Design and Development Methodology of C Language Markup Converter Based on XML

Takuya Toita*, Kazuki Yoshida** and Hirohide Haga*

(Received April 18, 2014)

The software testing support environment IST (Integrated Software Testing Environment) is an integrated environment for software testing support. In IST, source codes of programming languages are expressed in XML form, and stored in a program database. In order to construct the database easily, the software for converting a source code into XML form is required. The converter for Java language has already existed. However about C languege, there is no converter. This paper describes the design of the markup language CML (C Markup Language) for C language and development of the converter.

Key words : software testing, converter, C-languege, XML

キーワード : ソフトウェアテスト，コンバータ，C言語，XML

XMLに基づくC言語用のマークアップコンバータの設計と開発手法

砥板卓矢，吉田万輝，芳賀博英

1. はじめに

本論文では，C 言語のソースコードを XML 形式で表現するための変換系 (converter：コンバータ) の設計と開発について述べる．我々の研究室で開発しているソフトウェアテスト支援環境 IST[1)](Integrated Software Testing Environment) では，プログラミング言語のソースコードを XML 形式で表現し，データベースに格納している．このデータベースの構築を容易にするために，ソースコードを XML 形式に変換するための変換系が必要である．Java 言語についてはすでに JavaML[2)] と JJmlt[3)] という DTD とソフトウェアが存在しているが，C 言語については我々が必要としている粒度のものは存在しない．

そこで本研究では，C 言語用のマークアップ言語 CML(C Markup Language) の設計と処理系の開発を行った．変換方法は次の通りに設計を行った．構文規則は C 言語の規格として ISO 及び ANSI が定めた ANSI-C[4)] を利用した．この構文規則に基づいて，構文解析を yacc[5)](Yet Another Compiler Compiler)，字句解析を lex[5)](lexical analyser generator) で行う．この構文規則から抽象構文木を作成し，抽象構文木を走査することにより，C ソースコードを XML 形式に変換した．

本論文の構成は以下の通りである．2 章では本研究において必要となる CML，構文木それぞれの概要を述べる．3 章では設計と開発について述べる．開発対象は CML のコンバータである．

* Graduate School of Science and Engineering, Doshisha University, Kyoto
Telephone:+81-774-65-6979, E-mail:ttoita@ishss10.doshisha.ac.jp
** COOKPAD, Tokyo, Telephone:+81-3-6408-6143, E-mail:kzk.yshd@gmail.com

そして，実装機能としてC言語の機能の内，本研究において開発した機能を述べる．4章ではCMLの評価を行う．そして5章で結論を述べる．

2. CMLと木構造

2.1 C言語用マークアップ言語CMLの概要

CMLとはC言語のソースコードから構文情報を抽出し，その構文情報をXML形式で表現したマークアップ言語の一種である．CML及びその処理系は，我々の研究室で開発しているソフトウェア開発環境ISTのフロントエンドを担当するものであり，ソースコードをCMLに変換しておくことで，バックエンドの各ツール群がソースコードの情報を使うときに毎回ツールごとに構文解析をする必要がなくなる．

2.1.1 XML

XML[6](Extensible Markup Language / 拡張可能なマークアップ言語) とは1998年にW3C(World Wide Web Consortium) によって勧告されたマークアップ言語である．マークアップ言語ではタグと呼ばれる装飾によって情報の意味や構造を定義する．HTMLなどのタグが固定化されているマークアップ言語と比べて，XMLはタグ自体を自由に設計することができるのでマークアップ言語を作成するためのメタ言語としても用いられる．またXMLはデータのシリアライズや異なるソフトウェア間の通信が主な目的であるが，テキスト形式で表現されることや設計段階で読みやすさが考慮されているため，人にとっても読みやすい形式となっている．Source-code 1に個人データを表すXMLを表現した例を示す．

Source-code 1. Example of XML code.

```
1 <?xml version="1.0"?>
2 <person>
3     <name>Nizima Yae</name>
4     <birth>11/3</birth>
5     <country>Japan</country>
6 </person>
```

XMLは $< word >$ の形式の開始タグと $< /word >$ の形式の終了タグがあり，対になっている開始タグと終了タグで囲まれた要素は囲んでいる要素の子要素になる．XMLではこのように親子関係を表現することで構造を定義する．Source-code 1ではpersonという要素にname，birth，countryの要素が子要素として含まれることを表現している．

2.2 ソースプログラムの木構造表現

木構造は親子関係を持つ階層構造である．木構造は，節 (ノード) と枝 (エッジ) により構成され，一つのノードが子要素を持ち，その子要素が更に子要素を持つことにより階層構造を表す．親要素を持たないノードを根 (ルート) といい，子要素を持たないノードを葉 (リーフ) と呼ぶ．

2.2.1 構文木

構文木とは入力記号列を文法規則に従って構造解析し，それを木構造で表現したものである．例えば，次のような構文規則を定義する．

$$
\begin{array}{lcl}
E & \rightarrow & E \ = \ E \\
E & \rightarrow & E \ + \ E \\
E & \rightarrow & E \ * \ E \\
E & \rightarrow & x \mid y \mid z \mid 3
\end{array}
$$

この文法規則に従って，次の式を構造解析する．

$$x = y + z * 3 \tag{1}$$

式(1)の解析結果をFig. 1に示す．これが式(1)の構文木となる．

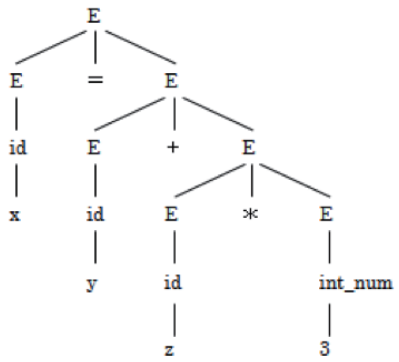

Fig. 1. Syntax tree of “x=y+z*3”.

