XMLを用いたCASEツール・プラットフォーム作成支援環境

CASE-tool Platform Development Framework based on XML

高橋透*　大久保弘崇†　粕谷英人‡　山本晋一郎§

Summary. In this paper, we propose a CASE-tool platform development framework. While lexical and syntactic analyzers can be generated easily with existing compiler compiler, semantic analysis varies with programming languages especially scopes of declaration. It is designed to express various scope rules and to support semantic analysis phase of CASE-tool platform. We adopt XML for structured external representation for flexibility.

1　はじめに

CASEツールの開発には対象となるソフトウェアの字句・構文・意味解析が必要となり多大な労力が必要である．この労力を軽減するため，様々なCASEツールで共通に利用できる解析器を提供し，またCASEツール間のデータ統合環境を提供するCASEツール・プラットフォームが開発されている [1] [2] [3]．その目的は，CASEツール作成者をツール本来の機能により専念させることにある．

しかし，情報化の進展に伴い新たに提案される多種多様なプログラミング言語を対象としたCASEツール・プラットフォームを用意し続けることの困難さは解決されていない．我々のプロジェクトでもC言語，Java，JavaScript等の広く使用されているプログラミング言語を対象としているが，例えばC++は未対応であり，近年注目を集めているWebアプリケーション向けのスクリプト言語への対応にも時間がかかっている．

本研究の目的は，多種多様なプログラミング言語を対象としたCASEツール・プラットフォームを効率よく作成するための基盤である，CASEツール･プラットフォーム・プラットフォームの要素技術を確立することにある．その中心的な課題は，(i) 比較的環境が整備されている字句・構文解析に比べて手付かずである記号表に関するライブラリと，(ii) 解析結果の出力方法にある．

本稿の構成を以下に示す．2節では現在のCASEツール・プラットフォームがかかえる問題を述べ，3節ではXMLリポジトリを作成するCASEツール・プラットフォームに必要な開発環境について述べる．4節ではCASEツール･プラットフォーム作成支援環境のを実装方法について述べ，5節でJava言語のサブセットを例に作成支援環境の流れを説明し，最後にまとめと今後の課題を述べる．

*Tooru Takahashi, 愛知県立大学大学院 情報科学研究科, takahashi@yamamoto.ist.aichi-pu.ac.jp

†Hirotaka Ohkubo, 愛知県立大学 情報科学部, ohkubo@ist.aichi-pu.ac.jp

‡Hideto Kasuya, 愛知県立大学 情報科学部, kasuya@ist.aichi-pu.ac.jp

§Shinichiro Yamamoto, 愛知県立大学 情報科学部, yamamoto@ist.aichi-pu.ac.jp

2　CASE ツール・プラットフォームの現状

我々の研究グループでは，細粒度ソフトウェアリポジトリに基づいた CASE ツール・プラットフォームとして，Sapid を開発している [1]．Sapid を始めとしたソースプログラム解析器は一般に図 1 の手順で解析を行う．その手順は，解析器にソースプログラムを入力として与えると，言語の仕様に基づき字句・構文解析を行い，構文解析木を得る．次に記号表を用いて意味解析を行い，名前の宣言と使用を結びつけた意味情報を構文解析木上で表現した意味解析木を得る．意味解析木は次のフェーズであるフロー解析へ渡される．フロー解析まではコンパイラと同様の解析が行われる．外部出力の際はコンパイラのコード生成の代わりに，意味解析木とそこから使用されている記号表，フロー解析により得られたフローグラフを基に CASE ツール作成者が利用できるよう提供する．Sapid では SDB (Software Database) と呼ばれるフォーマットで解析情報を提供する．

CASE ツール・プラットフォームの解析情報を汎用性の高い XML 文書として表現する既存研究には Java を対象とした JavaML [4] や JX-Model [5]，ANSI-C を対象とした ACML [6] などがある．

本節では CASE ツール・プラットフォーム作成手法と現状の問題点について述べる．

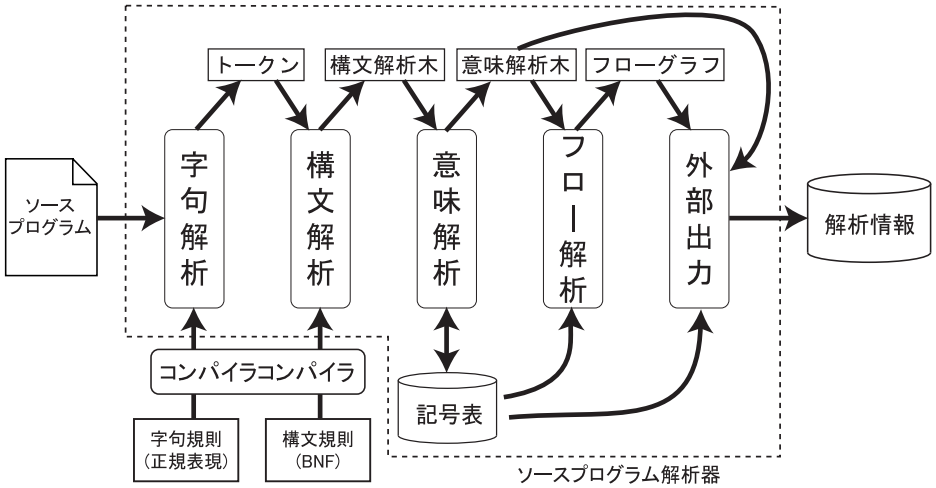

図 1　一般的なソースプログラム解析の流れ

2.1　字句・構文解析

字句・構文解析器の作成にはコンパイラコンパイラを用いられる．コンパイラコンパイラには，Lex/Yacc をはじめ，SableCC [7]，JavaCC [8]，ANTLR [9] などが挙げられる．字句解析器はソースプログラムの文字列をトークンと呼ばれる対象プログラミング言語の最小単位に分割する．構文解析器はトークンを元に構文を解析し抽象構文木を作成する．抽象構文木はソースプログラムに対する最小限の意味づけ，

