specifier declarator ( ";" | compound_statement ) ) *
  ;
type_specifier
  : "void" | "char" | "int" | "long" ;
declarator
  : IDENTIFIER [ "(" ")" ] ;
compound_statement
  : "{" (type_specifier declarator ";")* ( statement )* "}" ;
exp: primary [ "(" ")" ] ;
primary
  : INTEGER | CHARACTER | STRING | IDENTIFIER | "(" exp ")" ;
```

- 型は, void, char, int, long, 関数のみ
- 式は, 整数定数, 文字定数, 文字列定数, 識別子, カッコ, 関数呼び出しのみ



XC-smallのLL(1)文法 (2/2)

statement

```
: IDENTIFIER ":" // 注1  
| compound_statement  
| "if" "(" exp ")" statement [ "else" statement ] // 注2  
| "while" "(" exp ")" statement  
| "goto" IDENTIFIER ";"  
| "return" [ exp ] ";"  
| [ exp ] ";" ;
```

- 注1：expのfirstと衝突→2トークン先読みが必要
- 注2：ぶらぶらelse解消のため、次トークンが「else」なら if-else文と要解釈
 - followにも「else」は入っているが、それは無視

この2点の部分のみLL(1)では無い



first/follow集合

	first	follow
translation_unit	void char int long ϵ	\$
type_specifier	void char int long	IDENTIFIER
declarator	IDENTIFIER	; {
statement	{ if while goto return ; INTEGER CHARACTER STRING IDENTIFIER (else } { if while goto return ; INTEGER CHARACTER STRING IDENTIFIER (
compound_statement	{	\$ void char int long else } { if while goto return ; INTEGER CHARACTER STRING IDENTIFIER (
exp	INTEGER CHARACTER STRING IDENTIFIER () ;
primary	INTEGER CHARACTER STRING IDENTIFIER () ; (

頑張って計算しましたが、間違えてたらごめんなさい



実装する関数の型

```
static struct AST* parse_translation_unit (void);  
static void unparse_AST (struct AST *ast, int depth);
```

```
int main (int argc, char *argv[])  
{  
    // 中略  
    create_tokens (ptr);  
    reset_tokens ();  
    // main関数で以下の通り, 呼び出すこと  
    ast = parse_translation_unit ();  
    unparse_AST (ast, 0);  
}
```



parse_translation_unit関数

- argv[1]でファイル名を与えられたXCプログラムを構文解析した結果の構文木を返す関数
- 構文エラーを検出したら、parse_error関数呼んでプログラムを終了させる



unparse_AST関数

- 第1引数はstruct AST型の構文木へのポインタ
- 第2引数は表示するインデントの深さ
- 返値は無し
- 構文木を元のプログラムに逆変換した結果を標準出力に出力
 - コンパイル可能なものを要出力
 - コメントは出力不要
 - 空白類は元と同じで無くても良い
 - インデントをつける
 - if文とif-else文の内部のstatementは必ず { と } をつけて出力
 - ぶらぶらelseが正しく構文解析されたことを明示するため
- 構文木にエラーを見つけたら, unparse_error を呼んでプログラムを終了させる



unparse_AST関数（表示例）

```
int main ()
{
    if (1)
        if (2)
            return 0;
    else
        return 1;
}
```

入力



```
int main ()
{
    if (1)
    {
        if (2)
        {
            return 0;
        }
        else
        {
            return 1;
        }
    }
}
```

出力



拡張課題（オプション）

- 外付けのボーナス点（最大10点）を与える
- 内容は任意。以下は例。
 - XC-smallで省略した文法（宣言や式）の構文解析を加える。
 - XCに含まれないC言語の文法の構文解析を加える。
- 拡張課題をした場合は
 - レポート中で拡張課題があることを明記する
 - 拡張課題のソースコードは `xcc-small.c` とは別に提出する



提出方法

- T2SCHOLA の課題一覧から提出.
- 提出物一式（次スライド参照）をtarで1つのファイルに固めて、アップロードすること.
 - 学籍番号と同じディレクトリを作り、そこに送るファイルをすべてコピーする.
% mkdir 22B12345
% cp ファイル名 22B12345
 - tar で1つのファイルにまとめる.
% tar cvzf 22B12345.tgz 22B12345

このファイルを
アップロード.

オプションの意味
c 作成 create
v 詳細出力 verbose
z gzip圧縮
f ファイル名指定 file



提出物一式の確認

- 固めたファイルが正しく解凍できるかチェックする