2.2.2 抽象構文木

抽象構文木[7](AST：Abstract Syntax Tree) とは，前述した構文木から言語の意味に関して冗長な部分を取り除くことにより，抽象化した木構造のデータ構造である．理論的には分岐ノードに演算子を，終端ノー

ド (リーフ) に変数や定数を対応させたものである．更に，グループ化のために用いたカッコ，文字列であることを示すために用いたクォーテーションマークなども言語の意味に関して冗長な部分であり，AST では省略する．Fig. 1 に示した構文木を抽象構文木に変換した結果を Fig. 2 に示す．

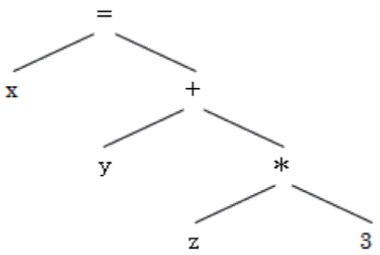

Fig. 2. AST of “x=y+z*3”.

2.2.3 構文木の走査

木の走査 (traverse) とは，ツリーにある全ノードを一回ずつ体系的にめぐっていく処理である．木の走査には，大きく分けてルートから階層の低い点へ走査を進める幅優先探索 (breadth-first search) と終端ノードに到達することを優先して木を走査していく深さ優先探索 (depth-first search) の二つがある．また，深さ優先探索には行きがけ順 (pre-order)，通りがけ順 (in-order)，帰りがけ順 (post-order) の三つの方式がある．以下に例を示す．

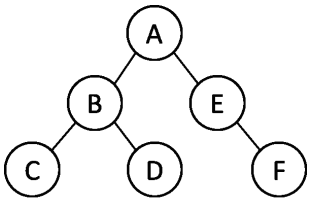

Fig. 3. Example of tree structure.

Fig. 3 の木構造を breadth-first search で探索すると ABECDF という順になる．また pre-order で探索すると ABCDEF，in-order では CBDAEF，post-order では CDBFEA の順になる．

3. CML コンバータの設計と開発

CML の開発はコンパイラの処理系を参考に行った．多くのコンパイラの処理系は字句解析，構文解析，意味解析，機械語出力の順で処理が行われる [5] が，CML の処理系も似たような処理系を備えている．C 言語のソースコードを字句解析してトークンに分割する．分割されたトークンの集合に構文解析を行い，最終的に CML の形式で出力している．CML の処理系では C 言語として動作するプログラムを対象としているので意味解析は行わない．また CML は字句解析処理を lex(lexical analyzer generator), 構文解析処理を yacc(Yet Another Compiler Compiler) というコンパイラ生成系によって開発している．

3.1 開発手法

3.1.1 lex

lex とは POSIX 標準になっている字句解析プログラムを生成するための字句解析生成系 [5] である．字句解析とはソースコードを解析して意味を持つ最小の要素 (トークン) に分解する処理である．Fig. 4 に字句解析の例を示す．

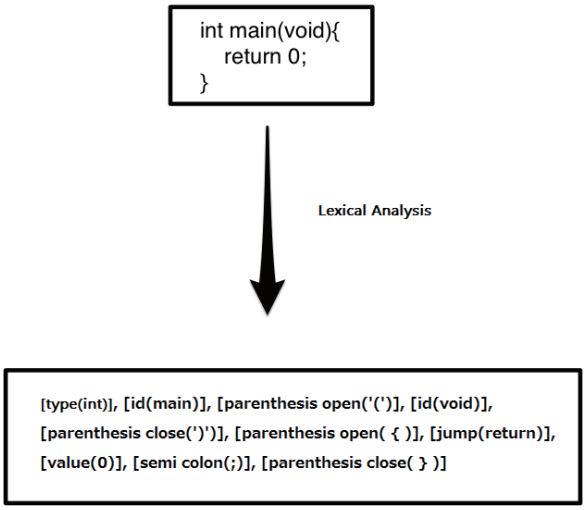

Fig. 4. Lexical analysis.

lex が字句解析プログラムを作るために，トークンの種類とそれぞれが検出された時に行う処理を予め記述する必要がある．記述方法の一例を Source-code 2 に示す．

Source-code 2. Definifion of tokens in lex.

```
1 "signed" { return SIGNED; }
2 "unsigned" { return UNSIGNED; }
3 "const"  { return CONST; }
4 [1−9]{DIG}*{LU}? { return
      DECIMAL_CONSTANT; }
```

Source-code 2 の 1 行目は “signed” という文字列が読み込まれた場合に SIGNED というトークンを返すことを表している．2,3 行目も同様に “unsigned” という文字列が読み込まれた場合 UNSIGNED トークンを返し “const” という文字列が読み込まれた場合に CONST トークンを返す処理が記述されている．また 4 行目のように文字列の判定には正規表現を用いることが可能で，この場合は数値が読み込まれた場合に DECIMAL_CONSTANT トークンを返すことを示している．

3.1.2 yacc

yacc とは構文規則をもとに構文解析プログラムを生成するためのプログラム生成系 [5] である．構文解析とは字句解析によって分割されたトークン列を順番に読み込み，どのような構文なのか解析する処理である．実際の処理は lex と yacc の処理は同時に行われており，yacc が生成した yyparse() が，構文解析を進めトークンが必要になるたびに，lex で生成された字句解析関数を call してトークンを切りださせる仕組みをとっている．また yacc では構文規則を BNF(バッカスナウア記法) に似た文法で記述する．Source-code 3 に構文規則の記述例を示す．

Source-code 3. Example of syntax rule in yacc.

```
1 TRANSLATION_UNIT
2     : EXTERNAL_DECLARATION   { }
3     | TRANSLATION_UNIT
          EXTERNAL_DECLARATION { }
4     ;
```