構造化である．後方参照が無い言語では構文解析の際に意味解析を同時に実行していたが，近年のプログラミング言語は後方参照を許すものが多いため，構文解析器では構文解析木の作成だけを念頭においているコンパイラコンパイラが多い．

字句解析によって得られるトークンには，ソースプログラム中のインデントや改行，コメントが含まれない．これらは構文を定式化する上では必要ないため構文解析器に利用されることなく捨てられる．CASE ツールの種類によってはスタイル情報の保持をプラットフォームに要求するため，解析情報から完全な形で元の解析対象のプログラムに復元できることが望ましい．Sapid プロジェクトでは SPIE (Source Program Information Explorer) [10] がこれにあたる．

2.2 意味解析

CASE ツール・プラットフォームの意味解析ではコンパイラと同様に記号表を用いて以下のことを行う．

- 名前の使用と宣言を対応付ける．名前を使用する前に宣言が必要となる言語の場合その検査を行うなど，プログラミング言語ごとに定まっている名前の宣言と使用に関する細かなルールのチェックを行う．
- 多重定義された手続きや関数の呼び出しでは，実引数の数や型を基に必要な範囲で名前の解決を行う．この時に実引数と仮引数でどのような場合に型が同じとみなされるかはプログラミング言語により様々に定義されている．さらに型の正当性のチェックも必要である．

このように，意味解析はプログラミング言語ごとに，記号表へ登録する名前の種類やスコープルールが異なるため構文解析と比較して一般化が難しい．

2.3 制御フロー・データフロー解析

制御フロー解析の段階ではソースプログラムをフローグラフと呼ばれる有向グラフによって表現し，その中からループや分岐などに対応する特定の性質を持つ部分グラフを見つける．これは大域的な最適化のために行われる．データフロー解析では，制御フロー解析と意味解析の結果を利用してデータの流れを調べる．

2.4 解析情報の外部出力

以上によって解析された情報を CASE ツールから利用するために外部出力する．従来は独自形式のファイルへのアクセスルーチンを提供していたが，解析情報を XML 文書として表現することにより，特定の言語や計算機環境に束縛されることなく CASE ツールが作成できる．2.1 項でも述べたように，CASE ツール・プラットフォームの提供する解析情報は様々な CASE ツールの要求に応えるために，字句情報の保持など，細粒度の情報を保持することが望まれる．対象のソースプログラムを直接マークアップすることで，字句情報が保持できる．このような要求に応えるためにも，XML 文書として解析情報を作成することが望ましい．

このようなソフトウェア文書に対して直接タグを挿入した，XML 文書による細粒度ソフトウェアリポジトリとして XSDML (eXtensible Software Document Markup Language) が提案されている [5].

3 CASE ツール・プラットフォーム作成支援環境への要求

2 節で述べた CASE ツール・プラットフォームの現状を踏まえ本節では各種言語を対象とした CASE ツール・プラットフォームを効率よく作成するための支援環境に必要な要素について議論する．

3.1 字句・構文解析

字句・構文解析器はコンパイラコンパイラを利用することで比較的容易に作成できる．ただし，2.1 項で述べたようにスタイル情報やコメントなどのコンパイラが捨ててしまう字句情報も必要である．

3.2 意味解析

前節で見たように CASE ツール・プラットフォームの意味解析は記号表処理によって行われる．例えば C 言語のブロック文はブロックごとに同名のローカル変数を宣言できる．また，Java では継承により名前のスコープが発生する．これは構文上の包含と無関係な名前空間である．このようなプログラミング言語ごとに異なる複雑なスコープルールに沿った解析を行うには，様々なスコープを表現できる柔軟な記号表が必要である．記号表に求められる特徴を以下に挙げる．

3.2.1 記号表

名前ごとに存在するスコープはプログラミング言語のスコープルールによって決められる．プログラマからの視点で見た場合，一つの名前のスコープを見るのではなく，ある限られた領域内で今どの名前が有効であるかを見る．この限られた領域が名前空間と呼ばれ，この中では名前の使用と宣言が一意に定まる．名前空間はプログラム中のあらゆる位置に存在するが，実際にはある二つのトークンに挟まれた領域が名前空間になる．入れ子構造の場合，入れ子に入ると新しい名前空間を作成し，抜けるときにはその名前空間が見えなくなる．記号表は細分化された名前空間を名前が衝突しないよう統合した名前空間を表す．

