平成 8 年度

筑波大学第三学群情報学類

卒業研究論文

インクリメンタルなコンパイラのための エディタと字句解析器の提案と実装

主専攻		情報科学
著者名		立堀 道昭
指導教員	電子・情報工学系	中田 育男
	電子・情報工学系	山下 義行

要旨

一度解析したプログラムの変更に対して、変更箇所から影響を受ける範囲のみの再解析を行なうインクリメンタルな解析について研究が行なわれている。このような解析では変更前のソースファイルと変更後のソースファイルとの差分を検出することが不可欠であり、そのために様々な方法が考えられ、また提案されている。本研究では、これらの方法を検証し、最適化された方法を提案する。また、本研究室で開発されている、インクリメンタルな構文解析器のために、提案した方法を使ったエディタと字句解析器を実装した。

本論文で提案する手法は Galaxy システムで用いられている KMN 法を拡張したものである。KMN 法はエディタから得られる編集情報から差分を検出するものであるが、編集された内容は見ないために実際には再利用できる部分も再利用不可部分としてしまう。本方法（Restorable-KMN 法）は得られた差分情報とソースファイルを検証することにより、より多くの部分を再利用できるような差分情報を編集時に残すようにしたものである。

もくじ

1 序論 **1**

1.1 研究の背景 1

1.2 研究の目的 1

1.3 構成 1

2 基礎概念 **2**

2.1 インクリメンタルなコンパイラ 2

2.1.1 一般的なインクリメンタルな LR 構文解析法 2

2.1.2 想定するパーザ 3

2.2 KMN 法 3

2.2.1 KMN リスト 3

2.2.2 仕様 4

3 ソースファイルのみからトークンの差分を抽出する方法の検証 **6**

3.1 ファイル間の差分抽出アルゴリズム 6

3.1.1 LCS 問題を解くアルゴリズム 6

3.1.2 適当な解で満足するアルゴリズム 7

3.1.3 実装と評価 7

3.2 トークンレベルの差分抽出への適用 9

3.2.1 予測 9

3.2.2 変換のアルゴリズム 9

3.2.3 実装と評価 10

4 KMN リストを用いたトークンの差分抽出 **12**

4.1 KMN リストを用いたトークンの差分抽出 12

4.1.1 編集の判定法 13

4.1.2 トークンの再利用と再解析の終了条件 15

4.1.3 アルゴリズム 16

4.2 編集されているが再利用可能なトークンの抽出 18

4.2.1 実現方法 18

4.3 実装と評価 19

5 Restorable-KMN 法 21

5.1 補正の方法 21

5.1.1 次々に補正してしまうことにより起きる問題 21

5.1.2 補正情報を別に保持する方法 23

5.1.3 バックグラウンドでの補正 23

5.2 復活のための差分抽出法 23

5.2.1 単純な文字列比較 23

5.2.2 トークンの文字列のパターンマッチ 23

5.2.3 字句解析 23

5.3 実装と評価 24

6 結論 26

参考文献 29

第 1 章

序論

1.1 研究の背景

プログラム開発時の修正に対し、修正箇所から影響を受ける範囲のみの再コンパイルで済めば開発にかかる時間的なコストを削減できる。このようなコンパイルをインクリメンタルなコンパイルという。本研究室において、インクリメンタルな構文解析及び意味解析については研究され実装されている [4, 5]。そのような解析器にはプログラムのソースファイルの変更箇所を指示してくれる字句解析器が不可欠である。

1.2 研究の目的

本研究室の中井らが提案し、実装したインクリメンタルな構文解析器が存在する。この解析器の要求を満たす、変更されたトークンを指示できる字句解析器、及び、差分抽出を補助するエディタを提案し、実装する。これまでにもそのようなエディタは研究されている [1] ので、それらを利用しかつよりよいものにする。

1.3 構成

本研究では、編集前と編集後のソースファイル間における、トークンの差分を抽出することが必要であり、先に 2章では、対象となるインクリメンタルなコンパイラについて触れる。また、基礎となる、エディタで編集時にソースファイルの差分を保持する方法について触れる。

3章では、編集前後のソースファイルのみからトークンの差分をとる方法で差分抽出器を提案し、実装して検証する。 4章では、 3章の方法での実行時間の遅さを解消するために、エディタから得られた情報からトークンの差分情報を構成する方法を提案し、実装して検証する。

5章では、 4章で差分の精度を上げようとして、実行時間が遅くなってしまった点を解消して、トークンの再利用率が高くかつ高速にトークンの差分を得るためにエディタから得られる情報を拡張することを提案し、実装して検証する。

最後に 6章で以上についてまとめ、結論を述べる。

第 2 章

基礎概念

まず、本論文で対象とするインクリメンタルなコンパイラについて説明し、次に、エディタでの編集時に編集箇所を保持する方法を説明する。

2.1 インクリメンタルなコンパイラ

通常、プログラムの開発時には繰り返してソースファイルの修正とコンパイルを行なう。このとき、修正箇所から影響を受ける範囲のみの再コンパイルで済めば開発にかかる時間的なコストを削減できる。このようなコンパイルをインクリメンタルなコンパイルという。

2.1.1 一般的なインクリメンタルな LR 構文解析法

図 2.1を使ってインクリメンタルな LR 構文解析の概略を説明する。

ある入力記号列 $w = xzy$ があり、これに対して解析木を作りながら構文解析が行われているとする。この解析木を t_1 とする。部分的な解析木の葉を左から右へ並べたのものは、その解析木のへり (frontier) と呼ばれるが、ここで t_1 において、z をへりの中に含むある部分木を $t_1^{'}$ とする。変更後の入力記号列を $w^{'} = xz^{'}y$ とし、解析木を作りながら構文解析するとし、その解析木を t_2 とする。t_2 の $z^{'}$ をへりとして含む部分木 $t_2^{'}$ として、図 2.1の影をつけた 2 つの部分が等しくなるものを見つければ、$t_1^{'}$ を $t_2^{'}$ で置き換えることで w の解析結果から $w^{'}$ の解析結果を得ることができる。

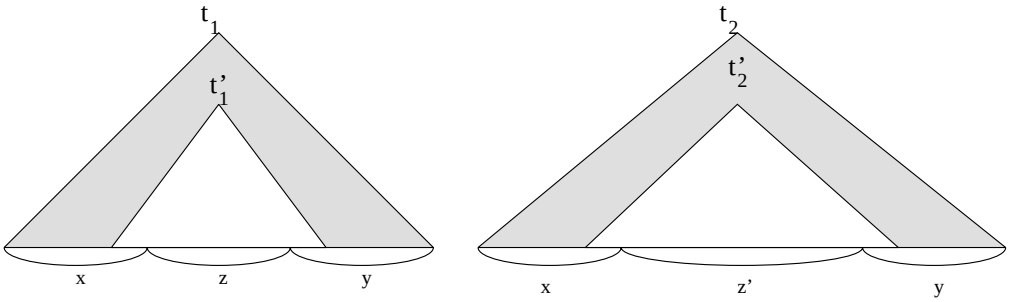


図 2.1: 入力 $w = xzy$ と $w^{'} = xz^{'}y$ の解析木

LR(k) 構文解析の場合、入力記号列 w の z を w' の z' に置き換えても、入力の最初から z の最初の記号より k 個前までの解析は以前と同じであることが保証される。このことから、 w' に関する解析は、 z' によって影響を受け始めるところから解析を始めることができる。このことは、 x のうち z から k 個前の部分までをへりとする部分木列を再利用することを意味する。

それ以降は、入力の終りまで通常の LR 構文解析をすればよい。しかし、実際にはそれ以降の部分木列中には以前の解析結果と同じ形になる部分木が多く含まれる可能性がある。すなわち、上で述べたようにして t'_1 を t'_2 に置き換えてもよいことがわかれば、部分木の置換えを行うことによって、それ以降は解析をしなくて良い。すなわち、それ以降の部分木を再利用できる。

2.1.2 想定するパーザ

ここで対象とするインクリメンタルなコンパイラは、構造エディタ [2, 3] を用いないでプログラムの自由な変更に基づいてインクリメンタルに構文解析を行なうものであり、本研究室[1]ではそのための手法を提案している。そこで、本論文では本研究室の中井らが設計・実装したパーザ [4, 5] を想定し、そのための字句解析器、字句差分抽出器を提案し、実装する。

このパーザは構文解析だけでなく、更に意味解析やコード生成もインクリメンタルに行なうものであり、再利用性を高めるなどの最適化がなされているが、基本的な部分では 2.1.1で挙げた一般的なインクリメンタルな LR 構文解析法に基づいている。字句差分抽出器はパーザとのやりとりを考えるだけでよく、編集前のソースファイルと編集後のソースファイルの間のトークンの差分情報を与えるだけでよい。