Source-code 3 の記述例においては，EXTERNAL_DECLARATION であるか，もしくは TRANSLATION_UNIT，EXTERNAL_DECLARATION というトークンの並びがあった場合，それらが TRANSLATION_UNIT に還元されることを表している．また還元元の集合の右側の中括弧には，還元が行われたタイミングで実行される処理を記述できる．CML の実装はこの位置に構文情報を保存する構造体を生成するコードを埋め込むことでプログラムを木構造に一旦変換し XML を生成している．

3.2 設計手法

3.2.1 CML の基本的な実装手法

C のソースコードから CML への変換は，yacc に構文情報を格納する構造体を生成する処理を埋め込むことで実現している．Source-code 4 に構文情報を格納する構造体の宣言を示す．

Source-code 4. Structure declaration for syntax information.

```
1 typedef struct Node{
2         char name[100];
3         char attr[100];
4         int type;
5         int children;
6         struct Node *child1, *child2, *child3, *
              child4;
7 }Node, *NP;
```

構文情報を格納する Node 構造体は，タグ名と属性情報の他に最大で 4 つの子要素を保持できる仕組みをとっている．CML は ANSI-C の構文規則に基づいて設計されており，Node 構造体も ANSI-C の文法規則に対応できる形に設計されている．Source-code 5 に Node 構造体を生成する Make 関数群を示す．

Source-code 5. Function declaration for Node.

```
NP MakeLeaf(char *name, char *attr, int type
    );
NP MakeN1(char *name, NP child1);
NP MakeN2(char *name, NP child1, NP child2);
NP MakeN3(char *name, NP child1, NP child2,
    NP child3);
NP MakeN4(char *name, NP child1, NP child2,
    NP child3, NP child4);
NP MakeN12(char *name, char *attr, NP
    child1);
NP MakeN22(char *name, char *attr, NP
    child1, NP child2);
NP MakeN32(char *name, char *attr, NP
    child1, NP child2, NP child3);
NP MakeN42(char *name, char *attr, NP
    child1, NP child2, NP child3, NP child4);
```

MakeLeaf 関数は，末端ノードを生成するための生成関数である．また MakeN1 から MakeN4 までが，属性情報を持たないノードの子ノード数別の生成関数，Make12 から Make42 が，属性情報を持つ場合の子ノード数別の生成関数である．Source-code 6 に実装例として MakeN42 の実装を示す．

Source-code 6. Implementation of MakeN42.

```
1  NP MakeN42(char *name, char *attr, NP
       child1, NP child2, NP child3, NP child4){
2          NP np = (NP)malloc(sizeof(Node));
3          strcpy(np->name, name);
4          strcpy(np->attr, attr);
5          np->type = 0;
6          np->children = 4;
7          np->child1 = child1;
8          np->child2 = child2;
9          np->child3 = child3;
10         np->child4 = child4;
11         return np;
12 }
```

このように Source-code 5 に示した関数を使って，yacc 内で Node 構造体を生成している．Source-code 7 に実際に Node 構造体を生成している yacc のプログラムの一部を示す．

Source-code 7. Creating structure of Node.

```
1  INITIALIZER
2  : ASSIGNMENT_EXPRESSION  { $$ =
       MakeN1("INITIALIZER", $1); }
3  | '{' INITIALIZER_LIST '}'  { $$ = MakeN1("
       INITIALIZER", $2); }
4  | '{' INITIALIZER_LIST ',' '}' { $$ =
       MakeN1("INITIALIZER", $2); }
5
6  INITIALIZER_LIST
7  : INITIALIZER    { $$ = MakeN1("
       INITIALIZER_LIST", $1); }
8  | INITIALIZER_LIST ',' INITIALIZER  { $$ =
       MakeN2("INITIALIZER_LIST", $1, $3); }
9  ;
```

yacc のプログラム内には特殊な変数が用意されている．$1 のような，$のあとに数値がついた変数は構成要素の意味値を表している．つまり，Source-code 7 の 2 行目の$1 は ASSIGNMENT_EXPRESSION を示しており，3 行目の$2 は INITIALIZER_LIST を示している．また$$は，還元した結果の意味値を格納する変数であり$$に値を格納することで次の還元処理に値を引き渡すことができる．例えば，ASSIGNMENT_EXPRESSION が Source-code 7 の 2 行目の規則で INITIALIZER に還元された場合，このタイミングで MakeN1 関数が実行され INITIALIZER というタグ名の Node 構造体が$$に格納される．次に，還元された INITIALIZER は 7 行目の規則で INITIALIZER_LIST にさらに還元されるが，そのタイミングで実行される MakeN1(“INITIALIZER_LIST”, $1) という関数命令の第二引数の$1 には，先ほど生成した INITIALIZER というタグ名の Node 構造体が格納されている．このような値の受け渡しによって構文情報が木構造の構造体に変換される．

3.2.2 タグの生成

タグは抽象構文木を pre-order 順でたどり，生成していく．Source-code 8 に基本的な二分木のタグ生成プログラムを示す．

Source-code 8. Tag generation program.

```
1  void genTag(NP np){
2      if(np == NULL) return;
3      if(np->children == 0) leafTag(np);
4      else{
5          preTag(np);
6          genTag(np->child1);/*左のノード*/
7          genTag(np->child2);/*右のノード*/
8          postTag(np);
9      }
10 }
```

調べるノードがない場合は return，ある場合はタグを生成する．preTag() では開きタグを生成し，postTag() は閉じタグを生成している．このタグ生成プログラムは再帰的な構造である．調べるノードに子ノードがあれば，一つ目の genTag() で左のノードのタグを生成し，二つ目では右のノードのタグを生成する．Fig. 5 をこのプログラムでタグ生成すると Source-code 9 のようになる．

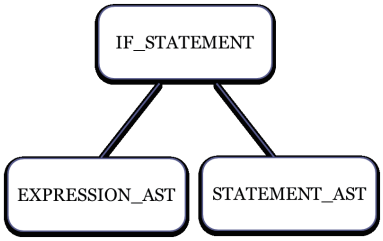

Fig. 5. Example of AST.