記号表の一般的な構造は，名前を全てひとつの記号表で管理するか名前の種類ごとに記号表を管理し，名前がどの名前空間で宣言されたかを保持させる．しかしこの記号表の構造ではプログラミングの際に存在するプログラマの視点から見た名前空間を直接表現しているとはいえない．よって，名前空間ごとに登録管理し，名前空間に相互関係を持たせたほうが自然である．

また，型の扱いが問題である．型には言語ごとに基本型と構造型がある．基本型とは C 言語の `int` や `float` のようにプログラミング言語に組込まれている型である．構造型は基本型と構造型の組み合わせで構成される型である．構造型の定義部分では名前空間が存在するため，構造型は記号表として扱う必要がある．

文献 [11] では多種多様なプログラミング言語に対応させるために，記号表を 3 つのレベルに分けて構成している．

1. ネストが無く，名前の種類が無い記号表．直接使用することは少なく，項目 2・3 から間接的に利用される．
2. ネストが無く，名前の種類が管理できる記号表．プログラミング言語における名前とは変数名，手続き名，型名など様々な種類があり，またアクセス修飾子

などの属性もありこれらを分離して管理する．

3. ネストした名前空間をはじめとした記号表間を結ぶ記号表．ブロック構造のようにネストした構造を表現するための記号表．またネストした構造以外にも，クラスの継承関係といったソースプログラム中の構造の包含とは無関係な名前空間の参照を解決するための記号表である．

しかし，項目 2 のように種類ごとに分類して管理していては場合分けが増えすぎる．またデータ構造の提案に留まっており，項目 2, 3 間の連携や後方参照の解決については触れられていない．そこで本稿では以下のように記号表を作成する．

記号表クラス 記号表はその中で名前の衝突が起こらない名前空間 1 つを表す．記号表クラスはプログラミング言語ごとに定められている名前の種類ごとに記号表を用意するのではなく，登録情報に名前の種類を持たせ管理する．

登録情報のクラス 記号表に登録される 1 つ 1 つを表わすクラスである．記号表への登録情報は名前の文字列だけでなく，名前の種類や型や可視性，手続きなら仮引数の数や型などである．また，リファクタリングなどソースプログラムを直接利用する CASE ツールのために，ソースプログラム中での宣言されている名前の位置 (オフセット，行数) を保持する．

記号表間の繋がりを管理するクラス スコープルールによって定められた検索順序を表現するために記号表間に関係を持たせるクラスである．ある記号表で検索をするときにより優先度の低い記号表をすべて含めた名前空間で検索をする．この管理クラスでは，検索の際にスコープ内で使用可能な名前を収集する機能が必要になる．プログラミング言語のスコープルールは木構造や DAG に対応づけられる．木構造は入れ子構造や，(単一) 継承の表現に用いることができ，DAG 構造は多重継承を表現するために用いることができる．

後方参照の解決 後方参照を解決するためには参照より前に宣言を記号表に登録する必要がある．宣言を登録するステップと参照を解決するステップを設け抽象構文木を複数回走査することにより後方参照を解決する．

3.3 XSDML の出力

構文解析木を XML 形式にマークアップしていく．その際に，記号表を用いて得られた宣言・使用の関係を，XML のリンクの属性値として出力する．宣言部に対応する XML 要素に ID 属性をつけるために，CASE ツール・プラットフォーム・プラットフォームには登録情報にユニークな ID を付加する機能が必要である．

4 実装

本稿で提案する CASE ツール・プラットフォーム作成支援環境の概要を図 2 に示す．字句・構文解析を使うコンパイラコンパイラには JavaCC [8] を用いた．JavaCC に同梱されている JavaCC のプリコンパイラで JJTree というツールは抽象構文木の生成を支援する．これは，JavaCC の文法ファイルに抽象構文木[1]を生成する Java プログラムを埋め込むとともに，抽象構文木を走査するクラスを生成する．本 CASE ツール・プラットフォーム・プラットフォーム利用者は言語 L の JJTree の文法 を

[1]生成される抽象構文木は非終端記号を木のノード，終端記号をトークンとして構成されている．抽象構文木のノードは Node クラスを継承し，ノードからはトークンにアクセスができる．

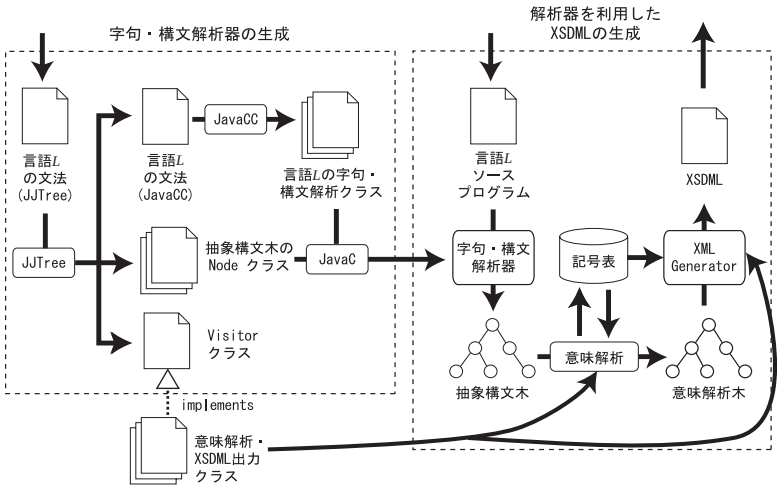

図 2　支援環境の概要

与えることにより L の字句・構文解析器を得る．

本 CASE ツール・プラットフォーム・プラットフォーム利用者が他に行うことは，意味解析及び XSDML 出力を行うクラスを作成することである．これらのクラスは抽象構文木を走査する必要があるが，これには JJTree の提供する Visitor クラスが利用できる．

