平成14年度学士論文

プログラムの意味構造と
相互の関連を意識した検索機能

東京工業大学 理学部 情報科学科
学籍番号 99-0553-0
沖 沙織

指導教官
千葉 滋 助教授

平成15年2月6日

概要

現在テキストエディタの検索機能や Unix の grep コマンドなど、 数多くの検索ツールが開発されている。これらの多くは文字列のパターンマッチを前提として開発されたもので、検索文字列の指定に正規表現を利用することができる。ところがこれらの多くはプログラムの字句構造を扱うだけで、意味構造を意識した検索を行うことができない。このためユーザにとって不必要な情報まで抽出されてしまうことがあり、正確さを犠牲にする。

一方、統合開発環境として知られる Eclipse の Java 検索機能は、プログラムの意味構造を意識した検索機能をサポートしている。意味構造を意識した検索とは、対象となる文字列がプログラムの中でどのような要素として利用されているかを意識した検索である。例えば検索文字列が Java 要素 (パッケージ、型、メソッド、フィールド) の宣言なのか参照なのかといった区別や、フィールドへのアクセスがリード · アクセスなのか、ライト · アクセスなのかといった区別を考慮に入れた検索である。

このような機能により、プログラムの字句構造のみを扱っていた従来の検索ツールに比べ、より柔軟で強力な検索が可能となった。ところがこのようなツールでさえも不必要な情報が抽出されてしまうことがある。例えば、特定の戻り値を持つメソッドの検索をしたいにもかかわらず、他の戻り値を持つメソッドが抽出されたり、あるフィールドアクセスを含むメソッドを検索したいにもかかわらず、それらを含まないメソッドが抽出されたりといった場合である。これは、検索対象となる文字列が持っている情報（プログラムの相互の関連性を意識した情報）を十分に絞り込むことができないことによる。このため、ユーザは検索された結果から欲しい情報を取り出すという労力と手間を費やさなくてはならない。

そこで我々はこれらの問題に対処するため、Java プログラムに対しより柔軟で強力な検索を行うことができる検索ツール highgrep を提案する。highgrep は、Eclipse がサポートするプログラムの意味構造を意識した検索機能を強化することに加え、相互の関連性を意識した検索を可能

とするシステムである。プログラムの関連性を意識した検索により、例えばあるメソッド呼び出しを含んだメソッドの呼び出しや、ある 2 つのフィールドの両方にアクセスしているメソッドの呼び出しあるいは宣言、などを検索できる。

highgrep の開発にあたり、我々は検索パターンを記述するための言語を作成した。我々が提案する言語で扱いたい検索パターンは、既存のツールでは扱うことができないプログラムの意味構造を意識したものと、相互の関連性を意識したものである。そこで、プログラムの意味構造を記述するための文法として AspectJ の pointcut 記述を参考にし、相互の関連性を記述するための文法として、入れ子構造を適用した。highgrep の実装には構造リフレクションを提供する Javassist を用いた。最後に我々は、 典型的なパターンをいくつか検索し、highgrep を用いてその実行時間を測定した。その結果、十分に実用範囲の時間内で検索を行えることを確かめた。

謝辞

本研究を進めるにあたり、研究の方向付けや論文の組み立て方など、大変細かい点にわたり指導して頂いた指導教官の千葉滋助教授に心より感謝致します。千葉助教授には、時には厳しく時には優しく、未熟者の私を指導して頂きました。

また、プログラムの基本的な書き方に始まり、設計や構成まで多岐にわたる指導を親身になって行って下さった、東京工業大学の佐藤芳樹氏、中川清志氏、西澤無我氏に感謝致します。佐藤芳樹氏には、研究の方向付けを指導して頂いたり、関連研究を紹介して頂いたりしました。中川清志氏には、プログラムの設計などの助言に始まり、本研究の多くを見て頂きました。西澤無我氏には、多くの疑問を聞いて頂き、様々な助言を頂きました。

また本研究に必要な多くの知識を与えて下さった筑波大学の横田大輔氏、東京工業大学の栗田亮氏、宇崎央泰氏、に感謝致します。そして論文のスタイルファイルを作り残して頂きました元東京大学の光来健一氏に感謝致します。

最後に、同室で苦楽を共にし励まし合ってきた研究室のみなさん、心より感謝致します。

目 次

第 1 章　はじめに　8

第 2 章　関連研究　11

2.1　字句構造のみを扱った検索機能と問題点 11
2.1.1　正規表現 11
2.1.2　grep コマンドの仕様 13
2.1.3　字句構造のみを扱った検索機能の問題点 13
2.2　意味構造を意識した検索機能 15
2.2.1　Eclipse 15
2.2.2　AspsctJ 17
2.2.3　既存の検索機能の問題点 22

第 3 章　highgrep の仕様と実装　24

3.1　アイディア 24
3.2　仕様 25
3.2.1　AspectJ の pointcut 記述による指定 25
3.2.2　新しいパターンを持つ pointcut の指定 25
3.2.3　新しい演算子による指定 26
3.3　実装方法 28
3.3.1　字句解析 28
3.3.2　構文解析 29
3.3.3　BNF による文法の定義 34
3.3.4　highgrep が認める正規表現 36
3.3.5　Javassist 36

第 4 章　実験と考察　42

4.1　実験 42
4.2　検索にかかる時間の測定および評価 42
4.3　実験の考察 43

第 5 章　まとめ **48**
5.1　本研究の貢献 48
5.2　今後の課題 48

付 録 A 利用例および出力結果 **52**

図 目 次

2.1 Eclipse のワークベンチ 16
2.2 Eclipse の Java 検索 17

3.1 入力画面と出力画面 27
3.2 字句解析ルーチンの流れ 29
3.3 オートマトン 30
3.4 パターンの分類 31
3.5 ソートおよび構文木の作成 34

4.1 検索時間 call(* * *.*(..)) 44
4.2 検索時間 get (* * *.*), call (* * *.*(..)) 44
4.3 検索時間 call (get (* * *.*) ,call (* * *.*(..))) 45
4.4 検索時間 call (call (* * *.*(..))) 45
4.5 検索時間 set (* int *.*) と set (* * * *.*) 46
4.6 検索時間 call (call (* int *.*(..))) と call (call (* * *.*(..))) 46
4.7 検索時間 4 パターンの検索時間の比較 47

表 目 次

2.1 メタ文字の意味 12
2.2 grep コマンドのオプション 14

3.1 優先順位 32
3.2 BNF 文法 40
3.3 CtClass クラスの内観用メソッド 41

5.1 各検索機能の比較 49

第1章　はじめに

我々が膨大なファイル（プログラム）を扱う際、どのファイルのどの部分に何が記述してあるのか、全てを把握することは難しい。このため、ファイルの中から特定の情報を検索するという作業を行わなくてはならない場面に遭遇することがよくある。このような動機から現在では、検索機能をサポートするツールが数多く開発されている。

検索機能をサポートするツールの代表例として、テキストエディタの検索機能や、Unix で使われる grep コマンドなどがある。これらの多くは、文字列のパターンマッチを前提として開発されたもので、検索文字列の指定に正規表現 [4] を利用することができる。正規表現とは文字列のパターンを表す手法であり、テキスト処理を強力に行えるようにするための仕組みのことを指す。この正規表現を利用することで、より柔軟なテキスト処理を行うことができ、我々は作業効率を上げることができる。

ところがこれらの検索ツールの多くは、構文解析プログラムを組み込んでおらず、プログラムの字句構造を扱うだけで、文字列の背後にあるプログラムの意味構造を意識した検索を行うことができない。確かに、簡単な検索を行う場合には、大変便利で軽量なシステムといえる。しかしこれらの検索ツールは、プログラムの意味構造を意識していないため、ユーザにとって関心のない情報まで抽出してしまうことがあり、正確さを犠牲にする。例えばメソッド A を検索したい場合、メソッドとしての A を検索したいにもかかわらず、A というフィールドやコンストラクタ、さらにはコメント内で使われている A という文字列まで抽出される可能性がある。このため、ユーザは抽出された検索結果から欲しい情報を取り出すという労力と手間を費やさなくてはならず、効率的であるとはいえない。

一方、統合開発環境として知られる Eclipse の Java 検索機能は、プログラムの意味構造を扱うことができる検索ツールをサポートしている。意味構造を意識した検索とは、対象となる文字列がプログラムの中でどのような要素として利用されているかを意識した検索である。例えば検索

文字列が Java 要素 (パッケージ、型、メソッド、フィールド) の宣言なのか参照なのかといった区別や、フィールドへのアクセスがリード · アクセスなのか、ライト · アクセスなのかといった区別を考慮に入れた検索である。

このようなプログラムの意味構造を意識した検索環境の実現により、先に述べた字句構造のみを扱った検索ツールに比べ、より柔軟で効率的な検索を行うことが可能となった。しかし、このような機能でさえも不必要な情報が抽出されてしまうことがある。例えば、戻り値が int 型のメソッドを検索したいにもかかわらず、他の戻り値を持つメソッドが抽出されたり、引数を持たないメソッドを検索したいにもかかわらず、引数を持ったメソッドが抽出されたり、といった場合である。これはプログラムの意味構造を意識した検索を十分に行うことができないことによる。また、あるフィールドアクセスを含むメソッドを検索したいにもかかわらず、それらを含まないメソッドが抽出されるという場合もある。これはプログラムの相互の関連性を意識した検索を行うことができないことによる。このため Eclipse の Java 検索機能でさえも、抽出された検索結果から欲しい情報を取り出すという労力と手間を費やさなくてはならない。

そこで我々はこれらの問題に対処するため、Java プログラムに対し、より柔軟で強力な検索を行うことができる検索ツール highgrep を提案する。highgrep は、検索パターンを記述するための言語であり、Eclipse がサポートするプログラムの意味構造を意識した検索機能を強化することに加え、相互の関連性を意識した検索を可能とするシステムである。プログラムの意味構造を意識した検索機能の実現により、例えば以下に示す検索個所の指定を可能にした。

- 検索対象がメソッドであった時、引数や戻り値、修飾子などを指定できる。
- 検索対象がフィールドであった時、型や修飾子を指定できる。
- 検索対象がコンストラクタであった時、引数や修飾子を指定できる。

またプログラムの相互の関連性を意識した検索機能の実現により、例えばあるメソッド呼び出しを含んだメソッド呼び出しや、ある 2 つのフィールド両方にアクセスしているメソッド呼び出し、などの検索を可能にした。このような機能により我々は、従来までの検索ツールで抽出されていた不必要な情報を取り除くだけでなく、字句レベルでは辿ることができないプログラムの深い構造を辿る検索を実現した。

