修 士 論 文

ソースコード理解支援のための表示自由度の高い視覚化ツールの研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

永井 路人

2006 年 3 月

修 士 論 文

ソースコード理解支援のための表示自由度の高い視覚化ツールの研究

指導教官　鈴木正人 助教授

審査委員主査　鈴木正人 助教授
審査委員　落水浩一郎 教授
審査委員　片山卓也 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

410088 永井 路人

提出年月: 2006 年 2 月

概 要

現代の情報社会システムでは、システムに恒常的に新しい機能追加が要求される。ソフトウェアの世界において、その要求された機能を実現する状況下では既存のシステムが存在する場合が多く、多くの場合、要求者はそのシステムの拡張という形を求める。よって、システム開発者は既存のプログラムを理解する必要がある。しかし、そのプログラムが自分以外の第三者によって作成されたものや、自分が作成したプログラムであっても複雑化したもの・作成より時間が経過したものである場合、プログラムの理解が困難になる。

その理解支援のためのソースコード可視化ツールが現時点でも存在するが、その操作性・表示能力に問題がある。よって、本研究ではその問題点を解決し、ソースコードの理解を支援するツールを作ることとする。

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

近年、ソフトウェアの高機能化は著しく、そのため、ソフトウェアのソースコードの量は増大する傾向にある。このような状況下でのソフトウェアの開発・保守には、必然的に多くの開発者を必要とする。この結果、ある開発者の書いたソースコードを他の開発者が読むという局面が増大することになる。したがって、ソースコードの構造を深く理解することの重要性が高まっている。

チームでの開発状況として、ソースコードの理解の共有は非常に重要である。その共有の手法としては、設計図の作成や実際にソースコードを見ながらその開発者が説明をするなど、話し合いをする方法がある。しかし、この状況はそのソースコードを開発した開発者が存在し、なおかつ話し合う場を持つ事ができる場合であり、その状況をいつも作ることができるとは限らない。このような場合、自分以外の第3者が開発したソースコードを読んで理解しなくてはならない。

その際、一度にソースコードの全てを見ることができないため、一部分を注目しながら読み進めることになるだろう。しかし、この方法は大きなソースコードを対象とする場合には多くの時間を必要とする。この問題を回避するために多くの研究が行われており、クラスビューアや構文エディタを備えた様々なツールが開発されてきた。また、今日のソフトウェア開発で主に利用されているJava言語やC++言語等のオブジェクト指向言語では、継承やポリモーフィズムなどの手法が用いられている。これらの手法はソースコードの再利用性や拡張性を高め、ソフトウェアの生産性の向上に大きく寄与している。

しかし、これら既存のツールには多くの問題点がある。まず、クラスビューアなどではクラス間の依存関係などの大局的情報をグラフ構造で図示するが、依存関係が増加すると表示するグラフが複雑になり、全ての情報を効率的に把握することが難しくなる。継承などの手法については、本来ならば記述されている働きが、その時編集しているソースコード上に書いていないため、その情報を読み取る必要性がある。

また、既存ツールではソースコードを表示する大きさとして、関数・行・変数の3段階しか存在しなかった。そのソースコードによることであるが、これは単一の関数が肥大化している場合などにおいては、この表示の大きさはあまり意味をなさない。なぜならその関数を表示したとして、次の表示の単位である特定の1行を表示してもその行だけで理解が進む場合があまり存在しないためである。よって、開発者はその前後の行を読み始めることになる。しかし、これでは最終的に肥大化した関数の全てを読むことになり、開発者

の負担は減ることはない。

加えて、現時点で存在する理解支援ツールはそのソースコードの理解を試みている利用者、一人だけを対象とした作りになっていることが問題であると思われる。チームで開発を行う場合など、ソースコードの理解は複数人に要求される。しかし、現在の理解支援ツールにおいては一人一人に閉じてしまっていて、人から人への理解の伝承を考慮に入れていない状況である。

このように、今日のソフトウェア開発現場では、これら既存の視覚化環境における問題点を克服し、開発者が既存のソースコードを効率的に理解できるようにする視覚化手法を開発することが急務である。

1.2 本研究の目的

本研究の目的は、C および C++言語を対象として、開発者が既存の大規模なソースコードを効率的に理解できるようにする視覚化ツールを開発することである。このようなツールを作成することにより、多様化するユーザの要求に対して的確に表示が可能となる。本論文のタイトルにある「表示の自由度が高い」とは、表示させるための条件の要求が様々な方法で指定できるということを言う。

背景で述べた問題点の一つである「表示の大きさ」について本研究で拡張する。今までの表示単位であった関数・行・変数の 3 段階において、関数と行の間に新しい表示単位を導入する。この間に注目した理由は、ユーザがその時見ている関数が多数の行で構成されている場合、ある 1 行が表示されそれを読んだとしてもその行以外の部分が多大であり、その行だけでソースコードの意味を理解するのが困難であるためである。それに比べ、行と変数の間には視覚的に多くの文字列が存在することはほとんどなく、人の記憶力でも十分に保持できるので、この 2 点間には新しい表示単位は必要ないと考えた。

また、既存の理解支援ツールは単一の利用者を対象に作られてきたが、本研究では複数人を対象に開発をする。これは、ある利用者が理解支援ツールを使用して得られた知見を、次にこの理解支援ツールを使用してソースコードを理解する利用者に、その情報を伝える機能を持たせるということである。あるソースコードの理解を試みる利用者は、チームで開発を行う場合などにおいても多く存在する。そのような場合、本研究で開発する理解支援ツールでは、以前に理解を試みたユーザが行った表示要求を保存しておき、次に理解を求める人間がその要求と近い場合にその表示方法を表示候補として提示することを行う。

本研究では、この 2 点について上で記述したような方法により解決する理解支援ツールを開発する。

1.3 本論文の立場

この節では本研究で使用する言葉の定義、および本研究の対象としている範囲について説明する。

まず、「視覚化」という言葉についてだが、従来では、視覚化と言うと Scientific Visualization を指していたが、現在の Web の発展や大規模記憶装置の普及、およびエンドユーザでも大量のデータを処理/表示可能になり、Xerox PARC(Palo Alto Research Center) の S. Card らは、Scientific Visualization に対してこれを Information Visualization と呼んだ [1]。本論文のタイトルでも使用されている「視覚化 (ビジュアライゼーション)」とは、断りがない限り以下では Information Visualization(情報視覚化) のことを指す。

また、プログラムの視覚化は、アルゴリズムアニメーションやデバッグ支援など 1980 年代半ばまでは Visual Programming と Program Visualization との区別が明確でなく混同して使用されていた。それを B.A.Myers[2] が、グラフィックスそれ自体がプログラムであるものを Visual Programming、プログラムはテキストで記述されプログラムのある側面や実行状態を表示するためにグラフィックスを利用するものを Program Visualization と定義した。本研究では、後者である Program Visualization について研究したものである。

さらに、Program Visualization では描画対象が静的なものか動的なものか、および視覚化するものがソースコードかデータかで分類することができる。この 4 分類によれば、本研究で対象としているものは、静的コード視覚化システムである。

また、本論文のタイトルの一部にもなっている「理解」という言葉であるが、この言葉に関しては様々な解釈ができ、ユーザがどのような状態になった時「理解した」とするのかは非常に難しいことである。よって、本論文では「ソースコードを理解した」ということを「ソースコードを変更する際、変更箇所の特定が可能」つまり、ソースコードを直したい時に直す場所がわかると言う事と同義として以下では使用することとする。

1.4 本論文の構成

本論文の構成について簡単に説明したものを以下に示す。

- 第 2 章では、ソースコードの視覚化に関する既存の研究やツールを紹介し、これら既存ツールに潜む問題点について指摘する。および、この問題点についての解決法を述べる。
- 第 3 章では、作成するツールの要求仕様について述べ、ツールの全体像を記す。
- 第 4 章では、ツールの最初の処理段階である解析について説明する。この段階は既存ツールを使用するので、主にそのツールの利点と本研究での使用方法を述べる。
- 第 5 章では、ツールに取り入れた新しい抽出単位とともに、情報の抽出方法を説明する。

- 第 6 章では、第 5 章で抽出した情報を表示する方法、および表示の制御について説明する。
- 第 7 章では、第 3 章において述べた技術を用いた視覚化ツールを設計・実装し、その有効性について考察する。
- 第 8 章では、本研究のまとめと今後の課題について述べる。

第2章　既存研究

現在、理解支援を行うツールは多数存在する。そして、それらの支援の方法はさまざまである。例えば、関数の呼び出し関係などをグラフィカルに表示することを目的としたもの、対象となっている変数などの定義部に簡単に移動できることを目的としたものなどが挙げられる。

この章では、それらのツールについて調査したこと、およびその問題点についての解決法を述べる。

2.1　既存ツール

2.1.1　Data Display Debugger

Data Display Debugger[4] (DDD: Andres Zeller, Free Software Foundation, 2003) は、現在も開発がおこなわれており GDB[5], DBX[6], XDB[7] などのフロントエンドである。特徴としては以下が挙げられる。

- gdb のコマンドラインを GUI でデバッグが可能
- データ構造をグラフに表示可能

DDD は GDB の機能を全て受け継いでおり、かつプログラムの操作・表示が GDB では CUI であったものを、このツールではグラフィカルに表示することができるという利点を持っている。ユーザはソースコードの実行されている行を見るだけでなく、データを見ながらプログラムを実行できる。

DDD の操作画面 (図 2.1) は、主に 3 つのウィンドウから成っており上から順にデータウィンドウ、ソースコードウィンドウ、コンソールウィンドウ、となっている。データウィンドウは、選択された変数などを図に表示する部分である。ソースコードウィンドウは、デバッグの対象となっているソースコードを表示する。コンソールウィンドウは、デバッグ操作を文字列として表示する。また、ソースコードウィンドウに重なっている部分はコマンドツールパネルである。ここに、ユーザがよく使用するコマンドがボタンで表示されている。