4.1　字句情報を保持した解析木の作成

JavaCC を用いることで 3.1 項で指摘したスタイル情報やコメントを保持することができる．このような字句情報は，構文解析器に渡されるトークンから参照ができ，抽象構文木作成後も参照できる．

JJTree は字句･構文規則を入力とし，抽象構文木を生成する動作部を含んだ JavaCC ファイルを出力する．JavaCC ファイルと同時に構文木の走査クラスも生成する．この走査クラス内では抽象構文木の各ノードでのアクションを記述する．次項の記号表処理，XSDML 出力処理はここに記述する．

4.2　記号表処理

記号表処理のための基本的なライブラリを 3.2 項の考察をもとに設計する．図 3 にライブラリのクラス図を示す．これまでにも述べたように，記号表に登録する情報やスコープの情報はプログラミング言語ごとに異なるが，記号表処理の基本操作は登録と検索により行われる．そこで，記号表ライブラリは，実際に登録と検索を行う記号表クラス，登録情報クラス，スコープルールクラスにより構成される．

記号表クラス (SymbolTable)　このクラスが登録情報を管理する．また記号表内で検索を行う．検索は **Key** クラスをキーとして検索し，検索結果は **Entry** クラスを返す．このクラスだけでは単純なスコープルールしか表現できないため，通常はスコープルールクラスがこのクラスを利用し検索を行う．

登録情報クラス (Entry)　記号表に登録される項目のクラス．検索の際に利用する

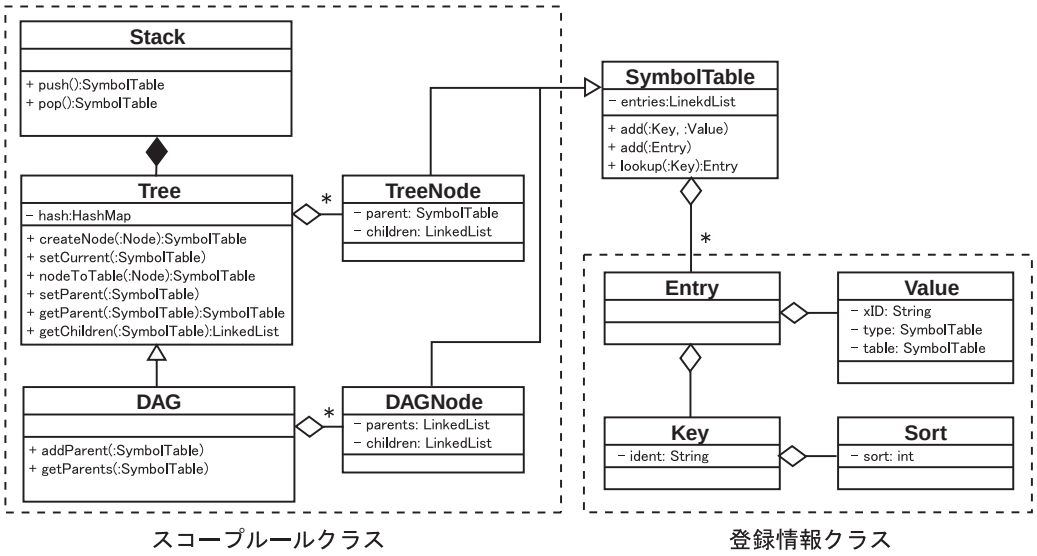

図 3　記号表ライブラリ

Key クラスと，検索後に利用する **Value** クラスによって構成される．

Key　識別子の名前が含まれており，メソッドの数や型といった情報は **Sort** クラスに登録する．

Value　登録情報の型，クラス定義やメソッド定義のように名前空間を持つときはその記号表への参照，XSDML 文書にする際に必要となる XML の ID 属性のためにソースプログラム内でのユニークな名前を保持する．

スコープルールクラス　スコープルールクラスは記号表を跨がった検索を可能するクラスである．これらのクラスはそのスコープ内にある複数の記号表に対し適当な優先順位をもって検索を行う．本稿で作成したスコープルールは，入れ子ブロック構造，クラスの (単一) 継承関係，クラスの多重継承関係である．各々を表すクラスとして **Stack**，**Tree**，**DAG** を設計する．

後方参照を解決するために，複数回抽象構文木を走査し，記号表へアクセスする必要がある．よって，現在の抽象構文木のノードとそれに対応する名前空間を管理する必要がある．**Tree** クラスはハッシュテーブル (図 3 の **Tree** クラスの `hash:HashMap`) を持っており，1 パス目で抽象構文木と記号表を対応付けて 2 パス目以降にこれを用いて登録や検索する記号表を引く．

4.3　XSDML の出力

意味解析木の各ノードはノード名をそのまま XML 要素名とする．トークンオブジェクトには記号表への参照があるため，宣言部のトークンを出力する際にはその宣言に対応する記号表の登録情報を埋め込む必要がある．参照部のトークンを出力する際には IDREF 属性を指定し宣言部と対応付ける必要がある．また，XSDML 出力に必要な要素であるコメントやスタイル情報を保持するために，トークンオブジェクト間の字句情報を埋めこむ．