我々は highgrep の開発にあたり、プログラムの意味構造を意識した検索パターンを記述するために AspectJ [5, 6] の pointcut 記述を参考にした。Aspect 指向とは, ログ出力やメモリ管理などプログラム全体に散らばってしまう処理を、アスペクトという新しいモジュール概念によりモジュール化する技法である。AspectJ は標準的な汎用アスペクト指向言語の一つであり、Java を拡張した言語である。またプログラムの相互の関連性を意識した検索パターンを記述するために、入れ子構造を適用した。highgrep の実装には、Javassist [2, 7, 9] を用いた。Javassist は、構造リフレクションを提供するクラスライブラリであり、クラス構造の内観を行うだけでなく変更を可能にする。本稿ではクラス構造の内観を行うために Javassist を利用した。最後に我々は、 典型的なパターンをいくつか検索し、highgrep を用いてその実行時間を測定した。その結果、十分に実用範囲の時間内で検索を行えることを確かめた。

以下 2 章で、既存の検索機能とその問題点を指摘し、AspectJ についての説明を加える。3 章で highgrep の仕様および実装方法を述べ、4 章で実験による考察をし、5 章で本稿をまとめる。

第2章　関連研究

本章では既存の検索機能についての説明をし、それらの問題点を指摘する。まず、字句構造のみを扱った検索機能の例として、正規表現 (Regular Expression) を利用できる Unix の grep コマンドを取り上げ説明する。次にプログラムの意味構造を扱った検索機能の例として Eclipse の Java 検索機能を取り上げ説明する。さらに本システム highgrep の文法を記述するために参考にした AspectJ の説明を述べる。

2.1　字句構造のみを扱った検索機能と問題点

2.1.1　正規表現

ここではまず正規表現について説明をする。正規表現とは文字列のパターンを表す手法であり、強力で柔軟、かつ効率的なテキスト処理を行うことを可能とする仕組みのことである。テキスト処理とは、文字列の検索、置換、削除といった作業のことを指す。

正規表現はさまざまな環境で使われており、各種ソフトウェアに組み込まれ利用されている。エディタやワープロに始まり、Unix 系コマンドラインツール、果てにはインターネット上のサーチエンジンにも実装されている。このように各方面にて利用されている正規表現であるが、標準化されていないという欠点を持っており、ツールによって実装方法やサポート状況に多少の違いがみられる。以下に代表的な検索機能として知られる Unix の grep コマンドで利用できる一般的な正規表現の使用例を示す。

正規表現の使用例

正規表現を記述するにはメタ文字と呼ばれる特別な文字を使う。このメタ文字を組み合わせる事で高度な文字列処理を行うことができる。メ

タ文字には以下のようなものが用意されている。

^　$　.　|　+　*　?　[ ]　{ }

各メタ文字が表す意味および利用例を以下に示す。

表 2.1: メタ文字の意味

	記号	意味
1	^	文字列の先頭 [ ] の中で使われた場合はその後の文字列の否定
2	$	文字列の末尾
3	.	改行以外の任意の 1 文字
4	|	2 者択一の演算子
5	+	直前の文字の 1 個以上の並び
6	*	前の文字の 0 個以上の並び
7	?	直前の文字の 0 個または 1 個
8	[ ]	[ ] でくくられた中にある任意の 1 文字
9	{a , b}	直前の文字の a 個以上、b 個以下の並び

1. ^java
   行の先頭が java という文字で始まっている場合

2. java$
   行の末尾が java という文字で終わっている場合

3. java..
   java の後に改行以外の任意の文字が 2 つ続く場合

4. JSP | Servle
   JSP または Servlet

5. .+
   1 個以上の任意の文字の繰り返し

6. .*
   0 個以上の任意の文字の繰り返し

7. javax?
   java または javax

8. [0 − 9]
   数字の中の 1 文字

9. j{1 , 3}
   j または j j または j j j (1 個以上 3 個以下の j の並び)

2.1.2　grep コマンドの仕様

ここでは、正規表現を利用できる代表的な検索ツールとして、Unix で使われる grep（Global Regular Expression Print） コマンドを取り上げ説明する。grep コマンドとは、プログラム内に出現する文字列とのパターンマッチを行うコマンドで、ファイル中から文字列検索を行なうためのツールである。

> grep [オプション] 検索パターン ファイル

コマンドラインから上記の書式で grep を実行すると、ファイルから指定した検索パターンでサーチを行ない、検索対象のキーワードが見つかったら、該当した行が表示される。オプションには検索方法を変更したり、検索結果を表示する際の挙動を変更するものが用意されている。表 2.2 は、Unix の grep コマンドに用意されている代表的なオプションである。

2.1.3　字句構造のみを扱った検索機能の問題点

前節で述べた grep のような字句構造のみを扱う検索機能は、確かに軽量で便利なシステムということができる。しかしこれらの検索ツールには、構文解析プログラム（パーサー）が組み込まれておらず、文字列の背

[1]gerp では正規表現を「基本正規表現」と「拡張正規表現」の 2 種類に分けて取り扱う。拡張正規表現では一通りのメタ文字が利用できますが、基本正規表現では、?, +, {, | (,) のメタ文字が単なる文字として扱われる。

表 2.2: grep コマンドのオプション

オプション	説明
-C num	マッチした行の前後 num 行もいっしょに表示する
-c	マッチした行数のみを表示
-E	検索パターンを拡張正規表現として扱う[1]
-i	大文字、小文字を無視する
-n	行番号表示
-v	マッチしなかった行を表示する
-w	単語境界で検索する
-x	完全に全体とマッチするもののみを表示する

後にある意味構造を意識した検索を行うことができない。すなわち、このような字句構造のみを扱う検索システムでは、字面の検索しか行えず、検索文字列がプログラムの中でどのような要素として記述されているかを意識することができない。このため、例えば次のような区別を考慮に入れた検索を行うことができない。

- 検索したい文字列がメソッドなのか、フィールドなのか、クラスなのか、パッケージなのか、といった区別
- メソッドが宣言なのか参照なのかといった区別
- フィールドがリード · アクセスなのかライト · アクセスなのかといった区別
- メソッドの返り値は何であるのかといった区別

従って、字句レベルでマッチした文字列が全て抽出されてしまうため、プログラム内の意味レベルでマッチしない文字列まで検索結果に入り込んでしまうということが十分考えられる。このためユーザは、必要な情報だけを検索結果から抽出するという労力と手間を費やさなくてはならない。

2.2　意味構造を意識した検索機能

本節ではまず、プログラムの字句構造に加え意味構造を意識した検索を可能とするツールの例として Eclipse の Java 検索機能を取り上げ説明する。意味構造を意識した検索とは、対象となる文字列がプログラムの中でどのような要素として利用されているかを意識した検索を指す。次に我々が提案する言語の検索パターンを指定する際、参考にした AspectJ の説明を加える。

2.2.1　Eclipse

Eclipse とは

Eclipse(エクリプス) とは、java などの複数の言語に対応したオープンソースの統合開発環境 (IDE) である。

Eclipse は、JBuilder[2] や Sun One Studio[3]などと同様に、GUI 環境で Java によるアプリケーション開発を行える統合開発環境である。コード生成（補完)、デバッグ、検索など、開発に必要な機能はもちろん、テスティング、フレームワーク「JUnit」、ビルドツール「Ant」、バージョン管理システム「CVS」との連携機能、リファクタリング機能など充実した機能を備えている。

また Eclipse は、plugin ベースで動作しており、多くの plugin を組み込むことで、XML や Web アプリケーション、JSP などの豊富な機能を備えた IDE を実現できる。Eclipse は GUI の構築に、Swing[4]を使わず SWT[5]（Simple Widget Kit）と呼ぶ独自の GUI フレームワークを利用しているため、動作は軽快である。Eclipse を起動した際に現れるウィンドウ画面を図 2.1 に示す。

Eclipse がサポートする Java 検索

Eclipse がサポートする Java 検索機能は、プログラムの意味構造を意識した検索を行うことができる。まず検索エリアで、検索対象の文字列が Java 要素 (パッケージ、型、メソッド、フィールド) のいずれを表して

[2]Borland Software の製品
[3]米 Sun Microsystem の製品
[4]Java で実装した GUI コンポーネント群。JDK1.2 以降に含まれている。
[5]Swing と違って OS のネイティブな API を利用して画面描画を行う。

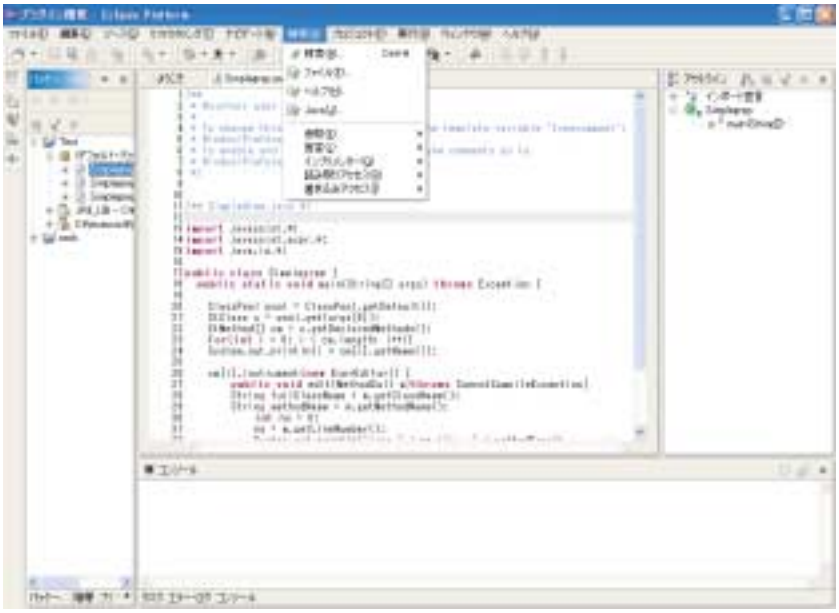

図 2.1: Eclipse のワークベンチ

いるかを指定できる。さらに制限エリアでその文字列が、宣言なのか参照なのか、あるいは両方なのかを指定したり、フィールドアクセスに関しては、読み取りアクセスなのか、書き取りアクセスなのかを指定して検索範囲を絞り込むことができる。ここで利用できる正規表現はアスタリスク (任意のストリング) のみである。図 2.2 は Eclipse の Java 検索を行う際に現れる画面である。

このように Eclipse は、字句構造のみを扱っていた従来の検索機能では対処することができない、プログラムの意味構造を意識したより柔軟で強力な検索機能を実現している。

Eclipse の問題点

Eclipse がサポートする Java 検索機能により、プログラムの字句構造のみを扱っていた既存の検索ツールに比べ、より柔軟で強力な検索が可能となった。ところが、Eclipse で利用できる正規表現はアスタリスクのみに限られている。このため、grep のような正規表現をサポートする既存の検索ツールに比べ、検索対象の文字列を柔軟に表現することができない。また、修飾子や戻り値、引数を指定した検索を行うことができな