図 2.1 では構造体 node(表 2.1) のメンバを図として表している。そのデータ図を表示する操作例としては、以下のように行う。

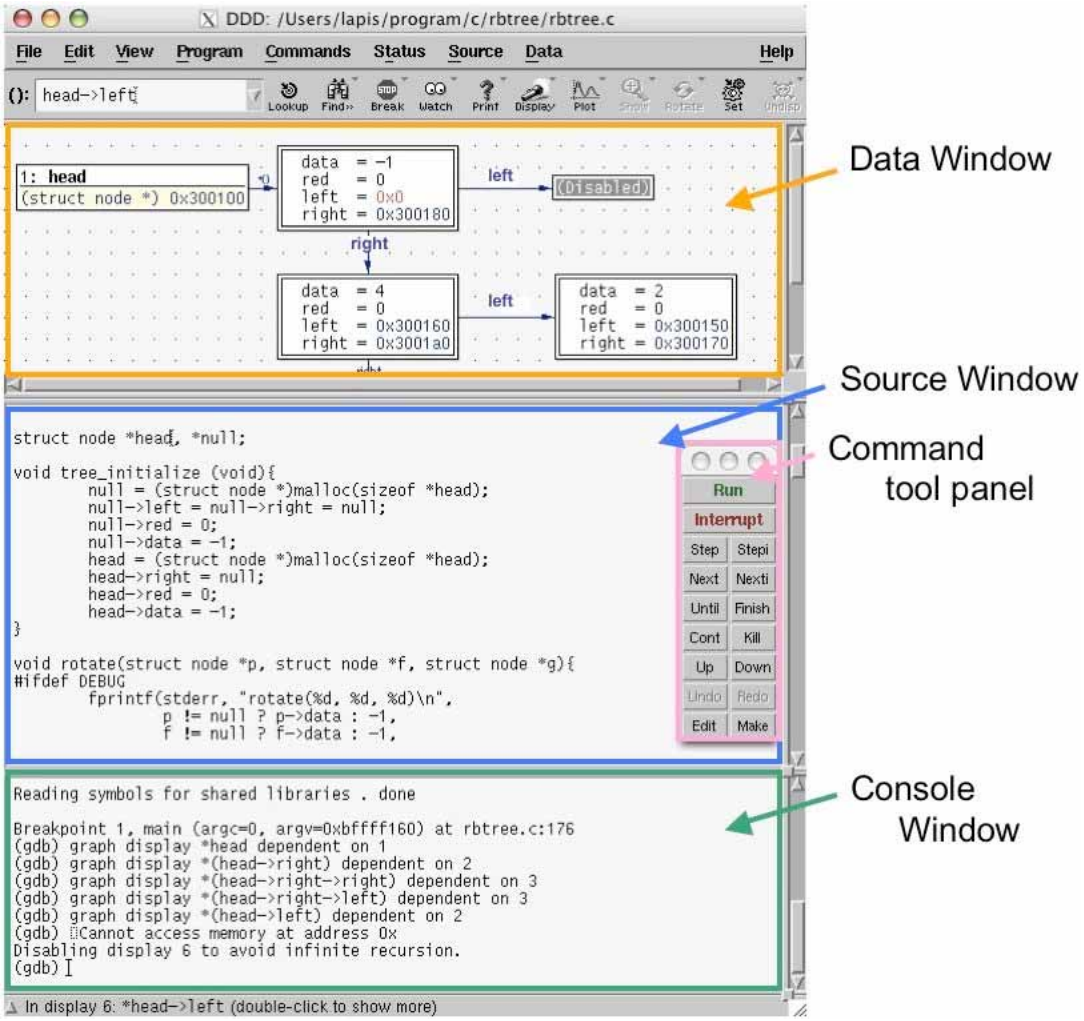

図 2.1: Data Display Debugger で 2-3-4 木のソースコードを解析した結果の例

1. ソースコードウィンドウ中の文字列 head をマウスによって選択
2. ツールバーの display ボタンを押す
3. データウィンドウに構造体 head の図が表示されるので、そのアドレス (この図の場合 0x300100) をダブルクリック
4. head が指している次の構造体が表示されるので、メンバ left または right のアドレスをダブルクリック

参照先が存在しない場合 (0x0 の時) は、アドレスをクリックすると Disabled が表示されるようになっている。また、この図を表示した段階で、コマンドツールパネル中にある Cont(continue) ボタンなどで処理を進めると、データウィンドウ中の値も自動的に表示が変わる。

表 2.1: 構造体 node のソースコード

```
struct node{
 int data;
 int red;
 struct node *left, *right;
};
```

DDD は、先に挙げた (1.3 節) 静的・動的およびソースコード・データの 4 分類から分けると、動的データ視覚化システムと言うことができるので、本研究が対象としている静的ソースコードとは異なる。しかし、GDB においては非常に広く使用され制御の単位を関数・行の段階で制御できる。この制御の単位に注目したため、ここに挙げることとした。

2.1.2 GNU GLOBAL source code tag system

GNU GLOBAL[8] は、ソースコードに索引付けを行うことで参照部分や定義部分が簡単に発見可能なことより理解支援を行うツールである。このツールは、以下の特徴を持っている。

- 様々なツールから呼び出して使用することが可能
- オブジェクトの定義部だけでなく、参照部分も検索が可能
- ソースコードをハイパーテキスト化可能

現在も開発が行われており、関数や変数をリンク付けして交互に参照することを可能にする。対応環境としてはシェルのコマンドラインや bash,vi,Web ブラウザなど様々である。注目すべきことは、索引付けの結果を HTML 形式に変換することが可能であり、その際、関数名・変数名をインデキシングする。このことから、ユーザは関数名などをインデキシングされたページより見つけ、ハイパーリンクによって対応するソースコードに移動することが容易にできることとなる。

図 2.2 は、前節 2.1.1 で使用したソースコードをこのツールでタグ付けした結果をブラウザで表示したものである。このソースコード中の関数 tree_insert の定義箇所を見たいとする。

1. 今の検索対象は関数 tree_insert であるので、頭文字である t を「DEFINITIONS」フレームより探してクリック
2. 頭文字 t を持つ関数一覧がアルファベット順に表示されるので、その中から tree_insert をクリック

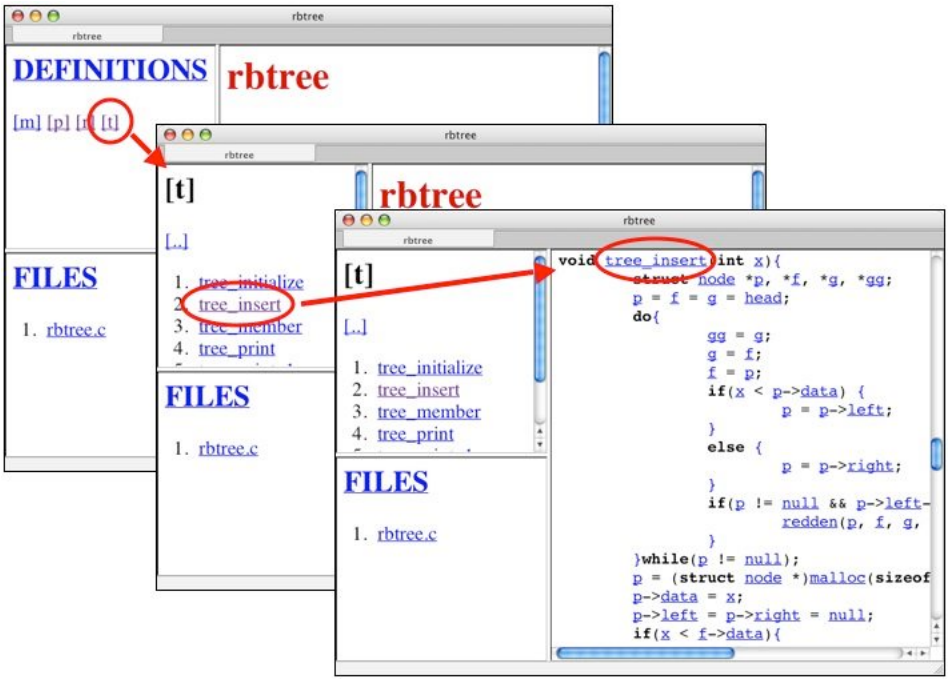

図 2.2: GNU GLOBAL で 2-3-4 木のソースコード中の関数定義部を探す例

3. DEFINITIONS フレームの右フレームにソースコード中の tree_insert が定義されている部分が表示される

このような操作方法により、容易に関数や変数の定義部を探すことが可能である。

しかし、画面を切り替える操作を繰り返し行うことにより、現在どこを見ているかわからなくなる「迷子問題」という問題を発生する可能性が高い。この問題は、以下のような表示方法から起こる。まず、ユーザはウィンドウ中のリンクをクリックし新しいページに移動する。すると画面は再描画され、以前の情報は隠されてしまう。この移動と再描画の繰り返しが少ない場合、ユーザは今までの道のりを記憶できるため現在の状態を正しく把握できる。しかし、この繰り返し行為が数多く起こると、ユーザは今の状態になるまでの経過を把握できなくなる。そしてその結果、ユーザは一度見た画面を忘れることにより再度見るという手間が必要となり、結果として理解支援としての利点が薄れてしまうことになってしまう。

2.1.3 ビジュアルモデリングツール Rational Rose

Rational Rose[9] はラショナルソフトウェア社の UML のビジュアルモデリングツールであり、そのツールの一部として、ソースコードの理解支援に関する以下の特徴を持っている。

- 設計モデルに基づいて各種言語用のコードを生成することが可能

Rose はモデリングツールであり、理解支援とは直接は結びつかないが、できたモデルを前提条件の知らない人に対して使うことによって、モデリングツールの一部を理解支援として応用している。特にオブジェクト指向言語の依存解析は高精度であり、ソースコードの解析結果をクラス図として出力することが可能となっている。しかし、解析結果の表示をする際に表示情報量の調節をすることが出来ないため、高精度の解析結果が生かせるはずの大型プログラムのクラス図の出力において、表示されたクラス図が大量かつ複雑になり、結果として開発者がソースコードを理解する際の助けとならない場合がある (図 2.3 参照)。表示する情報量の設定が可能であれば図 2.3 のような関係情報の爆発的な増加を押さえる事ができ、ソースコードの理解支援に有用であると思われる。

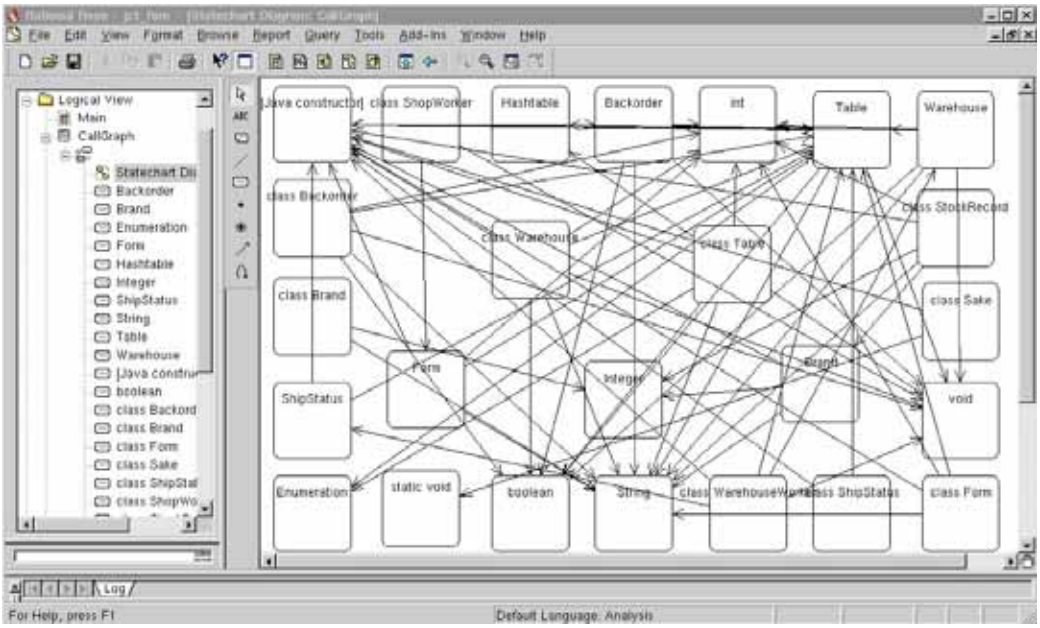

図 2.3: Rational Rose で酒屋問題を扱うプログラムを解析した結果の例

図 2.3 は、酒屋問題プログラムのクラスの呼び出し関係を図にしたものである。図中の四角形が各クラスを示しており、そこから出ている矢印が呼び出し関係を表している。図式化によりユーザの理解の容易性が本来ならば向上するのであるが、この場合 (図 2.3) はその利点が失われている。この原因としては、クラスの数が多いことと呼び出し関係の矢印が交差する線が多いことが挙げられる。よって、このツールでの理解支援については、このような問題があると言える。

