課題1
XCC-small 再帰下降型構文解析器

23B13163 宮本魁人

本レポートは以下のような構成になっている.

1. XC-small が LL(1) 文法ではないこと
2. `parse_translation_unit` 関数
3. `unparse_AST` 関数
4. 拡張課題(XC)
5. テスト
6. 議論

---

1. XC-small が LL(1) 文法でないこと

XC-small は以下の2点において, LL(1) 文法ではない.

1. `statement` において, 1トークン先読みでは一意に展開できない.
   - `statement: IDENTIFER ":" | [exp] ";" | ... ;`
   - `exp: primary ["(" ")"];`
   - `primary: IDENTIFER | ... ;`
     であるから,
   - `director(statement, IDENTIFIER ":") = {IDENTIFIER}`
   - `director(statement, [exp] ";") = first(exp) ∪ {";"} = {IDENTIFIER, ..., ";"}` となる. したがって, director(statement, IDENTIFIER ":") ∪ director(statement, [exp] ";") = {IDENTIFIER} `$\ne \phi$ つまり, 1トークン先が` IDENTIFIER` の場合に どちらに展開すればよいか判別できず、LL(1) 文法の定義に反する.

2. `statement` において, `if-else` 文に曖昧性がある.
   `statement: "if" "(" exp ")" statement ["else" statement ] | ...;` であるため, 次のような文を一意に解釈できない.
   `"if" "(" exp ")" "if" "(" exp ")" statement "else" statement`
   このとき, `"else"` がどちらの `if` につくのかを判別できず曖昧性が生じる.
   今回のXC言語では、常に一番近い `if` とくっつくように解釈する.

---

2. `parse_translation_unit` 関数

この関数は, 字句解析されたXCプログラムを受けて, 構文解析を行い, 結果として構文木（AST）を返す関数である.

XC言語の非終端記号に対応させるため，`parse_translation_unit` 関数を実装する際に，以下のような `parse_*` 関数群を用意した．

```
static struct AST *parse_type_specifier(void);
static struct AST *parse_declarator(void);
static struct AST *parse_compound_statement(void);
static struct AST *parse_exp(void);
static struct AST *parse_primary(void);
static struct AST *parse_statement(void);
```

これらの関数では共通して、非終端記号に対応するノードを構築し、構文規則に従って再帰的に子ノードを追加していく構成をとっている。
まず、終端記号と非終端記号の処理方法を説明し、その後に各構文パターンの処理方法について具体的に述べる。最後に，各 `parse_*` 関数において，どのように適用しているかを記述する．

A. 終端記号 a の変換

用意された `parse_translation_unit` では、終端記号の処理に `ast3 = create_AST(";", 0);` のようなコードが見られたが、以下のような問題を抱えていた:

- `ast_type` として文字列 `";"` を直接記述しているため，スペルミス（例: `";;"` ）をしてもコンパイルエラーが発生せず，バグに気付きにくい．
- `enum token_kind` の定数を変更したい場合に，一括での置換が困難．

これを解決するために，以下のような関数を定義した:

```
static struct AST *create_final_word(enum token_kind kind) {
  struct AST *ast = create_leaf(token_kind_string[kind], token_p->lexeme);
  consume_token(kind);
  return ast;
}
```

この関数では、トークンの種類 `kind` に応じて対応する葉ノードを生成し、トークンを1つ消費する。

- `token_kind_string[kind]` より、 `ast_type` が自動で決定される．
- `token_p->lexeme` によって、識別子などの具体的な文字列も保存される．

使用例

```
ast3 = create_final_word(';');
```

B. 非終端記号 A の変換

非終端記号 A の変換には、対応する `parse_*` 関数を呼び出す。

```
A = parse_*(); // A に対応する解析関数を呼び出す
```

使用例

```
ast3 = parse_compound_statement(); // compound_statement の処理
```

非終端記号ノードは、終端記号と同様に, 次のような `enum` を作成して管理している:

```
enum grammar_constructs_kind {
  CK_UNUSED,
  CK_TRANSLATION_UNIT = 0,
  CK_TYPE_SPECIFIER,
  CK_DECLARATOR,
  CK_COMPOUND_STATEMENT,
  CK_EXPRESSION,
  CK_PRIMARY,
  CK_STATEMENT,
};

char *grammar_constructs_kind_string[] = {
    "unused",
    "translation_unit",
    "type_specifier",
    "declarator",
    "compound_statement",
    "exp",
    "primary",
    "statement",
};

static struct AST *create_construct_AST(enum grammar_constructs_kind kind) {
  struct AST *ast = create_AST(grammar_constructs_kind_string[kind], 0);
  return ast;
}
```

各 `parse_*` 関数ではまず `create_construct_AST` によりノードを生成し、その後、子ノードを `add_AST` により追加していく。