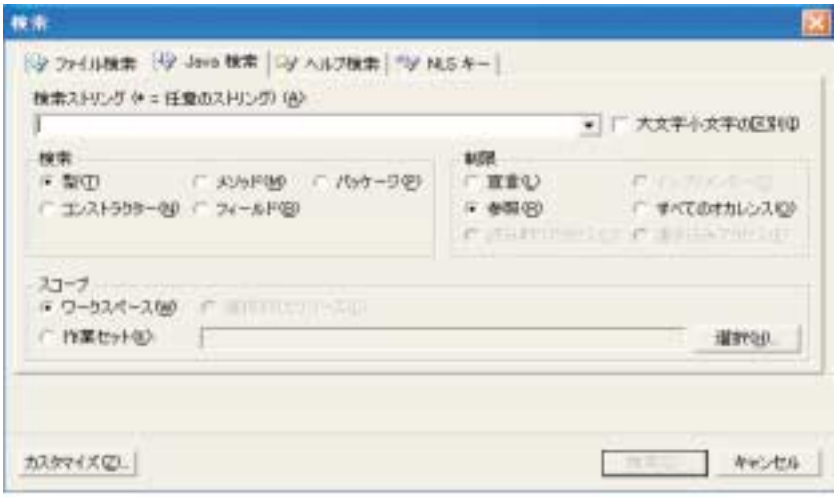

図 2.2: Eclipse の Java 検索

い。このためユーザにとって不必要な情報まで抽出されてしまうことがある。例えば、public なメソッドを検索したいにもかかわらず、private なメソッドが抽出されてしまったり、int 型のフィールドを検索したいにもかかわらず、他の型を持つフィールドが抽出されてしまったりといった場合が考えられる。これはプログラムの深い意味構造を扱うことができないことによる。また、あるフィールドアクセスを含むメソッドを検索したいにもかかわらず、それらを含まないメソッドが抽出されるという場合も考えられる。これはプログラムの相互の関連性を意識していないことによる。

このように、Eclipse がサポートする Java 検索機能では、プログラムの意味構造を扱った複雑な検索ができなかったり、相互の関連性を意識した検索を行うことができない。このためユーザにとって不必要な情報まで抽出されてしまうことがあり、効率的な検索であるとは言えない。

2.2.2　AspsctJ

先に述べた Eclipse の問題点を解決するため、我々はより強力な絞り込みを可能とする検索パターンの指定を実現した。検索パターンを表現するために、AspectJ の pointcut 記述を参考にすることで汎用性を高めた。本節では AspectJ の説明を行う。

アスペクト指向言語

オブジェクト指向におけるプログラミングは、オブジェクトという単位でプログラムをモジュール化していく。モジュールとは、ある意味を持ったコードのまとまりであり、人間にとって理解しやすい大きさに分割されたプログラムの単位である。このオブジェクト指向プログラミングでは、コードが複雑に絡み合ってしまうため、プログラムの拡張性が低下しコードの修正が困難となる。また、同じ関心事を扱うコードが複数のクラスに分散してしまうため、プログラムの保守性が低下する。このようなオブジェクト指向プログラミングではうまく取り扱うことができない問題を解決するために提案されたのがアスペクト指向という新しいパラダイムである。アスペクト指向は、ロギングや同期処理といった複数のオブジェクトに散らばってしまった関心事（横断的関心事）を、アスペクトという再利用可能なモジュールとしてカプセル化する手法である。オブジェクトとアスペクトは織り込み（weaver）と呼ばれる言語処理系により 1 つのプログラムとして合成される。このアスペクトという概念を用いることで、保守性や再利用性を高めることができる。

AspectJ は標準的な汎用アスペクト指向言語の 1 つであり、Java にアスペクト指向を取り入れて拡張したものである。

AspectJ の基本概念

ここでは、AspectJ を理解する上で必要となる 4 つの概念を説明する。

1. Joinpoint
   Joinpoint とは、プログラムの実行局面の中で適切に定義されたポイントを表し、他のプログラム（他の関心事）を割り込ませ、実行することのできるソースコード上の位置を意味する。Joinpoint として割り込み可能な位置は、AspectJ の仕様で決められており、メソッド呼び出し、変数へのアクセス、例外ハンドラの実行などがある。

2. pointcut
   pointcut は、Joinpoint の集合を表しプログラムの実行時の目立った時点として定義される。

3. Advice
   Advice は、pointcut において新しい命令を実行するコードとして定

義される。Advice は処理を実行するタイミングの種類により、pointcut の前 (before)、後 (after)、前後 (around) の 3 種類に分類される。

4. Aspect
   Aspect は、pointcut、Advice、および通常の Java を構成する要素の宣言から成り、横断的に実行されるモジュール単位として定義される。

AspectJ の pointcut 記述

上述したとおり、指定可能な Joinopint の位置は AspectJ の仕様で決められている。以下では、AspectJ で指定可能な Joinpoint と、それを実際に指定する際の pointcut の記述法およびその説明を示す。

- メソッドの呼び出し：call(MethodPat)

  1. call(public void ClassA.methodA())
     クラス ClassA のメソッドのうち、public で戻り値および引数を持たないメソッド methodA の呼び出しを指定。
  2. call(int ClassA.methodA(..))
     クラス ClassA のメソッドのうち、戻り値が int 型、アクセス・スコープおよび引数が任意のメソッド methodA の呼び出しを指定。(..) は引数が任意であることを示す。

     さらに正規表現を利用すると、以下のような指定ができる。

  3. call(* ClassA.method*(String))
     クラス ClassA 内で、戻り値およびアクセス・スコープが任意で、引数が String 型のメソッドのうち、メソッド名が”method”で始まる全てメソッドの呼び出しを指定。
  4. call(* methodA(..))
     戻り値および引数が任意であるメソッド methodA の呼び出しを指定。
  5. call(public * *(..))
     全ての public メソッド呼び出しを指定。

- コンストラクタの呼び出し：call(ConstructorPat)

  1. call(ClassA.new())
     クラス ClassA のコンストラクタのうち、引数をとらないコンストラクタの呼び出しを指定。
  2. call(*.new(int, String))
     全てのクラスのコンストラクタのうち、引数が (int, String) であるコンストラクタの呼び出しを指定。

- 特定メソッドの実行：execution(MethodPat)

  1. execution(void ClassA.methodA())
     クラス ClassA のメソッドのうち、アクセス・スコープが任意で、戻り値および引数を持たないメソッド methodA の実行を指定。
  2. execution(public !static * ClassA.*(..))
     クラス ClassA の static でないメソッドのうち、戻り値および引数が任意であるメソッドの実行を指定。
  3. execution(!static * *(..))
     全ての static でないメソッドの実行を指定。

     特定コンストラクタの実行も execution(ConstructorPat) という形で同様に指定できる。

- 属性へのアクセス：get(FieldPat) , set(FieldPat)

  1. get(int ClassA.x)
     クラス ClassA の属性のうち、int 型の属性 x へのリード・アクセスを指定。
  2. set(String ClassA.x)
     クラス ClassA の属性のうち、String 型の属性 x へのライト・アクセスを指定。
  3. set(!private * ClassA.*)
     クラス ClassA の private でない属性のうち、戻り値および属性名が任意である属性へのライト・アクセスを指定。

- 例外ハンドラ

  1. handler(Exception)
     クラス Exception の例外オブジェクトを catch する例外ハンドラの実行を指定。すなわち Exception 例外が投げられる瞬間を指定。

- static なメンバの初期化

  1. staticinitialization(ClassA)
     クラス ClassA の static イニシャライザ内のメンバの初期化を指定。
  2. staticinitialization(ClassA+)
     クラス ClassA を継承するクラスの static イニシャライザ内のメンバの初期化を指定。

- クラス/メソッド内の全 Joinpoint

  1. wihtin(ClassA)
     クラス ClassA の中の全 Joinpoint の指定。
  2. withincode(int ClassA.methd*())
     クラス ClassA 内で、アクセス・スコープが任意 、戻り値が int 型、引数を持たないメソッドのうち、メソッド名が”method” で始まるメソッドの中の全 Joinpoint を指定。

- コントロール・フロー上の全 Joinpoint

  1. cflow(call(* ClassA.methodA(..)))
     メソッド ClassA.methodA を呼び出した後のコントロール・フロー上に存在する全ての Joinpoint を指定。call(* ClassA.methodA(..)) 自体も pointcut に含める。
  2. cflowbelow(call(* ClassA.methodA(..)))
     メソッド ClassA.methodA を呼び出した後のコントロール・フロー上に存在する全ての Joinpoint を指定。call(* ClassA.methodA(..)) 自体 は pointcut に含め ない。

- this,target, 引数、通知する例外

1. this(ClassA)
   処理を実行している実体がクラス ClassA の場合を指定。
2. target(ClassA)
   処理の対象がクラス ClassA の場合を指定。
3. args(String, ..)
   1 番目の引数に String 型を持つ場合を指定。
4. args(Exception)
   1 番目の引数もしくは通知する例外にクラス Exception を持つ場合を指定。

さらに、上で示した pointcut を組み合わせることで、より複雑な pointcut を定義することができる。組み合わせる際に用いられる演算子は、or(||)、and(&&)、not!の 3 種類である。以下にそれらを用いた複雑な pointcut 記述の例を示す。

1. target(ClassA) && call(int *(..))
   クラス ClassA を処理の対象とし、戻り値が int 型、引数が任意である全てのメソッド呼び出しを指定。
2. call(* *(..)) && (within(ClassA) || within(ClassB))
   クラス ClassA またはクラス ClassB 内の全てのメソッド呼び出しを指定。
3. within(*) && call(*.new(int))
   全てのクラス内のコンストラクタのうち、引数に int 型をもつコンストラクタの呼び出しを指定。

本研究では、検索部分を指定するために、このような AspectJ の pointcut 記述を参考にした。

2.2.3　既存の検索機能の問題点

先に述べてきたように、既存の検索機能ではプログラムの字句構造しか扱うことができなかったり、たとえ意味構造を扱うことができたとしても、単純なパターンしか扱うことができなかった。このため、プログラムの深い意味構造や相互の関連性を意識した検索を行うことができない。小さなファイルを対象とする場合は、何ら問題は生じないかもしれ

ない。ところが膨大なファイルを対象とした場合、検索された結果から欲しい情報のみを選び出すために多大な手間と労力を費やさなくてはならない。

第3章　highgrep の仕様と実装

2 章で示した問題に対処するため、我々はより柔軟で強力な検索機能 highgrep を提案·開発した。本章では、highgrep を開発するにあたってのアイディアを最初に述べ、以降で highgrep の仕様および実装方法を示す。

3.1　アイディア

既存の検索ツールではユーザにとって不必要な情報まで抽出されてしまうことがあり、効率的な検索を行うことができない。そこで我々はこのような問題に対処するため、より柔軟で強力な検索ツール highgrep を提案する。highgrep は、検索パターンを記述するための言語であり、プログラムの意味構造と相互の関連性を意識した検索を可能とする。

