ソフトウェア工学 61－3
（１９８８．７．２７）

プログラム読解支援に対する自然言語理解的アプローチ

石井威望　　広瀬通孝　　小池英樹
東京大学工学部

現在、プログラム部品のデータベース化が盛んに行なわれているが、問題となるのは登録された部品の検索・読解・修正のプロセスをいかに支援するかである。本報告ではこのうち読解プロセスについて知識工学的支援という立場から、自然言語理解研究で用いられるプラン・ゴール理論をプログラムのソースコードに適用し、プログラマの意図(プラン)を抽出するシステムを試作したのでこれについて報告する。

具体的なインプリメントは、プラン、及びゴールをテンプレートを基本としたフレーム形式のスキーマとして表現し、各スロットに他のプラン・ゴールへの参照ポインタを持たせることにより、有機的な知識構造を構成できるようにした。システムはプログラムを１文ずつ読み込み、プラン・データベースに登録されたプランとマッチするものがあれば解析木に登録し、より上位のプランが後に判明した時は、以前の決定を変更することを繰り返し、最終的に解析木を出力する。

最後にプラン、及び試作したシステムに対して考察を加え、ゴールとプランと段階的詳細化設計法との関係を述べる。

Software Reuse Using Plan-Goal Theory

Takemochi ISHII　　Michitaka HIROSE　　Hideki KOIKE
Faculty of Engineering, University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo, Japan

Recently, many program parts are being registered in database. So it is important how to assist the programmer when he retrieves, understands and modifies them. This paper describes the system that abstracts programmer's intention (plan) from source code by using Plan-Goal theory developed in study of natural language understanding.

We represent plans and goals as frame-typed schemata which have program template in their slots. they also have some reference pointers to other plans and goals in their slots to construct organic knowledge structures. The system reads one statement at a time and examine if it matches against any plans in plan database. If it matches, the system registeres that plan to interpretation tree or modifies this tree when the system find more abstract plan. The system repeats this process until there is no more statement.

Finally we discuss about the plan and our system, and about the relation between Plan-Goal and stepwise refinement.

1 緒言

計算機ハードウェア技術の急速な進歩に伴う生産コストの低下により、現在、ソフトウェアの開発・保守費用が深刻な問題となっている。このため既存のソフトウェアの再利用の重要性に対する認識が広まり、多くの研究がなされているが必ずしも成功しているとは言い難い。

これまでのソフトウェア再利用研究はプログラムの部品化の研究が主体であった。プログラミング言語学のアプローチは、データ独立性・手続きの局所性・情報隠蔽度の高い部品をいかに作成するかであり、オブジェクト指向言語 ([12])、パッケージ ([11]) の研究などがこの範ちゅうに入る。一方、ソフトウェア工学としては、これらプログラム部品・仕様・ドキュメント等のデータベース化に解決策を求めている。しかるに、データベースに登録された部品は、プログラマによって検索され、読解された後に要求に合うように修正されねばならないが、この検索・読解・修正に対する支援が不十分であるために部品は有効に活用されていないのが実状であろう。

現在、自動プログラミングの研究 ([10]) 等、知識に基づきユーザを支援するシステムが開発され、ある程度の成果を上げている。ソフトウェアの再利用問題に対しても、単純なデータベース登録やドキュメントの整備といったいわば消極的な方法でなく、知識工学を用いた積極的な支援が必要だと思われる。

上述した検索・読解・修正のプロセスの内、検索の問題に対して演者らは、既にソフトウェア・カルチャーの概念に基づく部品の検索を提案している ([2])。本報告ではプログラマの読解の支援を目的とした上で、まずプログラミングに用いられる知識のうちプログラムの作成・読解に用いられていると思われるテンプレート型の知識構造に注目し、これと自然言語理解におけるプラン・ゴール理論を結び付けることによってプログラマの意図を推論しようとする試み、及びこの考えに基づき実際に試作したプラン解析システムについて説明し、最後に考察を行なう。

2 プログラミング・プラン

2.1 プログラミングにおける知識

プログラミングは人間の高度な知的作業である。その作業過程はソフトウェアの規模や、実験的か実用的かといったシステムの性質によって異なるが、要求仕様の理解・デザイン・コーディング・保守等の各過程において多くの知識が必要とされる。こうして作られたプログラムは箇条書のように各文書が無機的に並んでいるのではなく、ある処理毎にいくつかの文がグループをなして有機的に結合されている。こうして一まとまりになった知識構造のことをここではプログラミング・プランと呼ぶことにする。

「プログラムを書く」という作業はプログラマが与えられた問題 (要求仕様) を問題解決計画 (プラン) に関する知識とこれらを組み合わせるためのメタ知識を核とした、プログラミング言語に関する知識によるプランの実現と考えられる。逆に「プログラムを読む」という作業は、ソースコード内に存在するプログラマのプランを各人が持っているプランに関する知識に基づいて理解し、そこに存在するプログラム作成者の意図を理解する過程ともいえる (図1)。言い替えれば、プログラム中のプランを理解することがプログラムを理解することにつながると言えるであろう ([8], [4])。