2.2 KMN 法

本研究ではエディタからソースファイルの差分を抽出するために KMN 法を採用、拡張している。KMN 法はインクリメンタルなコンパイラシステム Galaxy [1] で提案され、使用されているもので、編集時に差分を残す方法として適している。以下では 2.2.1で KMN 法で使われる KMN リストの説明をし、 2.2.2で KMN 法の仕様を説明する。

2.2.1 KMN リスト

Galaxy ではエディタによる編集情報は KMN リストに保持されることになる。

通常、差分情報は同じ要素の対の列で表す。例えば、ある要素の列で、

$$abcdef \Rightarrow abDef$$

というように左の要素列が右の要素列に変更があったときには、

$$(1,1),(2,2),(5,4),(6,5)$$

[1]筑波大学電子情報工学系プログラミング言語研究室

という列で差分情報を表す。これは、左の要素列と右の要素列で、 1 番目が 1 番目に、 2 番目が 2 番目に、 5 番目が 4 番目に、 6 番目が 5 番目に、それぞれ同じ要素が対応していることを表している。

これに対し、 KMN リストは編集結果を 「 k 文字はそのままで、その次の m 文字を n 文字に置き替えた」という情報を (k,m,n) という要素として、その要素の列で表すものである。あくまでも編集結果として、もとのソースファイルから現在のソースファイルが変更されたところを保持しておくことになる。

例えば、 KMN リスト

$$(15,0,10),(20,10,5),(30,0,0)$$

は、「最初の 15 文字はそのままで、その後に 10 文字を挿入、次の 20 文字はそのままで、その後の 10 文字を 5 文字に置き替え、残りの 30 文字はそのまま」ということを表す。（図 2.2参照。）

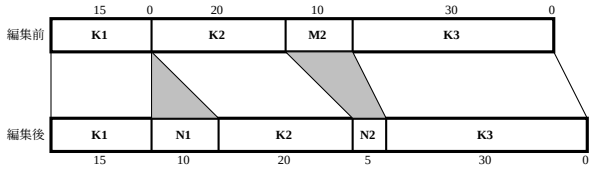


図 2.2: KMN リストの例

エディタで編集情報を保持するためには通常の差分情報よりも KMN リストのほうが都合がよい。通常の差分情報では、挿入または削除があるたびに、列のそれより右の要素を全て更新しなければならないが、 KMN リストでは編集部分を含んでいる要素のみを更新するだけでよい。

2.2.2 仕様

KMN 法では既に編集がなされたソースファイルに対して更に編集があったときに、エディタは最新の変更についての情報を与えるだけでよく、 KMN リストはその情報に基づいて更新される。

例えば、解析済みの最初のソースファイルに編集が加えられ、

$$abcdef \Rightarrow abDef$$

となったときに、 KMN リストは

$$(2,2,1),(2,0,0)$$

になる。（ab の 2 文字がそのままで cd の 2 文字が D の 1 文字に変わり、次の ef の 2 文字はそのまま、ということを示している。）更に編集が加えられソースファイルが

$$abcdef \Rightarrow abDef \Rightarrow abDeFG$$

となったときに、エディタは情報として KMN 要素

$$(4,1,2)$$

を与えるだけでよい。つまり、直前のソースファイルに対する変更の情報だけを与えればよく、この場合は2番目のソースファイルに対する変更の情報だけでよい。（2番目のソースファイルの $abDe$ の4文字がそのままで f の1文字が FG の2文字に変わった、ということを示している。）この情報から KMN リストは

$$(2,2,1),(1,1,2),(0,0,0)$$

に更新される。つまり、1番目のソースファイルに対して3番目のソースファイルがどう変わったかという変更の情報に更新される。

リストの終りは (kmn) の内、 m と n の両方が 0 になることで表されている。

KMN リストを更新するアルゴリズムについては [1] を参照されたい。

第 3 章

ソースファイルのみからトークンの差分を抽出する方法の検証

エディタからの編集情報がなくとも、編集前のソースファイルと編集後のソースファイルさえあればトークンの差分を抽出することができる。実際、本研究室の中井らのパーザ [4, 5] はソースファイルの差分をとるためにこの方法を用いて実現している。

この方法は、後述（4章）する KMN 法による方法に比べ、ファイル全体としてのグローバルな差分を抽出できる。そのため、編集が大幅に行なわれた場合には特に KMN 法によるものに比べトークンの再利用率は上がる。しかし、試験的実装ということもあり、中井らの採用した差分抽出アルゴリズム及び実装方法では効率が悪過ぎ、このままではコンパイラ全体としての実行時間が通常の（インクリメンタルでない）コンパイラより遅くなってしまっている。つまり、構文解析と属性評価のみの計算時間で通常のコンパイラよりもインクリメンタルなコンパイラは優れているが、差分抽出（通常のコンパイラでは必要ない）までを含めた場合に劣ってしまっている。

この章では得られた編集後のソースファイルと編集前のソースファイルのみからトークンの差分を抽出するために考えられる手法を検証する。

3.1 ファイル間の差分抽出アルゴリズム

以下ではファイル間の差分抽出をするためのアルゴリズムについて検討する。

3.1.1 LCS 問題を解くアルゴリズム

トークンの並びには重要な意味があり、２つのソースファイルの要素（トークン）を単調に（交差せずに）対応づける必要がある。したがって、編集の前と後のソースファイルにおける、トークンの差分を抽出する問題は、最適解（最小差分）を求める場合には LCS 問題 (Longest Common Subsequence Problem) [6] に相当する。 LCS 問題には様々なアルゴリズムが提案されている [7] が、よく似たファイルを比較する場合には、高速法 [8] が、計算時間、計算領域ともに少なくてすむ。中井らが採用したのはこれより単純で、計算時間、計算領域ともに劣る 2 次法 [9] というアルゴリズムであった。

3.1.2　適当な解で満足するアルゴリズム

一方で、本研究の問題では必ずしも最小差分を求める必要はない。そこで、計算時間が短くて済むなら適当に最小に近い差分で満足した方がよい。なぜなら、インクリメンタルなコンパイラはコンパイルの時間を短くするためのものであり、確かに差分が小さくて済むほどインクリメンタルな構文解析や属性評価は早く完了するが、そのための字句解析でコンパイル全体の計算時間が長くなっては意味がないからである。

このようなヒューリスティックに適当な解で満足するようなアルゴリズムは数々提案されている[10]。

3.1.3　実装と評価

3.1.1で挙げた最適解を求めるアルゴリズムと、3.1.2で挙げたヒューリスティックに適当な解で満足するアルゴリズムとの比較を行なう。

GNU により実装されている diff コマンド[1]では、最適解を求めるアルゴリズムと適当な解で満足するアルゴリズム [11] の両方が採用され選んで使えるようになっている。本研究ではそのソースプログラム（diffutils version 2.7.1）を参考にし、その差分抽出ルーチンに手を加え、任意の要素単位で差分抽出できるようにし、最適解を求めるものと、適当な解で満足するものの２つを実装した。

差分をとる対象を Pascal-S プログラムのトークン列とした。ソースファイルとして、2000 行程度の Pascal-S で書かれたプログラムファイルのトークン列（10000 トークン程度）を用意し、それとほとんど内容の同じプログラムファイルのトークン列との差分、及び、ほとんど内容の異なるプログラムファイルのトークン列との差分をとった。各種のトークン総数（1000 個から 8000 個）で比較するために、それらのうちの一部分を与えた。

内容の異なる度合を、「一方のファイルからもう一方のファイルに変換するのに最小限必要な削除と挿入の回数」を「２つのファイルの要素数の総和」で割ったもので表すことにする。本実験では、ほとんど内容の異なるファイルとして、70% 程度のトークンが異なるものを採用し、ほとんど内容の同じファイルとして、1% 程度のトークンが分散して異なるものを採用した。

なお、作成した適当な解で満足する差分抽出器は最小の差分よりも冗長な差分を抽出するが、最小の差分に対する冗長な分の割合（((冗長な差分 – 最小差分)/最小差分）は図 3.1に示される程度であった。ほとんど内容の同じファイル間では全て最適な解が得られている。ほとんど内容の異なるファイル間でもトークンの数が 1000 個程度までは最適な解がえられており、トークンの数がそれ以上の場合でも異なる箇所の分散による（分散が大きいほど精度は高くなる。）が、5% 程度冗長なだけである。

実行時間は Sillicon Graphics 社の Indy (OS:IRIX5.2,CPU:MIPS R4000, 200MHz) 上で pixie コマンド[2]を用いて測定した。