2.2 問題点に対する解決法

上記のように、既存の視覚化ツールには以下のような問題点が存在する事が分かった。

1. 表示する大きさが、関数・行・変数という単位でしか表示することができないこと

2. 経過が保存されないことでの迷子問題が発生すること

3. 表示対象が多数存在する場合、ユーザが必要とする情報の選択が難しいこと

これらの問題点の 1 については、表示する大きさが問題となっている。しかしこの表示については、要するに、より細かいレベルで抽出可能であればよいということと同じと捕らえることができる。また、3 についても、ユーザが必要とする情報だけを抽出できないことが課題であるとわかる。よって、これらの問題点については、新しい抽出方法を取り入れることで解決する。

以下に例を挙げる。今、以下のような関数 tree_insert(表 2.2) があるとする。この時、例えば「if 文を表示して欲しい」というユーザの要求があったとする。以前であれば、if 文が出現する場所でしか表示 (表 2.3) することができなかったものが、本研究で提案する柔軟な表示方法では、表 2.4 のような if 文の全体を表示するような表示方法が可能である。

表 2.2: 既存ツールと本研究との表示方法の違いを説明するためのソースコード

```
 1  void tree_insert(int x){
 2      struct node *p, *f, *g, *gg;
 3      p = f = g = head;
 4      do{
 5          gg = g;
 6          g = f;
 7          f = p;
 8          if(x < p->data) {
 9              p = p->left;
10          }
11          else {
12              p = p->right;
13          }
14          if(p != null && p->left->red && p->right->red){
15              redden(p, f, g, gg);
16          }
17      }while(p != null);
18      p = (struct node *)malloc(sizeof *p);
19      p->data = x;
20      p->left = p->right = null;
21      if(x < f->data){
22          f->left = p;
23      }
24      else{
25          f->right = p;
26      }
27      redden(p, f, g, gg);
28  }
```

表 2.3: ソースコードから if 文だけを表示した結果 (既存ツール)

```
 8          if(x < p->data) {
14          if(p != null && p->left->red && p->right->red){
21      if(x < f->data){
```

表 2.4: ソースコードから if 文だけを表示した結果 (本研究で開発するツール)

```
 8          if(x < p->data) {
 9              p = p->left;
10          }
11          else {
12              p = p->right;
13          }
14          if(p != null && p->left->red && p->right->red){
15              redden(p, f, g, gg);
16          }
21      if(x < f->data){
22          f->left = p;
23      }
24      else{
25          f->right = p;
26      }
```

また、

- ツールによって得られた知見を、他の開発者に伝えることができない

という問題もある。もし知見を伝承することができたならば、他ユーザはその知見を基に類似する要求を短時間で見つけることができる可能性がある。よって、その知見を本研究では一時的に「ツール使用のノウハウ」と呼んで、これを伝承することを機能として取り入れることとする。このツール使用のノウハウについては第 5.2 節で詳しく述べる。

つまり、既存ツールは同じソースコードにおいて複数の利用者間での知識の共有を想定してないということが問題であると言える。これは、抽出のための操作が複雑な場合で同じような抽出要求があった際には、再び同程度の複雑な操作をしなければならないことを示している。これでは、仮に抽出する新しい方法をとりいれたとしても、ユーザはツールの操作方法の複雑さから、目的となる部分を抽出をすることができなくなり、ツールを使用しなくなってしまう。よって本研究では、利用者の以前の操作のノウハウを保存してお

き、これを次の利用者に提示し、その利用者の抽出要求をする際の基準とする。この機能を取り入れることは、他のユーザに対しての理解支援はもちろんのこと、同じユーザが時間が経過した後にツールを使用して再び同じような抽出を行う場合においても有効な方法となりえる。この機能を取り入れることでこの問題を解決する。

以下に例を示す (図 2.4)。この例では、ユーザ B があるソースコードのある部分を抽出したいと思い、ツールを操作する。試行錯誤の結果、その要求にあった部分が抽出できたとする。その場合、ユーザ B のツールに対する試行錯誤の結果 (ノウハウ) を次にツールを使用するユーザ C に伝える。これにより、ユーザ C はユーザ B の結果を 1 つの例として、より少ない操作で自分の要求する情報を引き出すことができる。

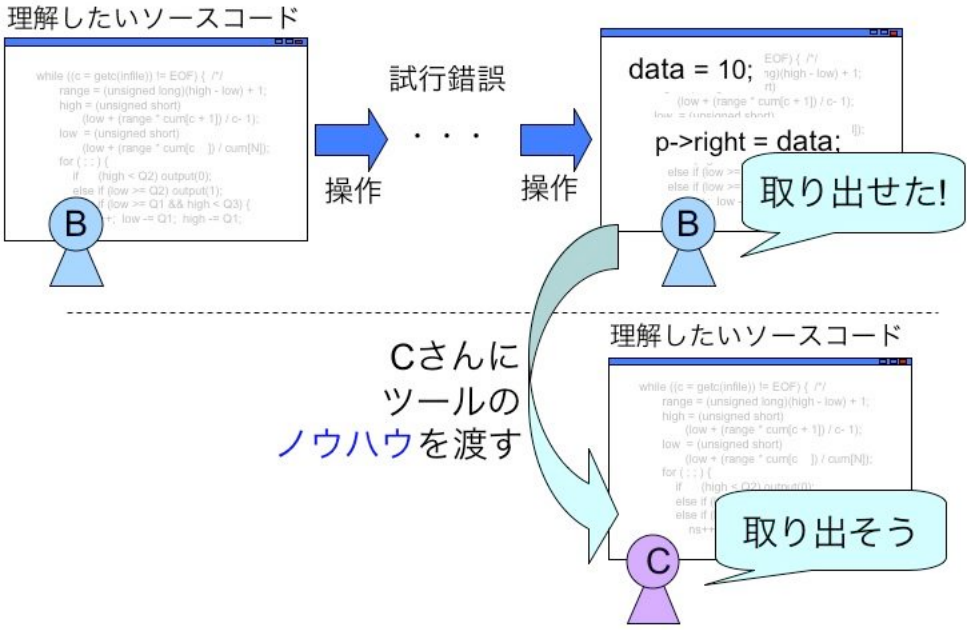

図 2.4: 他者に知見を渡す例

第3章　視覚化ツールの要求仕様

本章では、本研究で作成するツールの概要について紹介する。ここで挙げる技術を用いて、表示が柔軟な視覚化システムを作成する。視覚化対象言語は C 言語および C++言語である。

3.1　要求

要求として第一点目として考えられることは、

- 高い表示自由度

である。これは、表示を細かく制御できることという意味である。この要求を実現するために取る方法としては、言葉通り表示することに関して自由度を高くする方法と、抽出する段階で情報を選別し結果として表示の自由度を高くする方法の 2 点が考えられる。本研究では、この高い表示自由度実現のために、後者からの観点で開発を行う。

つまり、柔軟な表示をするために、本研究では様々な抽出方法で表示を制御するということである。表示を行うとは、抽出された結果をどのように表すか、という意味で表示処理を行うことである。高い表示自由度とは、例えば、あるソースコード中の「for 文の中にある if 文について抽出を行いたい」という要求があった場合、今まではそのような抽出ができなかった。既存の抽出法であるクロスリファレンサにおいては言うならば、「if 文について抽出を行いたい」という単純な抽出でしかなく、更にその抽出は文字列に対して行うというものであった (この場合であると i と f が連続して現れる箇所を抽出するというもの)。よって、より柔軟に抽出ができる方法が本研究のツールには要求される。

上記の表示自由度を高くしたことにより、ツールはより高い表示方法をもつことができた。しかし、その表示をするための操作が複雑になると、逆にユーザの要求となる箇所を表示させる方法が難しくなる。よって、その操作の煩雑さを解消するためのツール対する要求として、

- ツール使用のノウハウの保持および提示

が挙げられる。これは、既存ツールの問題点と解決法 (第 2.2 節) の二点目に挙げたものである。この機能を取り入れることにより、次のユーザはより容易に自身の要求する箇所を見つけることができる。

これは実際に、操作の仕方を他者にツールを操作する時に立ちあって教えることができれば、そのツール使用のノウハウを保持する必要はない。しかし、実際にその場に立ちあうことは難しい。よって、他ユーザにノウハウ (第 5.2 節参照) を伝承する必要があるのであれば、それを記録し保持する必要がある。そして、それを記録した後に同じような要求があった場合には、そのノウハウを提示する機能が要求として考えられる。

この 2 点を取り入れることが本研究では開発する理解支援ツールの要求としてあげられる。

3.2 解決の方針

本研究で開発する視覚化ツールは、「解析・抽出・表示」の 3 段階から構成される。

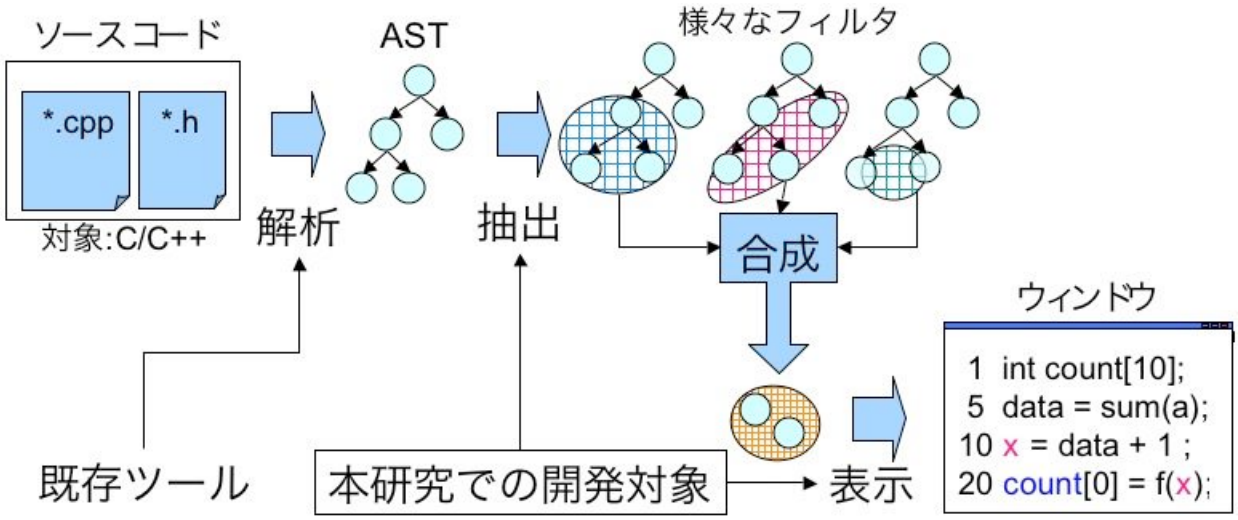

図 3.1: 本研究で作成するツールの全体構成

まず、既存ツールを使用し理解したいソースコードの構文・意味解析を行う (図 3.1 参照)。解析対象言語は C 言語および C++言語である。このステップは独自に開発を行うことはせず、既存にある Sapid[10] を使用して行う。これは、このツールが本研究で行う理解支援ツールにおいて必要な情報を全て保存する機能を備えていることが理由である (第 4 章参照)。

次に、ソースコードを解析した結果である抽象構文木 (以下 AST) から抽出を行う。このステップでは様々なフィルタを使用し、複数のフィルタによって抽出された情報を合成する。そして、最終的に抽出する情報を決定することを行う (第 5 章参照)。