使用例（`parse_exp` より抜粋）:

```
static struct AST *parse_exp(void) {
  struct AST *ast, *ast1;
  ast = create_construct_AST(CK_EXPRESSION);
  ast1 = parse_primary();
  ast = add_AST(ast, 1, ast1);
  return ast;
}
```

C. 記号列 α の変換

文法規則の右辺に現れる記号列 α は、終端記号、非終端記号、空文字列(ε)、繰り返し、選択などを含む。

- 例：`exp ";"`，`type_specifier declarator (";" | compound_statement)`
  以下では、典型的な構文パターンごとの変換方法を示す。

ここで, 記述を簡単にするため, 記号列α を構文解析して, その構文木を返す関数という意味合いで `parse_alpha` の様に表現する. 必ずしも実際に関数を作成しているわけではなく, 単に処理を抽象化して, 記述しやすくしているだけに過ぎない.

D. $\alpha : \alpha_1\alpha_2\cdots\alpha_k$ の変換（連接）

記号列が連続して並ぶ場合、それぞれの部分を個別に解析してから、まとめて `add_AST` で結合する。

```
alpha_1 = parse_alpha_1();
alpha_2 = parse_alpha_2();
...
alpha_k = parse_alpha_k();

alpha = add_AST(alpha, k, alpha_1, alpha_2, ...);
```

使用例

```
ast1 = create_final_word('(');
ast2 = parse_exp();
ast3 = create_final_word(')');
ast = add_AST(ast, 3, ast1, ast2, ast3);
```

E. $\alpha : \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$ の変換（選択）

選択肢が存在する場合には、`switch` 文を用いて先読み記号に応じた処理を分岐する。

```
switch(lookahead(1)) {
    case director(alpha, alpha_1):
        ast1 = parse_alpha_1();
        break;
    ...
    default:
        parse_error();
}
alpha = add_AST(alpha, 1, ast1);
```

使用例（`parse_type_specifier`）

```
static struct AST *parse_type_specifier(void) {
  struct AST *ast, *ast1;
  ast = create_construct_AST(CK_TYPE_SPECIFIER);
  switch (lookahead(1)) {
    case TK_KW_INT:
    case TK_KW_LONG:
    case TK_KW_CHAR:
    case TK_KW_VOID:
      ast1 = create_final_word(lookahead(1));
      ast = add_AST(ast, 1, ast1);
      break;
    default:
      parse_error();
  }
  return ast;
}
```

F. $\alpha : [\alpha_1]$ の変換（省略可能）

省略可能な構文は、`if` 文で条件をチェックして、対応する構文を追加するかどうかを判断する。

```
if (lookahead(1) == director(alpha, alpha_1)) {
    alpha_1 = parse_alpha_1();
    alpha = add_AST(alpha, 1, alpha_1);
}
```

ここで，`director(alpha, alpha_1) = {a_1, a_2, ...}` のように複数ある場合は，以下の様に記述する．

```
if (lookahead(1) == a_1 || lookahead(1) == a_2 || ..
```

使用例（`parse_declarator`）

```
static struct AST *parse_declarator(void) {
  struct AST *ast, *ast1, *ast2;
  ast = create_construct_AST(CK_DECLARATOR);
  ast1 = create_final_word(TK_ID);
  ast = add_AST(ast, 1, ast1);

  if (lookahead(1) == '(') {
    ast1 = create_final_word('(');
    ast2 = create_final_word(')');
    ast = add_AST(ast, 2, ast1, ast2);
  }
  return ast;
}
```

G. $\alpha : \alpha_1^*$ の変換（0回以上の繰り返し）

繰り返しは，`while` 文を使ってトークンの先読みにより繰り返し条件を判定し，子ノードを逐次追加していく．

```
while (1) {
    if (lookahead(1) != director(alpha, alpha_1)) break;
    alpha_1 = parse_alpha_1();
    alpha = add_AST(alpha, 1, alpha_1);
}
```

ここで，`director(alpha, alpha_1) = {a_1, a_2, ...}` のように複数ある場合は，Gと同様に `if` 文を用いて以下の様に記述できる．

```
if (!(lookahead(1) == a_1 || lookahead(1) == a_2 || ..)) break;
```

あるいは `switch` 文を用いて以下の様に記述できる．（`alpha_1` が$\beta_1 \mid \beta_2 \mid \cdots \beta_k$ 型の時は特に有用．）

```
while(1) {
    switch(lookahead(1)) {
    case a_1 :
    case a_2:
    ...
    alpha_1 = parse_alpha_1();
```

```
        alpha = add_AST(alpha, 1, alpha_1);
            break;
        default:
            goto loop_exit;
        }
    }
    loop_exit:
```