Source-code 9. Creating tags of Fig. 5.

```
1  <if_statement>
2    <expression>
3      ...
4    </expression>
5    <statement>
6      ...
7    </statement>
8  </if_statement>
```

3.2.3 冗長な構文木の修正

構文木は ANSI-C をという文法規則をもとに作成している．しかし，ANSI-C は一般的な C 言語の構文に対応するために，文法が詳細に定義されており，木構造で表現した時に無意味に木の高さが高くなる．その

ためこの文法規則をそのまま構文木作成に使用すると，すべての文法にタグ付けを行うため，非常に膨大な量となる．Source-code 10 に C 言語のサンプルプログラムである return 文を，Source-code 11 に ANSI-C の全文法規則に従って，すべてにタグ付けを行ったプログラムを示す．

Source-code 10. Sample program of “return”.

```
1  int main(void){
2    return 0;
3  }
```

Source-code 11. Creating tag of “return”.

```
1  <external_declaration>
2    <function_definition>
3      <declaration_specifiers>
4        <type_specifiers>
5          <type name= ”int ”/>
6        </type_specifiers>
7      </declaration_specifiers>
8    </function_definition>
9  </external_declaration>
10 <declarator>
11   <direct_declarator>
12     <var name= ”main ” >
13   </direct_declarator>
14   <parameter_type_list>
15     <parameter_list>
16       <parameter_declaration>
17         <declaration_specifiers>
18           <type_specifiers>
19             <type name= ”int ”/>
20           </type_specifiers>
21         </declaration_specifiers>
22       </parameter_declaration>
23     </parameter_list>
24   </parameter_type_list>
25 </declarator>
26 …(以下省略)
```

以上より文法規則の全てにタグ付けを行うと膨大な量となることがわかる．したがって，冗長な構文部分を判別しまとめる必要がある．Fig. 6 に Source-code 10 の構文解析木を示す．

本論文で提案する手法では，ANSI-C の文法規則から作成した構文解析木の冗長な部分を削除していき CML の構文解析木を作成していく．Fig. 7 に ANSI-C の構文規則に忠実に従って“expression”から文字“x”を出す場合の構文解析木を示す．

Fig. 6 と Fig. 7 から，“return 0;” の“0”のような数字や文字，文字列などを構文解析木で表現する場合，構文解析木の“expression”から“primary_expression”までの文法規則は同じになることがわかる．したがってこの場合，“expression”から“primary_expression”

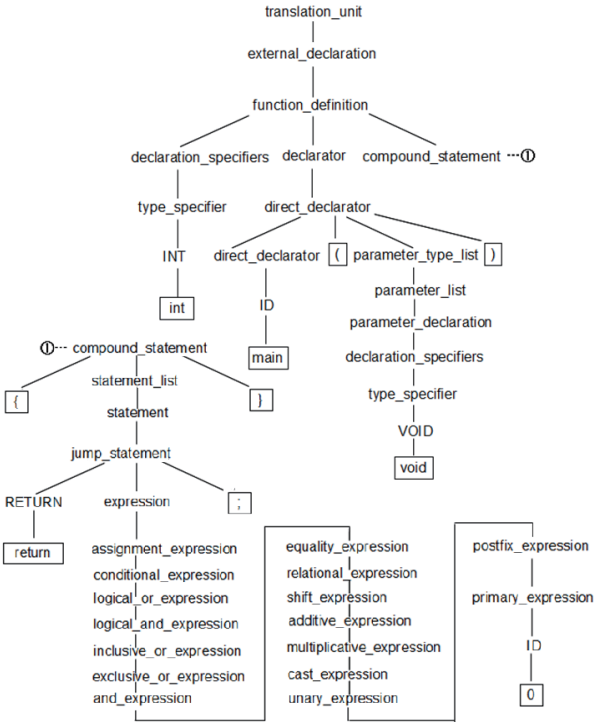

Fig. 6. Original syntax tree.

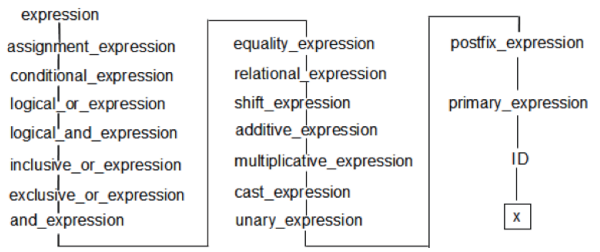

Fig. 7. Syntax tree of character.

までの文法規則は冗長であると判断し，省略するように修正を加えた．このようにいくつかのプログラム構文解析木を作成し，構文木の比較をして冗長と判断した部分に修正を加えた．Fig. 8 に Fig. 6 に対して修正を加えた構文解析木を示す．また Source-code 12 に省略等をし修正を加えたタグ名を示す．

Source-code 12. Modified tags.

```
ADDITIVE_EXPRESSION
AND_EXPRESSION
ASSIGNMENT_EXPRESSION
CAST_EXPRESSION
CONDITIONAL_EXPRESSION
CONSTANT
CONSTANT_EXPRESSION
DECLARATION
DECLARATION_SPECIFIERS
DECLARATOR
```