最後に、抽出された情報を表示する。このステップは、抽出された情報を全てユーザに提示し、その提示方法を変えることを行う。これは、抽出された情報をどのように表すかを示しており、抽出された情報についての選択は行わない。つまり、抽出された情報は

全て画面に表示を行い、その表示方法だけを変えることである。情報の表示するしないはこのステップでは考慮しない。表示方法とは例えば、文字を着色することや、Fish-Eye View[11] などの表示方法などを指している (第 6 章参照)。

また、本研究の開発する範囲は解析結果の抽出から情報の抽出までである。

3.3 本研究が抽出および表示する対象範囲

本研究が対象としている範囲を述べる。以下が既存ツールの抽出できる範囲と本研究のツールが抽出できる範囲とを合わせた図である (図 3.2)。そして、この図が本研究の提案するフィルタの抽出できる範囲を示す。

まず、図 3.2 中の左の円について説明する。これは、クロスリファレンサにより抽出できる範囲を示している。クロスリファレンサはソースコードの文字列を抽出対象とし、対象とする文字列を含む行毎に出力することができる。例を挙げる。あるソースコード (表 3.1) があり、ユーザの注目している部分が変数 count だとする。その場合、クロスリファレンサは文字列 count について抽出を行い、その結果は「1,20,30 行目」である。

次に、図 3.2 中の右の円について説明する。これは、スライサにより抽出できる範囲を示している。スライサはプログラムの意味を解析する技術であり、対象となる変数を操作する箇所を全て出力することができる。先ほどのソースコード (表 3.1) を例として、スライサの出力結果を示すならば、「1,10,11,20,30 行目および 10,11 行目の右辺を定義している部分全て」が抽出することができる。この場合であると、10 行目の右辺にある data および 11 行目の右辺にある buf を定義している部分が抽出されることになる。

そして、図 3.2 のクロスリファレンサとスライサによる範囲を示している部分を内包し四角で囲まれた部分が、本研究で作成するフィルタが抽出できる範囲を示している。本研究のフィルタは意味情報に基づく抽出を行い、ソースコード (表 3.1) の例から抽出範囲を示すと、「1,10,20 行目および 10 行目の右辺を定義している部分の内、指定された範囲」が抽出することができる。この”範囲”については第 5 章で説明を行う。

表 3.1: 抽出対象となる行

```
 1  int count[10];
   ...
10  x = data + 1;
11  y = buf;
   ...
20  count[0] = f(x);
   ...
30  count[1] = g(y);
```

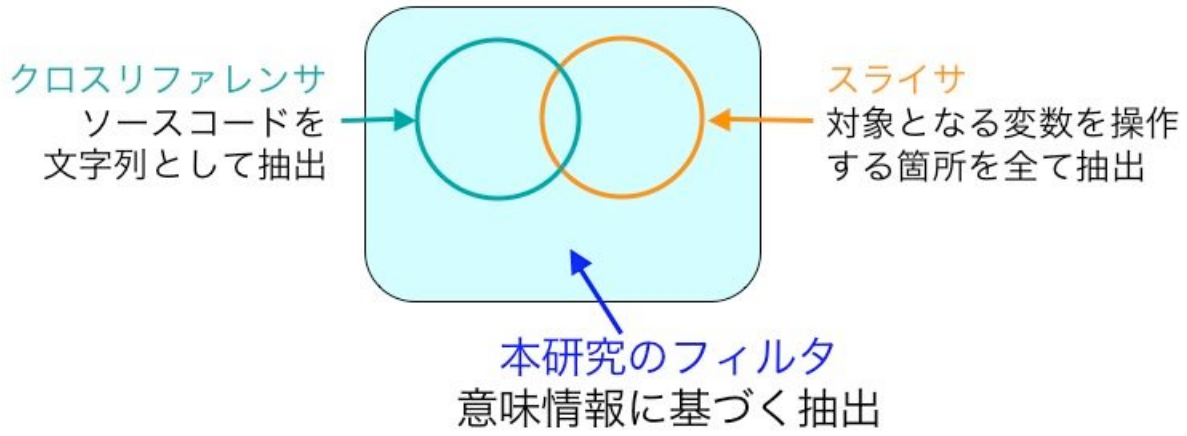

図 3.2: 本研究の抽出および表示する範囲のイメージ

3.4 ユーザサイド

ユーザと本研究の視覚化ツールとの関わり合いについて記す。

ユーザは解析時には、ソースコードの解析をツールに指示する。ソースコードは理解したい対象であるので、ここでは既に決まっているものとしてある。

次にユーザが抽出を制御する際について関わる点は 2 点ある。1 点目は、ソースコードの解析結果からどのような情報を抽出するかを決める点である。もう 1 点は、返された結果を見る点である。以下に簡単なユーザとツールの抽出の順序を示す。

1. User:要求する視点や目的などの抽出基準をツールに指示
2. Tool:その要求に対する抽出結果を返す
3. User:その結果を表示するようにツールに指示
4. Tool:その抽出結果を返す

4 で返された結果を変更したいのであればまた 1 に戻り、抽出基準を変更してさらに抽出を行う。このようにな操作を繰り返し行い、ユーザはソースコードの理解を深めていく。

例 (図 3.3) として、ユーザが制御文 if について条件文だけを表示対象として抽出を行い、then 部および else 部は抽出しないという基準を設ける。するとツールはユーザの指示どおり if 文の条件文だけを抽出した表示 01 を行う。しかし、ユーザは条件文だけでは理解することができなかったため、今度は if 文の条件文だけでなく then 部 else 部も表示をするように指示をする。ツールはまた表示を更新し、ユーザに提示する (表示結果 02)。しかし今度は、then 部 else 部の実行部が長過ぎて理解が難しくなった。よって、今度は then 部 else 部の実行部の変数 right が使われている行だけを表示するよう、ユーザはツールに指示をする。すると、次に表示された結果は適度な情報量であってユーザにとって見やす

い表示であった。このように、ユーザはツールに対して抽出基準の決定と閲覧を繰り返し行うこととなる。

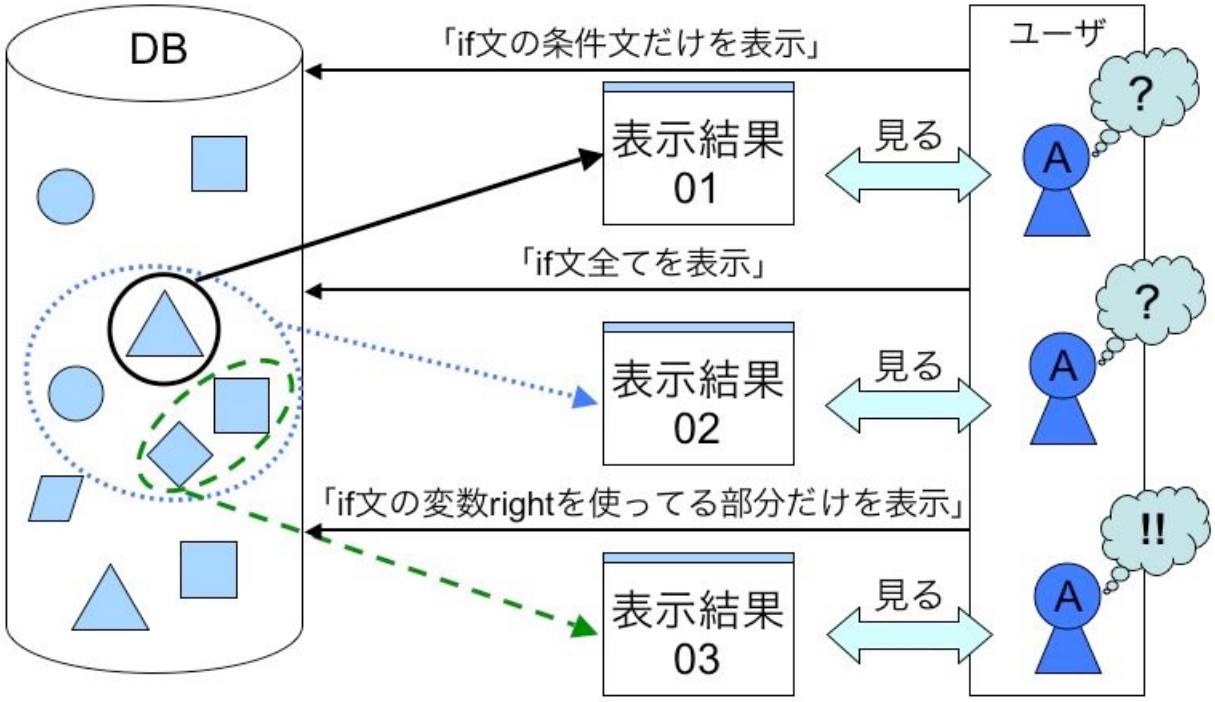

図 3.3: ユーザの表示要求の例

最後は、表示の段階であるが、この時には抽出結果があるので、その結果を表示するようにツールに指示をする。

第4章　解析

この章では、本研究で使用する既存解析ツールの説明とそのツールを使用して解析した例をあげる。第 3 章、3.2 節でも書いたが、ソースコードの解析器の開発は、本研究では行わない。これは、視覚化ツールにとって解析は必須であるもののそれの作成は開発者に労力がかかり、かつ理解支援にとっては本質的なものでない段階であるためである。

4.1　概要

本研究では解析器として Sapid[10] を用いる。このツールは、現在も開発されている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である。

Sapid は、ソフトウェアデータベース (SDB:Software Database)、アクセスルーチン (AR:Access Routines)、ソフトウェア操作言語 (SML:Software Manipulating Language) から構成される。SDB は Sapid の基盤であり，要求されたソフトウェアをソフトウェアモデルに基づいて解析する解析器と解析の結果得られた実体・関連情報を格納するデータベースからなる。AR は SDB に対するアクセス機能を提供するとともに、ソフトウェア開発において多用される前処理に対応するするための PIDB(PIDB: Preprocess Information Database) を含んでいる。SML はソフトウェアに対する各種の操作を簡潔で直観的にわかりやすく記述するための言語である。クロスリファレンサなどの比較的単純な CASE ツールについては AR を用いて実現し、複雑な CASE ツールは SML を用いて実現する。

Sapid の大きな特徴の 1 つとしては、上記でも挙げたが、細粒度リポジトリに基づくことが挙げられる。細粒度とは、ソースコードを行 1 つやコメント文 1 つ 1 つまでも保存することであり、リポジトリとはその保持された情報を管理する仕組みのことを言う。この細粒度リポジトリは可能な限り細かい情報を保持することを求められるが、細かすぎる粒度は管理するオブジェクトが増加することなどの弊害も生ずる。このツールでは C 言語の構文規則に基づいて作られ、かつある程度の細かい粒度を持つ I-model と呼ばれるモデルを使用することで、その双方のバランスをとっている。このモデルは、構文構造と静的意味の情報をファイルに格納し、ソフトウェアのライフサイクルにおいて重要でないものを切り捨てた簡潔なモデルになっている。

4.2 I-model

I-model はソフトウェアの構造を基にソフトウェアを捉えたモデルであり、プログラマからの視点に重点を置いている。ただし、C 言語の文法をそのまま構造として用いたわけではなく、プログラマの視点から抽象化を図り、必要ないと思われる情報を削除している。また、関連の多くは構成関連として捉えることが自然であるため、構成関連を的確に表現するモデルとして、OMT モデルのオブジェクト図の記法を採用している。