使用例（`parse_translation_unit` より抜粋）

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

  while (1) {
    switch (lookahead(1)) {
      case TK_KW_INT:
      case TK_KW_LONG:
      case TK_KW_CHAR:
      case TK_KW_VOID:
        ast1 = parse_type_specifier();
        ast2 = parse_declarator();
        ast3 = create_final_word(';'); // または compound_statement など
        ast = add_AST(ast, 3, ast1, ast2, ast3);
        break;
      default:
        goto loop_exit;
    }
  }
loop_exit:
  return ast;
}
```

以下では, 具体的な `parse_*` 関数について、これらの変換をどのように用いているかを説明する. ただ, 規則になぞらえて機械的に記述しているだけなので, 上記の変換とは外れた処理を行っている箇所について重点的に説明を行う.
コードの貼り付けは行わないので, 添付したソースファイル `xcc-small.c` を参照してほしい.

H. `translation_unit`

対応文法:

```
translation_unit
  : ( type_specifier declarator ( ";" | compound_statement ))*
;
```

構文解析方針:

- 非終端記号としてのノードを生成し（`CK_TRANSLATION_UNIT`）、その中に繰り返し構文の結果を追加していく.
- 繰り返し構文の判定には、`type_specifier` の先頭トークンである `TK_KW_INT` などによる `while` 判定を使う.
- 本体は `type_specifier` → `declarator` → `";"` or `compound_statement` の順に構築.
- 関数定義（`{}`）と関数宣言（`;`）の両方に対応.

I. `type_specifier`

対応文法:

```
type_specifier
  : "void" | "char" | "int" | "long"
;
```

構文解析方針:

- トークン種別を `switch` で判定し、該当する終端記号を `create_final_word` で葉ノードとして処理.
- キーワードがそのままASTに現れるため、処理は単純.

J. `declarator`

対応文法:

```
declarator
  : IDENTIFIER [ "(" ")" ]
;
```

構文解析方針:

- 最初に識別子 `IDENTIFIER` を処理.
- 続いて `"(" ")"` が現れるかどうかを `if` 文でチェック（省略可能構文に対応）.

K. `compound_statement`

対応文法:

```
compound_statement
  : "{" (type_specifier declarator ";")* ( statement )* "}"
;
```

構文解析方針:

- `"{"` と `"}"` に挟まれるブロック構文を処理.
- 前半の `type_specifier declarator ";"` は局所変数定義、後半の `statement` は文を表す.
- 各セクションを `while` 文で繰り返し処理.
- `type_specifier` による先読みで前半のループを制御し、`statement` の先頭トークンの集合で後半ループを制御.

L. `exp`

対応文法:

```
exp
  : primary [ "(" ")" ]
;
```

構文解析方針:

- `primary` を先に解析し、続いて `"(" ")"` の存在を確認して関数呼び出しのような形に対応.
- `"(" ")"` は省略可能なため、`if` 文でチェック.

M. `primary`

対応文法:

```
primary
  : INTEGER
  | CHARACTER
  | STRING
  | IDENTIFIER
```

```
  | "(" exp ")"
;
```

構文解析方針:

- 先頭トークンに応じて、定数（整数・文字・文字列）や識別子を処理.
- `"(" exp ")"` の場合は括弧付き式を再帰的に処理.
- `switch` 文により、分岐と構文木構築を行う.

N. `statement`

対応文法:

```
statement
  : IDENTIFIER ":"                                    // 注1: ラベル文
  | compound_statement
  | "if" "(" exp ")" statement [ "else" statement ] // 注2: else は省略可能
  | "while" "(" exp ")" statement
  | "goto" IDENTIFIER ";"
  | "return" [ exp ] ";"
  | [ exp ] ";"                                       // 空文または式文
;
```

構文解析方針:

- ラベル構文(注1) は, 1. で説明した通り, 2つ先読み記号を見て判断する必要がある. そのため, `switch` 文で扱うと処理が煩雑になるので, `if` 文を用いて特別に処理をした.
  それ以降の選択肢は, `switch` 文による分岐処理を行った.
- `if-else` 文は上記の文法では複数の解釈ができるが, 今回は指定通り, 次トークン `lookahead(1)` が `else` なら `if-else` 文として解釈した.
- `return` 文や `if-else`, `while` においては、それぞれの構成要素を順に `add_AST` に追加.
- `[exp] ";"` は, コード上では, `exp ";" | ";"` の様に解釈して記述した.

3. `unparse_AST` 関数

`unparse_AST` 関数は, `parse_translation_unit` によって, 構文解析された構文木 `struct AST *` を受けて, 元のプログラムに逆変換した結果を標準出力に出力する関数である.
この際, `depth` はインデントの深さを表しており, フォーマットを整えるために必要となる.
こちらも非終端記号に対応させて, `unparse_*` 関数に分離して作成した.
これらの関数を直接呼び出しても良いのだが, 課題の内容が一つにまとめた `unparse_AST` 関数を作成することだったので, やや冗長的ではあるが, 以下の様に記述した.