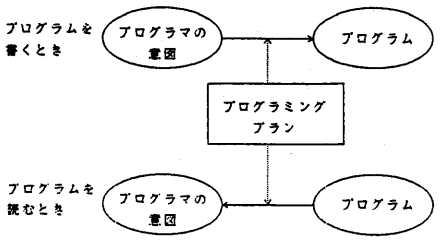

図 1: 作成・読解に用いられるプラン

2.2 プログラミング・プランの特徴

プログラミング・プランには次に述べるいくつかの特徴がある。

1. プランの非局在性
   プログラミング・プランは一箇所に局在するのではなく、図2に示すように散在している。そして、他のプランと互いにオーバーラップし合い、また他のプランをプランの一部として含むこともある。

2. 再利用可能な知識としてのプラン
   いくつかのプログラムを書く際に、共通して用いられる知識構造が存在し、プログラマはこの知識構造の探索・再利用によってプログラムを作成・読解するものと考えられる。

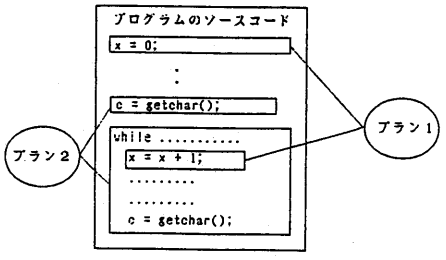

図 2: プログラム内に散在するプラン

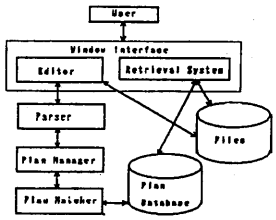

図 3: システム概要

2.3 他の人工知能研究におけるプランとの比較

STRIPS に代表されるロボットのプランニングは、初期状態から始めて目標状態に到達するまでの最適なプラン列を求めるものである。一方、自然言語理解におけるプランニングは、断片的に観測される事実から途中過程を推論・補完することによって文脈を理解するものである。プログラムからプログラマの意図を推論するプロセスは後者に近く、R. C. Schank らが用いたプラン・ゴール理論([7]) による推論が有効だと思われる。ただし、プログラムでは文脈理解における EVENT のような動作単位が明らかでない。よって、本システムではテンプレートをベースとしたフレーム型のデータ構造を採用しプランの解析を行なった。

3 プラン解析システムのインプリメンテーション

3.1 システム概要

以上の考察に基づき、プラン解析システムを試作した。システムの概要を図 3に示す。 本システムにおいてはパターン・マッチング等の文字列処理が頻繁に行なわれるためシステムの実現言語には Lisp を採用した。解析するターゲット言語としては Parser の負担を軽くするため、C のサブセットで入出力関数として read() と write() しか持たない仮想的な構造化言語 (便宜上 C' と呼ぶ) を選定した。 C' は Parser によってリスト形式に変換され、 Plan Matcher によって Plan Base 中のプランとのマッチングが行なわれる。これらの結果はトップ・レベルの Plan Manager によって登録・変更される。

3.2 プランとゴールの表現

プラン、およびゴールは図4に示すようにフレーム形式のデータ構造で表す。 ある目的に対し実現法が複数

```
(Plan while-process-read-plan
  (instance-of (read-and-process))
  (constances (?const))
  (variables (?data))
  (template
    (init (subgoal (input ?data)))
    (main (while (!= ?data ?const)
                 ?*
                 (subgoal (input ?data))))))
(Goal read-and-process
  (form (read-and-process ?new ?stop))
  (instance (while-process-read-plan)))
```

図 4: プランとゴールのフレーム表現

存在する場合、それらをまとめて一つの概念として表したのがゴールであり、具体的な実現手段がプランである。端的に言えばオブジェクト指向におけるクラスとインスタンスの関係に相当する。ゴールの instance スロットとプランの instance-of スロット、および template スロットによってゴールとプランは互いに参照し合っている (図5)。以下、プランの各スロットについて解説する。

1. template スロット
   template スロットは解析の際にプログラムとのマッチングに使用される部分で、ここに書かれるステートメントには、以下のような特殊な表現法が用いられる。

   (a) コンポーネント・ラベル
   init、 main 等、スロットの各要素の car 部はそのプランにおける各ステートメントの役割を表し、特に main で始まるステートメント

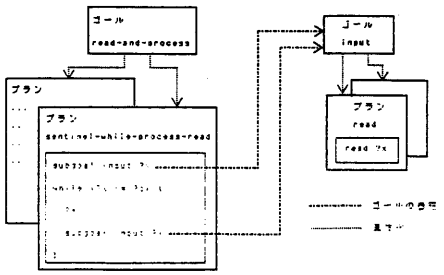

図 5: ゴールとプランの参照関係

はプログラムとのパターン・マッチングの際、最初に調べられる。

(b) オペレータ
+、−、! =等のオペレータは、C' と異なり前置記法で記述される。つまり、

```
(?data != ?const) --> (!= ?data ?const)
          (3 + 4) --> (+ 3 4)
```