I-model の例として、ある構造体をモデル化したグラフを図 4.1 に示す。この図は tree.c という C 言語で記述されたソースコードの宣言部の一部である「struct node *p;」をモデル化したものである。モデル図の declaration というノードから、解析された結果が有向グラフとして下につながっている。宣言「struct node *p;」はデータ型と変数から成っているので、declarator(宣言子) として「node」と「*p」がその下に続く。データ型「node」はタグであるので declarator での sort 名は「DECL_TAG」という名が付けられ、最終的な型の種類としては「TYPE_STRUCT_TAG」という sort 名としてみなされる。また、変数「*p」それ自体は変数 (DECL_VARIABLE) として扱われ、その変数を表す文字そのものである「p」は ID_VARIABLE となり、その変数の型の種類はポインタなので最終的に「TYPE_POINTER」となる。

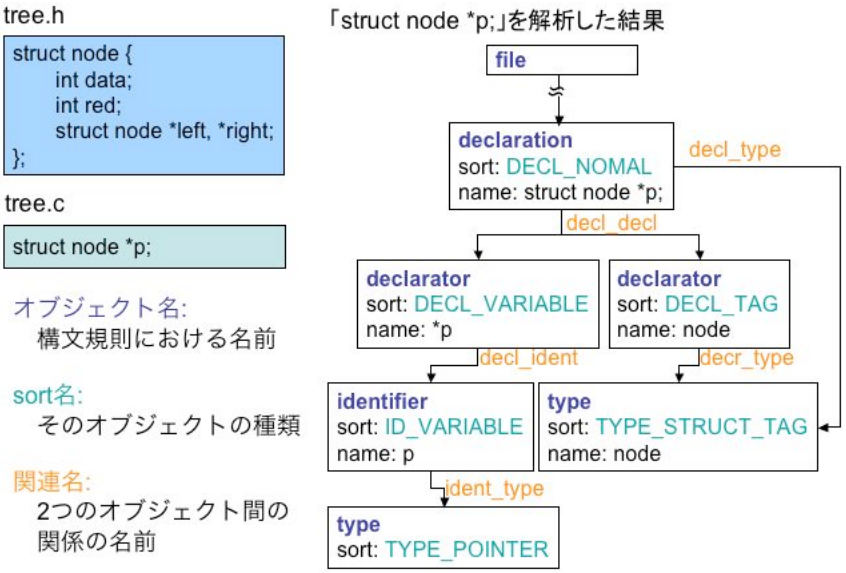

図 4.1: I-model の例 (struct node *p; をモデル化した場合)

4.3 本研究での使用範囲

4.1 節でも記述したが、Sapid は SDB,AR,SML から成っており、本研究では解析にコマンド *sdb* を使用し SDB(Software Database) を作成した後、AR を使用してその解析結果から抽出を行う。

Sapid はソースコードを細粒度で静的に解析し、コメントを保持することができる。このコメントは理解支援にとっては有用であり、この情報を保持できることは非常に価値のあることである。また、解析結果を XML 形式でリポジトリに格納することもできる。XML 形式で格納する理由としては、二次利用の容易性が挙げられる。一般に、解析ツールから得られる情報は、メモリ上に記憶されるため一時的なものでしかなく、特定のアプリケーションでしか使用することができない。よって、今後の研究のことも考慮し、XML を用いてプログラム解析情報のデータベース化を行う手法をとる。

第5章　抽出

本章の抽出は、本視覚化ツールの構成において解析と表示の間にあたる。ここでは、ソースコードの情報量と複雑さをどのように制御するかについて説明する。

5.1　フィルタの作成

まず、第3.3節の最後で述べた”範囲”という言葉について説明をする。”範囲”とは、フィルタにより抽出されるソースコード中の複文のことであり、このフィルタにより抽出された複文を本研究では「意味ブロック」と定義する。そして、ユーザはフィルタを組み合わせることによりユーザに必要となる情報の抽出を行う。例として図3.1(15ページ)において言うならば、ASTからフィルタにより抽出されたものが意味ブロックであり、複数の意味ブロックを合成して最終的な抽出結果を出すことがフィルタの組み合わせということになる。

そして、この意味ブロックをソースコード中から決定する基準としては以下の5点がある。

1. 制御文
2. 関数の宣言部、実行部
3. 複文中の変数の出現範囲
4. 代入文の右辺に出現する式
5. 着目行の近傍

これらのうち、構文木と対応して抽出できるものは1〜4であり、5点目の「着目行の近傍」については構文木と対応して抽出は行うことができないものとして分類することができる。

また、意味ブロックの最小単位をI-modelのexpressionにする。このexpressionは標準的な言語でいうところのstatementである。例としてC言語では

「statement = expression + ; (セミコロン)」

である。この意味ブロックと I-model との関係として、以下の図 5.1 を例として挙げる。この図 (5.1) は左側の if 文をモデル化したものである。通常 if 文は条件部・then 部・else 部から成り、「条件部=expression、then 部 else 部=statement」である。しかし、上でも言ったように C 言語では expression と;（セミコロン）で statement であり、I-model も C 言語に準拠しているので、条件部も statement になる。よって、条件部・then 部・else 部全てが意味ブロックとなりえる。

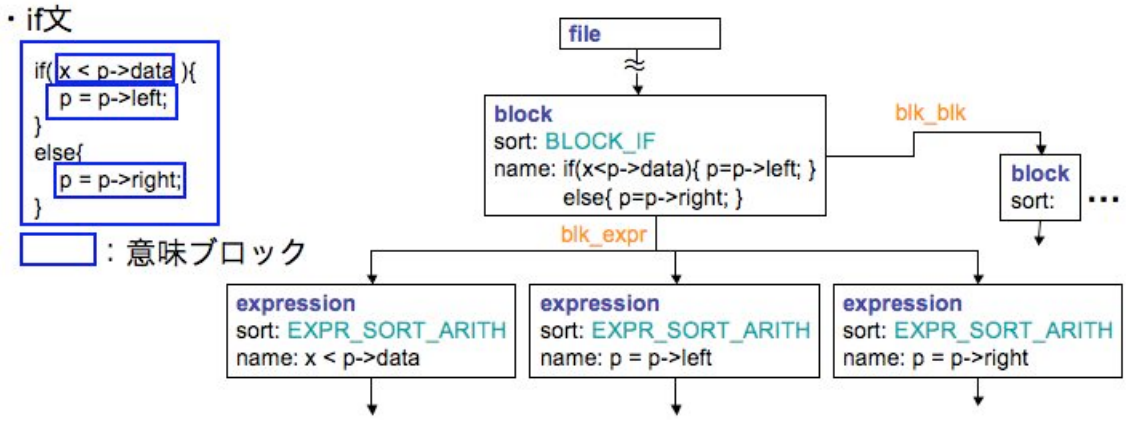

図 5.1: I-model と意味ブロックとの関係

また、上で述べた意味ブロックの抽出の仕方より分かることではあるが、意味ブロックはネストすることが可能である。これにより、ユーザの多様な要求に対しても柔軟に対応することが可能になる。

以下の節で上記の意味ブロックを抽出する 5 つの基準について説明していく。

5.1.1 制御文

まず、抽出基準 1 の「制御文」についてである。この基準で意味ブロックを抽出するということは、構文木上の特定の関係にある構文要素を取り出すことを示しており、ここで言う制御文とは if,while などの制御構文のことを指している。その制御構文として抽出基準に挙げられるものは以下の、

if, while, switch, for

の 4 種類とする。

以下に、制御文で抽出することの例を示す (図 5.2)。図中左がソースコードの例であり、ユーザから「for 文中の if 文のみを抽出したい」という要求があった場合、3,4 行目が抽出されることになる。6 行目の if 文は for 文の中でないので抽出されない。また、右が構文木のイメージを示した図であり、「for 文中の if 文」という要求であるので図中の for の下に位置する if が意味ブロックとなる。

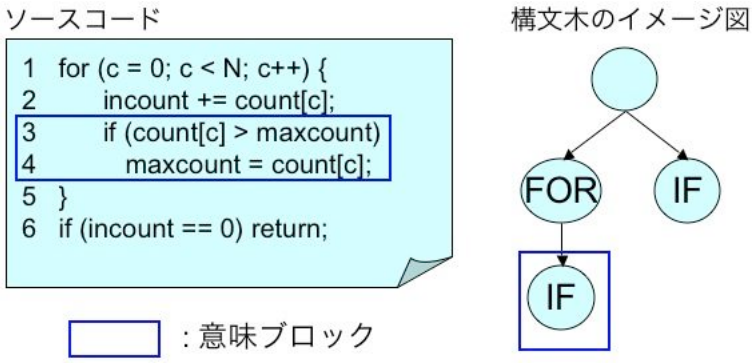

図 5.2: 「for 文の中にある if 文のみ」を抽出する例

5.1.2 関数の宣言部と実行部

次に、抽出基準 2 の「関数の宣言部, 実行部」について述べる。これは、変数などの宣言をしている部分を宣言部、処理を行っている部分を実行部として抽出を行うことである。宣言部は、基本的に値の変更は行われないので、処理とは切り離して抽出することができる。また、C 言語において宣言部は関数の最初においてのみ出現するが、C++言語においては関数中の複数の場所に出現する可能性があるため、この抽出法を取り入れた。

この基準でソースコードから意味ブロックを抽出する例を載せる (表 5.3 参照)。今、「宣言部を抽出したい」というユーザの要求があった場合、宣言部である 1 行目、および 3 行目が抽出される。構文木のイメージとしては構文要素 dec(declaration) の部分が意味ブロックとして抽出される。

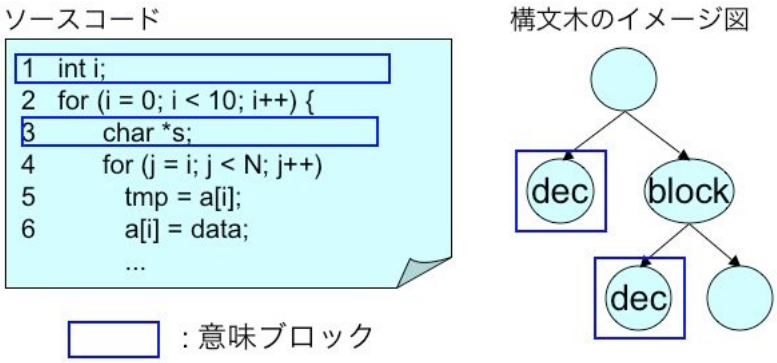

図 5.3: 「宣言部」を抽出する例

5.1.3　複文中の変数の出現箇所

次に抽出基準の 3 について説明する。これは、ある変数に注目した場合に、その変数が出現する箇所で抽出するというものである。変数の中には、関数中である一定の箇所でしか出現しないものが存在する。よって、その変数が使われている行だけを抽出し、それらを 1 つの意味ブロックとして抽出できる。しかしこの抽出基準は、変数によって関数全体で使われているものもあるので、一概に全ての変数に適用できるというものではない。また、この抽出法は grep などを代表とするクロスリファレンサと同じ手法であり、本研究が新しく提案する抽出法ではない。