DIRECT_DECLARATOR
EXCLUSIVE_OR_EXPRESSION
EXPRESSION
EXPRESSION_STATEMENT
GROUP_SEU_IDENTIFIER
INCLUSIVE_OR_EXPRESSION
LOGICAL_EXPRESSION
MULTIPLICATIVE_EXPRESSION
OPTIONAL_EXPRESSION
POSTFIX_EXPRESSION
RELATIONAL_EXPRESSION
SHIFT_EXPRESSION
STATEMENT
STORAGE_CLASS_SPECIFIER
STRUCT_OR_UNION
TRANSLATION_UNIT
TYPE_SPECIFIER
TYPE_QUALIFIER
UNARY_EXPRESSION

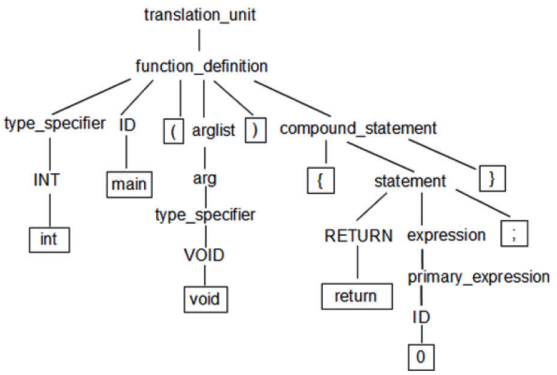

Fig. 8. Simplified syntax tree.

3.2.4 変数定義の拡張

C 言語のソースコードにはポインタや構造体内の変数にアクセスする文法が用意されているが，それらのソースコードを ANSI-C の文法規則に基づいて変換すると非常に扱いづらい冗長なタグが生成される．例えば Source-code 13 を ANSI-C の文法規則にしたがって生成した XML が Source-code 14 である．

Source-code 13. Sample code containing pointer or structure.

```
1 **node;
2 node->name;
```

Source-code 14. XML before improvement.

```
1 <UNARY_OPERATOR kind = "*"/>
2 <UNARY_EXPRESSION>
3     <UNARY_OPERATOR kind = "*"/>
4     <PRIMARY_EXPRESSION identifier = "
          node"/>
5 </UNARY_EXPRESSION>
6 <POSTFIX_EXPRESSION op = "->" identifier
      = "name">
7     <PRIMARY_EXPRESSION identifier = "
          node"/>
8 </POSTFIX_EXPRESSION >
```

この問題を解決するために，変数トークンである Identifier の定義を拡張した．本来の C 言語の変数として扱う文字列は [a-zA-Z_][0-9a-zA-Z_]*が正しい定義であるが，ポインタやアクセス演算子を包含する形式に変更した．

Source-code 15. Regular expression of Identifier.

```
1 // 変更前の表現
2 [a−zA−Z\_][0−9a−zA−Z\_]*
3 // 変更後の表現
4 \**[a−zA−Z_][0−9a−zA−Z_]*(\[[0−9a−zA−Z_
      ]\])*((\.|\−>)[a−zA−Z_][0−9a−zA−Z_]*)*
      {return Identifier}
```

Identifier の定義を変更したことにより，Source-code 16 のように XML の構造がシンプルになり扱いやすくなった．

Source-code 16. XML after improvement.

```
1 <PRIMARY_EXPRESSION identifier = "**node
      "/>
2 <PRIMARY_EXPRESSION identifier = "node->
      name"/>
```

3.2.5 typedef の対応

C 言語には typedef という文法があり，プログラム内で新たに型を定義することができる．Source-code 17 に C 言語の typedef の文法の例を示す．

Source-code 17. Example of typedef.

```
1  // (1)既存の型にtypedefを用いた例
2  typedef unsigned int unit;
3
4  // (2)構造体の宣言にtypedefを用いた例
5  typedef struct Node{
6      char name[100];
7      char attr[100];
8      int type;
9      int children;
10     struct Node *child1, *child2, *child3, *
           child4;
11 }Node, *NP;
```

Source-code 17 の一つ目の例の場合は符号なし int 型を unit という名前で利用することが出来るようになり，２つ目の構造体の例は “struct Node” という型を Node もしくは*NP というデータ型名で利用できるようになる．このように typedef の文法は非常に有用だが，字句解析においては変数と新たに定義した型の区別が，文字列上はできないため扱いが難しい．もち

ろん ANSI-C の文法規則において typedef を表すトークン TYPE_DEF を返すべきところを変数を表すトークン IDENTIFIER を返してしまうと文法エラーになり，また型全般を表す TYPE_SPECIFIER に IDENTIFIER を還元できる文法規則を加えると文法規則の衝突が起こるので，type_def の文法を扱うためには，変数と新たに定義された型を区別することが必須である．

この問題を解決するために，lex 内にフラグ変数を複数用意することで文脈情報を保持し，新たに用意した名前表に typedef で定義された型名を格納する実装を行った．Source-code 17 の例 (1) の場合，既存の型に typedef を用いた時に新たに定義された型名は uint なので,unit を名前表に格納する必要がある．uint の出現位置をトークンの並びで定義すると，“typedef” を読み込み型名を読み込んだ次のトークンであることがわかる．そこで typedef を検出したことを示すフラグ typedefTokenFlag と，既存の型を読み込んだフラグ typedefNonStructIdentifierFlag を用意し，両方のフラグが true の状態で Identifier トークンが読まれた場合にその文字列を typedef 用の名前テーブルに格納する処理を実装した．また例の (2) の構造体の宣言に typedef を用いた場合は，新たに定義された型名は末尾の Node と NP なので，これらを名前表に格納する必要がある．この場合も新たに定義された型の出現位置をトークンの並びで定義すると，“typedef” を読み込み “{”(開き中括弧) を読み込み “}”(閉じ中括弧) を読み込んだ次のトークンから，“;” までに出現した Identifier トークンであることがわかる．そこで新たに “{”(開き中括弧) の検出用のフラグ変数 typedefOpenBracketFlag と “}”(閉じ中括弧) 検出用のフラグ変数 typedefCloseBracketFlag を用意した．これらのフラグを利用した型名の名前表への登録と参照の処理を Source-code 18 に示す．