```
static void unparse_AST(struct AST *ast, int depth) {
  if (is_CK_type(ast, CK_TRANSLATION_UNIT)) {
    unparse_translation_unit(ast, depth);
  } else if (is_CK_type(ast, CK_TYPE_SPECIFIER)) {
    unparse_type_specifier(ast, depth);
  } else if (is_CK_type(ast, CK_DECLARATOR)) {
    unparse_declarator(ast, depth);
  } else if (is_CK_type(ast, CK_COMPOUND_STATEMENT)) {
    unparse_compound_statement(ast, depth);
  } else if (is_CK_type(ast, CK_EXPRESSION)) {
    unparse_exp(ast, depth);
  } else if (is_CK_type(ast, CK_PRIMARY)) {
    unparse_primary(ast, depth);
  } else if (is_CK_type(ast, CK_STATEMENT)) {
    unparse_statement(ast, depth);
  } else {
    unparse_error(ast);
  }
}
```

ここで `is_CK_type` 関数は, 詳細は後述するが, 構文木と非終端記号が対応しているかを判定して返す関数である.
これは, 第二節の `create_final_word` 関数や `create_construct_AST` 関数に対応したもので, 保守性を高めるために作成した. 関連する関数として以下の4つの関数を作成した.

- `is_TK_type`

```
static int is_TK_type(struct AST *ast, enum token_kind kind) {
  return !strcmp(ast->ast_type, token_kind_string[kind]);
}
```

この関数は, 構文木(`*ast`) と, 終端記号(`enum token_kind`) を引数で受け取って, 構文木の `ast_type` が終端記号に対応しているかを判定して返す関数である.

- `is_CK_type`

```
static int is_CK_type(struct AST *ast, enum grammar_constructs_kind kind) {
  return !strcmp(ast->ast_type, grammar_constructs_kind_string[kind]);
}
```

この関数は, 構文木(`*ast`) と, 終端記号(`enum grammar_constructs_kind`) を引数で受け取って, 構文木の `ast_type` が非終端記号に対応しているかを判定して返す関数である.

- `check_TK_type`

```
static void check_TK_type(struct AST *ast, enum token_kind kind) {
  if (!is_TK_type(ast, kind)) unparse_error(ast);
}
```

この関数は，`is_TK_type` 関数を用いて，構文木の `ast_type` と引数で受け取った終端記号が一致していなかったらエラーを返す関数である．対応していることが必須である場合はこちらを用いる．

- `check_CK_type`

```
static void check_CK_type(struct AST *ast, enum grammar_constructs_kind kind) {
  if (!is_CK_type(ast, kind)) unparse_error(ast);
}
```

この関数も同様に，`is_CK_type` 関数を用いて，構文木の `ast_type` と引数で受け取った非終端記号が一致していなかったらエラーを返す関数である．

また，補助的な関数として，`depth` を受け取り，`4*depth` の空文字列を出力することで，インデントを表示する `printf_ns` 関数を利用した．(私が作成したものではないので，コードの提示は控える．)

以下では，2. `parse_translation_unit` と同様に，各構文パターンについて，どのように逆変換をするかを述べた後に，各 `unparse_*` 関数でどのように用いているかを説明する．

A. 終端記号 a の逆変換

`unparse` する際に，正しい構文木であることも確認するので，`is_TK_type` あるいは，`check_TK_type` を用いて，構文木の `ast_type` が意図した終端記号に対応していることを保証してから逆変換を行う．
終端記号 a の逆変換は `printf` によって出力し，対応する文字列をべた書きあるいは `lexeme` を用いて行う．

```
if (is_TK_type(ast, a)) {
    priintf("a") // 対応する文字列を出力
}

//あるいは
check_TK_type(ast, a);
printf("%s ", ast->lexeme);
```

使用例

```
if (is_TK_type(ast->child[i + 2], ';')) {
  printf(";\n");
}
```

B. 非終端記号 A の変換

非終端記号も同様に，`is_CK_type` または `check_CK_type` によって，構文木(`*ast`) が非終端記号に対応していることを保証してから，逆変換を行う．
非終端記号 A の逆変換は，`unparse_AST` を用いて行う．`parse` の時と同様に，A に対応する `unparse_*` 関数を呼び出した方が直感的ではあるが，先述の通り，まとめた `unparse_AST` 関数の作成が課題となっているので，`unparse_AST` 関数を呼び出している．

```
if (is_CK_type(ast, A)) {
    unparse_AST(ast, depth);
}

//あるいは
check_CK_type(ast, A);
unparse_AST(ast, depth+1);
```

使用例

```
else if (is_CK_type(ast->child[i + 2], CK_COMPOUND_STATEMENT)) {
  unparse_AST(ast->child[i + 2], depth);
  printf("\n\n");
}
```

C. 記号列 α の逆変換

`parse` でも説明したが，文法規則の右辺に現れる記号列 α は、終端記号、非終端記号、空文字列(ε)、繰り返し、選択などを含む。

- 例：`exp ";"`，`type_specifier declarator (";" | compound_statement)`
  以下では、典型的な構文パターンごとの逆変換方法を示す。

同様に，記号列 α を逆変換する関数という意味合いで，`unparse_alpha` の様に表現する．
また，記号列 α のはじめの終端記号・非終端記号を `kind(alpha)` の様に表現し，
`is_TK_type` と `is_CK_type` を合わせて `is_type` 関数として便宜上表記する．

D. $\alpha : \alpha_1\alpha_2\cdots\alpha_k$ の逆変換（連接）

記号列が連続して並ぶ場合、単に順番に逆変換を行えばよい.