この抽出法の例を以下に載せる (図 5.4)。この例では、10 行目にある変数 data についてユーザが注目しており、「変数 data についてこの抽出基準により抽出を行いたい」とする。この場合、抽出できる箇所は変数 data が出現する最初である 6 行目から最後に出現する 11 行目までの出現箇所である。この箇所が意味ブロックをして抽出される。構文木のイメージにおいては、変数 data に対応する構文要素 var を含んだ木が条件に対応するものとして抽出される。また、この例における変数 index のように、ソースコード中のほぼ全域に出現するようなものに対しては、この方法による抽出はあまり意味をなさない。

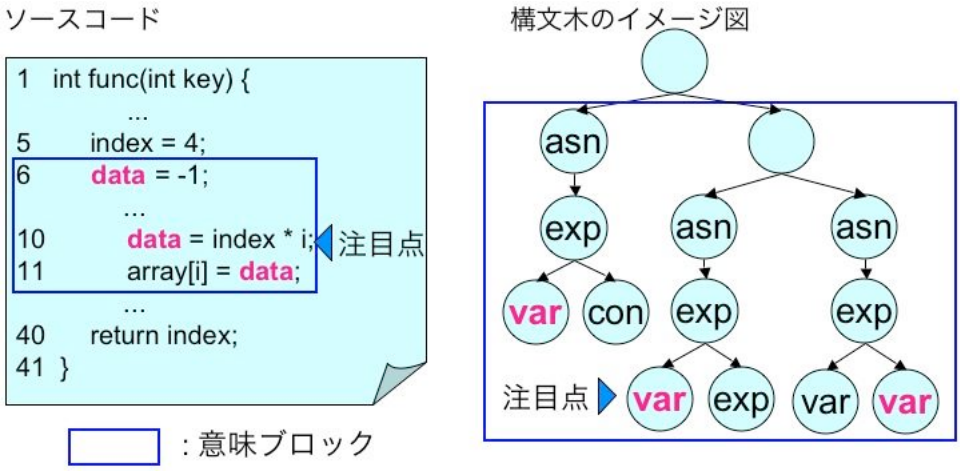

図 5.4: 「変数 data の出現箇所」で抽出する例

5.1.4　代入文の右辺に出現する式

4 点目に、「代入文の右辺に出現する式」について説明する。これは、構成要素に対応する箇所を抽出することを示している。注目している文の右辺を構成する関数および変数を定義している部分を、意味ブロックとして抽出する。そしてこの抽出法は、この定義部をどの深さまで抽出範囲に加えるか、および関数の定義部を抽出するか否かを設定することができる。つまり、一言で言うならばスライシングの段階的実現による抽出法。

この抽出法で抽出した結果の例が図 5.5 である。この例では、ユーザが「変数の定義部の深さを 1、関数の定義部は抽出しない」という要求であった場合に四角で囲った部分が抽出される。ユーザが 21 行目に出現する変数 p について、この方法により抽出を行うとすると、変数 p を定義している 10 行目がまず 1 つ目の意味ブロックとして抽出できる。次にその 10 行目の右辺である $f(x) + v$ の変数 v の定義部が抽出される。この場合であると 2 行目である。構文木のイメージ図としては、一番右下にある構文要素 exp(expression) が変数 v であり、その定義部である中段真ん中にある変数 v の定義部 exp が意味ブロックとして中段されることとなる。

今回、ユーザは変数の定義部の深さを 1 として設定してあるが、ここで深さを 2 とした場合には、2 行目の右辺を定義している部分の変数についても抽出が行われる。また、関数定義部も抽出するという設定であれば、$f(x)$ の変数 x の定義部である 4 行目も抽出されることとなる。

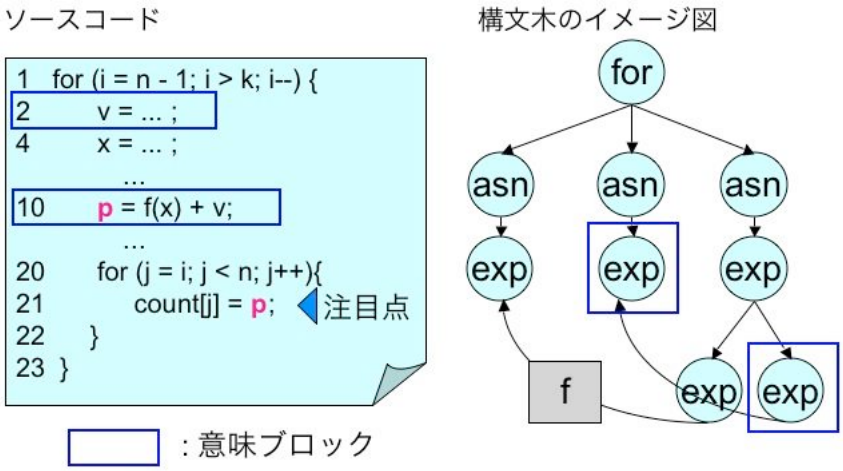

図 5.5: 「依存深度 1, 関数定義含まず」で抽出する例

5.1.5 着目行の近傍

意味ブロックの抽出基準の最後である、着目行の近傍について説明する。この抽出法は、ソースコード中の注目点から上下の行に必要な情報を含むように抽出することを指している。要するに、見ている行の前後に視野を広げるということである。この抽出法が有効となる場合は、ソースコード中の物理的に近い位置に必要な情報が存在する場合である。よって、この抽出法に関してのみ、構文木とは対応しないものとして分類している。しかし、厳密に言えばこの抽出法も構文木より抽出が可能である。これは解析に使用した Sapid が行の情報も持つ AST を作成するためである。

以下に例を挙げる (図 5.6)。今、ユーザは 5 行目に注目をしており、「変数 count につい

て上下に視野を広げる」という要求であるとする。その場合、変数 count を定義している 4 行目から、その変数が出現している 8 行目までを意味ブロックとして抽出を行う。

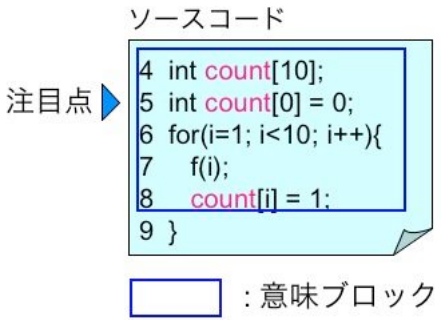

図 5.6: 「変数 count の着目行の近傍」で抽出する例

5.2 ノウハウの伝承

第 2.2 節で述べた”ノウハウ”とは、上記であげた 5 つの抽出法で使用されるパラメータのことである。このパラメータを保持し次の利用者に提示することで、理解の伝承を行う。つまり、第 2.2 節で挙げた図 2.4(13 ページ) のユーザ B がユーザ C に渡すものは、フィルタの作り方と組み合わせ方、つまり「意味ブロックを決めるパラメータ」のことである。そして、本研究での抽出の柔軟さは、このパラメータを組み合わせることにより実現するものである。また、ここでの”組み合わせ”は、論理和 (or) もしくは論理積 (and) での組み合わせを言う。

以下にそれぞれのフィルタに使用するパラメータを載せる (表 5.1)。

表 5.1: パラメータの一覧

filter	parameter
制御文 (5.1.1 節)	親または子の構文要素:element
関数の宣言部, 実行部 (5.1.2 節)	宣言部 / 実行部
複文中の変数の出現範囲 (5.1.3 節)	変数:var
代入文の右辺に出現する式 (5.1.4 節)	関数定義を [含む:TRUE / 含まず:FALSE] 追跡する範囲:n
着目行の近傍 (5.1.5 節)	着目行:row 前に b 行 後に f 行

5.3 抽出の制御例

この節では、上記で挙げたフィルタを用いてソースコードよりユーザの要求する部分を抽出する例を示す。まず、表 5.2 に理解したいソースコードを載せる。この関数の全てを理解する手順をこの論文に書くことは無意味であるので、ここではある変数に注目している場合をとりあげ、その変数について理解する手順の一部を以下で説明することとする。

例えば、ユーザはこの関数中の 48 行目で標準出力されるに変数 incount について知りたいとする。まず、ユーザはこの変数 incount の出現範囲について知りたいとする。その場合、
ステップ 01:(01)

- フィルタ「複文中の変数の出現範囲」、パラメータ「var=incount」

という風に使用するフィルタおよびパラメータを設定する。これにより、抽出された結果が表 5.4 になる。変数 incount が出現している行が全て抽出されているのがわかる。ここで、ユーザは抽出された部分の 12 行目 *incount += count[c]* を見て、現在注目している変数 incount は変数 count により書き換えられれていることがわかり、次に変数 count について見てみたいとする。よって、今度は先ほどのフィルタを
ステップ 02:(01 → 02)

- フィルタ「複文中の変数の出現範囲」、パラメータ「var=count」

と設定する。すると、表 5.5 が出力される。しかし、次に見てみたい変数 count の書き換えられることのない行まで抽出されてしまったので、ステップ 01 の設定に戻り、今度は変数 count の定義されている部分を抽出するようにする。その場合、
ステップ 03:(01 → 03)

- フィルタ「複文中の変数の出現範囲」、パラメータ「var=incount」
- フィルタ「代入文の右辺に出現する式」、パラメータ「関数定義:含まず、追跡範囲:1」

というもう 1 つの別のフィルタをかける設定にする。その抽出結果は変数 count の定義部が出現している表 5.6 になる。これにより、変数 count の書き換えている箇所はわかった。しかし、この抽出において、変数の型定義部は必要ないと思い、今度はユーザは宣言部を抽出しないようにする。その設定は、
ステップ 04:(01 → 03 → 04)

- フィルタ「複文中の変数の出現範囲」、パラメータ「var=incount」
- フィルタ「代入文の右辺に出現する式」、パラメータ「関数定義:含まず、追跡範囲:1」
- フィルタ「関数の宣言部、実行部」、パラメータ「実行部」

となる。この結果、表 5.7 が表示される。これにより、注目していた変数 incount の書き換えられている箇所のみを抽出することができた。しかしこの時更に、ユーザは変数 count の右辺を構成する変数 d について知りたいと思い、右辺に出現する式の変数の追跡範囲を一段階増やす。
ステップ 05:(01 → 03 → 04 → 05)

- フィルタ「複文中の変数の出現範囲」、パラメータ「var=incount」
- フィルタ「代入文の右辺に出現する式」、パラメータ「関数定義:含まず、追跡範囲:2」
- フィルタ「関数の宣言部、実行部」、パラメータ「実行部」

この結果、表 5.8 という範囲が抽出される。

このように、ユーザはフィルタを変更しながら抽出を繰り返し、理解を深めていく。この際、読まなければならない行数の視点からすると、関数の総数 52 行から、ステップ 05 までに読んだ全ての行は 12 行であり、最終的には 9 行程度を見ればいいことになる。よって、行を読む量が最終的には 2 割弱、途中で読んだ行の割合としても 2 割強ということになる。この割合は関数または要求により様々であるが、本研究のフィルタの組み合わせにより、読むコード量の減少という点においても、理解支援をすることができたと言える。