Source-code 18. Processing when lex detects Identifier token.

```
1  \**[a-zA-Z_][0-9a-zA-Z_]*(\[[0-9a-zA-Z_
       ]\])*((\.|\->)[a-zA-Z_][0-9a-zA-Z_]*)*
       {
2      yylval.Name = yytext;
3      strcpy(tmptext, yytext);
4
5      // typedef struct {} [rename_identifier]
         pattern
6      if( typedefTokenFlag==1 &&
         typedefCloseBracketFlag==1 ){
7          strcpy(tydefName[typedefCounter],
             yytext);
8          typedefCounter++;
9          return IDENTIFIER;
10     }
11
12     // typedef [identifier] [renameIdentifier]
         pattern
13     if( typedefTokenFlag==1 &&
         typedefNonStructIdentifierFlag==1 ){
14         strcpy(tydefName[typedefCounter],
             yytext);
15         typedefCounter++;
16         typedefNonStructIdentifierFlag = 0;
17         typedefTokenFlag=0;
18         return IDENTIFIER;
19     }
20
21     for( iter=0; iter<=typedefCounter; iter++
         ){
22         strcpy(NameWithStar, "*");
23         strcat(NameWithStar,yytext);
24         strcpy(NameWithOutStar, "*");
25         strcat(NameWithOutStar,tydefName[iter
             ]);
26         if( !strcmp(yytext, tydefName[iter]) ||
27             !strcmp(NameWithStar, tydefName[
                 iter]) ||
28             !strcmp(yytext, NameWithOutStar)
29           ){
30             strcpy(structtmptext, yytext);
31             return TYPEDEF_NAME;
32         }
33     }
34
35     return IDENTIFIER;
36 }
```

Source-code 18 の 6 行目から 10 行目で，構造体に typedef が利用された場合に新たに定義された型を名前表に登録する処理をしており，13 行目から 19 行目で，既存の型に typedef が定義された場合に新たに定義された型を名前表に登録する処理を行っている．また，21 行目から 33 行目では Identifier が読み込まれた際に一度名前表を参照する処理を行っており，名前表内に処理中の文字列が検出された場合，“TYPEDEF NAME” トークンを返す処理を行っている．また検出されなかった場合は 35 行目で Identifier トークンを返す．

4. 評価実験

作成した CML コンバータの評価実験を行った．評価は，次の 3 つの文法においてソースコードの階層構造を表現できているか確認する．

・宣言部分

・typedef

・if-else 文と switch 文

4.1 実験内容

上記の評価内容が含まれているソースコードを，作成した CML コンバータで XML へ変換した．以下に評価するソースコードを示す．

Source-code 19. Sample code of “declaration”.

```
1 double b = 3.0; //小数
2 int d[] = {1,2,3,4,5}; //配列の初期値代入
3 char *str; //ポインタ
```

Source-code 20. Sample code of “typedef”.

```
1 typedef unsigned int uint;
2 uint e;
```

Source-code 21. Sample code of “conditional branch”.

```
1  if (a<b) {
2      a = 0;
3  } else if(a==b){
4      a = 1;
5  } else{
6      a = 2;
7  }
8  switch (a) {
9      case 1:
10         x = 0;
11         break;
12     default:
13         x = 1;
14         break;
15 }
```

4.2 実験結果

以下に Source-code 19 から 21 のソースコードを作成して，それらを CML に変換した．それぞれが適切に文法構造を表現できているか検証した．

4.2.1 宣言部分

Source-code 22. CML of source-code 19.

```
1  <INIT_DECLARATOR>
2      <DECLARATION type = "double"
           identifier = "b" />
3      <INITIALIZER>
4          <PRIMARY_EXPRESSION>
5              <CONSTANT kind = "
                   floating_point_constant"
                   value = "3.0"/>
6          </PRIMARY_EXPRESSION>
7      </INITIALIZER>
8  </INIT_DECLARATOR>
9  <INIT_DECLARATOR>
10     <DECLARATION isarray="true" type = "
           int" identifier = "d" />
11     <INITIALIZER>
12         <INITIALIZER>
13             <PRIMARY_EXPRESSION>
14                 <INTEGER_CONSTANT kind
                       = "decimal_constant"
                       value = "1"/>
15             </PRIMARY_EXPRESSION>
16         </INITIALIZER>
17         <INITIALIZER>
18             <PRIMARY_EXPRESSION>
19                 <INTEGER_CONSTANT kind
                       = "decimal_constant"
                       value = "2"/>
20             </PRIMARY_EXPRESSION>
21         </INITIALIZER>
22         <INITIALIZER>
23             <PRIMARY_EXPRESSION>
24                 <INTEGER_CONSTANT kind
                       = "decimal_constant"
                       value = "3"/>
25             </PRIMARY_EXPRESSION>
26         </INITIALIZER>
27         <INITIALIZER>
28             <PRIMARY_EXPRESSION>
29                 <INTEGER_CONSTANT kind
                       = "decimal_constant"
                       value = "4"/>
30             </PRIMARY_EXPRESSION>
31         </INITIALIZER>
32         <INITIALIZER>
33             <PRIMARY_EXPRESSION>
34                 <INTEGER_CONSTANT kind
                       = "decimal_constant"
                       value = "5"/>
35             </PRIMARY_EXPRESSION>
36         </INITIALIZER>
37     </INITIALIZER>
38 </INIT_DECLARATOR>
39 <INIT_DECLARATOR>
40     <DECLARATION type = "char" identifier
           = "*str" />
41 </INIT_DECLARATOR>
```