[1] ファイル間の差分を行単位で表示するコマンド

[2] 実行にかかったマシンインストラクション数とサイクル数を得るためのプロファイルをとるための実行ファイルを作成するコマンド

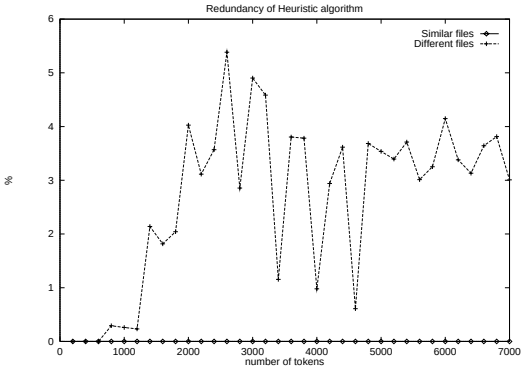
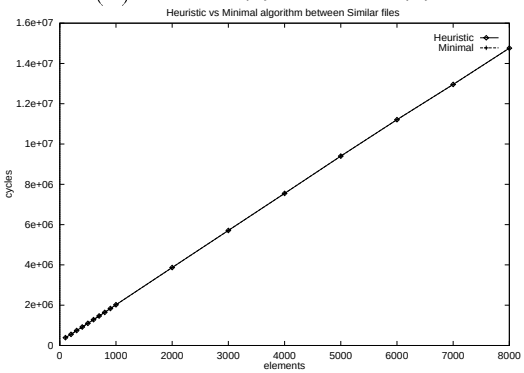


図 3.1: 適当な解で満足する差分抽出器で得られる差分の冗長度

図 3.2(1) は、ほとんど内容がおなじファイル間において、また、図 3.2(2) は、ほとんど内容が異なるファイル間において、それぞれ、最適解を求めるアルゴリズムと適当な解で満足するアルゴリズムを実行時間で比較したものである。

(1) ほとんど同じファイル間



(2) ほとんど異なるファイル間

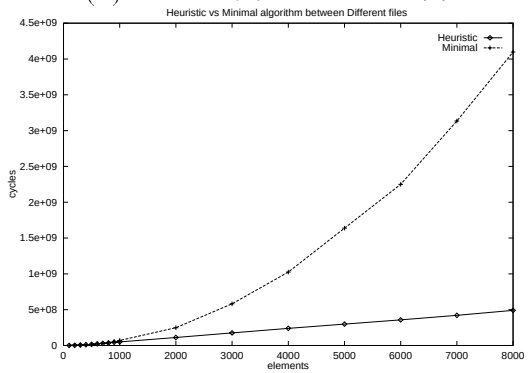


図 3.2: 差分抽出のアルゴリズムによる比較

図 3.3(1) は、最適解を求めるアルゴリズムにおいて、また、図 3.3(2) は、適当な解で満足するアルゴリズムにおいて、それぞれ、ほとんど内容の同じ（1% 程度異なる）ファイル間の差分を抽出する場合とほとんど内容の異なる（70% 程度異なる）ファイル間の差分を抽出する場合を比較したものである。

ほとんど内容の同じファイル間では実行時間にはほぼ差がない。ほとんど内容が異なるファイル間でその差がでており、ヒューリスティックに適当な解で満足する差分抽出器のほうが高速である。対象のファイルのトークン数が多くなるほど、その差が開くのが分かる。

実行時間の利に比べ、ヒューリスティックに適当な解で満足する差分抽出器で得られる差分の冗長度はそれほど高くなくパーサーの実行時間に及ぼす悪影響は小さいとしてよいであろう。

したがって、ファイル間の差分をとるためにはヒューリスティックに適当な解で満足する差分抽出器を採用するのが適当である。

(1) 最適解を求めるアルゴリズム　　(2) 適当な解で満足するアルゴリズム

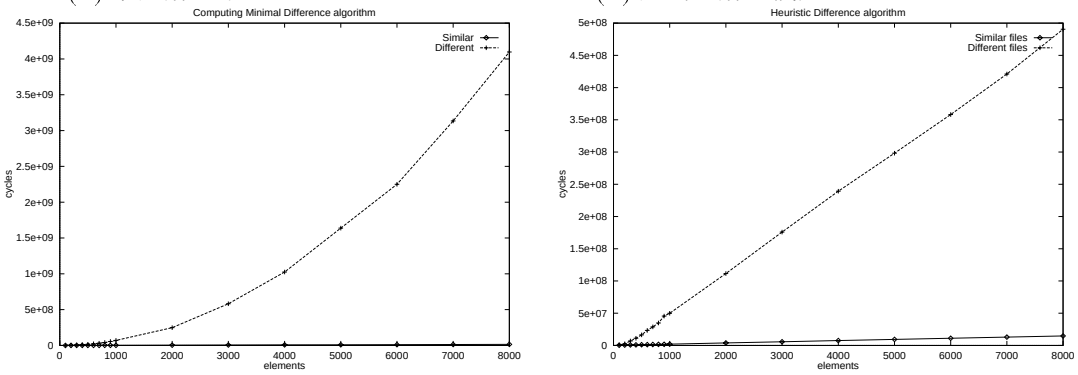


図 3.3: 差分抽出のアルゴリズムによる比較

3.2 トークンレベルの差分抽出への適用

トークンレベルの差分を抽出するためには、編集前のファイルと編集後のファイルにおいて、以下のような方法がとれる。

1) テキストレベルの差分をとりトークンレベルの差分に変換する方法

2) それぞれのファイルをトークン列に変換してからトークンレベルの差分をとる方法

3.2.1 予測

1) の方法ならば、あらかじめ獲得されている編集前のソースファイルのトークン列を利用して、インクリメンタルに字句解析が可能である。一方、 2) の方法ではインクリメンタルな字句解析はできないが、差分抽出の問題規模が小さくなるので、差分抽出の時間は 1) より短くなる。

また、 2) で得られる差分情報は 1) よりも有益で、トークンの再利用率が上がる。

3.2.2 変換のアルゴリズム

得られた差分は編集前と編集後のソースファイル間の同じトークン番号の対の列で表されており、構文解析器が必要としている情報、すなわち、トークン列のうち再利用できる範囲を与える情報に変換する必要がある。変換のアルゴリズムを示しておく。

```
type diffInfo = record
    begin
        oldtokenI  /* index of old token */
        newtokenI  /* index of new token */
        next       /* pointer to next this element */
    end

procedure diffToLexer( diffInfo D, var DiffForLex )
begin
    oldtp = 0;
    newtp = 0;
    k := 0 ;
    m := 0 ;
    n := 0 ;
```

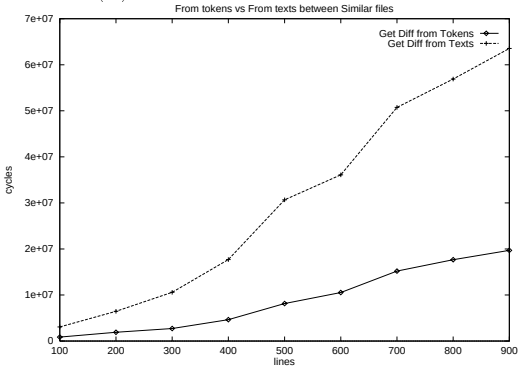
```
    while di != null
        /* does deleted ? */
        oldtp := oldtp + 1 ;
        if oldtp <= D.oldtp
            m := D.oldtp - oldtp + 1 ;
            oldtp := oldtp + m ;
        endif
        /* does inserted ? */
        newtp := newtp + 1 ;
        if newtp <= D.newtp
            n := D.newtp - newtp + 1 ;
            newtp := newtp + n ;
        endif
        /* if deleted or inserted, add new element to list */
        if m = 0 and n = 0
            k := k + 1 ;
        else
            add ( k m n ) list DiffForLex ;
            k := 1 ;
            m := 0 ;
            n := 0 ;
        endif
    end while
    m := length of old tokens - oldtp ;
    n := length of new tokens - newtp ;
    if m != 0 or n != 0
        add ( 0 0 0 ) to list DiffForLex ;
    endif
end
```

3.2.3 実装と評価

ヒューリスティックに適当な解で満足するアルゴリズムを用いた差分抽出器を作成し、まず、差分抽出の時間計算量を比較した。ソースファイルとして、 Pascal-S の 1000 行程度の、内容がほとんど同じものとまったく異なるものの２種類２組を用意し、その一部で各種の行（100 行から 800 行）を作成し与えた。

結果は図 3.4のようになった。

(1) ほとんど同じファイル間



(2) ほとんど異なるファイル間

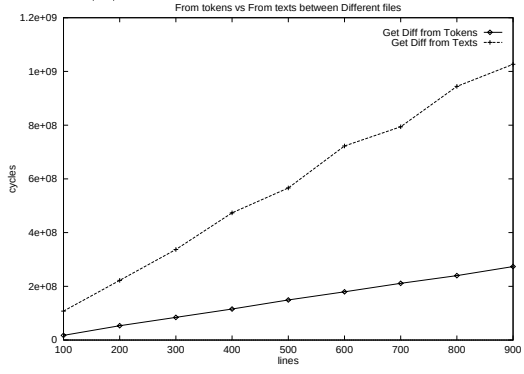


図 3.4: 差分抽出の段階（テキストレベル、トークンレベル）による比較

トークン列に変換してから差分をとったほうがテキストの差分をとるより３倍ほど高速である。

さらに、絶対的な実行時間の差が最も小さい、内容がほとんど同じファイル間で差分をとる場合について、字句解析までを含めて検証する。トークン列に変換する分の実行時間をトークン列に変換し