このように、情報を取捨選択していき、抽出を制御することを「抽出の制御」と言う。

表 5.2: 理解対象のソースコード

```
1    void encode(void)  /* 圧縮 */
2    {
3        int c;
4        unsigned long range, maxcount, incount, cr, d;
5        unsigned short low, high;
6        static unsigned long count[N];
7
8        for (c = 0; c < N; c ++) count[c] = 0;  /* 頻度の初期化 */
9        while ((c = getc(infile)) != EOF) count[c]++;  /* 各文字の頻度 */
10        incount = 0;  maxcount = 0;  /* 原文の大きさ, 頻度の最大値 */
11        for (c = 0; c < N; c ++) {
12            incount += count[c];
13            if (count[c] > maxcount) maxcount = count[c];
14        }
15        if (incount == 0) return;  /* 0バイトのファイル */
16        /* 頻度合計がQ1未満, 各頻度が1バイトに収まるよう規格化 */
17        d = max((maxcount + N - 2) / (N - 1),
18                (incount + Q1 - 257) / (Q1 - 256));
19        if (d != 1)
20            for (c = 0; c < N; c ++)
21                count[c] = (count[c] + d - 1) / d;
22        cum[0] = 0;
23        for (c = 0; c < N; c ++) {
24            fputc((int)count[c], outfile);  /* 頻度表の出力 */
25            cum[c + 1] = cum[c] + (unsigned)count[c];  /* 累積頻度 */
26        }
27        outcount = N;
28        rewind(infile);  incount = 0;  /* 巻き戻して再走査 */
29        low = 0;  high = USHRT_MAX;  ns = 0;
30        while ((c = getc(infile)) != EOF) {  /* 各文字を符号化 */
```

表 5.3: 理解対象のソースコードの続き

```
31              range = (unsigned long)(high - low) + 1;
32              high = (unsigned short)
33                     (low + (range * cum[c + 1]) / cum[N] - 1);
34              low  = (unsigned short)
35                     (low + (range * cum[c    ]) / cum[N]);
36              for ( ; ; ) {
37                  if      (high < Q2) output(0);
38                  else if (low >= Q2) output(1);
39                  else if (low >= Q1 && high < Q3) {
40                      ns++;  low -= Q1;  high -= Q1;
41                  } else break;
42                  low <<= 1;  high = (high << 1) + 1;
43              }
44              if ((++incount & 1023) == 0) printf("%12lu\r", incount);
45          }
46          ns += 8;  /* 最後の 7 ビットはバッファフラッシュのため */
47          if (low < Q1) output(0);  else output(1);  /* 01 または 10 */
48          printf("In : %lu bytes\n", incount);  /* 原文の大きさ */
49          printf("Out: %lu bytes (table: %d)\n", outcount, N);
50          cr = (1000 * outcount + incount / 2) / incount;  /* 圧縮比 */
51          printf("Out/In: %lu.%03lu\n", cr / 1000, cr % 1000);
52      }
```

表 5.4: 抽出例 01:「複文中の変数 incount の出現範囲」で抽出

```
4           unsigned long range, maxcount, incount, cr, d;
```

```
12              incount += count[c];
```

```
18                  (incount + Q1 - 257) / (Q1 - 256));
```

```
28          rewind(infile);  incount = 0;  /* 巻き戻して再走査 */
```

```
44              if ((++incount & 1023) == 0) printf("%12lu\r", incount);
```

```
48          printf("In : %lu bytes\n", incount);  /* 原文の大きさ */
```

表 5.5: 抽出例 02:「複文中の変数 count の出現範囲」で抽出

```
6       static unsigned long count[N];
8       for (c = 0; c < N; c ++) count[c] = 0;  /* 頻度の初期化 */
9       while ((c = getc(infile)) != EOF) count[c]++;  /* 各文字の頻度 */
12          incount += count[c];
13          if (count[c] > maxcount) maxcount = count[c];
21              count[c] = (count[c] + d - 1) / d;
24          fputc((int)count[c], outfile);  /* 頻度表の出力 */
25          cum[c + 1] = cum[c] + (unsigned)count[c];  /* 累積頻度 */
```

表 5.6: 抽出例 03:抽出例 01 に「代入文の右辺に出現する式」のフィルタを加えた抽出

```
4       unsigned long range, maxcount, incount, cr, d;
6       static unsigned long count[N];
8       for (c = 0; c < N; c ++) count[c] = 0;  /* 頻度の初期化 */
9       while ((c = getc(infile)) != EOF) count[c]++;  /* 各文字の頻度 */
12          incount += count[c];
18              (incount + Q1 - 257) / (Q1 - 256));
21              count[c] = (count[c] + d - 1) / d;
28       rewind(infile);  incount = 0;  /* 巻き戻して再走査 */
44          if ((++incount & 1023) == 0) printf("%12lu\r", incount);
48       printf("In : %lu bytes\n", incount);  /* 原文の大きさ */
```

表 5.7: 抽出例 04:抽出例 03 に「関数の宣言部、実行部」のフィルタを加えた抽出

```
8         for (c = 0; c < N; c ++) count[c] = 0;  /* 頻度の初期化 */
9         while ((c = getc(infile)) != EOF) count[c]++;  /* 各文字の頻度 */
```

```
12            incount += count[c];
```

```
18                (incount + Q1 - 257) / (Q1 - 256));
```

```
21                count[c] = (count[c] + d - 1) / d;
```

```
28         rewind(infile);  incount = 0;  /* 巻き戻して再走査 */
```

```
44            if ((++incount & 1023) == 0) printf("%12lu\r", incount);
```

```
48         printf("In : %lu bytes\n", incount);  /* 原文の大きさ */
```

表 5.8: 抽出例 05:抽出例 04 に「代入文の右辺に出現する式」のパラメータを変えた抽出

```
8         for (c = 0; c < N; c ++) count[c] = 0;  /* 頻度の初期化 */
9         while ((c = getc(infile)) != EOF) count[c]++;  /* 各文字の頻度 */
```

```
12            incount += count[c];
```

```
17         d = max((maxcount + N - 2) / (N - 1),
18                (incount + Q1 - 257) / (Q1 - 256));
```

```
21                count[c] = (count[c] + d - 1) / d;
```

```
28         rewind(infile);  incount = 0;  /* 巻き戻して再走査 */
```

```
44            if ((++incount & 1023) == 0) printf("%12lu\r", incount);
```

```
48         printf("In : %lu bytes\n", incount);  /* 原文の大きさ */
```

第6章　表示

この章は、視覚化ツールを構成する「解析・抽出・表示」の3段階の最後であり、第5章で抽出された情報を表示する方法について述べる。本研究では「高い表示自由度」を実現する方法として、抽出段階で解析結果より複数のフィルタを組み合わせて抽出し、この表示ステップで表示するという実現方法を取っている。

6.1　表示の方法

今、本研究でとる表示方法としては、着色と文字サイズの変更のみである。Fish-Eye View[11] などの特別は表示方法は用いていない。これは、情報量の制御は抽出の段階で解決されているからである。

表示方法としてまず、複文中の変数の出現範囲により抽出された変数をボールドで表示する。また、代入文の右辺に出現する式により抽出された関数, 変数のそれぞれの文字色を変更することとする。つまり、今ユーザが注目している変数が太文字で表示され、それを定義している変数が着色されるというものである。

6.2　表示制御の例

以下の例は、第5.3節の例05で抽出された表5.8である。この抽出は

- フィルタ「複文中の変数の出現範囲」、パラメータ「var=incount」
- フィルタ「代入文の右辺に出現する式」、パラメータ「関数定義:含まず、追跡範囲:2」
- フィルタ「関数の宣言部、実行部」、パラメータ「実行部」

というパラメータで成り立っている。これを上記で記述した表示方法により表示を行うと、以下のようになる (表6.1)。まず、複文中の変数の出現範囲により抽出された変数:incountをボールドで表示する。次に、12行目の代入文の右辺に出現する変数countをそれぞれを着色し、その変数countを定義している21行目の変数dも着色する。この時、それぞれの色は違うものとする。

このように表示の制御をしていくことなり、この方法により抽出されたコードを見る際の見やすさが向上する。

表 6.1: 抽出例 05 を表示制御した例

```
8         for (c = 0; c < N; c ++) count [c] = 0;  /* 頻度の初期化 */
9         while ((c = getc(infile)) != EOF) count [c]++;  /* 各文字の頻度 */
```

```
12              incount  += count [c];
```

```
17        d  = max((maxcount + N - 2) / (N - 1),
18                  (incount  + Q1 - 257) / (Q1 - 256));
```

```
21                  count [c] = (count [c] +d  - 1) /d ;
```

```
28         rewind(infile);  incount  = 0;  /* 巻き戻して再走査 */
```

```
44             if ((++incount  & 1023) == 0) printf("%12lu\r", incount );
```

```
48         printf("In : %lu bytes\n", incount );  /* 原文の大きさ */
```

第7章　理解支援ツールの設計と実装

7.1　方針

本研究では、理解対象ソースコードはC/C++言語とし、Java言語によりツールの実装を行う。

図7.1は本研究で作成する視覚化ツールの全体像であり、ユーザの要求する箇所を表示するまでの流れは以下のように進む。

01:解析

ユーザの理解したいソースコードがまず存在する (図7.1中、左上)。これを解析するよう、ユーザがシステムに指示をする。するとシステムは、そのソースコードを解析器Sapidにより解析を行う。その結果をソフトウェアデータベース (SDB:Software Database) に保存する。このSDBは、解析の結果得られた実態・関連情報を格納するデータベースである。

02:抽出

抽出を行うフィルタのパラメータの流れは、ユーザからインタフェース部 (図中のGUI)→フィルタ作成部→フィルタ合成部→インタフェース部となり、最終的にユーザに抽出結果として文字列として返ってくる。

まず、解析が終わると、ユーザは表示したい情報を抽出するようにフィルタのパラメータを決定してシステムに指示をする。そのパラメータがユーザより送られてくると、システムはインタフェース部でそれを受ける (図7.1,2の矢印)。

次に抽出の指示を、そのパラメータと共に、インタフェースがフィルタ作成部に伝える。すると、フィルタ作成部は解析結果であるSDBの中から渡されたパラメータにあう情報を抽出する。このパラメータが複数であった場合、それぞれのパラメータに対応した情報を一つ一つ抽出することになる。

フィルタ作成部より抽出された情報をフィルタ合成部が受ける。この部分は抽出した情報、つまり意味ブロックが複数の場合においてのみ機能し、それらの組み合わせによる抽出を行うこととなる。その組み合わせは、ユーザによりフィルタ毎に論理和か論理積かを決められている。また、この時合成した結果をツール使用のノウハウとして、フィルタの合成記録データベースに記録する。

抽出の最後に、合成された抽出結果がインタフェースを通じてユーザに返される。ここで、抽出された結果をユーザが眺める。その際、余分な情報もしくは情報が不足している

状態であると判断した場合、フィルタのパラメータもしくは合成方法を変更して更に抽出の指示を行う。
　この流れをユーザは繰り返し、抽出する部分を決定する。

03:表示

ユーザの要求にあった箇所が抽出された場合、ユーザはシステムに抽出箇所の表示を指示し、最終的にシステムは抽出された情報をユーザに提示する。
　まず、表示処理をさせるために抽出結果となる文字列をシステムに送る。その文字列が送られてくると、システムはインタフェース部で受ける。
　次に表示選択部が抽出結果をインタフェース越しに受け取り、表示する方法にのっとりそれを処理する。現在、本研究では表示方法はフィルタにおいて 1 種類ずつであるが、その表示方法を採用しない場合には、その設定もこの表示選択部で行う。