意味構造を意識した検索パターンを記述するために、AspectJ の pointcut 記述を参考にし、相互の関連性を意識した検索パターンを記述するために入れ子構造を適用した。参考にした AspectJ の pointcut 記述は、メソッド/コンストラクタの呼び出し、および実行、属性へのアクセス、クラス/メソッド内の全 Joinpoint の指定である。またそれらの pointcut を組み合わせる際の演算子として、or(||) や、and(&&)、正規表現としてアスタリスク (*) の記述法を参考にした。

例えば highgrep では以下のような検索ができる。

call(get(* * Point.x) , get(* * Point.y))

これは、フィールド x へのリード·アクセス、およびフィールド y へのリード·アクセスがともに Point クラスの同じメソッド内で宣言されているとき、そのメソッドを呼び出している部分を全て検索できる。

このように我々が提案する highgrep は既存の検索ツールで実現していたプログラムの意味構造を意識した検索機能を強化し、さらに相互の関連性を意識した検索を実現したツールである。以下に highgrep の仕様および実装方法を示す。

3.2　仕様

本システム highgrep を起動させるには、まずコマンドプロントから以下のように入力する。

> java MainGrep *.class

第一引数には、検索対象となるファイルのクラスファイル名を入力する。ファイルの指定にはワイルドカードを利用できる。次に入力画面になるので、ここで検索個所の指定を規定の文法 (表 3.2 に示す BNF 文法) に従って打ち込む。ただし、'help' と入力すると本システムが認める BNF 文法および利用法が表示される。以下に入力例をいくつか示す。

3.2.1　AspectJ の pointcut 記述による指定

2 章で示した AspectJ の pointcut 記述とほぼ同じ記述で、検索個所を指定できる。各パターンが表す意味は AspectJ が表すものと同じであるため、詳細は省略する。

- call(MethodPattern) , call(ConstructorPatternn)
  指定されたメソッドパターン / コンストラクタパターンにマッチするメソッド / コンストラクタの呼び出しを検出

- execution(MethodPattern) , execution(ConstructorPatternn)
  指定されたメソッドパターン / コンストラクタパターンにマッチするメソッド / コンストラクタの実行を検出

- get(FieldPattern) , set(FieldPattern)
  指定されたフィールドパターンにマッチするフィールドへのリード · アクセス / ライト · アクセスを検出

- within(ClassPattern)
  指定されたクラスパターンにマッチするクラス内全ての pointcut を検出 pointut を検出

3.2.2　新しいパターンを持つ pointcut の指定

- call(QualifiedMethodPatern)

  - call(public int Point.methodA(..) , methodB(..))
    クラス Point 内のメソッドのうち、引数任意のメソッド methodB の直後に現れる、修飾子 public、戻り値 int、引数任意のメソッド methodA を呼び出している部分を検出。ここでの直後とは、バイトコードの中で現れる順序における直後を意味する。

- call(pointcutPattern) , execution(pointcutPattern)

  パターンに pointcutPattern を指定できるようにし、入れ子構造を実現した。この指定により、プログラムの関連性を意識した検索を可能とする。以下にいくつか例を示す。

  - execution(call(* * Point.getX(..)))
    クラス Point 内のメソッドのうち、修飾子、戻り値、引数が任意であるメソッド getX の呼び出しを含むメソッドを宣言している部分の検出。
  - call(call(* * Point.getX(..)))
    クラス Point 内のメソッドのうち、修飾子、戻り値、引数が任意であるメソッド getX の呼び出しを含むメソッドを呼び出している部分の検出。
  - call(call(call(* * Point.getX(..))))
    クラス Point 内のメソッドのうち、修飾子、戻り値、引数が任意であるメソッド getX の呼び出しを含むメソッドを呼び出しているメソッドをさらに含んでいるメソッドを呼び出している部分の検出。

3.2.3　新しい演算子による指定

- pointcut , pointcut

  - get(* * *.x),get(* * *.y)
    フィールド x へのリード・アクセス、およびフィールド y へのリード・アクセスが共に同じクラスの同じメソッド内で出現する場合その両者を検出。
  - execution(get(* * *.x),get(* * *.y))
    フィールド x へのリード・アクセス、およびフィールド y へ

のリード・アクセスが共に同じクラスの同じメソッド内で出現する場合、メソッドを宣言している部分を検出。

- call(get(* * *.x),get(* * *.y))
  フィールド x へのリード・アクセス、およびフィールド y へのリード・アクセスをともに含むメソッドを呼んでいる部分の検出。

- pointcut ＞　pointcut , pointcut ＜　pointcut

  - execution(* * *.*(..)) ＞ call(* * Point.getX(..))
    クラス Point 内のメソッドのうち、修飾子、戻り値、引数が任意であるメソッド getX の呼び出しを含むメソッドを宣言している部分の検出。

  - execution(* int *.*(..)) ＞ call(* * Point.getX(..))
    クラス Point 内のメソッドのうち、修飾子、戻り値、引数が任意であるメソッド getX の呼び出しを含むメソッドのうち戻り値が int 型であるメソッドを宣言している部分の検出。

上に示したような検索指定をすると、指定したクラスファイルの中からそれらにマッチする部分を検索し、行番号、ファイル名、ターゲット名（メソッド名、フィールド名、コンストラクタ名）を含めて結果を出力し、次の入力を待つ画面に戻る。終了したいときは、'exit' と打ち込む。以下に入力例および出力例を示す。

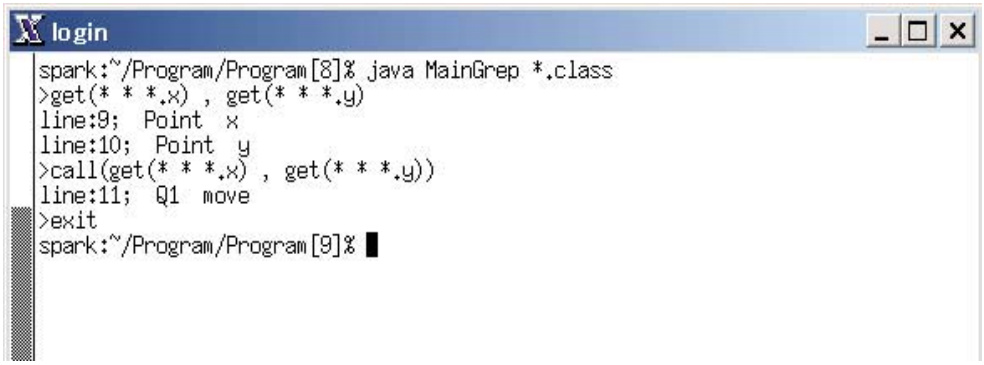

図 3.1: 入力画面と出力画面

3.3　実装方法

3.3.1　字句解析

本研究では、検索対象として入力される文法を解析するために、まず字句解析を行う。字句解析とは、入力された文字ストリームを読み込み、トークン（意味をなす最小の単位）に分割する処理のことを指す。分割されたトークンは、その後構文解析に引き渡され解析が行われる。字句解析は、構文解析のサブルーチンとして実現されており、構文解析が次のトークンの獲得を要求すると入力文字を読み込み、次のトークンを返す。

字句解析を行う際に、先読みが必要になることがある。これは、1 文字先の文字を読まなくてはトークンの種類を判断できない時に利用される。例えば、文字 & が現れたら、その先の文字まで読まなくてはなそのトークンが何を表すのか判断できない。もし次の文字が & なら、トークンは && として判定される。そうでなければ、トークンは & として判定され、字句解析ルーチンは 1 文字余計に呼んでしまったことになる。この読みすぎた文字は、次の字句の先頭である可能性があるため、入力に戻しておかなくてはならない。このように、読んでしまった文字を読まなかったことにする操作を押し戻しと呼ぶ。先読み、および押し戻しを実現するため、読み込んだ文字を保持しておくバッファを用意した。すなわち、読みすぎてしまった文字をこのバッファに戻し、次の文字はバッファから取り出すという仕組みである。

また、1 文字読み込む関数 getChar() は以下のように実装されている。

```
public int getChar() throws IOException {
    int indt;
    if(buffer == EMPTY) {
        indt = input.read();
    } else {
        indt = buffer;
        buffer = EMPTY;
    }
    return indt;
}
```

このような、先読みおよび押し戻しの操作は、構文解析でも必要となる。字句解析では 1 文字単位でそれらの操作を行ったが、構文解析ではトーク

ン単位で行われる。我々は、1 トークンを読み込む関数として getToken()、トークンの先読みを行う関数として lookahead()、トークンの押し戻しを行う関数として pushback() を用意した。

以上、入力から構文解析までの流れを図示すると図 3.2 のようになる。

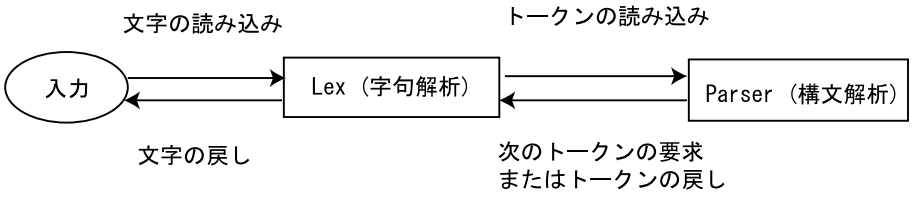

図 3.2: 字句解析ルーチンの流れ

3.3.2　構文解析

字句解析により入力をトークン列に分割することができた。次に入力文の構文構造を特定しなくてはならない。そのために構文解析を行う。構文解析とは，字句解析から受け渡されるトークンを読み込み、トークンの組み合わせを文法の意味にマッチさせ，構文木 (parser tree) に変換する処理のことである．我々が提案する highgrep の文法は 2 章で説明した AspectJ のポイントカット記述を参考にした。 ここで定めた文法を直感的に理解するために、簡単なオートマトンを図 3.3 に示す。

PCD は AspectJ のポイントカット指定子、例えば call や get などを表す。また Pattern は 図 3.4 に示すように pointcutPattern や MethodPattern などを表す。Pattern の説明として MethodPattern を例に取り、以下に簡単な説明を加える。他のパターンについてもほぼ同じ設計で実装されているため詳細は省略する。また構文解析を行う際に必要となる文法は、BNF (Backus Normal Form) 文法を用いて記述し、詳細は、3.3.3 で示す。

MethodPattern

MethodPattern は要素として修飾子 · 戻り値 · メソッド名 · クラス名 · 引数を持つ。また makegrep() は、パターンが MethodPattern であった