てから差分をとる実行時間に加えたものと、さきほどと同じテキストの差分をとる方法の実行時間と比較すると図 3.5のようになった。

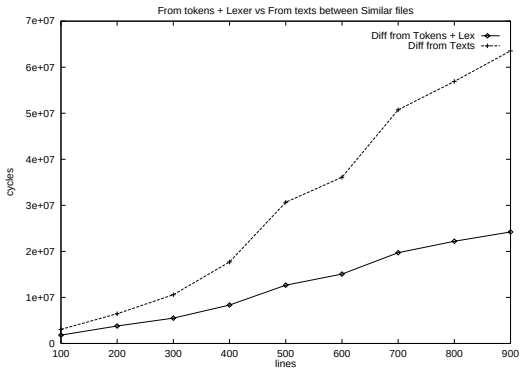


図 3.5: トークン化 + トークンの差分抽出対テキストの差分抽出

インクリメンタルに字句解析を行なう時間を入れるまでもなくテキストの差分抽出は時間がかかり、トークン列に変換してからトークンの差分をとったほうが高速であることがわかる。

第 4 章

KMN リストを用いたトークンの差分抽出

この章では、編集前後のソースファイルに加えて、エディタから得られた KMN リストを用いて構文解析器のために編集前後のトークンの差分を抽出する方法を述べる。

4.1では、編集された部分のトークンは変更されたものとして単純に差分を抽出する方法を述べる。4.2では、そこから更に編集されている部分のうち再利用できる部分を再利用する方法を述べる。

4.1 KMN リストを用いたトークンの差分抽出

編集時に KMN 法を用いてエディタで作成された KMN リストの i 番目 $(i = 0, 1, 2, \ldots)$ の要素を (K_i, M_i, N_i) とする。

また、編集前のソースファイルから得られたトークン列の j 番目 $(j = 0, 1, 2, \ldots)$ のトークンを T_j で表し、編集後のソースファイルのトークン列の j' 番目のトークンを $T'_{j'}$ で表す。$|T_j|$ はトークン T_j の文字数とその直前の意味のない空白文字列（スペース、タブ、コメント等、トークンの区切りにはされるがトークンにならない文字列）の文字数の和を表すものとする。$|T'_{j'}|$ も同様である。

KMN リストの i 番目の要素に対応する部分の文字列中の α 文字目が、編集前のソースファイル全体の先頭から $X_i(\alpha)$ 文字目、及び、編集後のソースファイル全体の先頭から $X'_i(\alpha)$ 文字目に対応しているとする。（図 4.1参照）このとき、$X_i(\alpha)$ と $X'_i(\alpha)$ は i と α を用いて以下のように表せる。

$$X_i(\alpha) = \sum_{p=0}^{i-1}(K_p + M_p) + \alpha$$

$$X'_i(\alpha) = \sum_{p=0}^{i-1}(K_p + N_p) + \alpha$$

本章では説明のためにこの $X_i(\alpha)$ と $X'_i(\alpha)$ を用いる。

トークンはその文字列自体は変更されていなくてもそれより左方のトークンの変更の影響を受けて変更されてしまう可能性がある。そのため、トークンが変更されたか否かを特定するためには、編集前のトークン列の先頭からみていくことが必要である。

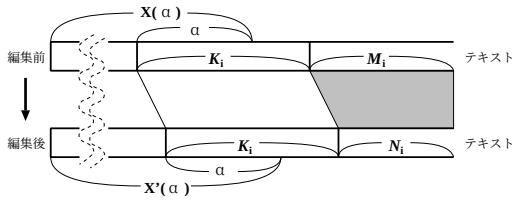


図 4.1: KMN の要素の文字列とソースファイル全体の文字列の関係

ここでは、差分検出のための解析はトークンの再利用と再解析をトークン列の先頭から交互に繰り返すことにより行なうことにする。すなわち、 KMN リストの先頭から参照していき、 K に対応する部分を再利用し、 M に対応する部分は再利用せず、代わりに N に対応する部分の字句解析をしてトークンの変更後の部分とする、という解析を繰り返していく。（図 4.2参照）

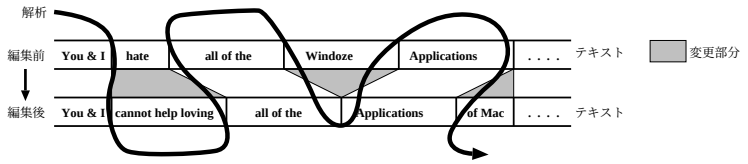


図 4.2: 解析順の概念図

4.1.1 編集の判定法

先に、文字が編集されているか否か、そして、トークンが編集されているか否かを検査する方法を述べておく。

文字の編集の判定

$X_i(\alpha)$ において、 α 文字目が K_i に対応する部分にあるとき、すなわち、 $1 \leq \alpha \leq K_i$ のときは編集前のソースファイルの $X_i(\alpha)$ 文字目は編集されていないといえる。このとき、編集前のソースファイルの $X_i(\alpha)$ 文字目は編集後のソースファイルの $X'_i(\alpha)$ 文字目として残っていることになる。

一方、 α 文字目が M_i に対応する部分にあるとき、すなわち、 $K_i + 1 \leq \alpha \leq K_i + M_i$ のときは編集前のソースファイルの $X_i(\alpha)$ 文字目は編集されたといえる。

トークンの編集の判定法

「トークン T_j が KMN リストの i 番目の要素に対応」を「トークン T_j の直前の空白文字列の先頭の文字が KMN リストの K_i または M_i に対応する文字列に含まれていること」と定義する。すなわ

ち、以下を満たすとき、トークン T_j は KMN リストの i 番目の要素に対応している。

$$X_i(1) \leq \sum_{p=0}^{j-1}(|T_p|) + 1 \leq X_i(K_i + M_i)$$

編集後のトークン $T'_{j'}$ についても同様に定める。すなわち、以下を満たすとき、トークン $T'_{j'}$ は KMN リストの i 番目の要素に対応している。

$$X'_i(1) \leq \sum_{p=0}^{j'-1}(|T'_p|) + 1 \leq X'_i(K_i + N_i)$$

この定義により、全てのトークンはある KMN リストの要素に対応づけられる。

いま、トークン T_j が左方のトークンの変更で影響を受けておらず、KMN リストの i 番目の要素に対応しているとする。このとき、トークンの文字列が K_i に対応する部分の文字列に完全に含まれ、かつ、そのトークンが前の（左方の）トークンの変更で影響を受けていなければ、再利用できる部分としてよい。したがって、トークン T_j が編集されたか否かは以下の 3 つの場合のどれにあてはまるかで決めることができる。

(1) トークン T_j の文字列は編集されておらず、トークンは変更されていない場合。

$$\sum_{p=0}^{j}(|T_p|) < X_i(K_i)$$

このトークン T_j は再利用してよい。（図 4.3(1) 参照）

(2) トークン T_j の文字列は編集されていないが、右方の文字列の編集の影響をうけ、トークンが変更された可能性がある場合。

$$\sum_{p=0}^{j}(|T_p|) = X_i(K_i)$$

編集後のソースファイル中のトークン T_j の先頭に対応する部分から字句解析をし、得られたトークンが T_j と同じであればトークン T_j は変更されていないとしてよく、(1) の場合と同様に処理できる。そうでなければ、再利用できないものとする。（図 4.3(2) 参照）

(3) トークン T_j の文字列が編集されている場合。

$$\sum_{p=0}^{j}(|T_p|) > X_i(K_i)$$

このトークンは再利用できないものとする。（図 4.3(3) 参照）

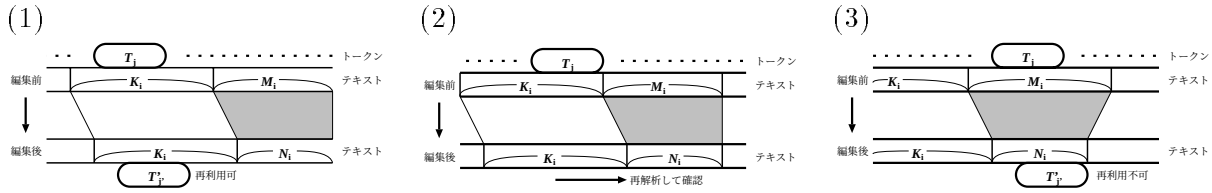


図 4.3: KMN リストとトークンとの関係

4.1.2 トークンの再利用と再解析の終了条件

再利用と再解析の繰り返しにおいて、再利用を終了し再解析に移る条件と、再解析を終了し再利用に移る条件を述べる。

ただし、再利用可不可の検査が KMN リストの i 番目の要素に対応する編集前のトークンから始まっているとする。

再利用

いま、再利用を始めるトークンが KMN リストの i 番目の要素に対応しており、再利用を始める最初のトークンのインデックスとして u が与えられたとする。

このとき、$T_u, T_{u+1}, T_{u+2}, \ldots$ の順に T_u から始めて左側からトークンが再利用できるか否かを検査していく。再利用不可であるトークンか、i 番目の要素に対応しないトークンが検出されるまで、検査を行ない、検出したトークンを T_v とすると、その直前までのトークン T_j $(u \leq j < v)$ を再利用する。