　最終的に表示された結果が不満足なものであったなら、抽出もしくは表示から繰り返すこととなる。

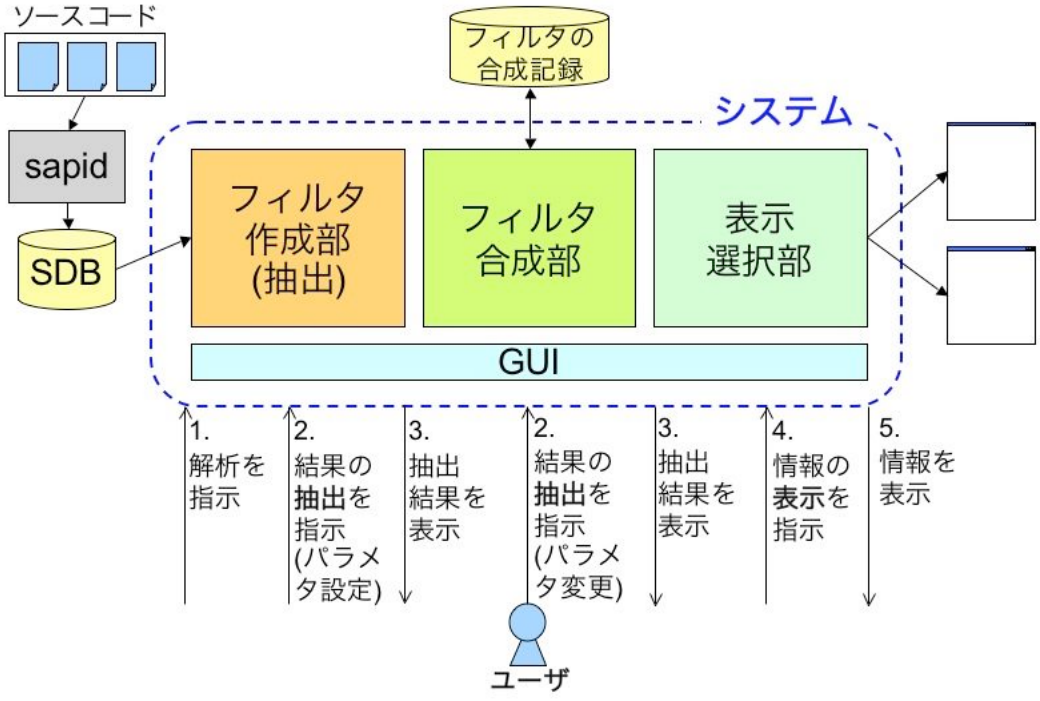

図 7.1: ツールの全体構成

7.2 設計

以下にユーザが抽出から表示までを行うユースケース図を載せる (図 7.2)。
要求箇所を表示する

1. ユーザがフィルタのパラメータを決める
2. ユーザが決めたパラメータで抽出を指示
3. システムが抽出結果を表示
4. ユーザが表示方法を決める
5. ユーザがその表示方法で表示することを指示
6. システムが結果を表示

バリエーション:抽出結果が不満な場合
手順 3 で、システムが表示した抽出結果が不満であった場合ユーザは再度手順 1 でパラメータを決定できる

以前にツールが使用されフィルタのパラメータが記録されている場合、ユーザはパラメータを決める際に以前のパラメータを参考にすることができる。

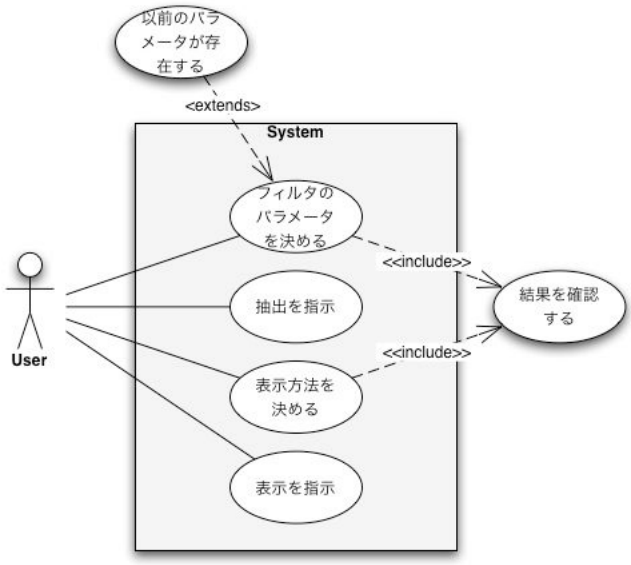

図 7.2: 要求された箇所を抽出・表示するユースケース図

7.3 実装

フィルタに使用するパラメータとその型を載せる。フィルタの使用は、そのフィルタ自体を使うかどうかを決めるための変数 state がありその型は Boolean である。この変数が TRUE になった場合にのみ、パラメータの値が有効になる。

表 7.1: パラメータとその型

	filter	parameter	type
1	制御文	状態:state 親または子の構文要素:element	Boolean char
2	関数の宣言部, 実行部	状態:state 宣言部:dec 実行部:exe	Boolean Boolean Boolean
3	複文中の変数の出現範囲	状態:state 変数:var	Boolean char
4	代入文の右辺に出現する式	状態:state 関数定義を含む:fanc 追跡する範囲:n	Boolean Boolean int
5	着目行の近傍	状態:state 着目行:row 前に b 行 後に f 行	Boolean int int int

またフィルタは論理積・論理和それぞれで ${}_5C_2$ 通りの組み合わせを持つため、それぞれについてフィルタ毎に組み合わせを設定できなくてはならない。以下にその設定の例を載せる (図 7.3)。5 種類のフィルタが格子状に並んでおり、項目が交差する部分のラジオボタンにチェックを入れるとフィルタリングが合成される。現在、この図では Filter の 1,3,4 が使用されており、この 3 種類の中での組み合わせを設定する状態であり、右上半分が論理積 (and) による合成、左下半分が論理和 (or) による合成となっている。この場合であると、Filter1 と 2 が論理積で、Filter1 と 4 および Filter3 と 4 が論理和で合成されている。

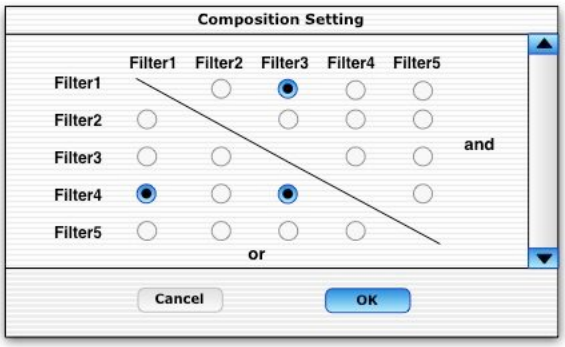

図 7.3: フィルタの合成を設定する GUI

7.4 評価

表示自由度については、行と関数の中間の表示単位として意味ブロックを定義し、クロスリファレンスとスライシングとの中間結果を表示できるようになった。更に、スライシングにおいては抽出の範囲を制御した。それらの複数のフィルタの組み合わせ (論理積もしくは論理和) を可能にしたことにより、表示自由度が高くなった。

理解支援については、意味ブロックを決める際の具体的なパラメータを決め、これを保持・提示することで他者に抽出法の例を示した。

第8章　終わりに

8.1　まとめ

本研究では、複数のフィルタを定義することにより、様々な粒度で情報を取り出すことに成功した。定義したフィルタの内の一部はクロスリファレンスやスライシングによるものであるが、それらの手法を改良し更に複数のフィルタの組み合わせにより、ソースコードの情報量と複雑さを制御可能となった。これによりユーザの視点に応じた柔軟な表示が可能となり、目的であった表示自由度の高い視覚化ツールを実現することができた。

理解支援という観点で本研究のツールを述べると、ツール使用のノウハウ、つまりフィルタのパラメータを記録し他者に提示することにより、自由度の高いツールにおいても要求に近いコード片を抽出することができた。この方法は、同じソースコードで他者と同程度の理解をしたい場合においても有効である。他者の抽出したパラメータを閲覧するだけであるので、より容易に実現できる。これにより、自由度の高さから発生する操作性の低下を防ぐことができ、また、他者との知見の伝承も行うことができ、理解支援という点においてもユーザの補助をすることができた。

8.2　今後の課題

本研究での表示方法は文字による簡単なものだけであったことより、先にも挙げたFish-Eye View[11]などの技術を用いた表示方法の拡張が必要だと思われる。本研究では抽出により情報量の制御をしたが、存在する様々な視覚化手法を適用することにより、表示の段階での情報の制御を行うことができる。よって、表示においても情報の制御ができるようになれば、抽出と表示の両方において表示自由度の高いツールとなる。

また、特定の状況に対応したフィルタの構成と組み合わせ（パターン）の抽出もこの視覚化ツールにより発見したい。これは、例えば「注目している変数が書き換えられている値の出現箇所」などという抽出を行うならば、その抽出の際に使用するフィルタとパラメータの組を発見したいということである。このパターンを発見することができたなら、更なるユーザの理解支援になるだろう。加えて、そのパターンはソースコード中のユーザの知りたい箇所、つまりユーザがソースコードを理解する上でわかりにくい箇所であることがわかる。よって、この視覚化ツールが、今までは要求を受けその要求に応じた箇所を表示するという消極的支援が、積極的にユーザの理解支援を行えることとなるだろう。

謝辞

最後に、本研究を行うにあたり、終始ご指導頂ました鈴木正人助教授に感謝申し上げます。時間のある限り研究について議論をしていただき、また研究以外に関しても熱心に様々なことを深い知識によりご教授いただきました。

また、研究の節目節目において、適切な助言を下さいました落水浩一郎教授に深く感謝致します。特に、高い立場より本研究の趣旨を的確に示して下さった時は、ある種の感動を覚えました。

そして、落水研究室の小谷正行氏にも日頃から様々な助言をいただくことによりスムーズに学生生活を送ることができたことを感謝致します。

並びに、大学時代に情報科学への道を示していただき数々の助言をも頂きました原田拓氏にも心より感謝致します。

最後に、進学への援助をしてくださった両親家族、そして私生活の面でお世話になった友人に感謝を申し上げます。

参考文献

[1] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone Trees, "Animated 3D visualizations of hierarchical information", In Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'91), pp. 189-194. ACM Press, 1991.

[2] B. A. Myers, "Visual programming, programming by example and program visualization", A taxonomy, In Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'86), pp. 59-66. ACM Press, 1986.

[3] Hideaki K., "Fractal Views: A Fractal-based method for Controlling Information Display", ACM Transactions on Office Information Systems, Vol. 13, No. 3, pp. 305-323, 1995.

[4] DDD - Data Display Debugger - GNU Project - Free Software Foundation (FSF), http://www.gnu.org/software/ddd/, Accessed 2005 Dec 19.

[5] GDB: The GNU Project Debugger, http://www.gnu.org/software/gdb/gdb.html, Accessed 2006 Jan 17.

[6] The dbx debugger, http://www.physics.utah.edu/ p573/hamlet/lessons/dbx/dbx/ dbx.html, Accessed 2006 Jan 30.

[7] The xdb debugger, http://docs.hp.com/en/5969-0905/ch01s04.html, Accessed 2006 Jan 30.

[8] GNU GLOBAL source code tag system, http://www.gnu.org/software/global/, Accessed 2005 Dec 19.

[9] developerWorks : Rational : Products : Rose, http://www-128.ibm.com/ developerworks/rational/products/rose/, Accessed 2005 Dec 19.

[10] Sapid Home Page, http://www.sapid.org/, Accessed 2005 Dec 19.

[11] G. W. Furnas : Generalized fisheye views, In Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'86), pp. 16–23. ACM Press,1986.