```
% cp 22B12345.tgz /tmp 別ディレクトリにコピー
% cd /tmp
% tar xvzf 22B12345.tgz 解凍・展開
% cd 22B12345
% ls
xcc-small.c report.pdf
%
```

オプションの意味

x 解凍（展開） extract



提出物

ファイル分割しない

- プログラム（ソースコードのみ，各ファイル1つで）
 - xcc-small.c：XC言語の再帰下降型構文解析器
 - struct ASTの宣言も xcc-small.c に含める
- レポート
 - report.txt または report.pdf
(図が必要な場合は PDF形式の report.pdf中に含めること)

コンパイルにオプションが必要な場合は，
Makefile を書いて提出することを推奨．



提出してはいけないもの

- バイナリファイル (a.out, *.o)
- 昔のソースコード, バックアップファイル (*~)
- winmail.dat (Outlook 使いの人は要注意)
- その他, レポートに関係ないファイル.



レポートに書く内容

無駄にダラダラ長い文章は減点.
ページ稼ぎ禁止.
ソースコード引用は最低限に.
内容が少ないものも減点.

- 次のことをアピールする文章を（簡潔に）書く.
 - 「私はこの課題を深く理解して、しっかり取り組んだ」
 - 「私は様々な試行錯誤をし、様々な工夫もした」

情報少ないと「分かってないな」
とこちらは判断

特にプログラムが未完成の場合は、
その分、レポートも頑張って書く！

- 書くことの例

- 設計上の取捨選択とその理由・結果.
- 何をどこまで作ったか. 既知のバグ（もしあれば）.
- 改良や拡張すべき点とその方法.
- 議論（考察）.
- 感想（採点外）.

単に「プログラムをきれいに直したい」
と書くだけでは、議論として意味がない.



入力と出力のお約束

- こちらで自動的にテストするために必要
- 守らなかった場合は減点
- XCCは入力ファイル名を **argv[1]** から受け取る
- XCCは**unparse結果**を標準出力に出力
 - 余計なデバッグ情報を出力しないこと

```
% gcc xcc-small.c
% ./a.out test/t1.c
/* comment #1 */
int printf ();
...
%
```



テスト用の入力ファイル

- test1/t*.c
 - 構文解析に成功する入力ファイル
- test2/t*.c
 - 構文解析に失敗する入力ファイル



プログラムの書き方

- 他人が読みやすく理解しやすいプログラムを書く.
- 定石：
 - きれいにインデント（字下げ）する.
 - 分かりやすい変数名をつける.
 - 良いコメントを書く.
 - モジュール化する.

参考書：プログラミング作法

<http://www.amazon.co.jp/dp/4756136494>





カンニングはダメ

- アイデアレベルでの議論はOK.
- ソースコードを見るのはカンニングと見なす.
- 警告：剽窃チェッカーを使います.
見せた方も同罪. 不合格にします.
- 見せない努力も必要.
 - ソースコードを不用意に印刷して放置しない.
 - ソースコードを表示したまま, 席を離れない.
 - ファイルやディレクトリの他人の読み取りを不許可にする.
 - 議論する際にソースコード (疑似コード) を使わない.
 - デバッグを助けてもらう時もソースコードは見せない.
 - Github等でパブリックに公開しない.



その他の注意

- 情報工学科計算機室（CSC）の使用
 - 他の授業や演習を行っていない時間だけ使用すること.
 - その他, CSCの利用規則を遵守すること.
 - 違反した場合は厳罰を科す.
- リスク管理をすること.
 - **×切りギリギリを狙わない.**
 - ・ 根拠無く「×切一週間前からやろう」などと思わない.
 - 風邪, 停電, 学会発表, CSCの保守など, 課題に取り組めない事態はいくらでもありうる.
 - 自己責任で, 早めに課題に取り組むこと.
 - リスク管理は社会人に必要な重要なスキル.



ヒント

- 権藤が書いた parse_translation_unit (1/2)

```
static struct AST*
parse_translation_unit (void)
{
    struct AST *ast, *ast1, *ast2, *ast3;

    ast = create_AST ("translation_unit", 0);
    while (1) {
        switch (lookahead (1)) {
            case TK_KW_INT: case TK_KW_CHAR: case TK_KW_VOID:
                ast1 = parse_type_specifier (); // あなたが実装
                ast2 = parse_declarator ();      // あなたが実装
                switch (lookahead (1)) {
                    case ';':
                        consume_token (';');
                        ast3 = create_AST (";", 0);
                        break;
```



ヒント

- 権藤が書いた parse_translation_unit (2/2)

```
        case '{':
            ast3 = parse_compound_statement (); // あなたが実装
            break;
        default:
            parse_error ();
            break;
    }
    ast = add_AST (ast, 3, ast1, ast2, ast3);
    break;
default:
    goto loop_exit;
}
}
loop_exit:
    return ast;
}
```



ヒント

- 権藤が書いた unparse_AST の一部 (1/2)

```
static void
unparse_AST (struct AST *ast, int depth)
{
    int i;
    if (!strcmp (ast->ast_type, "translation_unit")) {
        for (i = 0; i < ast->num_child; i++) {
            printf_ns (depth, ""); // インデント
            unparse_AST (ast->child [i], depth+1);
            unparse_AST (ast->child [i+1], depth+1);
            if (!strcmp (ast->child [i+2]->ast_type, ";")) {
                printf (";\n");
            } else if (!strcmp (ast->child [i+2]->ast_type,
                                "compound_statement")) {
                printf ("\n");
                unparse_AST (ast->child [i+2], depth);
            }
        }
    }
}
```



ヒント

- 権藤が書いた `unparse_AST` の一部 (2/2)