再解析

編集前のファイルのトークン T_v が最初に再利用不可となったトークンのとき、再利用または再解析により編集後のファイルのトークン列として既に得られている一番右（後）のトークン $T'_{u'-1}$ の直後から字句解析を行なっていく。

字句解析は、字句解析で得られたトークンの直後のトークン $T'_{v'}$ の（直前の空白文字列も含めて）先頭の文字が KMN リストの i 番目の要素に対応する部分に含まれず、かつ、編集前のファイルのトークンのどれかの先頭の文字と対応している場合に終了して再利用の解析に移ってよい。

字句解析で得られた最後のトークン $T'_{v'-1}$ の次のトークン $T'_{v'}$ は以下の 3 つの条件を満たす。

(1)

KMN リストの i 番目の要素より右側の要素に対応、すなわち、

$$\sum_{p=0}^{v'-1}(|T'_p|) + 1 \geq X'_{i+1}(1)$$

を満たす。このとき、トークン $_l$ は少なくとも K_i と M_i には対応しないことになり、少なくとも、 KMN リストの i 番目の要素に対応する文字列内では編集されてはいない。

(2)

先頭の文字が編集されていない、すなわち、ある $i_+ > i$ が存在して、

$$X'_{i_+}(1) \leq \sum_{p=0}^{v'-1}(|T'_p|) + 1 \leq X'_{i_+}(K_{i_+})$$

を満たす。このとき、得られたトークンは少なくとも先頭はそのまま残っているので、編集前のテキストに対応する文字が存在する。したがって、再利用の可不可が検査可能である。

(3)

編集前のトークンのどれかと先頭が一致、すなわち、ある u_+ が存在して、

$$\sum_{p=0}^{v'-1}(|T'_p|) + 1 = \sum_{p=0}^{u_+-1}(|T_p|) + 1$$

を満たす。これにより、トークン T_{u_+} は左方のトークンの変更の影響で変更されてはいないことが保証される。

トークン $T'_{j'}$ $(u' \leq j' < v')$ が再解析で得られたトークンである。

4.1.3 アルゴリズム

以上に基づいて構成したアルゴリズムは以下のようになる。

```
procedure getDiffFromKMN( oldT[], newT[], KMN[],)
begin
    old_x = 1;
    new_x = 1;
    oldtp = 0;
    newtp = 0;
    i := 0;
    repeat
        /* reuse old tokens */
        while oldT[oldtp] within KMN[i].k
            newT[newtp] := oldT[oldtp] ;
            old_x := old_x + tokenLength( oldT[oldtp] ) ;
            oldtp := oldtp + 1 ;
            new_x := new_x + tokenLength( newT[newtp] ) ;
            newtp := newtp + 1 ;
        end while
        /* get new tokens */
        repeat
            /* lexical analyze new tokens */
            while new_x within KMN[i].k or KMN[i].m
                newT[newtp] := getToken( new_x ) ;
                new_x := new_x + tokenLength( newT[newtp] ) ;
                newtp := newtp + 1 ;
            end while
            /* shift KMN to next */
            if KMN[i].m and KMN[i].n = 0 then return ;
            i := i + 1 ;
            /* dispose old tokens */
            while  KMLocal( old_x ) < KNLocal( new_x )
                old_x := old_x + tokenLength( oldT[oldtp] ) ;
                disposeToken( oldT[oldtp] ) ;
                oldtp := oldtp + 1 ;
```

```
            end while
            /* lexical analyze new tokens */
            while new_x within KMN[i].k
                   and  KMLocal( old_x ) != KNLocal( new_x ) )
                newT[newtp] = getToken( new_x ) ;
                new_x := new_x + tokenLength( newT[newtp] ) ;
                /* dispose old tokens */
                while  KMLocal( old_x ) < KNLocal( new_x )
                    old_x := old_x + tokenLength( oldT[oldtp] ) ;
                    disposeToken( oldT[oldtp] ) ;
                    oldtp := oldtp + 1 ;
                end while
            end while
        until oldx within KMN[i].k
             and KMLocal( oldx ) = KNLocal( oldy )
    until new_x > max_new_x
end
```

ただし、

- $oldT$、 $newT$ は、それぞれ、編集前と編集後のソースファイルのトークン列を表す配列である。
- $tokenLength(T[i])$ は $|T_i|$ を表す。
- $KMLocal(old\_x)$ 、 $KNLocal(new\_x)$ は、それぞれ、編集前と編集後のソースファイルにおける、 KMN 要素内の文字列の何文字目にあたるかを表す。
- $getToken(x)$ は編集後のソースファイルの x 文字目から字句解析を行ない、１個分のトークン情報を返す。

解析時には、編集されたか否かというようなトークンの差分情報を保持していく。編集後のトークン列における再利用の始まりの（編集されていない）トークンのインデックス、及びに再利用できる（編集または変更されていない）トークンの個数を１つの要素として、リスト形式で保持する。

再利用の始まりのトークンのインデックスを $start$、再利用できるトークンの個数を $recycle$ とすれば、図 4.4のようになる。

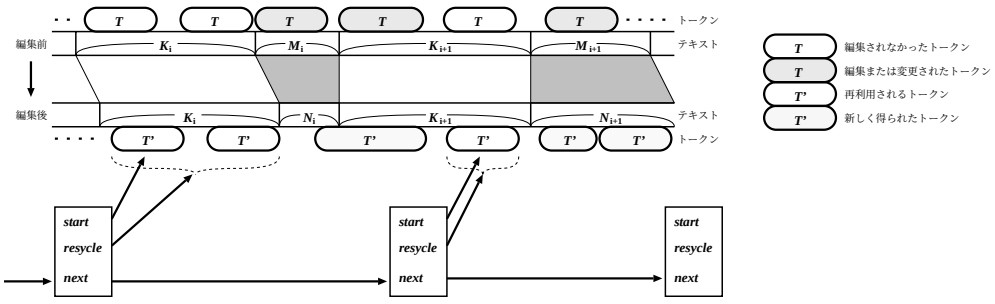


図 4.4: トークンの変更情報保持の概念図

4.2 編集されているが再利用可能なトークンの抽出

4.1で挙げた方法は、ユーザの編集が冗長でない場合、すなわち、プログラムの意味を変えるための必要最小限の編集であった場合には、得られる差分情報も最小になり、最高の効果を発揮する。

ところが、大概ユーザの編集は冗長であり、エディタによっては差分が冗長になり易いものも存在する。例えば、インデントの変更や編集時にある文字列を一度削除してからまた同じ文字列を挿入した場合なども、トークンは変わってしまったものとして扱っている。その場合には得られるトークンの差分情報も冗長なものになってしまう。

ここでは、冗長に編集が行なわれた場合も考慮に入れ、KMN リストの情報に加えてソースファイルの編集された文字列を検査して、トークンの再利用性を高める方法を挙げる。

4.2.1 実現方法

まずは 4.1で挙げた方法を用いてトークンの編集情報を得るが、その際変更されたとして捨ててしまっていた、編集前のソースファイルの部分トークン列も変更情報とともに保持しておく。（図 4.5）

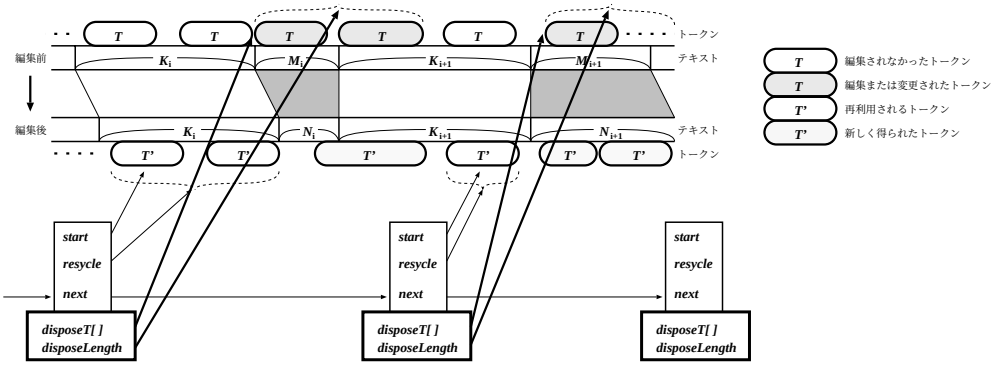


図 4.5: トークンの変更情報及び破棄する予定のトークン列の保持の概念図

その上で、再利用不可となった部分トークン列とそれに対応する新しく得られた部分トークン列の差分をとる。ここでは、連続した編集の範囲が大きい場合も考慮して、差分抽出のアルゴリズムとして、ヒューリスティックに適当な解で満足するもの（3.1.2参照）を採用する。