5 実行例

本節では，CASE ツール・プラットフォーム開発支援環境を利用して，Java 言語のサブセットを例に，ソースプログラム (図 4) から XSDML 出力 (図 5) の作成までの手順を示す.

```
01: class C {
02:   int a;
03:   void m1() {
04:     int a;
05:     a;
06:   }
07:   void m2() {
08:     a;
09:   }
10: }
```

図 4　入力ファイル例

```
 1: <Input><ClassDecl><TClass>class</TClass><sp> </sp>
<TID type='class' id='class1'>C</TID><sp> </sp><TLBrace>{</TLBrace><nl>
 2: </nl><sp> </sp><ClassBody><VarDecl>
<Type><TINT type='prim'>int</TINT></Type><sp> </sp>
<TID type='var' id='field1'>a</TID><TSemicolon>;</TSemicolon></VarDecl><nl>
 3: </nl><sp> </sp><MethodDecl><ResultType><TVOID>void</TVOID></ResultType><sp> </sp>
<TID type='method' id='method1'>m1</TID><TLParen>(</TLParen><TRParen>)</TRParen><sp> </sp>
<Block><TLBrace>{</TLBrace><nl>
 4: </nl><sp>  </sp><VarDecl><Type><TINT type='prim1'>int</TINT></Type><sp> </sp>
<TID type='var' id='var1'>a</TID><Tsemicolon>;</TSemicolon></VarDecl><nl>
 5: </nl><sp>  </sp><Expr><TID type='var' ref='local1'>a</TID></Expr><TSemicolon>;</TSemicolon><nl>
 6: </nl><sp> </sp><TRBrace>}</TRBrace></Block></MethodDecl><nl>
 7: </nl><sp> </sp><MethodDecl><ResultType><TVOID>void</TVOID></ResultType><sp> </sp>
<TID type='method' id='method2'>m2</TID><TLParen>(</TLParen><TRParen>)</TRParen>
<sp> </sp><Block><TLBrace>{</TLBrace><nl>
 8: </nl><sp>  </sp><Expr><TID type='var' ref='field1'>a</TID></Expr><TSemicolon>;</TSemicolon><nl>
 9: </nl><sp> </sp><TRBrace>}</TRBrace></Block></MethodDecl></ClassBody><nl>
10: </nl><TRBrace>}</TRBrace></ClassDecl></Input>
```

図 5　XSDML 出力例

5.1 サブセット Java 言語の文法

まず，ここで対象とする言語であるサブセット Java 言語は，Java のクラス宣言 (継承なし)，メソッド宣言，フィールド宣言，メソッドの中はローカル変数宣言と名前の使用だけからなる．図 6 はサブセット言語を JJTree の文法で記述したものである．JJTree の文法は JavaCC の文法とほぼ同じである．今回は JJTree を用いて作成する抽象構文木生成器のために，3,4 行目に非終端記号のノードクラスが生成されるオプションと，抽象構文木を走査する際にノードごとの動作を記述する Visitor インタフェースを生成するオプションを付けている．

```
01: PARSER_BEGIN(SubsetJavaParser)
02: options {
03:   MULTI = true;
04:   VISITOR = true;
05: }
06: public class SubsetJavaParser {
07:   public static void main(String[] args)
08:    throws ParseException {
09:     new SubsetJavaParser(System.in).Start();
10:   }
11: }
12: PARSER_END(SubsetJavaParser)
13:
14: SPECIAL_TOKEN: {
15:   " "|"¥t"|"¥n"|"¥r"|"¥f"
16: }
17: TOKEN: {
18:   <ID: ( ["a" - "z", "A" - "Z"] )+>
19: | <NUM: ["1" - "9" ] ( ["0" - "9"] )+>
20: | <INT: "int">
21: | <VOID: "void">
22: }
23: void Start(): {} { // トップ
24:   ( ClassDecl() )* <EOF>
25: }
26: void ClassDecl():{} { // クラス宣言
27:   "class" <ID> "{"
28:     [ ClassBody() ]
29:   "}"
30: }
31: void ClassBody(): {} { // クラスメンバ
32:   ( VarDecl()
33:   | MethodDecl() )*
34: }
35: void VarDecl(): {} { // 変数宣言
36:   Type() <ID> ";"
37: }
38: void Type(): {} {
39:     <ID>
40:   | <INT>
41: }
42: // メソッド宣言
43: void MethodDecl(): {} {
44:   ResultType() <ID> "(" ")"
45:   ( Block()| ";" )
46: }
47: void ResultType(): {} {
48:   Type()
49: | <void>
50: }
51: void Block(): {} {
52:   "{"
53:     ( VarDecl()
54:     | Expr() ";" )*
55:   "}"
56: }
57: void Expr(): {} { // 式
58:     <ID>
59:   | <NUM>
60: }
```

図 6　JJTree (JavaCC) 記法によるサブセット言語の構文規則

5.2 意味解析

JJTree によって生成された Visitor インタフェースを実装して意味解析プログラムを作成する．この時に本稿が提案する記号表ライブラリを用いる．ここでは，記号表処理は以下の 3 ステップを必要とする．