```
unparse_alpha_1();
unparse_alpha_2();
...
unparse_alpha_k();
```

使用例

```
printf("(");
check_CK_type(ast->child[1], CK_EXPRESSION);
unparse_AST(ast->child[1], depth);
check_TK_type(ast->child[2], ')');
printf(")");
```

E. $\alpha : \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$ の逆変換（選択）

`parse` では `switch` 文を用いて分岐していたが, `token` は文字 `char` であったのに対して, `ast_type` は文字列 `char*` であるから, `switch` 文を用いることが難しい.
したがって, やや冗長的ではあるが, `if` 文と先述の `is_TK_type`, `is_CK_type` 関数を用いて分岐して逆変換を行う. この際, どのパターンにも該当しない場合はエラーを返す.

```
if (is_type(ast, kind(alpha_1)) {
    unparse_alpha_1();
} else if(is_type(ast, kind(alpha_2))) {
    unparse_alpha_2();
}
...
  else if(is_type(ast, kind(alpha_k))) {
     unparse_alpha_k();
} else {
    unparse_error();
}
```

使用例（`unparse_translation_unit` より抜粋）

```
if (is_TK_type(ast->child[i + 2], ';')) {
  printf(";\n");
} else if (is_CK_type(ast->child[i + 2], CK_COMPOUND_STATEMENT)) {
  unparse_AST(ast->child[i + 2], depth);
  printf("\n\n");
```

```
} else {
  unparse_error(ast);
}
```

F. $\alpha : [\alpha_1]$ の逆変換（省略可能）

省略可能な構文は，`parse` と同様に `if` 文で条件をチェックして，対応する構文を逆変換するかどうかを判断する．

```
if (is_type(ast, kind(alpha_1))) {
    unparse_alpha_1();
}
```

使用例（`parse_declarator` より抜粋）

```
if (ast->num_child >= 3 && is_TK_type(ast->child[1], '(')) {
    check_TK_type(ast->child[2], ')');
    printf("()");
}
```

`unparse_error` 以外のエラーが起きると，それは予想外のバグであり，正しく実装できていないことになる．したがって，セグメンテーションフォルトを避けるため，こうした比較を行う際には，`num_child` を確認することで，範囲外のメモリにアクセスしないようにしている．

G. $\alpha : \alpha_1^*$ の逆変換（0回以上の繰り返し）

繰り返しは、`while` 文を使って逆変換を行う．

```
while (is_type(ast, kind(alpha_1))) {
    unparse_alpha_1();
}
```

使用例（`parse_translation_unit` より抜粋）

```
static void unparse_translation_unit(struct AST *ast, int depth) {
  int i = 0;
  while (i+3 < ast->num_child) {
    if(!is_CK_type(ast->child[i], CK_TYPE_SPECIFIER)) break;
    printf_ns(depth, "");
    unparse_AST(ast->child[i], depth + 1);
```

```
      unparse_AST(ast->child[i + 1], depth + 1);

      if (is_TK_type(ast->child[i + 2], ';')) {
        printf(";\n");
      } else if (is_CK_type(ast->child[i + 2], CK_COMPOUND_STATEMENT)) {
        unparse_AST(ast->child[i + 2], depth);
        printf("\n\n");
      } else {
        unparse_error(ast);
      }
      i += 3;
    }
  }
```

同様に, セグメンテーションフォルトを避けるため, インデックスが `num_child` を超えないことも `while` 文の条件式に含めている.

`while((i+3 < ast->num_child) && (is_CK_type(ast->child[i], CK_TYPE_SPECIFIER)) {...}` の様に記述しても良かったが, 条件式が長くなりすぎるため, 上記の様に分割して記述した.

以下では, 具体的な `unparse_*` 関数について、これらの逆変換をどのように用いているかを説明する. ただ, 規則になぞらえて機械的に記述しているだけなので, 上記の変換とは外れた処理を行っている箇所について重点的に説明を行う.
コードの貼り付けは行わないので, 添付したソースファイル `xcc-small.c` を参照してほしい.

H. `translation_unit`

対応文法:

```
translation_unit
  : ( type_specifier declarator ( ";" | compound_statement ))*
;
```

構文逆解析方針:

- 繰り返し構文を `while` 文を用いて逆解析している.
- `while` 文の各ループは, 関数宣言(`;`) あるいは関数定義 `{}` に対応しているため, はじめに, `printf_ns(depth, ")` によりインデントを行い, 各ループの終わりには, `\n` を出力して改行している. (関数定義の場合は, 読みやすさ向上のため二回改行を行っている.)

I. `type_specifier`

対応文法:

```
type_specifier
  : "void" | "char" | "int" | "long"
;
```

構文逆解析方針:

- 構文木 `*ast` の `ast_type` が上記のいずれかであれば, 対応する定数を出力する. その際, 定数は `lexeme` に格納されているので, `ast->child[0]->lexeme` を出力する.
- `type_specifier` の後は必ず `declarator` が出現するので, `int x` のようになるように, 型の後には空白を出力する.

J. `declarator`

対応文法:

```
declarator
  : IDENTIFIER [ "(" ")" ]
;
```

構文逆解析方針:

- 最初に識別子 `IDENTIFIER` を出力する. 実際の値は `lexeme` に格納されているので, 同様に `ast->child[0]->lexeme` を出力する.
- 続いて `"(" ")"` が現れるかどうかを `if` 文でチェック（省略可能構文に対応）.

K. `compound_statement`

対応文法:

```
compound_statement
  : "{" (type_specifier declarator ";")* ( statement )* "}"
;
```

構文逆解析方針:

- `"{"` と `"}"` に挟まれるブロック構文を逆解析している.
- まず, `{\n` を出力するが, この時ネストが一段深くなるので, `depth++` を行っている.

- 前半の各ループは, 局所変数定義である. まず, インデントを行い, 最後に `;\n` を出力して改行している.
- 後半の各ループは, 文である. まずインデントを行い, 最後に `\n` を出力して改行を行っている.
- 最後に, ネストが元に戻るため, `depth--` を行って, `}\n` を出力して閉じている.

L. `exp`

対応文法:

```
exp
  : primary [ "(" ")" ]
;
```

構文逆解析方針:

- `primary` を先に逆解析し、続いて `"(" ")"` の存在を確認して関数呼び出しのような形に対応.

M. `primary`

対応文法:

```
primary
  : INTEGER
  | CHARACTER
  | STRING
  | IDENTIFIER
  | "(" exp ")"
;
```

構文逆解析方針:

- 識別子や定数の場合はまとめて逆解析を行っている.
- `(exp)` の場合はあらかじめ `num_child` が `3` より大きいことを確認して, セグメンテーションフォルトを避けている.

N. `statement`

対応文法:

```
statement
  : IDENTIFIER ":"                                    // 注1: ラベル文
  | compound_statement
  | "if" "(" exp ")" statement [ "else" statement ] // 注2: else は省略可能
  | "while" "(" exp ")" statement
  | "goto" IDENTIFIER ";"
  | "return" [ exp ] ";"
  | [ exp ] ";"                                       // 空文または式文
;
```

構文逆解析方針:

- ラベル構文(注1) は, 二つめの `child` まで見て判断している.
- `if` 文と `while` 文は共通した構造を持っており, 今回は実装していないが `for` 文などその他の構文にも応用できるので以下の様な共通関数を作成した.
- `unparse_condition_statement`

```c
static void unparse_condition_statement(struct AST *ast, int depth) {
    check_TK_type(ast->child[1], '(');
    printf("(");
    check_CK_type(ast->child[2], CK_EXPRESSION);
    unparse_AST(ast->child[2], depth);
    check_TK_type(ast->child[3], ')');
    printf(")");
}
```

この関数は `"(" exp ")"` の様な条件式を逆解析する関数である. さらに抽象化するのであれば, `child_index` を引数で受け取り, `ast->child[child_index]` のようにした方がよいかもしれないが, 今回は上記の様な形式で記述した.

- `unparse_wrapped_block`

```c
static void unparse_wrapped_block(struct AST *ast, int depth) {
  check_CK_type(ast, CK_STATEMENT);
  if (is_CK_type(ast->child[0], CK_COMPOUND_STATEMENT)) {
    unparse_AST(ast, depth);
  } else {
    printf(" {\n");
    printf_ns(depth + 1, "");
    unparse_AST(ast, depth + 1);
    printf("\n");
    printf_ns(depth, "");
    printf("}");
  }
}
```

この関数は，`if` 文や `while` 文の中身である `statement` を逆解析する関数である．
今回の課題では，`if` 文，`if-else` 文の内部の `statement` は `{, }` をつけて出力するという指定だったが，`while` 文も共通した構造を持つため，`while` 文に関しても `{, }` をつけて出力した．
また，`statement` が `compound_statement` に展開する場合，`{}` が二重に出力されてしまうため，`ast->child[0]` が `CK_COMPOUND_STATEMENT` であるかを判定して処理している．

`if` 文と `while` 文に関して以下の様に解釈して逆解析を行っている．