この差分抽出アルゴリズムにより、同等なトークンの対応リストが以下の形で得られる。

$$(y_0, y'_0), (y_1, y'_1), (y_2, y'_2), \ldots$$

これは y_i 番目のトークンが $y'_{i'}$ 番目のトークンと同等であり、再利用可であることを示している。この情報を編集の範囲を表す情報に変換する方法は 3.2.2で示したのと同じ方法である。

以上で得られた差分情報を差分情報のリストに挿入する。

4.3 実装と評価

KMN リストを残すエディタを、 GNU により提供されている emacs version 19.34 [1] 上に emacs lisp で KMN 法を実装することにより実現した。 KMN リストはエディタの終了時にソースファイルとは別のファイルに保存されるようにした。

字句解析器は構文解析器や属性評価器とメモリを共有するようにし、先に新しいソースファイルと KMN リストのファイルを読み込んで解析後、メモリ上に保持しておき、パーザからの要求でトークン情報とトークンの差分情報を渡すようにした。

解析時間は 3章と同様に pixie コマンドを用いて得た。

対象のファイルは Pascal-S で書かれた 500 行のファイルを用意して、それから、それぞれ 2%、10%、 20%、 30% 程度の冗長でない編集が加えられたファイルと、編集はされているが、実際の変更は 2% 程度であるファイルを作成した。その際、エディタでの編集情報を残した。

グラフの横軸は編集の程度を表しており、 (1) はその編集で実際にトークンが変更されている場合、 (2) はその編集で変更されたトークンが全体の 2% 程度の場合、である。すなわち、ユーザの編集が、 (1) は冗長でない場合、 (2) は冗長な場合、である。また、全てのグラフについて、（インクリメンタルでない）コンパイラの実行時間を折れ線グラフで掲載し、比較した。

図 4.6は、 3章の、先にトークンに変換してから、ヒューリスティックに適当に小さい差分で満足するアルゴリズムを用いてトークンの差分をとる差分抽出器で構成したコンパイラの実行時間を示している。特に実際の変更の程度が大きい場合に長くなる。したがって、インクリメンタルなコンパイラ全体として、通常のコンパイラよりも遅くなってしまっている。実際の変更の程度が小さい場合でもそれほど速くない。

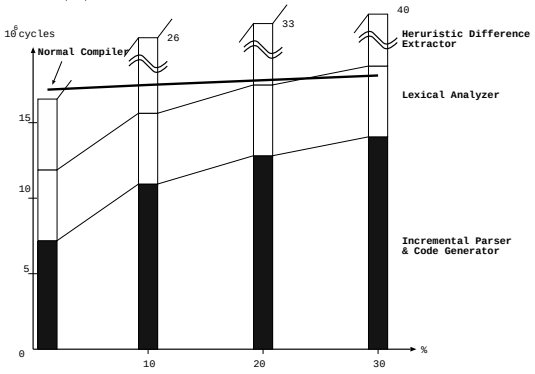
図 4.7は、単純に KMN リストを用いた差分抽出器で構成したコンパイラの実行時間を示している。変更が大き過ぎない限り、十分高速であるといえる。編集の程度は大きいが実際の変更の程度は小さい場合（(2) の 30 コンパイラ全体として、ソースファイルのみから差分抽出を行なうものより、速くなっている。

図 4.8は、 KMN リストを用い、冗長な差分情報を補正する差分抽出器で構成したコンパイラの実行時間を示している。ユーザの編集が冗長な場合（(2) の場合）ではかなり実行時間が短く済んでいる。一方、ユーザの編集がそれほど冗長でないときは、単純に KMN リストから差分を抽出するものに較べて、差分補正のオーバーヘッドのため遅くなってしまっている。

以上をまとめると、ユーザの編集が冗長な場合は単純に KMN リストを用いた差分抽出器を、ユーザの編集が冗長でない場合は KMN リストを補正する差分抽出器を、インクリメンタルなコンパイラに組み込んだときに、全体としてのコンパイルの実行時間が短くなる。

[1] 広く使われている多機能スクリーンエディタ

(1) ユーザの編集が冗長でない場合



(2) 編集が冗長な場合（実際は 2% 程度）

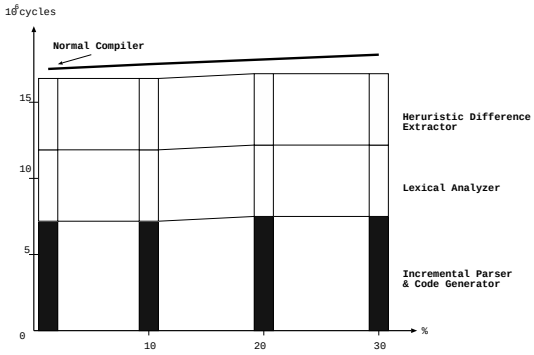
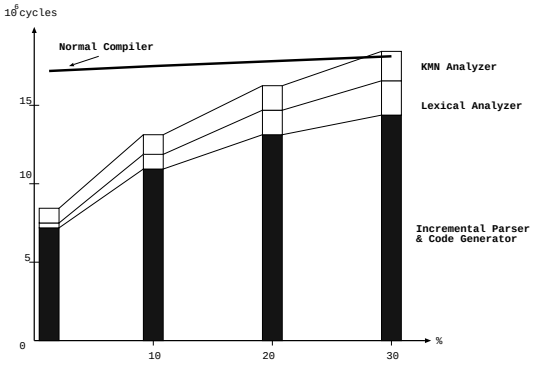


図 4.6: ソースファイルのみからの差分抽出器で構成したコンパイラの実行時間

(1) ユーザの編集が冗長でない場合



(2) 編集が冗長な場合（実際は 2% 程度）

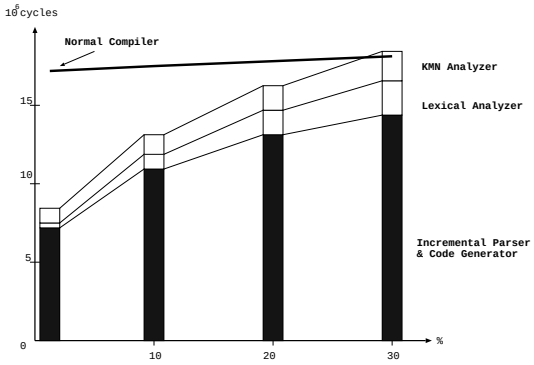
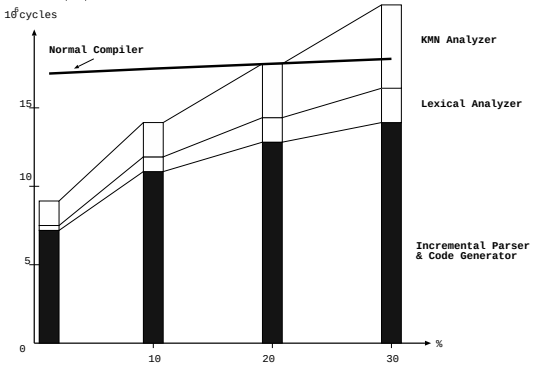


図 4.7: KMN リストからの差分抽出器で構成したコンパイラの実行時間

(1) ユーザの編集が冗長でない場合



(2) 編集が冗長な場合（実際は 2% 程度）

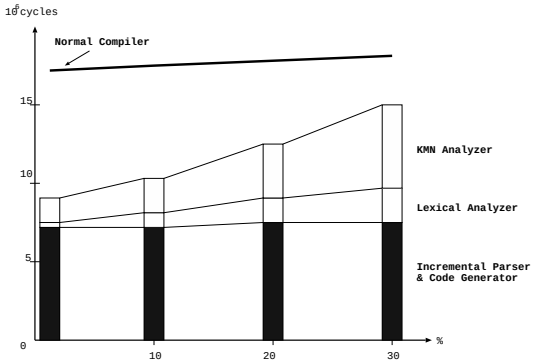


図 4.8: KMN リストを補正する差分抽出器で構成したコンパイラの実行時間

第 5 章

Restorable-KMN 法

ユーザの編集は冗長な場合が多く、 KMN 法で得られる差分もまた冗長になりやすい。編集された部分に元のソースファイルから変わっていない部分がある場合、その部分を編集されていないものとして扱えれば再利用性は高まり、インクリメンタルなコンパイルの実行時間を短くできる。