```
        } else {  
            unparse_error (ast);  
        }  
        i+=2;  
    }  
    // ここに書く．残りコードはあなたが実装  
} else {  
    unparse_error (ast);  
}  
}
```



付録

- ・ コツ：ちょっとずつ作る
- ・ SEGMENTATION
FAULT



コツ：ちょっとずつ作る

- 一般的にプログラミングは以下でやるとうまくいく
 - 少しコードを書く→実行して動作確認→動くと嬉しい
- こまめに実行しないと、デバッグが大変だから
- この課題の場合はどうする？
 - × `parse_*`を全部作ってから、`unparse_AST`を作る
 - ○ （カッコ以外の）`parse_primary`を作って、（`primary`だけを出力できる）小さな `unparse_AST`を作る
→ 少しずつ扱える文法を増やしていく



Segmentation fault 対策 (1)

- ポインタ処理を間違えると、実行時エラーとなる
 - 例：Segmentation fault, Bus error
 - ただし、必ず実行時エラーとなるわけでは無い
(暴走する可能性もある、こっちの方がやっかい)
- 実行時エラーはデバッガで調べれば、簡単に原因が分かることが多い
 - ググれば、やり方はいくらでも出てくる！

文法エラーや実行時エラーは、バグとしては簡単な部類.

講義資料 lldb.pdf を読み、デバッガの使い方を覚えよう



Segmentation fault 対策 (2)

- とある学生さんのバグの調査例

```
% gcc -g xcc-small.c
% lldb ./a.out
(lldb) run ../test1/t1.c
Process 20269 stopped
* thread #1: tid = 0x5e637f, 0x0000000100002ad1
a.out`unparse_AST(ast=0x00000001001045d0, depth=0) + 3809 at
xcc-small.c:1454, queue = 'com.apple.main-thread', stop reason =
EXC_BAD_ACCESS (code=1, address=0x0)
    frame #0: 0x0000000100002ad1
a.out`unparse_AST(ast=0x00000001001045d0, depth=0) + 3809 at
xcc-small.c:1454
    1451             for (i = 0; i < ast->num_child;){
    1452                 unparse_AST(ast->child[i], depth);
    1453                 i++;
-> 1454                 if (!strcmp(ast->child[i]->ast_type,
"*"))){
```

1454行目でNULLアクセス



Segmentation fault 対策 (3)

- とある学生さんのバグの調査例

```
(lldb) print ast->child[i]
(AST *) $0 = 0x0000000000000000
```

この値がNULL

```
(lldb) bt
```

```
* thread #1: tid = 0x5e637f, 0x0000000100002ad1
a.out`unparse_AST(ast=0x00000001001045d0, depth=0) + 3809 at
xcc-small.c:1454, queue = 'com.apple.main-thread', stop reason
= EXC_BAD_ACCESS (code=1, address=0x0)
```

関数の呼び出し列を表示

```
* frame #0: 0x0000000100002ad1
a.out`unparse_AST(ast=0x00000001001045d0, depth=0) + 3809 at
xcc-small.c:1454
```

```
frame #1: 0x00000001000029c6
a.out`unparse_AST(ast=0x00000001001045a0, depth=0) + 3542 at
xcc-small.c:1437
```

(以下略)

```
(lldb) quit
```

```
%
```



ASLRとPIEの無効化：概要

- ASLR = address space layout randomization
 - ライブラリを置くアドレスなどをランダム化
- PIE = position independent executable
 - 位置独立実行可能ファイル
- ASLRとPIEのせいで、実行毎にアドレスが変わる
 - デバッグ時には ASLR と PIE を無効化すべし

```
$ gcc -g foo.c
$ ./a.out
0x1073aaf30, 0x7ff800da3fbb, 0x600002f10030
$ ./a.out
0x10911af30, 0x7ff800da3fbb, 0x60000068c030
$ ./a.out
0x10bf5bf30, 0x7ff800da3fbb, 0x600003d8c030
```

アドレスが毎回変わる

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int *p = malloc (sizeof (int));
    printf ("%p, %p, %p\n", main, printf, p);
}
```



ASLRとPIEの無効化：macOS

- lldb だと自動で無効化される
 - (lldb) settings set target.disable-aslr true がデフォルト
- -fno-PIE オプションでも無効化

アドレスが固定に

```
$ gcc -g -fno-PIE foo.c
$ ./a.out
0x100003f30, 0x7ff800da3fbb,
0x600000004030
$ ./a.out
0x100003f30, 0x7ff800da3fbb,
0x600000004030
$ ./a.out
0x100003f30, 0x7ff800da3fbb,
0x600000004030
```



ASLRとPIEの無効化：Linux

```
$ sudo sysctl -w  
kernel.randomize_va_space=0  
$ gcc -g -no-pie foo.c  
$ ./a.out  
0x401146, 0x7ffff7e1dc90, 0x4052a0  
$ ./a.out  
0x401146, 0x7ffff7e1dc90, 0x4052a0  
$ ./a.out  
0x401146, 0x7ffff7e1dc90, 0x4052a0
```

再起動するたび

コンパイルするたび

アドレスが固定に