1. クラス宣言を抽出し，クラスの名前空間を作成する．
2. メソッドやフィールド宣言をクラスの名前空間に登録する．メソッドの場合はメソッドの名前空間を作成する．メソッド内でのローカル変数宣言はメソッドの名前空間に登録する．また，メソッドの返り値の型やフィールドの型情報はステップ 1 で登録してあるため，検索し型情報に含めることができる．
3. 1，2 の操作が完了したら，変数の使用部分で記号表を検索し，対応する宣言の ID を取得する．

図 7 の SemanticAnalyzer1〜3 はそれぞれ上に述べた意味解析のステップを行う．その結果として図 8，9 が得られる．**Tree** クラスは記号表間に親子関係を持たせるクラスで，今回は入れ子構造の表現に用いる．記号表を **Tree** クラスのノードとし

```
001: import symboltable.*;
002:
003: public class SemanticAnalyzer
004:  implements SubsetJavaParserVisitor {
005:   // 記号表管理クラス
006:   private symboltable.Tree tree;
007:   public static final int CLASS  = 0;
008:   public static final int VAR    = 1;
009:   public static final int METHOD = 2;
010:
011:   public SemanticAnalyzer() {
012:     tree = new Tree();
013:     // 基本型を登録しておく
014:     tree.add(new Key("int", new Sort(CLASS)), new Value());
015:   }
     (省略：各ノードで走査するためのメソッド)
055: }
056:
057: class SemanticAnalyzer1 extends SemanticAnalyzer {
     (省略：コンストラクタ)
061:   public Object visit(ASTClassDecl node, Object data) {
062:     TID tID  = node.getID();
063:     Entry entry = new Entry(new Key(tID.image, new Sort(CLASS)),
064:                             new Value());
065:     tID.setAttr(entry);
066:     tree.add(entry);
067:     SymbolTable parent = tree.getCurrent();
068:     // nodeと名前空間を対応付け、
069:     // 現在の名前空間を親にもつ名前空間を作成し、
070:     // 現在の名前空間をクラス内にする
071:     tree.setCurrent(tree.createNode(node));
072:     // 子ノードを走査する
073:     node.childrenAccept(this, data);
074:     tree.setCurrent(parent);
075:     return null;
076:   }
077: }
078: class SemanticAnalyzer2 extends SemanticAnalyzer {
079:  public SemanticAnalyzer2(Tree tree) {
080:      this.tree = tree;
081:  }
082:  public Object visit(ASTClassDecl node, Object data) {
083:    SymbolTable parent = tree.getCurrent();
084:    // nodeに対応する名前空間を引き
085:    // その名前空間を現在の名前空間にする
086:    tree.setCurrent(tree.nodeToTable(node));
087:    node.childrenAccept(this, data);
088:    tree.setCurrent(parent);
089:    return null;
090:  }
091:  public Object visit(ASTVarDecl node, Object data) {
092:    Entry type = (Entry)node.getType().jjtAccept(this, data);
093:    // 変数名を現在の記号表に登録
094:    TID tID  = node.getID();
095:    Entry entry = new Entry(new Key(tID.image, new Sort(VAR)),
096:                            new Value(type));
097:    tID.setAttr(entry);
098:    tree.add(entry);
099:    return null;
100:  }
101:  public Object visit(ASTType node, Object data) {
102:    Token t = node.getFirstToken();
103:    Entry entry = tree.lookup(new Key(t.image, new Sort(CLASS)));
104:    t.setRef(entry);
105:    return entry;
106:  }
107:  public Object visit(ASTMethodDecl node, Object data) {
108:    // 戻り値の型名を検索
109:    Entry type = (Entry)node.getType().jjtAccept(this, data);
110:    // メソッド名を記号表に登録
111:    TID tID  = node.getID();
112:    Entry entry = new Entry(new Key(tID.image, new Sort(METHOD)),
113:                            new Value(type));
114:    tID.setAttr(entry);
115:    tree.add(entry);
116:
117:    SymbolTable parent = tree.getCurrent();
118:    // 現在の名前空間をメソッド内にする
119:    tree.setCurrent(tree.createNode(node));
120:    node.childrenAccept(this, data);
121:    tree.setCurrent(parent);
122:    return null;
123:  }
124:  public Object visit(ASTResultType node, Object data) {
125:    ASTType type = node.getType();
126:    return (type != null ? type.jjtAccept(this, data) : null);
127:  }
128: }
129:
130: class SemanticAnalyzer3 extends SemanticAnalyzer {
     (省略：コンストラクタ)
134:   public Object visit(ASTClassDecl node, Object data) {
     (省略：SemanticAnalyzer2#visit(ASTClassDecl, Object)と同じ)
140:   }
141:   public Object visit(ASTMethodDecl node, Object data) {
     (省略：上のASTClassDeclノードと同じ処理)
147:   }
148:   public Object visit(ASTExpr node, Object data) {
149:     TID tID = node.getID();
150:     if (tID != null) {  // ExprがIDならば
151:       String id = tID.image;
152:       // 宣言部を検索し、宣言部を登録する
153:       tID.setRef(tree.lookup(new Key(id, new Sort(VAR))));
154:     }
155:     return null;
156:   }
157: }
```