しかし、コンパイル時に冗長な差分を抽出するのは時間的なコストが高く、逆に遅くなってしまう場合も少なくない。

そこで、本章では、そのような冗長な差分をエディタでの編集時に補正する方法をを提案する。この方法を Restorable KMN 法と呼ぶ。

アルゴリズムとして最も簡単なのは編集の操作があるたびに編集前のソースファイルと編集中のソースファイルの最小差分をとってしまうことであるが、これは時間が掛かりすぎて現実的ではない。

したがって、差分補正を行う範囲を決める必要があり、本研究では、基本的には KMN 法に基づくことにして、編集情報の内、編集されていない部分を編集された部分としてしまうような補正は行わないことにした。

5.1 補正の方法

編集の操作があるたびに補正できるところを次々に編集されていない部分としてしまえば、簡単であるが、そうすることによって不都合が生じる。ここでは、それについて述べる。

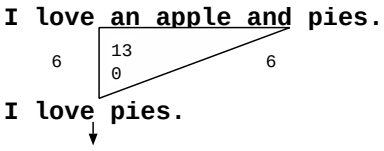
5.1.1 次々に補正してしまうことにより起きる問題

編集の操作の都度 KMN リストを補正してしまうことにより起きる不都合の例を挙げる。

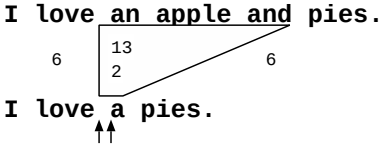
いま、以下の文字列が編集前の文字列であるとする。

```
I love an apple and pies.
```

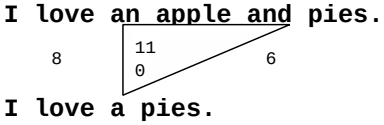
編集（削除）があって以下に変わったとする。このとき KMN リストは $(6,13,0),(6,0,0)$ になり、冗長な差分は含まれていない。



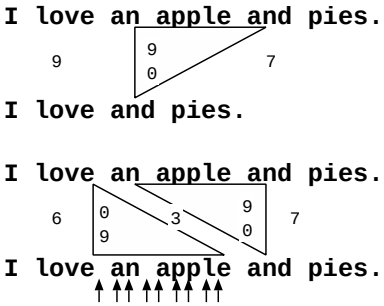
この後、編集（挿入）があって、文字列が以下に変わったとする。 KMN リストは $(6,13,2),(6,0,0)$ になり、冗長な差分が含まれている。



そこで、冗長な差分を補正すれば、以下になる。 KMN リストを $(8,11,0),(6,0,0)$ とでき、冗長な差分は含まれなくなった。



ところが、さらに、文字列が以下のように編集されていったとする。



このような編集がなされたときには補正ができなくなってしまい、結果としてかなり、冗長な差分になってしまった。

これを補正するためには、 KMN リストで編集がされていないとなっている部分をくずせばよいのだが、先に述べたように、補正する範囲を決めないときりがない。

補正の際に対応づけの順序を与えることにより、多少の改善はみられるであろうが、完全な順序というものはこの場合には存在しない。完全に近い順序を与えるためには、ユーザの行動を予測する必要があるため、実現は困難に思われる。面白い課題ではあるが、今回は挙げるにとどめる。

5.1.2 補正情報を別に保持する方法

KMN リストを補正してしまうことはせずに、補正情報を別個に残しておく方法が考えられる。

KMN リストの要素に変更がある都度差分補正の解析はするが、KMN リストの修正は行わないというものである。

KMN リストの性質から、通常の編集によって更新された KMN リストの要素のうち、差分補正を行うのは１つでよい。したがって、差分補正の範囲はそれほど広くならずに済む。ただし、編集が重なると、KMN 要素の M と N、特に M は大きくなっていく一方である。そこで、差分補正に用いる差分抽出アルゴリズムは 3章で挙げた、ヒューリスティックに適当な解で満足するアルゴリズムを用いるのが適当である。

5.1.3 バックグラウンドでの補正

編集している部分から離れた部分の差分補正は、バックグラウンドで行うことも可能である。ただし、差分補正中の部分に編集が及んでしまった場合の処理等を考える必要がある。今回、この方法については挙げるだけで、実装は行えなかった。

5.2 復活のための差分抽出法

本研究では、5.1.2で挙げた方法をとりあげることにした。

編集があった部分の差分をとる方法について、いくつかの方法が考えられる。

5.2.1 単純な文字列比較

最も単純な方法として、単に部分文字列の差分をとる方法が挙げられる。この方法で得られる差分情報は、文字列の差分としては冗長性がないとしても、トークン列の差分としての冗長性が生じてしまう。

この場合、差分抽出アルゴリズムのヒューリスティック関数に文字列の連続性などのパラメータを与えるなどの工夫をするしかない。

5.2.2 トークンの文字列のパターンマッチ

編集前のコンパイラで得られているトークン列の情報をエディタで保持するようにし、差分抽出時にその情報を用いることでトークン列の差分としての冗長性を抑えることができる。

トークン単位でトークンの文字列を編集中の部分文字列に対してパターンマッチを行ない、得られた対に対して最小差分を（または、それに近いものをヒューリスティックに）求める。

5.2.3 字句解析

5.2.2に加え、エディタに字句解析の機能を持たせることにより、少なくとも部分的には、冗長性のないトークン列の差分を抽出できる。

そこで、エディタでの差分補正時に字句解析を行なうことにより、トークンレベルでの差分抽出を行なう。

このとき得られたトークン情報を編集完了時に出力し、コンパイラで利用すれば、コンパイラの字句解析での計算時間の消費を抑えることができるかもしれないが、本研究ではそこまでは触れない。

5.3 実装と評価

Restorable KMN 法で得られた情報を解析して利用できるように、 4章の差分抽出器を拡張した。

テキストレベルの差分補正情報を残す Restrable KMN 法を、 4章で実装したエディタを拡張することにより実装した。また、トークンレベルの差分補正情報を残す Restrable KMN 法を、さらに字句解析の機能を持たせる拡張を行ない実装した。

解析時間は 3章と同様に pixie コマンドを用いて得た。

コンパイルの対象とするソースファイルは 4章と同様に与えた。

グラフの横軸は編集の程度を表しており、 (1) はその編集で実際にトークンが変更されている場合、 (2) はその編集で変更されたトークンが全体の 2% 程度の場合、である。すなわち、ユーザの編集が、 (1) は冗長でない場合、 (2) は冗長な場合、である。

また、全てのグラフについて、（インクリメンタルでない）コンパイラの実行時間を折れ線グラフで掲載し、比較した。

図 5.1は、テキストレベルの差分補正情報が KMN リストに加わったものから差分抽出をした場合の、インクリメンタルなコンパイラの実行時間を示している。

図 5.2は、トークンレベルの差分補正情報が KMN リストに加わったものから差分抽出をした場合の、インクリメンタルなコンパイラの実行時間を示している。

差分補正がコンパイル時に行なわれない分、ユーザの編集が冗長な場合にはかなりの実行時間の短縮がみられる。ユーザの編集が冗長でない場合のオーバヘッドもほとんどないので、単純に KMN リストから差分抽出をするものと同程度の実行時間で済んでいる。