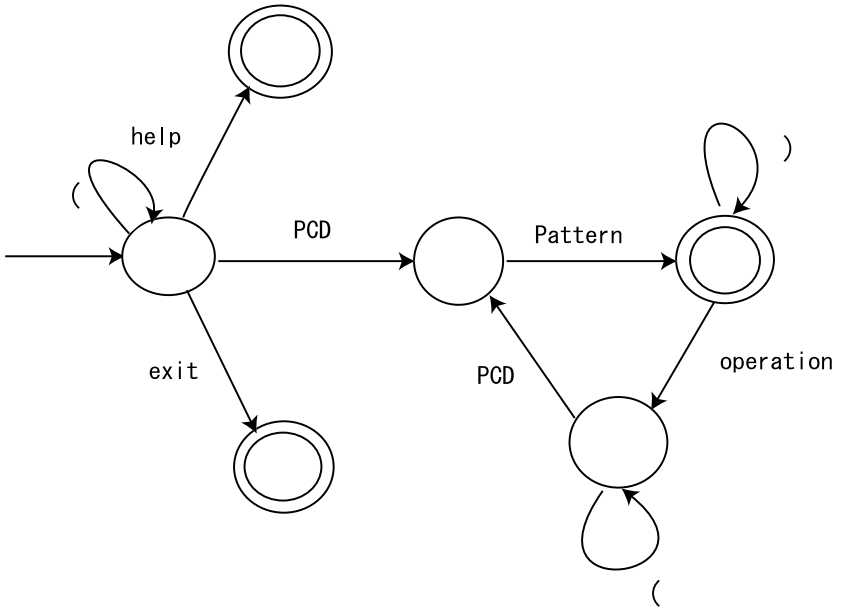

図 3.3: オートマトン

ときに、検索を実行するクラス MethodGrep (スーパークラスは Grep クラス) を返すメソッドである。以下に MethodPattern クラスのコードの一部を示す。

```
class MethodPattern extends Pattern {
    int modifier;
    String returnType;
    String methodName;
    Vector args;
    String className;
           :
           :
    public Grep makegrep(int i){
        return new MethodGrep(getClassName(), getModifierName(),
                              getReturnType(),getMethodName(),
                              getArgsVector(), i);
    }
           :
    ( 変数 i には ポイントカット指定子を識別する
      番号が入れられている。)
```

このように各パターン毎に検索を行う Grep クラスを生成することができる。

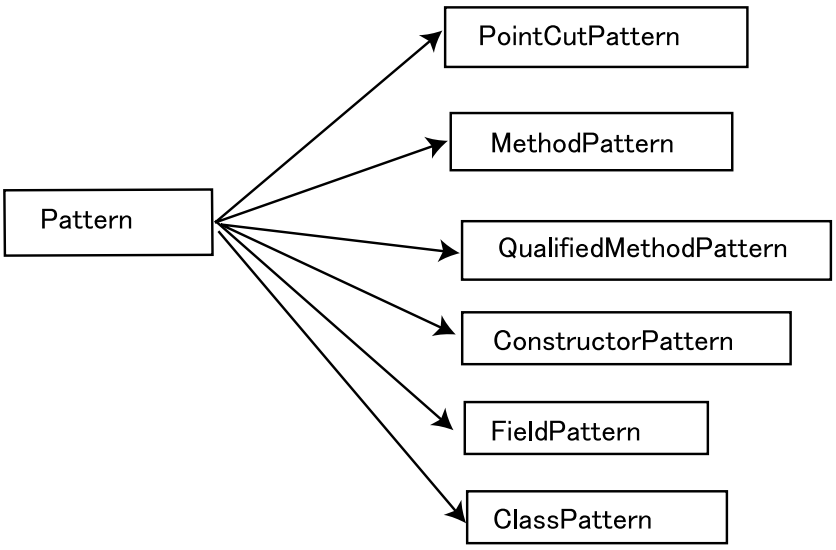

図 3.4: パターンの分類

逆ポーランド記法への変換

規定の文法に従った入力データは、構文解析（Parser）を通してリスト構造に変換される。このリスト構造には、条件式 (pointcut オブジェクト)、演算子 (&&, || など)、(、) が格納されている。次に Parser から返されたリスト構造を逆ポーランド記法に変換する。逆ポーランド記法に変換することで、リストを先頭から読んでいくことができるようになるとともに、カッコを全て取り除くことができる。さらに、各条件式を結ぶ演算子の優先順位もこの段階で意識してソートされるので、次の段階で構文木に変換することが容易になる。表 3.1 にリストから取り出される各要素の優先順位を示す。

表 3.1: 優先順位

因子（factor）	優先順位
(	6
pointcut	5
<, >	4
,	3
&&	2
||	1
)	0

数字が大きい方が優先順位が高い

構文木の作成

逆ポーランド記法に変換されたリスト構造から構文木を作成する。構文木 (syntax tree) は、入力データの内部表現を木構造で表したもので、構文解析の結果として生成される。

構文木の１番始めの節点である根（root）には、リスト構造に格納されている１番最後の要素が入れられる。また、条件式 （ pointcut オブジェクト) は構文木の葉に当たる。走査の再帰呼び出しは、この葉に到達するとリターンされるが、ここで検索対象として与えられる条件式は計算され LinkedList に格納される。計算を行う際には Javassist を利用した。

Javassist の詳細は 3.3.5 で示す。以下に構文木を作成するアルゴリズムを示す。

- アルゴリズム
  - 逆ポーランド記法に変換されたリスト構造の後ろから要素を 1 つ取り出し、それをノード (Node) として割り当てる。
  - もしその要素が pointcut オブジェクトなら左右のポインタに NILL を入れて再帰呼び出しから戻る。
  - もしその要素が演算子 (operator) なら、次の要素を右の子として接続する再帰呼び出しを行い、続いて次の要素を左の子として接続する再帰呼び出しを行う。

  このアルゴリズムに従って作られた構文木を帰りがけ順に走査し、得られた結果が検索結果となる。

  上で述べてきた規則に従い入力をソートし構文解析木を作る例を図 3.5 に示す。

- 各演算子に対する計算法
  - &&
    ２つの条件式が&&で結ばれている場合、左右の検索結果の共通部分を結果として返す。
  - ||
    ２つの条件式が || で結ばれている場合、左右の検索結果の部分和を結果として返す。
  - >
    ２つの条件式が > で結ばれている場合、右の結果が左の結果に含まれるような左の検索結果を結果として返す。
  - <
    ２つの条件式が < で結ばれている場合、左の結果が右の結果に含まれるような右の検索結果を結果として返す。
  - ,
    ２つの条件式が , で結ばれている場合、左右の検索結果が同じクラスの同じメソッド内で記述されている場合のみその両者を結果として返す。

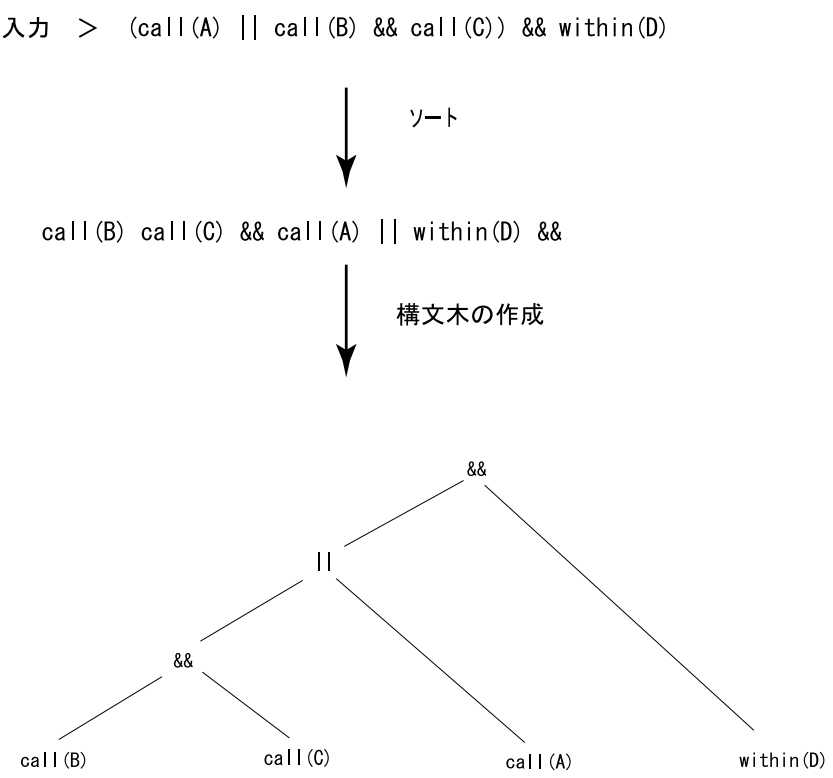

図 3.5: ソートおよび構文木の作成

3.3.3　BNF による文法の定義

BNF 文法とは、Algol60 の構文を形式的に定義するために考案された、文脈自由プログラムのシンタックスを記述するための表記法のことである。BNF 文法の規則は以下のように定められている。

- 終端記号は、そのまま書く
- 非終端記号は、名前を <> で囲んで書く
- 生成規則は、「非終端記号 ::= 右辺」という形
  右辺は記号列を書き、複数の可能性がある場合は縦棒で区切って並

べる

(BNF 記法の簡単な例)
　識別子（identifier）の構文
　　〈識別子〉::＝〈英字〉{〈英字〉|〈数字〉}
　　〈英字〉::＝　A｜B｜C |...｜Z｜a｜b｜c |...｜z
　　〈数字〉::＝　0｜1｜2 |...｜9

本システムは、表 3.2 に示す BNF による文法定義に従い入力の構文を定義する。

3.3.4　highgrep が認める正規表現

本システムを実装するにあたりサポートした正規表現は、AspectJ がサポートするアスタリスクである。アスタリスクは、任意文字列として扱うことができる。例えば、修飾子 / 戻り値の指定にアスタリスクを利用すると、全ての修飾子 / 戻り値にマッチすることを意味する。またクラス名やフィールド名など Java 要素の名前指定にアスタリスクを利用するとアスタリスクは任意の文字列にマッチすることを意味する。例えば、

get (* * * . get*)

は、修飾子 · 型 · クラス名が任意で、かつフィールド名が get で始まっているリード · アクセスを意味する。このように正規表現にアスタリスクの利用を許すことで、検索パターンをより柔軟にそしてより強力に指定することができる。ところが、highgrep は grep で扱える正規表現を全てサポートしていないため、字句レベルでの検索機能は grep のそれより劣る。

3.3.5　Javassist

本システムは Javassist を用いて実装されている。Javassist は、構造リフレクションを提供する Java 言語のみで記述されたクラスライブラリである。リフレクションとは、計算システムが自分自身の構成や計算過程 · 計算方式に関する計算を行うことを指し、大きくいって以下に示す 3 つの機能を提供する。

- あるクラス（正確にいうと、Class クラスのオブジェクト）が、どのような Field、Method、 Constructor から構成されているかを検出する機能
- あるクラスのオブジェクトの Fields の値を読み/書きする機能
- あるクラスのオブジェクトの Method や Constructor に、適当な引数を与えて呼び出す機能

Java は、標準 API (Application Programming Interface) の一部としてこれらのリフレクション機能を提供するプログラミング言語である。しか

しながら提供される機能は、上で述べたようなプログラムの内観（ introspection ）を行うことが主で、プログラムの振る舞いを変更することはできない。

Javassist は、標準のリフレクション API を拡張して、プログラムの振る舞いを変更できるようにしたものである。すなわち、実行時あるいはコンパイル時に、既存のクラスを修正したり、新規に生成することができる。このように、プログラムの内部のデータ構造を、必要に応じて変更できる機能を構造リフレクションという。