```
"if" condition_statement wrapped_block ["else" wrapped_block]
"while" condition_statement wrapped_block
```

- `return` 文は，`return;` の場合は間に空白を入れないが，`exp` を返す場合は，`"return" + " " + exp` の様に空白を入れたいので，`printf(" ");` を行っている．
- `[exp] ";"` は，コード上では，`exp ";" | ";"` の様に解釈して逆解析を行った．

4. 拡張課題(XC)

拡張課題として `xcc-small-extend.c` を作成した．これは，XC-small言語ではなく，XC 言語を構文解析できるように `xcc-small.c` を修正したものになる．
課題の本筋ではないことと，`xcc-small` とほとんど同じ構造を持つため，異なる部分について簡単に説明を行う．
コードの貼り付けは行わないので，`xcc-small-extend.c` を参照してほしい．

A. `declarator`

対応文法

```
declarator : IDENTIFIER | "*" declarator | "(" declarator ")" |
    declarator "(" [ parameter_declaration ( "," parameter_declaration )* ] ")";
```

`declarator` は引数付きの関数を宣言できるようになっている．
`IDENTIFIER, *declarator, (declarator)` `xcc-smallc` と同様に記述できる．
しかし，`declarator` が展開先の一番初めにある場合，`director` が他の展開先と被ってしまうため，今までと同じように構文解析できない．つまり，上記の文法に含まれる左再帰を除去する必要がある．

そこで，`declarator "(" [ parameter_declaration ( "," parameter_declaration )* ] ")"` について以下のように考えた．

- `declarator` が `IDENTIFIER | "*" declarator | "(" declarator ")"` のとき，`(IDENTIFIER | "*" declarator | "(" declarator ")") "(" [ parameter_declaration ( "," parameter_declaration )* ] ")"`

- `declarator` が `(IDENTIFIER | "*" declarator | "(" declarator ")") "(" [ parameter_declaration ( "," parameter_declaration )* ] ")"` のとき，`(IDENTIFIER | "*" declarator | "(" declarator ")") ( "(" [ parameter_declaration ( "," parameter_declaration )* ] ")" )^2` となる．
- 上記の様に繰り返して，

```
declarator : (IDENTIFIER | "*" declarator | "(" declarator ")")
    ("(" [ parameter_declaration ( "," parameter_declaration )* ] ")")*;
```

の様に解釈することが出来る．

ただこのように解釈すると，導出できるものは確かに一緒になるが，構文木としてみた時に元の `declarator` の文法と多少異なる部分が生まれると考えた．
例えば，`main()` は 元の構文木であれば，`declarator->IDENTIFIER "(" ")"` のようになるはずであるが，左再帰をなくした文法だと，`IDENTIFIER "(" ")"` となる．つまり，`IDENTIFIER` と `"(", ")"` が同列に並んでしまう．

そこで，以下のような手法をとった．
まず，`(IDENTIFIER | "*" declarator | "(" declarator ")")` の解析は今まで同様に行う．このとき，`declarator->IDENTIFIER` のようになっている．
次に，繰り返し構文を解析することになるが，その解析結果をそのまま `add_AST` を用いて追加すれば，先ほどの懸念通り，`declarator-> IDENTIFIER "(" ")"` の様になってしまう．
そこで，新しく構文木 `dec` を作成し，既存の `ast` を `dec` の配下に配置することで解消している．これにより，一回目のループが終わると，`dec -> (declarator->IDENTIFIER "(" ")")` のようになる．そして，`ast` を `dec` に置き換える．これにより次のループでは，`declarator->IDENTIFIER "(" ")"` までがセットで `declarator` として `dec` の配下に配置される．つまり，このループが終わると，`dec -> (declarator -> (declarator->IDENTIFIER "(" )) "(" ")")` のような構造になる．これを繰り返す．
各ループの終わりでは，`ast` を `dec` に置き換えているので，繰り返し構文のループが終了した時，`ast` に構文解析の結果が格納されているので，`ast` を返して終了する．

`unparse_declarator` についてもここで説明を行う．
左再帰が含まれるが，対応文法通りに構文木を構築したため選択型の構文として扱い，`else if (is_CK_type(ast->child[0], CK_DECLARATOR)` によって条件分岐を行っている．

B. `expression`

- 対応する文法

```
expression : IDENTIFIER | INTEGER | CHARACTER | STRING
| unary_operator expression
| expression binary_operator expression
```