差分補正情報がトークンの差分抽出にとって適切であるトークンレベルの差分補正情報のほうが、テキストレベルの差分補正情報よりも効果をあげているのは当然といえよう。

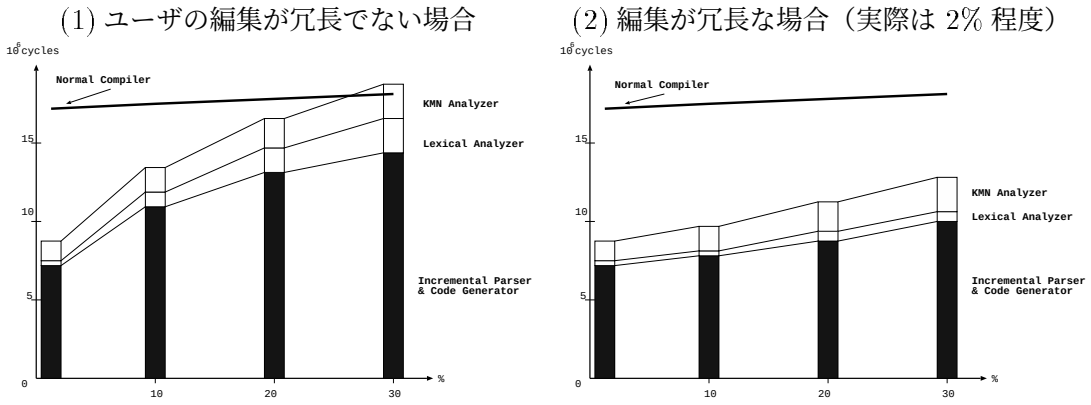


図 5.1: 文字列レベルの Restorable KMN 法からの情報を与えたコンパイラの実行時間

(1) ユーザの編集が冗長でない場合　　(2) 編集が冗長な場合（実際は 2% 程度）

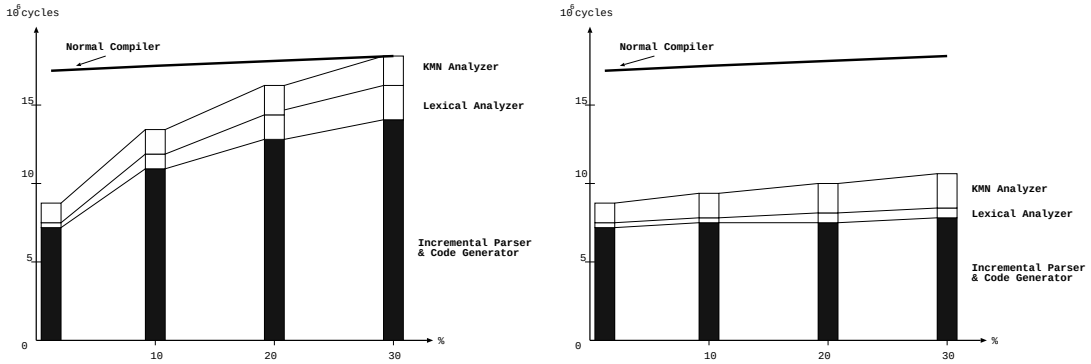


図 5.2: トークンレベルの Restorable KMN 法の情報を与えたコンパイラの実行時間

最後に、単純に KMN リストから差分を抽出する場合、差分抽出器で KMN リストを補正する場合、エディタで KMN リストにトークンレベルの補正が加えられた情報から差分を抽出する場合、のコンパイルの実行時間の 3 つを較べたグラフを図 5.3に示す。

グラフの横軸は編集の程度を表しており、ユーザの編集が、(1) は冗長でない場合、(2) は冗長な場合、である。

編集時に差分を補正することでかなりの効果がみられていることがわかる。

(1) ユーザの編集が冗長でない場合　　(2) 編集が冗長な場合（実際は 2% 程度）

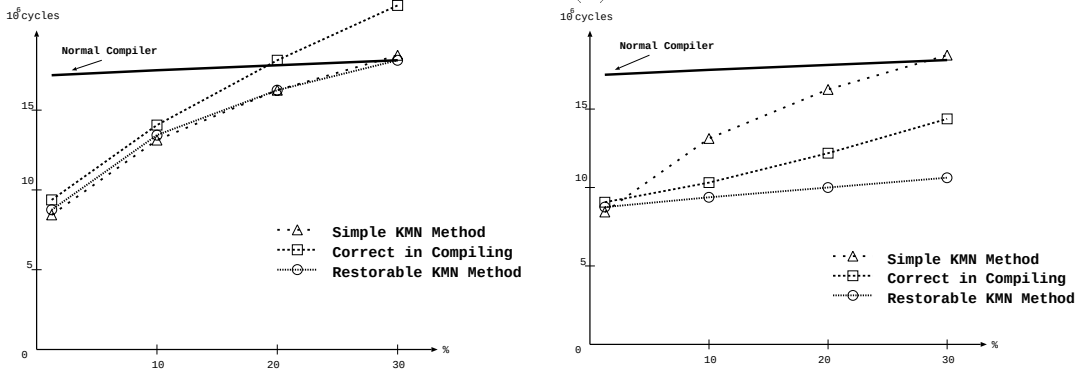


図 5.3: コンパイラの実行時間の比較

第 6 章

結論

本研究では、インクリメンタルなコンパイルを行なうために必要な字句解析器及び差分抽出器に着目した。

差分抽出のために、まずは単純に編集前後のソースファイルを比較して差分をとる方法について検証した。差分を最小にする問題は LCS(Longest Common Subsequence) 問題であり、優秀なアルゴリズムを用いても解くための計算時間がかなりかかる。インクリメンタルなコンパイル全体の計算時間において、ソースファイルやその差分が大きいときには差分抽出の時間がかかり過ぎてしまうため、最小差分を求めるアルゴリズムでは不都合が生じる場合がある。そこで、ヒューリスティックに差分計算を打ち切り適当な差分で満足するアルゴリズムを採用することにより、対処した。

差分をとる段階はテキストレベルとトークンレベルの 2 つが考えられるが、インクリメンタルに字句解析を行なえない不利を含めても、トークンレベルで差分をとった方が計算時間が少なくて済むことを実験から得た。さらには、トークンレベルで差分をとる方が得られる差分情報が有益になるので、計算時間と得られる情報量の両方でテキストレベルでの差分抽出に比べ有利である。

ソースファイルを比較する方法の計算時間は、インクリメンタルな構文解析、属性評価、コード生成全部含めたものと同等以上かかるため、全体のインクリメンタルなコンパイラシステムとしては十分な効果を得られない。そこで、エディタでソースファイルの編集情報を残し、それを用いる方法を採用した。この方法では、ユーザの編集が冗長でないならば、コンパイルの計算時間は十分な効果が得られる。

一方で、実際にはユーザの編集は冗長である場合が多く、その場合、エディタから得られる情報も冗長になってしまう。それを解決するために、エディタでの情報の解析時に部分的に差分抽出を行なう差分抽出字句解析器を提案し、実装した。解析時間の実験を行なったところ、場合によっては効果があるが、逆に遅くなってしまうこともある。

そこで、エディタでの編集情報を残すと同時に冗長な差分を減らすようなエディタを提案した。これは、編集があった部分について、差分の補正を編集時に行なうものである。先に得られた研究結果を利用して様々な実現方法を提案し、一部を実装した。

コンパイル時の差分補正を省くことができるため、差分抽出器の精度、実行速度の両方で効果がみられるようになって、コンパイラ全体の実行時間が短縮できた。

今後の課題としては以下のものが挙げられる。

1. 字句解析の機能を持ち、得られる差分情報をより有益にするエディタの実現方法の提案と実装。
2. エディタ上でのユーザの編集に支障を出さずにバックグラウンドで計算コストの高い解析を進められるようなシステムの構築。
3. エディタ、コンパイラ、デバッガ、各種ツールなどを含めた統合的な環境でインクリメンタルなコンパイルを実現するためのデータ形式の考案。

謝辞

本研究を行うにあたり、筑波大学電子・情報工学系中田育男教授、同山下義行助教授の両先生方には多くの御指導、助言を頂き、心より深く感謝致します。

また本論文を仕上げるにあたって、プログラミング言語研究室の先輩方の多くの御力添えを頂きました。特にインクリメンタルなコンパイラシステムの構成について先導頂いた中井央氏に感謝致します。加えて、OS・知識ベースシステム研究室、SCORE 研究室、IP Lab. の皆様、プログラミング言語研究室の同級生の諸氏にも感謝致します。

参考文献

[1] John F.Beetem and Anne F.Beetem: Incremental Scanning and Parsing With Galaxy, IEEE Transactions on Software Engineering, vol.17, No. 7, July 1991.

[2] Michael L. Van De Vanter, Susan L. Graham and Robert A. Ballance: Coherent user interfaces for language-based editing systems, Structure-based editors and environments, p19-69 (Academic Press Ltd 1996).

[3] Sten Minör and Boris Magnusson: Using Mjølner Orm as a structure-based meta environment, Structure-based editors and environments, p71-106 (Academic Press Ltd 1996).

[4] 中井央, 山下義行, 中田育男: インクリメンタルな LR 構文解析の一方式の提案とその評価, 情報処理学会論文誌 Vol.37 No.3 p371-383, Mar 1996.

[5] 中井央, 佐々政孝, 山下義行, 中田育男: LR 属性文法に基づいたインクリメンタルな属性評価, 情報処理学会論文誌 Vol.37 No.12 p2254-2265, Dec 1996.

[6] Hirschberg, D.S.: Algorithms for the Longest Common Subsequence Problem, J. ACM, Vol.24, No.4, pp.664-675 (1977).

[7] 角田博保: ファイル間の相違検査法, 情報処理 Vol.24 No.4 p514-520, Apr 1983.

[8] Hunt, J.W.: Algorithms for Computing Longest Common Subsequences, Comm. ACM, Vol.20, No.5, pp.350-353 (1977).

[9] Wagner, R.A. and Fischer, M. J.: The String-to-String Correction Problem, J. ACM, Vol.21, No.1, pp.168-173 (1974).

[10] Eugene Myers: An O(ND) Difference Algorithm and its Variations, Algorithmica Vol.1 No.2, pp.251-266 (1986).

[11] E. Ukkonen: Algorithms for Approximate String Matching, Information and Control Vol.64, pp.100-118 (1985).