さらに Javassist は、Java 標準のリフレクション API では検出できないメソッド呼び出しやコンストラクタ呼び出しなどの検出を可能としており、本研究で必要とされる機能を多くサポートしている。

Javassist を使用する第 1 段階は、JVM にロードするクラスファイルを表す CtClass（compile-time class）オブジェクトを生成することである。CtClass の生成は以下のように行う。

```
ClassPool pool = ClassPool.getDefault();
CtClass cc = pool.get("Point");
```

ClassPool オブジェクトは、CtClass の入れ物のようなものであり、バイトコードの変更を管理する。ClassPool は必要に応じてクラスファイルを読み込み、CtClass オブジェクトを生成する。そしてそのオブジェクトをファイルなどに出力するまで保持しておく。

Javassist は、クラスの内観については、標準リフレクション API とほぼ同等な機能を提供する。例えば、標準リフレクション API の Field、Method、Constructor は、Javassist の CtField、CtMethod、CtConstructor に相当する。また、Class クラスが CtClass クラスに相当し、CtClass が提供するメソッドにより、CtField, CtMethod, CtConstructor などを得ることができる。表 3.3 に、CtClass クラスが提供する代表的なメソッドを示した。

また、本研究で必要となるメソッド呼び出しや、コンストラクタ呼び出しを検出するために、 Javassist の Javassist.expr パッケージを利用した。Javassist.expr.ExprEditor を用いてメソッド呼び出しを検出する簡単な例を以下に示す。

```
CtMethod cm = ...;
cm.instrument(new ExprEditor() {
    public void edit(MethodCall m) throws CannotCompileException {
```

```
        if (m.getClassName().equals("Point")) {
            System.out.println(m.getMethodName()  " line: "
                               m.getLineNumber());
    }
});
```

このコードは、メソッド cm 内に現れる全てのメソッド呼び出しを検出し、そのメソッドが Point クラス内で定義されていたら表示するというコードである。このように、Javassist.expr パッケージに用意されている ExprEditor, FieldAccess, MethodCall, NewExpr クラスを用いて本システムは実装されている。以下に、パターンが Method パターンであった場合の検索を実装する コードの一部を示す。

```
public class MethodGrep extends Grep implements TokenNum {
                    :
    public LinkedList getResult(String[] targetScope) throws Exception {
        for(int q = 0; q < classFileSize; q++) {
            fileName = targetScope[q];
            ClassPool pool = ClassPool.getDefault();
            CtClass c = pool.get(targetScope[q]);
                      :
            if (pcdNumber == CALL){
                   //ポイントカット指定子が call の場合
                CtMethod[] cm = c.getDeclaredMethods();
                for(int i = 0; i < cm.length; i++) {
                    _method = cm[i];
                    cm[i].instrument(new ExprEditor() {
                        public void edit(MethodCall m)
                                    throws CannotCompileException {
                        (matchType(ctm.getReturnType(), returnTypeName)) &&
                        (matchName(ctm.getName(), methodName)) &&
                        (matchclassName(cn,className)) &&
                        (matchArgs(ctm.getParameterTypes(),argsArray))){
                                    :
                                    :
    /**
```

```
*  修飾子、戻り値、引数、クラス名、メソッド名をチェック
*  し、検索対象にマッチしているもののみを取り出す
**/
          else if(pcdNumber == execution){
             //ポイントカット指定子が execution の場合
              CtMethod[] cm = c.getDeclaredMethods();
              for(int i =0; i < cm.length; i++){
                 :

          else if(pcdNumber == withincode){
             //ポイントカット指定子が withincode の場合
                        :
                 public void edit(MethodCall m) throws Exception {
                        :
                 public void edit(NewExpr n) throws Exception {
                        :
                 public void edit(NewExpr n) throws Exception {
                        :
```

後のコードは省略する。

関数 getResult() により、検索対象にマッチする箇所のみが抽出され、LinkedList に格納される。

表 3.2: BNF 文法

⟨ Statemetnt ⟩	::=	⟨ pointcut ⟩⟨ Operation ⟩⟨ Statement ⟩ |
		⟨ pointcut ⟩ |
		help | exit
⟨ pointcut ⟩	::=	⟨ PCD ⟩ (⟨ Pattern ⟩)
⟨ Pattern ⟩	::=	⟨ MethodPattern ⟩ |
		⟨ ConstructorPattern ⟩ |
		⟨ FieldPattern ⟩ |
		⟨ QualifiedMethodPattern ⟩ |
		⟨ pointcutPattern ⟩ |
⟨ pointcutPattern ⟩	::=	⟨ pointcut ⟩
⟨ MethodPattern ⟩	::=	⟨ Modifier ⟩⟨ Type ⟩⟨ ClassName ⟩ .
		⟨ MethodName ⟩ (⟨ Args ⟩)
⟨ ConstructorPattern ⟩	::=	⟨ Modifier ⟩⟨ ClassName ⟩ . new (⟨ Args ⟩)
⟨ FieldPattern ⟩	::=	⟨ Modifier ⟩⟨ Type ⟩⟨ ClassName ⟩ . ⟨ FieldName ⟩
⟨ QualifiedMethodPattern ⟩	::=	⟨ Modifier ⟩⟨ Type ⟩⟨ ClassName ⟩ .
		⟨ MethodName ⟩ (⟨ Args ⟩) <
		⟨ MethodName ⟩ (⟨ Args ⟩)
⟨ ClassPattern ⟩	::=	⟨ ClassName ⟩
⟨ PCD ⟩	::=	call | execution | get | set |
		wuthin | withincode
⟨ Modifier ⟩	::=	⟨ Modifier ⟩ | ⟨ Modifier ⟩⟨ Modifier ⟩
⟨ Modifier ⟩	::=	public | private | protected | static |
		abstract | final | synchronize | native |
		∗ | ε
⟨ MethodName ⟩	::=	⟨ Identifier ⟩
⟨ ClassName ⟩	::=	⟨ Identifier ⟩
⟨ FieldName ⟩	::=	⟨ Identifier ⟩

⟨ Args ⟩ ::= ⟨ Identifier ⟩ , ⟨ Args ⟩ | ⟨ Identifier ⟩ | ..
⟨ Operation ⟩ ::= || | && | , | < | <
⟨ Type ⟩ ::= ⟨ PrimitiveType ⟩ | ⟨ Identifier ⟩
⟨ PrimitiveType ⟩ ::= boolean | char | byte | short |
int | long | float | double
⟨ Identifier ⟩ ::= ⟨ Identifier ⟩ | ⟨ Identifier ⟩⟨ Identifier ⟩
⟨ Identifier ⟩ ::= ⟨ letter ⟩ | ⟨ digit ⟩ | * | _ | [|]
⟨ digit ⟩ ::= 0 | 1 | 2 ... | 8 | 9
⟨ digit ⟩ ::= a | b | c ... | y | z |
A | B | C ... | Y | Z

表 3.3: CtClass クラスの内観用メソッド

メソッドの概要	
CtConstructor[] getConstructors()	コンストラクタを得る
CtField[] getDeclaredFields()	宣言されているフィールドを得る
CtField[] getFields()	フィールドを得る
CtMethod[] getDeclaredMethods()	宣言されているメソッドを得る
CtMethod[] getMethods()	メソッドを得る
int getModifiers()	修飾子を得る
String getName()	クラス名を得る
String getPackageName()	パッケージ名を得る
CtClass getSuperclass()	親クラスを得る
void instrument(ExprEditor editor)	全てのメソッドおよびコンストラクタ本体を変更する

第4章　実験と考察

4.1　実験

highgrep を用いて検索にかかる時間を測定するため、我々は実験を行った。実験に用いた計算機は、Sun Blade 1000(UltraSPARC III 750MHz × 2, 1024MB, Solaris 8) であり、JVM は $JavaHotSpot(TM)ClientVM(build1.4.0_01-b03, mixedmode)$ を用いた。またサンプルプログラムとして、tomcat3.3.1 のクラスファイルを用意した。

4.2　検索にかかる時間の測定および評価

我々は highgrep を用いて以下に示す典型的なパターンを対象に検索を行い、実行時間の測定を行った。対象としたクラスファイルは、およそ 1 KByte から 200 KByte の大きさのファイルで、tomcat のファイルからランダムに取り出した。また検索パターンの指定にアスタリスクを用いることで、各パターンの検索にかかる最悪時間の測定を行った。

- 表 4.1:　call(* * *.*(..)
- 表 4.2:　 get (* * *.*), call (* * *.*(..))
- 表 4.3:　call (get (* * *.*) ,call (* * *.*(..))
- 表 4.4:　call (call (* * *.*(..)))
- 表 4.5:　set (* int *.*)
  set (* * *.*)
- 表 4.6:　call (call (* int *.*(..)))
  call (call (* * *.*(..)))

4.3　実験の考察

図 4.1 から 図 4.7 のグラフより、検索対象として与えるクラスファイルの大きさは、検索にかかる時間に大きく影響を与えていることがわかる。クラスファイルのサイズが大きくなると、それに比例し、検索にかかる時間も増大している。これは実装に用いた Javassist が、検索対象として与えるクラスファイルの中を最初から最後まで全て解析するというステップをパターンに応じて繰り返し行うので、当然の結果と考えられる。

また図 4.7 より、検索対象となるパターンが入れ子構造の形をしているか否かが、検索時間に大きく影響を与えていることがわかる。図 4.1 、図 4.2 のグラフより、入れ子構造をとらないパターンを対象として検索を行った際にかかった時間は、クラスファイルの大きさの増加に伴い、ほぼ 1 次曲線に近似した形で増大していることがわかる。一方、図 4.3 、図 4.4 のグラフより、入れ子構造をとるパターンを対象として検索を行った際にかかった時間は、クラスファイルの大きさの増加に伴い、2 次曲線に近似したした形で増大していることがわかる。

ファイルのサイズが小さい時は、それらの違いは微々たるものであったが、ファイルのサイズが大きくなるとその違いは顕著に現れた。検索パターンに入れ子構造、すなわち pointcutPattern を指定すると、最初に行われた検索の結果を次の検索の条件に利用するため、一連の検索処理を各結果に対して再度行わなくてはならない。このため、入れ子構造の形をとらないパターンにかかる検索時間を $O(n)$ と表すと、2 重の入れ子にかかる検索時間は $O(n^2)$ と推測できる。また図 4.5、 4.6 からも分かるように、検索パターンにより明確な条件を記述すると検索にかかる時間はかなり短くなることが分かる。これはパターンにマッチする検索個所が絞りこまれることで、検索にマッチするものの数が減らされることによると考えられる。