図 7　抽象構文木の各ノードでのアクションを記述する Visitor クラス群

て登録する際には，71,119 行目のように`tree.createNode()` を用いて抽象構文木のノードと名前空間の対応を管理する．これは次のステップで抽象構文木を走査するときに，86 行目のように`tree.nodeToTale()` を用いて現在のノードに対応する記号表を取得する．登録の際には対応する記号表に登録し，検索の際にはこの記号表から順に検索を行う．

`SemanticAnalyzer1,2` によってクラス，フィールド・ローカル変数，メソッドが登録される．図 4 の入力プログラムに対し表 1 の左側の登録が行われる．記号表に登録された **Entry** オブジェクトは抽象構文木の `TID` トークンの `attr` フィールドから参照できる．図 6 の 32,53 行目からわかるようにフィールドとローカル変数は同

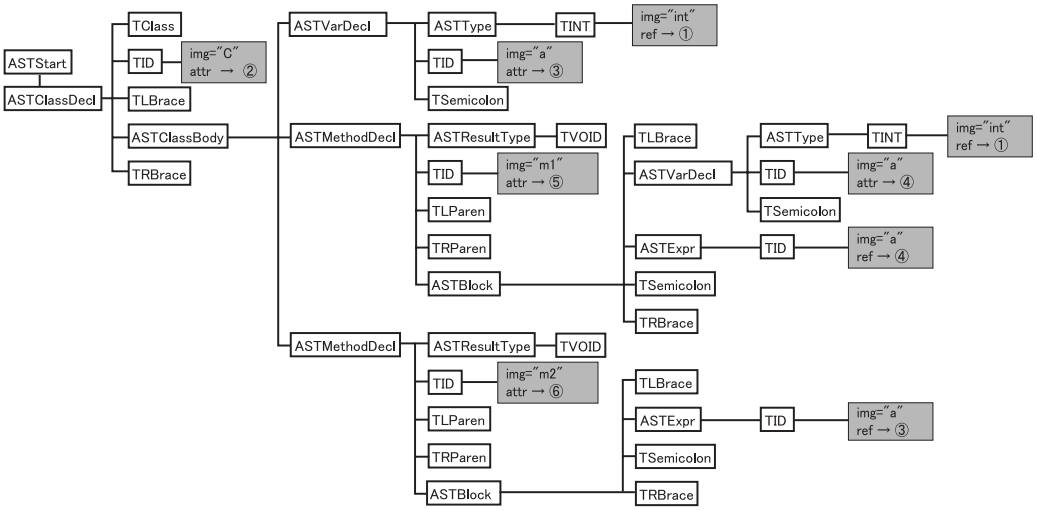

図 8　意味解析により得られる意味解析木

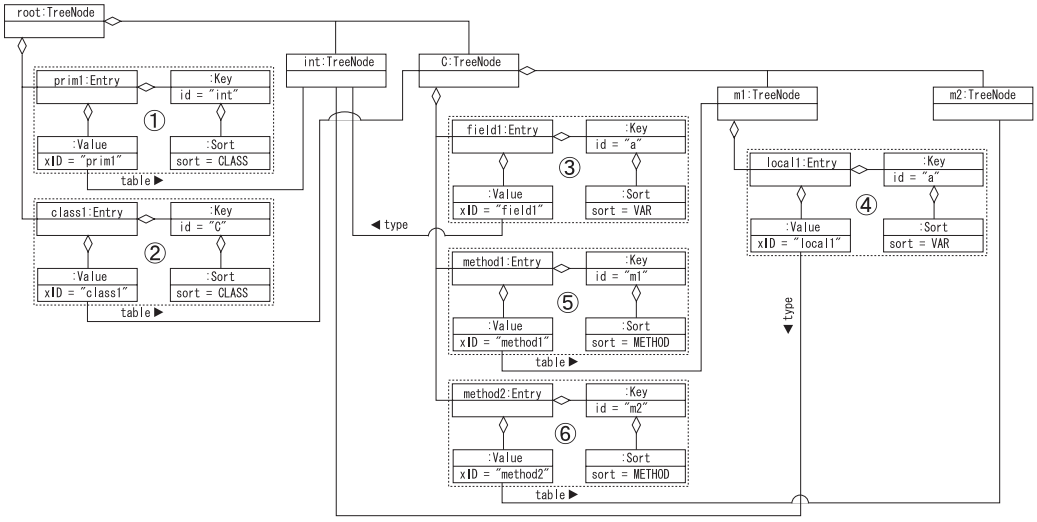

図 9　意味解析により得られる記号表

図 4 中の宣言	図 7 内で登録された **Entry** の番号	図 4 中の参照	図 7 内で検索された **Entry** の番号
l.1 の `class C`	②	l.2 の `int a`	①
l.2 の `int a`	③	l.4 の `int a`	①
l.4 の `int a`	④	l.5 の `a`	④
l.3 の `void m1()`	⑤	l.8 の `a`	③
l.7 の `void m2()`	⑥		

表 1　意味解析木から記号表への参照関係

一の構文要素である `VarDecl` によって宣言されている．`SemanticAnalyzer2` 内では走査中に同じメソッドが呼ばれている (91–100 行目) が，登録する際のノードに対応した正しい記号表に登録される (図 9 の ④ と ⑤).