```
  | expression "(" [ argument_expression_list ] ")"
  | "(" expression ")" ;
```

`operator` を含んだ式や, 引数を持つ関数に対応した文法となっている. `unary_operator` `expression` などXC-Small と異なる文法も含まれるが, 使用する構文は変わっていないため, 説明は割愛する.
また, `expression binary_operator expression | expression "(" [ argument_expression_list ] ")"` の様に二種類の左再帰が含まれるが, A. `declarator` と同様に左再帰を除去し, 構文木としても等価になるように, 新しく構文木 `exp` を用いるなどして対処した.
`unparse_exp` についても `declarator` と同様に左再帰を対処した.

その他, `parameter_declaration, unary_operator, binry_operator, argument_expression_list` など, XC-small言語には含まれなかった文法も存在するが, すでに説明した構文で構成されているので, 説明は省略する.
また, 文法の変更に伴って, `exprimary` など `first` が変わった箇所があるため, 対応する箇所の修正は行った.

5. テスト

テストは, 配布された `test1/t*.c, test2/t*.c` で行った.
`xcc-small.c` については, `test1` は正しく実装されていることを確認し, `test2` は文法が対応していないためエラーを返すことを確認した.
`xcc-small-extend.c` については, `test1` は正しく実装をされていることを確認し, `test2` は `t3.c, t4.c` についてのみエラーを確認した.

- `t3.c` については, ぶらぶら `else` が存在するため, エラーが起きた.
- `t4.c` については, `compound_statement` が, 文 `statement` のあとに局所変数定義 `type_specifier declarator ";"` を記述することを許容していないため, エラーが起きた.

また, 各テストケースを手入力で試すのは面倒なため, 以下の様な `Makefile` を作成して `make` を実行することで, 一括で全てのテストケースを試せるようにした. (`Makefile` は添付した資料に含まれる.)

```
CC = gcc
CFLAGS = -g

SRC = xcc-small_extend.c
BIN = a.out

TEST1 = $(wildcard test1/t[1-6].c)
TEST2 = $(wildcard test2/t[1-5].c)
TESTS = $(TEST1) $(TEST2)
```

```
all: $(BIN) run-tests

$(BIN): $(SRC)
    $(CC) $(CFLAGS) $(SRC)


run-tests: $(BIN)
    @for file in $(TESTS); do \
        echo "=== Running $$file ==="; \
        ./$(BIN) $$file; \
        echo ""; \
    done

clean:
    rm -f $(BIN)
```

上記の `Makefile` では `SRC = xcc-small_extend.c` となっているため, この箇所を変更することで, 任意のファイルを試すことが出来る.

6. 議論

今回構文解析のコードを記述したが, 本レポートの構成の様に, 各構文に対応する処理を考えそれを組み合わせることで, 機械的に構文解析機を作成できる. そこで, EBNFで記述された文法を簡単な構文解析を行うことで, 自動的に, 構文解析機を作成するコンパイラ・コンパイラを作成できるのではないかと考えた. しかし, `statement` のようなLL(1) 文法に反する文法が含まれると, 実装が難しくなることが予想される.

また, 上記の様な自動的に構文解析器を作成することが出来なくとも, 以下のようなことをすれば, 今回のコードの保守性をあげられると考えられる.

- `first, follow` を分離する.
  現在のコードでは, `case '(':` の様に, `declarator` をべた書きしているので, 拡張課題に取り組む際に `first` が変わると対応する箇所を探して変更する必要があった. (現に変更しきれいていない箇所があるかもしれない.) なので, `first, follow` を分離して記述することで, 一か所を変更することで, 自動的に `director` を計算して, `case director(A, B)` の様な記述が出来ると保守性の向上につながると考えた.
- `is_TK_type` や `is_CK_type` などでOR機能を実装する.
  `is_TK_type` などを可変長引数を持てるようにして, `ast->ast_type` がいずれかの `kind` であれば `true` を返す様な関数にすれば, 捕手性が向上すると考えた. 例えば以下のようなコードがきれいに書けるようになる.

```
if (is_TK_type(ast->child[0], '=') || is_TK_type(ast->child[0], TK_OP_OR) ||
      is_TK_type(ast->child[0], TK_OP_AND) ||
      is_TK_type(ast->child[0], TK_OP_EQ),
  is_TK_type(ast->child[0], '<') || is_TK_type(ast->child[0], '+') ||
      is_TK_type(ast->child[0], '*') ||
```

```
      is_TK_type(ast->child[0], '/')) {
printf(" %s ", ast->child[0]->lexeme);
return;
}

//実装後
if (is_TK_type(ast->child[0], 8, '=', TK_OP_OR, TK_OP_AND, TK_OP_EQ, '<', '+',
'*', '/')) {
    printf(" %s ", ast->child[0]->lexeme);
return;
}
```

今回は時間の都合上ここまでとなったが, 上記の様な実装を行えば保守性が向上すると考えらえる.

ここからはただの感想になるが, エディタの自動フォーマッタや色付けは, 構文解析までで実装できそうだと思った. また, エディタ上でのエラー表示は, 意味解析まで行っているのかわからないが, 構文解析までででもかなり機能しそうだと思った.