しかし実際には、検索対象となるファイルが持っている Java 要素の種類や、プログラムの構造といった、ファイル自体の性質にも検索時間は大きく依存すると考えられるが、今回の実験ではファイルサイズのみの違いを考慮に入れた実験しか行わなかった。そして十分に実用範囲の時間内で検索を行えることを確かめた。

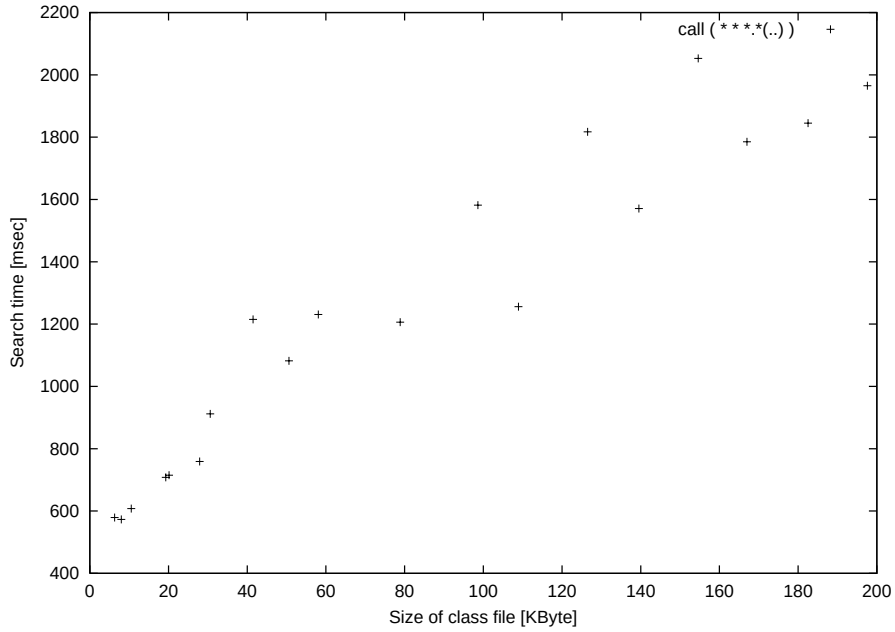

図 4.1: 検索時間　call(* * *.*(..))

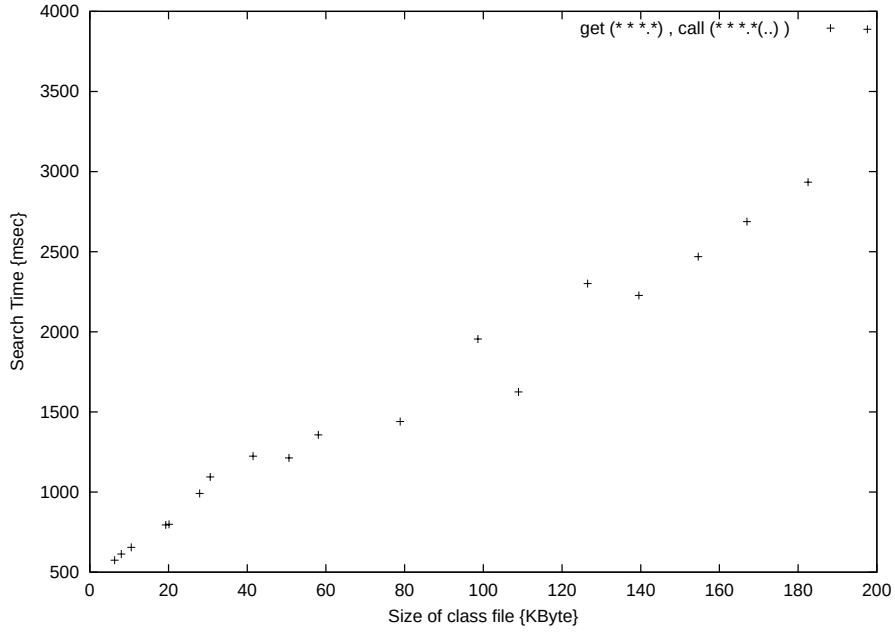

図 4.2: 検索時間　get (* * *.*), call (* * *.*(..))

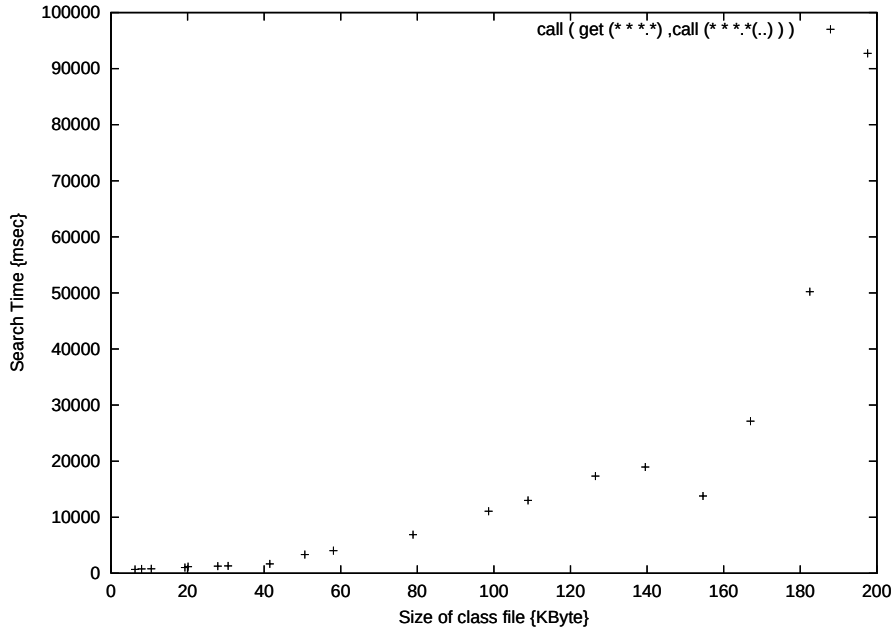

図 4.3: 検索時間　call (get (* * *.*) ,call (* * *.*(..)))

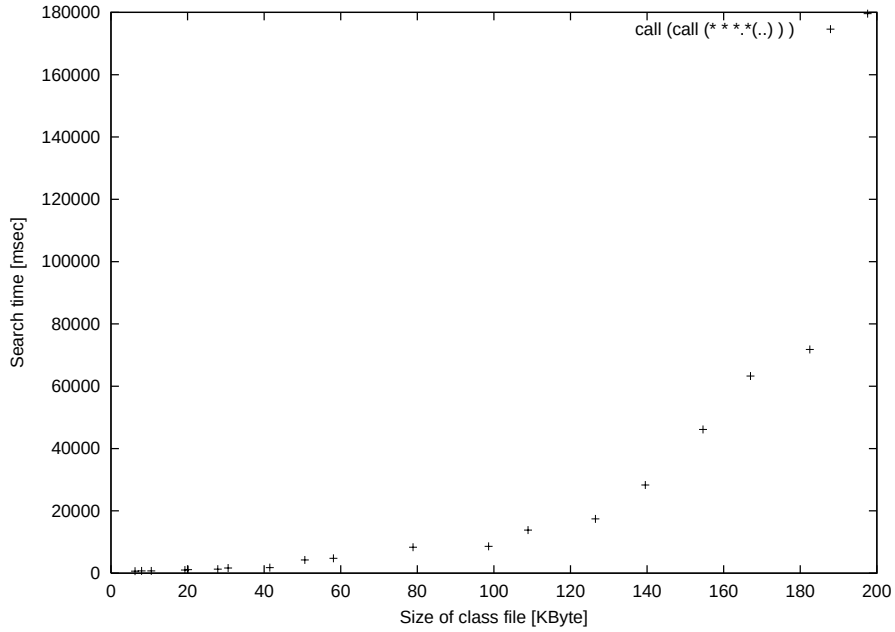

図 4.4: 検索時間　call (call (* * *.*(..)))

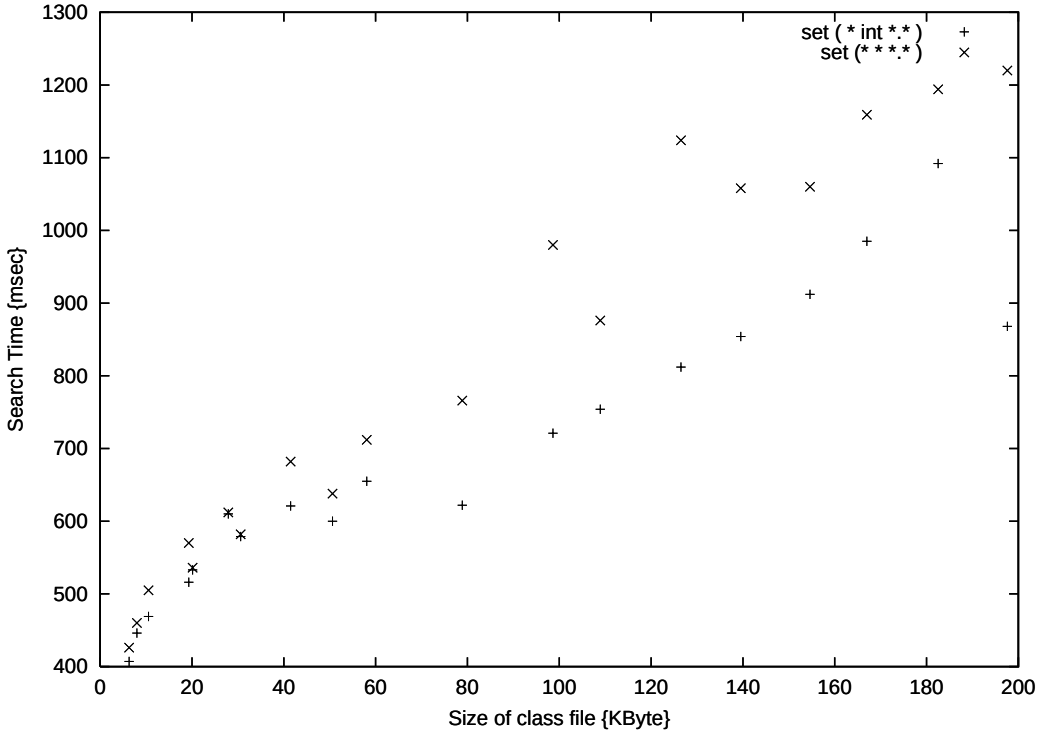

図 4.5: 検索時間　set (* int *.*) と set (* * *.*)