Source-code 22 の 1 行目から 7 行目までは，Source-code 19 の 1 行目の double 型の宣言部分を表現している．Source-code 22 の 1 行目の INIT_DECLARATOR タグが変数宣言及び初期値代入を行うことを表している．2 行目の DECLARATION タグは変数宣言を表しており，type 属性には型名，value 属性には変数名に記述されている．3 行目からの 7 行目の PRIMARY_EXPRESSION は式表現を表しており，これが初期値代入にあたる．5 行目の CONSTANT タグは数値を表しており，kind 属性には数値の種類，value 属性には値が記述されている．

Source-code 22 の 9 行目から 38 行目は，Source-code 19 の 2 行目を表現している．Source-code 22 の 10 行目のように，配列の宣言の場合には，変数宣言を表す DECLARATION タグに isarray 属性が追加さ

れる．また変数代入が複数ある場合には，11 行目から 36 行目のように INITIALIZER タグ内に値が列挙される．

また Source-code 19 の 3 行目ではポインタ表現が含まれる．このような場合 3.2 で述べたとおり，IDENTIFIER トークンが拡張されているので，Source-code 22 の 40 行目のようにポインタは変数名の部分に格納される．

以上より INIT_DECLARATOR タグで様々なパターンの変数宣言及び初期値代入の階層表現が適切であることがわかる．

4.2.2 typedef

Source-code 23. CML of source-code 20.

```
1 <INIT_DECLARATOR>
2     <DECLARATION class = "typedef" range = "unsigned" type = "int" identifier = "uint" />
3 </INIT_DECLARATOR>
4 <INIT_DECLARATOR>
5     <DECLARATION type = "uint" identifier = "e" />
6 </INIT_DECLARATOR>
```

Source-code 23 の 1 行目から 3 行目までは typedef による型の宣言を表現している．型宣言の場合は DECLARATION に class 属性に typedef が記述されている．また 3 章 2 節より lex が生成する字句解析において typedef で宣言された型の名前表への参照を行う実装を行っているので 5 行目で型を表す type 属性にユーザの任意の型が格納できている．

以上より，typedef の文法が適切に表現できているとわかる．

4.2.3 条件分岐文

Source-code 24. CML of source-code 21.

```
1 <SELECTION_STATEMENT kind = "if_else">
2   <TEST>
3     <RELATIONAL_EXPRESSION op = "&lt;">
4       <PRIMARY_EXPRESSION identifier = "a"/>
5       <PRIMARY_EXPRESSION identifier = "b"/>
6     </RELATIONAL_EXPRESSION>
7   </TEST>
8   <TRUE_CASE>
9     <COMPOUND_STATEMENT>
10      <ASSIGNMENT_EXPRESSION op = "=">
11        <PRIMARY_EXPRESSION identifier = "a"/>
12        <PRIMARY_EXPRESSION>
13          <INTEGER_CONSTANT kind = "decimal_constant" value = "0" />
14        </PRIMARY_EXPRESSION>
15      </ASSIGNMENT_EXPRESSION>
16    </COMPOUND_STATEMENT>
17  </TRUE_CASE>
18  <FALSE_CASE>
19    <SELECTION_STATEMENT kind = "if_else">
20      <TEST>
21        <RELATIONAL_EXPRESSION op = "==">
22          <PRIMARY_EXPRESSION identifier = "a"/>
23          <PRIMARY_EXPRESSION identifier = "b"/>
24        </RELATIONAL_EXPRESSION>
25      </TEST>
26      <TRUE_CASE>
27        <COMPOUND_STATEMENT>
28          <ASSIGNMENT_EXPRESSION op = "=">
29            <PRIMARY_EXPRESSION identifier = "a"/>
30            <PRIMARY_EXPRESSION>
31              <INTEGER_CONSTANT kind = "decimal_constant" value = "1"/>
32            </PRIMARY_EXPRESSION>
33          </ASSIGNMENT_EXPRESSION>
34        </COMPOUND_STATEMENT>
35      </TRUE_CASE>
36      <FALSE_CASE>
37        <COMPOUND_STATEMENT>
38          <ASSIGNMENT_EXPRESSION op = "=">
39            <PRIMARY_EXPRESSION identifier = "a"/>
40            <PRIMARY_EXPRESSION>
41              <INTEGER_CONSTANT kind = "decimal_constant" value = "2"/>
42            </PRIMARY_EXPRESSION>
43          </ASSIGNMENT_EXPRESSION>
44        </COMPOUND_STATEMENT>
45      </FALSE_CASE>
46    </SELECTION_STATEMENT>
47  </FALSE_CASE>
48 </SELECTION_STATEMENT>
49 <SELECTION_STATEMENT kind = "switch">
50   <PRIMARY_EXPRESSION identifier = "a"/>
51   <COMPOUND_STATEMENT>
52     <LABELED_STATEMENT name = "case" kind = "decimal_constant" value = "1">
53       <ASSIGNMENT_EXPRESSION op = "=">
54         <PRIMARY_EXPRESSION identifier = "x"/>
55         <PRIMARY_EXPRESSION>
56           <INTEGER_CONSTANT kind = "decimal_constant" value = "0"/>
57         </PRIMARY_EXPRESSION>
58       </ASSIGNMENT_EXPRESSION>
59     </LABELED_STATEMENT>
60     <JUMP_STATEMENT kind = "break" />
```

```
61          <LABELED_STATEMENT name = "
                default" >
62          </LABELED_STATEMENT>
63          <JUMP_STATEMENT kind = "break"
                />
64        </COMPOUND_STATEMENT>
65  </SELECTION_STATEMENT>
```