`SemanticAnalyzer2,3` によって型とフィールド・ローカル変数の使用と宣言が対応付けられる．型の識別子である `TINT` トークンとフィールド・ローカル変数の識別子である `TID` ノードの `ref` フィールドは，宣言部と同じ `Entry` クラスを指している．表 1 の右側が実際に行われる検索で，入力プログラムの 5,8 行目の `a` は，意味解析の結果それぞれ記号表内のエントリである ④ と ③ を表している．

5.3　XSDML の出力

構文解析木の各ノードに対して XSDML 出力を行う．この際，意味解析と同様に Vistor クラスを利用する．2.4 項で述べたように，元のプログラムの字句を XML 中に完全に再現するため，BNF 中の終端記号 (トークン) の場合には，ひとつ前のトークンとの間に存在するスタイル情報やコメントを挿入する．図 5 は図 8 を XSDML 出力したものである．CASE ツール・プラットフォームで XSDML への変換を可能とするため，細粒度で解析情報を提供する必要があり，冗長で繁雑なものになっている．CASE ツール・プラットフォーム作成者は必要に応じてより抽象度の高い XSDML へ変換し，CASE ツール作成者に提供する．

6　関連研究

記号表処理に関する研究には文献 [11] [12] が挙げられる．文献 [11] は 3.2.1 項で見たように多種多様なプログラミング言語に対応させるために，記号表を 3 つに分類して構成している．

文献 [12] は，宣言的な記述で意味解析を書くことによって意味解析器の自動生成を行う．宣言的な記述により従来の意味解析記述の 1/6 に減らすことで意味解析器生成系の有用性を示している．しかし，対象としているスコープルールがブロック構造のみで，後方参照のあるスコープについての対応は困難である．本稿では様々なスコープルールを対象としており，入れ子構造以外のルールや後方参照のあるスコープも対象としている．

7 おわりに

本稿では，多様なプログラミング言語を対象となるように再利用可能な CASE ツール・プラットフォーム支援環境を提案した．また，従来単一のプログラミング言語を対象に作成されている CASE ツール・プラットフォームでは再利用が困難である記号表処理について考察した．ソースプログラム中に存在する複雑な名前空間を表現するために，記号表間の関係をモデル化することで多くのプログラミング言語での利便性が高まった．

また，CASE ツール・プラットフォームの解析情報を XSDML として表現することで開発言語と開発環境に束縛されることがなくなった．

現在意味解析部を Java により手続き的に表現しているが，スコープルール及び名前の登録に関する事項を表現する宣言的な記法が望まれる．また，構文解析器の構成上の理由で存在する無意味な非終端記号の削除を行うなど，XSDML の冗長性を減少する手法も今後の課題である．さらに XSDML の機能として複数のソースファイルからなるプログラムを処理するためには，各々のソースファイルから生成される XSDML ファイルの情報の統合を行う必要がある．

謝辞 御指導して頂いた愛知県立大学情報科学部の稲垣康善教授および山本研究室の皆様に感謝します．また，研究のベースとなる Sapid プロジェクトの皆様に感謝します．

参考文献

[1] 福安直樹, 山本晋一郎, 阿草清滋. 細粒度ソフトウェア・リポジトリに基づいた CASE ツール・プラットフォーム Sapid. 情報処理学会論文誌, Vol. 39, No. 6, 1998.
[2] Hachisu Yoshinari, Yamamoto Shinichirou, and Agusa Kiyoshi. A CASE Tool Platform for an Object Oriented Language. *IEICE Trans. on Information and Systems*, Vol. E82–D, No. 5, pp. 977–984, 5 1999.
[3] Eclipse. `http://www.eclipse.org/`.
[4] Greg J. Badros. JavaML: a markup language for Java source code. *Computer Networks (Amsterdam, Netherlands: 1999)*, Vol. 33, No. 1–6, pp. 159–177, 2000. `http://www.cs.washington.edu/homes/gjb/JavaML/`.
[5] 吉田一, 山本晋一郎, 阿草清滋. XML を用いた汎用的な細粒度ソフトウェアリポジトリの実装. 情報処理学会論文誌, Vol. 44, No. 6, pp. 1509–1516, 6 2003.
[6] 川島勇人, 権藤克彦. XML を用いた ANSI C のための CASE ツールプラットフォーム. コンピュータソフトウェア, Vol. 19, No. 6, pp. 21–34, 2002.
[7] Étienne Gagnon. SableCC, March 1998. `http://www.sablecc.org/`.
[8] JavaCC (Java Compiler Compiler) Java Parser Generator. `https://javacc.dev.java.net/`.
[9] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software–Practice and Experience*, Vol. 25, No. 7, pp. 789–810, 1995. `http://www.antlr.org/`.
[10] 大橋洋貴, 山本晋一郎, 阿草清滋. ハイパーテキストに基づいたソースプログラム・レビュー支援ツール. 電子情報通信学会ソフトウェアサイエンス研究会, Vol. 98, No. 28, pp. 15–22, 1998.
[11] Pei-Chi Wu, Jin-Hue Lin, and Feng-Jian Wang. Designing a Reusable Symbol Table Library. Technical report, CSIE-93–1010, November 1993.
[12] 亀山裕亮, 中井央, 山下義行, 田中二郎. コンパイラのための意味解析器生成系. 日本ソフトウェア科学会 18 回大会, 9 2001.