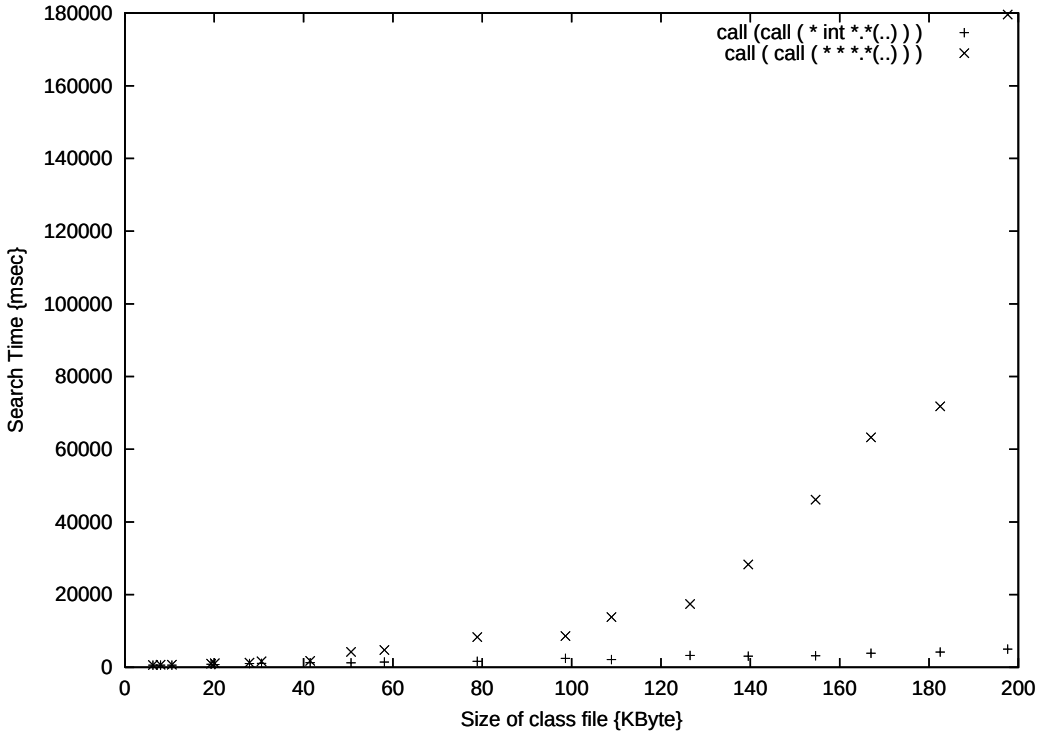

図 4.6: 検索時間　call (call (* int *.*(..))) と call (call (* * *.*(..)))

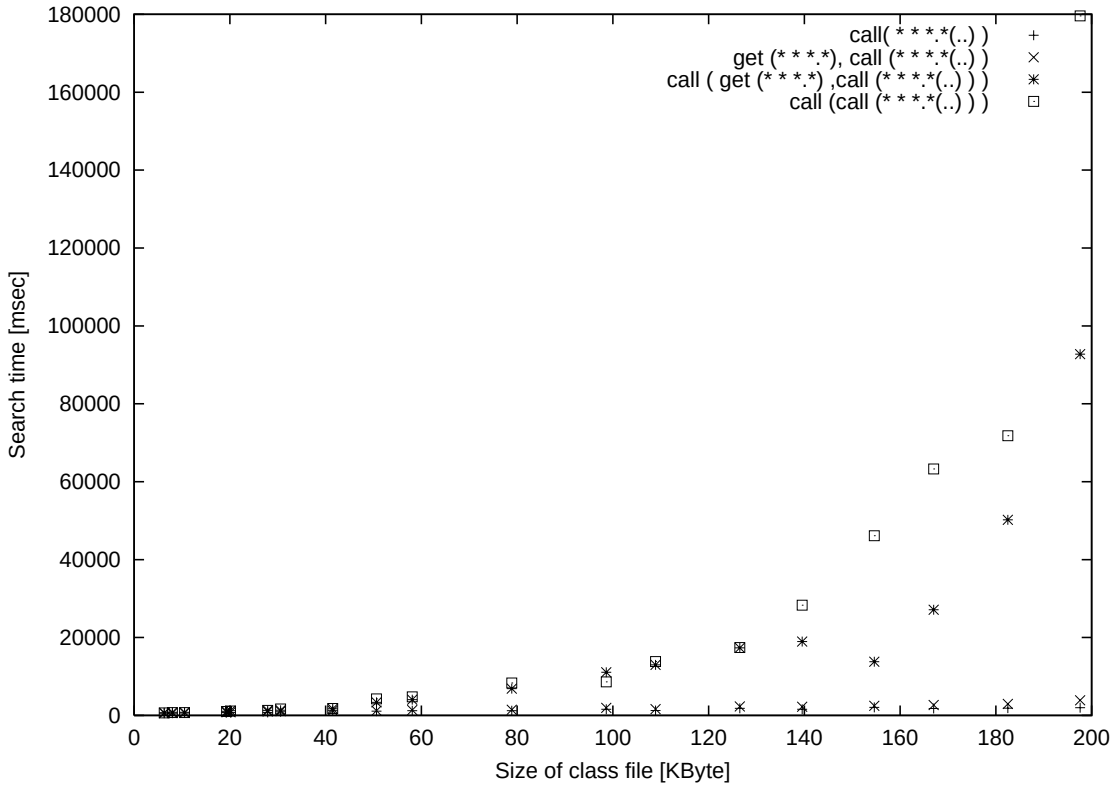

図 4.7: 検索時間　4 パターンの検索時間の比較

第5章　まとめ

5.1　本研究の貢献

本稿では、既存の検索機能では検索することができないプログラムの意味構造、および相互の関連性を意識した検索システム highgrep について述べた。highgrep は、検索パターンを記述するための言語であり、字句レベルでは追うことができないプログラムの構造を意識した検索を可能とする。

表 5.1 に、既存の検索機能と本研究で開発した highgrep の機能面での比較をまとめた。highgrep は 正規表現にアスタリスクしか利用できないため grep に比べ字句レベルでの検索機能は劣っている。しかし grep では扱うことができない、プログラムの意味構造および相互の関連性を意識した検索を行うことができるため、grep に比べより強力な検索が可能である。一方 highgrep は、Eclipse がサポートしている Java 検索機能を全てサポートしているだけでなく、より詳細な意味構造を意識した検索や相互の関連性を意識した検索を可能としているため、Eclipse に比べより強力な検索が可能である。このように我々が提案 · 開発した highgrep は、既存の検索ツールを強化するだけでなく、プログラムの相互の関連性を意識できる新たな検索個所の指定を実現した。

本システムは Javassist を使って実装されており、プログラムの内観を行っている。highgrep の開発により、ユーザにとって不必要な情報が取り出される可能性は少なくなり、より柔軟で強力な検索環境が実現した。最後に検索にかかる時間の測定を行い、十分に実用範囲の時間内で検索を行えることを確かめた。

5.2　今後の課題

現在 highgrep がサポートする検索機能は限られている。そこで、検索の幅や応用範囲を広げるために我々は以下に示す課題に取り組んでいか

表 5.1: 各検索機能の比較

	字句構造	意味構造	関連性
grep	◯	×	×
Eclipse	△	△	×
highgrep	△	◯	◯

なくてはならない。

1. 正規表現の強化
   現在 highgrep がサポートする正規表現はアスタリスクのみである。差しあたり、パーズを行なう際に支障が出ないメタ文字も扱うことができる機能をサポートすることで強力なパターン記述を実現させる。

2. AspectJ の pointcut 記述の完全サポート
   本研究でサポートしきれなかった AspectJ の pointcut 記述（例えば cflow など）を 全てサポートすることで、幅広い検索パターンの記述を実現させる。

3. テキスト処理のサポート
   検索結果を置換 · 削除できるような一般的なテキスト処理を行える機能のサポートを実現させる。

4. 他言語への拡張
   本研究が対象とした検索対象のファイルは、Java 言語に依存したものであった。しかしプログラム言語に依存しないファイルを指定できるような機能拡張が行える検索システムのサポートを実現する。

参考文献

[1] AspectJ: Aspect-Oriented Programming (AOP) for Java, `http://www.eclipse.org/aspectj`.

[2] Chiba, S.: Load-time Structual Reflection in Java, *In ECOOP 2000 - Object-Oriented Programming, LNCS 1850*, pp. 313–336 (2001).

[3] Eclipse: eclipse.org, `http://www.eclipse.org`.

[4] Friedl, J. E. F.: *Mastering Regular Expressions, 2nd Edition*, O'Reilly (2002).

[5] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G.: An Overview of AspectJ, *Proceedings of the European Conference on Object-Oriented Programming* (2001).

[6] Kiczales, G., Lamping, J., Anurag Mendhekar, C. M., Lopes, C., Loingtier, J.-M. and Irwin, J.: Aspect-Oriented Programming, *Proceedings of the European Conference on Object-Oriented Programming*, pp. 220–242 (1997).

[7] Tatsubori, M., Sasaki, T., Chiba, S. and Itano, K.: A Bytecode Translator for Distributed Excecution of Legacy Java, *Proceedings of the European Conference on Object-Oriented Programming*, pp. 236–255 (2001).

[8] 千葉滋: Javassist Home Page, `http://www.csg.is.titech.ac.jp/~chiba/javassist/index.html`.

[9] 千葉滋, 立堀道昭: Javaバイトコード変換による構造リフレクションの実現, 情報処理学会 論文誌, Vol. 42, No. 11, pp. 2752–2760 (2001).

[10] 中川清志, 立堀道昭, 千葉滋: Josh：バイトコードレベルでの Java 用 Aspect Weaver, 第 5 回プログラミングおよび応用のシステムに関するワークショップ (2002).

付 録A　利用例および出力結果

検索の対象としたプログラム

```
/** Point.java **/
public class Point {
    private int x;
    private int y;
    public Point(int x0, int y0) {
        x = x0;
        y = y0;
    }

    public void move(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
    public int getX() { return x; }
    public int getY() { return y; }
}

/** Q1.java **/
public class Q1 {
    int aaa = 0;
    int bbb = 0;

    public void move(int x0, int y0, int x, int y) {
        Point p = new Point(x0, y0);
        move(p,x,y);
    }
```

```
    private void move(Point p, int x, int y) {
        System.out.println(aaa);
        System.out.println(bbb);
        p.move(x,y);
    }
    public static void main(String[] args) {
        Q1 q = new Q1();
        q.move(3,4,3,1);
    }
}
```

入力および出力結果

```
spark:~/Program/Program[8]% java MainGrep Q1.class Point.class
>call(* * *.*(..))
line:6;  Q1  move
line:9;  Q1  println
line:10;  Q1  println
line:11;  Q1  move
line:15;  Q1  move
>set (* * *.*)
line:9;  Point  x
line:10;  Point  y
>call(* * *.move(..))
line:6;  Q1  move
line:11;  Q1  move
line:15;  Q1  move
>get(* * *.x)
line:9;  Point  x
line:12;  Point  x
>get(* * *.x),get(* * *.y)
line:9;  Point  x
line:10;  Point  y
>call(call(call(* * *.*(..))))
line:15;  Q1  move
>execution(* * *.*(..)) > call(* * *.move(..))
line:4;  Q1  move
line:8;  Q1  move
line:13;  Q1  main
>execution(* * *.move(Point, int, int)) >call(* * *.move(..))
line:8;  Q1  move
  ....
```