Source-code 24 の 1 行目から 48 行目までは，Source-code 21 の 1 行目から 7 行目の if 文による条件分岐文，Source-code 24 の 49 行目から 65 行目までは，Source-code 21 の 8 行目から 15 行目の switch 文による条件分岐文を表現している．1 行目の SELECTION_STATEMENT タグは，条件分岐命令があることを表現しており，kind 属性には if-else, switch のなどの条件分岐命令が記述されている．2 行目の TEST タグは条件式を表しており，関係演算式，論理演算式を用いる式が入っている．3 行目の RELATIONAL_EXPRESSION タグは，関係式を示しており，op 属性には'>' や'==' のような関係演算子が記述されている．3 行目の TRUE_CASE タグから条件式が真であったときの処理が記述され，18 行目の FALSE_CASE タグから else 節，つまり条件式が偽であったと処理が記述されている．49 行目の SELECTION_STATEMENT タグが switch の kind 属性持っている場合には，その子要素の先頭に判定する値を属性に持ったタグが挿入されている．52 行目の LABELED_STATEMENT タグはラベルを表しており，name 属性にはそのラベルの名前，kind 属性と value 属性には条件値が記述されている．以上より，条件分岐文の階層構造は if 文を適切に表現できているが，switch 文は条件部分を TEST タグで囲めていないため，適切に表現できているとはいえないことがわかる．

4.3 結果の検討及び考察

Source-code 19 から 21 までの C 言語のプログラムとそれらを変換した Source-code 22 から 24 までの CML について考察を行う．出力された CML を変換元となったソースコードとそれぞれ比較した結果，ほとんど全ての CML において元のプログラムの文法構造を適切に表現できていることがわかった．しかしいくつかの文法の XML 表現で改良すべき点が見つかった．ひとつが switch 文である．switch 文は if 文などと同じくプログラムの中で条件分岐を行う文法である．if 文の XML 表現では条件分岐を行う部分が TEST タグで囲まれており，また while 文や for 文などの繰り返し処理においても条件分岐を行う部分が TEST タグで囲まれている．一方 switch 文の条件分岐を行う部分は TEST タグで囲めていないため，CML 全体として仕様の統一感がなく扱いにくい印象がある．if 文と switch 文のような似た文法の XML 表現を統一していくことで CML 全体の仕様がわかりやすくなると思われる．

出力された CML の行数と元のソースコード (コメントアウトを除く) の行数を比較して Table 1 に示す．

Table 1. Comparison of the number of lines.

	original source code	CML
row1	3	41
row2	1	14
row3	2	6
row4	6	22
row5	7	13
row6	15	65
row7	7	45
tag.c	274	1422

7 つのソースコードとコンバータによって出力された CML の行数を比較した結果，CML は元のソースコードの平均 5.0 倍の長さになることがわかった．処理される文法によって CML の行数に大きく傾向があり，変数宣言部分では CML が元のソースコードの 10 倍近くまで大きくなるが，関数定義や宣言部分では CML は元のソースコードの 2 倍程度の大きさにとどまることがわかった．またコンバータ内でタグの生成を行っている tag.c というソースコードをコンバータで CML に変換したところ，これも約 5.2 倍の長さになったことからコンバータが生成する CML はおおよそ元のソースコードの 5 倍近くになると思われる．今後 CML をデータベースに格納することを考慮しても，CML が元のソースコードの 5 倍程度の大きさになることは許容範囲だと思われるが，更に CML の容量を小さくするためには，10 倍近く大きくなってしまっている変数の宣言部分の XML 表現について，設計を再度検討することで実現できると考える．

5. 結論

本論文ではソフトウェアテスト支援環境 IST のフロントエンドにあたる C 言語用のマークアップ言語 CML(C Markup Language) の定義と, 変換系 (converter) の設計と開発を行った. 今回開発した CML の処理系は ANSI-C(C90) の構文規則に基づき設計を行い，構文解析に構文解析生成器 yacc，及び字句解析生成器 lex を用いた．ANSI-C の構文規則を我々の必要とする粒度まで簡略化した後，構文解析で得られた構文情報を一度構造体の木構造への変換を行い，トップダウンで木構造の走査を行うことにより木構造のタグを表示できるよう設計をし，コンバータを開発した.

評価より，C 言語のほぼすべての文法を XML 形式で表現することができたことがわかった. これは十分な成果であると言えるだろう．しかし，一方で define によるソースコードの置換処理や include による外部ファイルの読み込みなどその他いくつかのプリプロセッサ処理への対応が実装されていない. 大規模なソフトウェアはファイルが複数に分割されていることが一般的であるので，より大きなソフトウェアを解析するためにプリプロセッサ処理の対応について早急に検討する必要がある.

参 考 文 献

1) 末広 暁久，佐々木 亮太，芳賀 博英，“XML によるプログラムの表現とそれに基づく統合テスト支援環境の提案”，電子情報通信学会技術研究報告.KBSE, 知能ソフトウェア工学，**110**(61)，33-38(2010).

2) Greg J. Badros，“JavaML: A Markup Language for Java Source Code”，http://www.badros.com/greg/papers/badros-javaml-www9.pdf，参照日時 (2014/05/11).

3) H.Aman，“JJmlt”，http://se.cite.ehime-u.ac.jp/tool/JJmlt/，参照日時 (2014/05/11).

4) 一般社団法人情報処理学会，プログラム言語Ｃ，http://www.jisc.go.jp/app/pager?id=478754，参照日時 (2014/05/11).

5) 芳賀 博英，コンパイラ講義資料，(同志社大学，京都，2013)，p.9-85.

6) XML 入門，http://gogodiet.net/z/xml/index.htm，参照日時 (2014/05/11).

7) 宮本 大輔，ブラン グレゴリー，秋山 満昭，“抽象構文木を用いた Javascript ファイルの分類に関する一検討”，https://www.iwsec.org/mws/2011/manuscript/2A4-2.pdf，参照日時 (2014/05/11).