というように記述される。

(c) subgoal ステートメント
subgoal で始まるステートメントは他のゴールへの参照を意味し、このゴールを満たすようなプランならどれとでもマッチする。ただし引数のパターン変数に対する変数束縛は参照されたゴールにも影響を与える。例えば、図4のプランの main パートは、

```
(while (!= x EOF)
       (= s (+ s 1))
       (read x))
```

とはマッチするが、

```
(while (!= x EOF)
       (= s (+ s 1))
       (read y))
```

とはマッチしない。その理由は、(!= ?data ?const) が (!= x EOF) とまずマッチしてパターン変数?data は x に束縛され、?const は EOF に束縛される。次に後述するワイルド・カード・パターンである?* が (= s (+ s 1)) とマッチし、システムは次に subgoal ステートメントのマッチングを試みる。いま、ゴール input のインスタンスに (read ?var) なるテンプレートを持ったプランがあるとすると、?var は?data の変数束縛を継承し、x に束縛される。このため、ゴール (input ?x) は前者とはマッチするが、後者とはマッチしない。

(d) ワイルド・カード・パターン
1 文とだけマッチする?+ と、1 文以上の複数の文とマッチする?* の 2 つがある。

(e) パターン変数
? で始まる単語はパターン変数で、プログラムの識別子等とマッチする。パターン変数には定数としかマッチしないものと、変数とマッチするものの 2 種類ある。前者はプラン・フレームの constants スロットに宣言されており、後者は variables スロットに宣言されている。最初これらのパターン変数は如何なる値にも束縛されていないが、一度ある値とマッチするとその値に束縛され、以後はこの値としかマッチしない。

2. pregoals と postgoals
それぞれプランが行なわれるために満たしていなければならない前提条件と後続条件を表す (例: 図6)。

```
(Plan average-plan
  (instance-of (average))
  (variables (?avg ?total ?count))
  (pregoals (count ?count)
            (sum ?total ?new))
  (template
    (main (= ?avg (/ ?total ?count)))))
```

図 6: 平均を求めるプラン

3.3 プラン・マッチング

各プランとプログラムのマッチングは Plan Matcher が行なう。Plan Matcher は Plan Manager によって作られた解析木を深さ優先で調べ、各ノードとプランとのマッチングをテストする。その順番は次のとおり。

1. template スロットの main コンポーネント
2. template スロットの main コンポーネントより前の文
3. template スロットの main コンポーネントより後の文
4. pregoals スロット
5. postgoals スロット

各段階において束縛されたパターン変数は、その変数束縛をそれ以降の段階にまで影響をもたらす。

templateスロット内のC'のステートメントとのマッチングは、以上のように行われるが、その際subgoalステートメントとプログラムとのマッチング、あるいはpregoalsスロットやpostgoalsスロットに書かれたゴールとのマッチングは、次のようにして行われる。もし、テストすべきプラン側の文がゴールへの参照文の場合、プラン・マッチャーは対応するゴールを捜し出し、instanceスロットに書かれたプランに対して順番に、プログラムの文とマッチするかどうかを、上述の過程と同様にしてテストしていく。

こうしてPlan Matcherはプログラムの一文に対し、登録されたプランをすべてテストする。そして、複数のプランがマッチした場合には、ある評価関数に基づいて最も適当なプランを決定する。評価関数としては、ここではプラン・フレームの中の要素の個数を評価関数として用いている。つまり、候補となるプランの内で最も複雑なものが最適なプランであるとしている。プラン・ゴールを保持する知識ベースが大規模化し、一つの文に対して多くのプランがマッチするようになると、よりプランの詳細を考慮するような評価関数が必要だが、今回のような小規模な実験システムの場合は、この評価関数で十分であると思われる。

以上のようにして、Plan Matcherは最適なプランを探し、結果をPlan Managerに返す。

3.4 Parser

ParserはC'の関数を1単位として受け取り、リスト形式のC'プログラムを出力する。基本的に1センテンスが1リストに対応し、while、if文等のブロックはリストのレベルが1つ深くなる。ただし、代入演算子等の演算子は、前置記法に変換される。

3.5 解析例

図7のプログラムの解析例をAppendixに示す。意識的に変数名などには意味のない名前をつけている。これはプラン解析システムの主要な動機の一つである。つまり本論文で例示するような、人間が見て即座にいかなるプログラムか判断できるものは別として、わけの判らない省略形を用いた識別子の羅列は、プログラマの読解を妨げる。しかし識別子名に関係なく機械的にテンプレート・マッチングを使って必要な知識だけを抽出することができれば、それはプログラマの読解を大幅に助けるものとなるであろう。

3.6 制約条件の役割

図8に示したプログラムの場合には

```
foo()
{
   int   s, x, i;
   float xx;
   s = 0;
   x = 0;
   i = 0;
   read(x);
   while (x != 99999) {
     s = s + x;
     i = i + 1;
     read(x);
   }
   if (i != 0) {
     xx = s / i;
     write(xx);
   }
}
```

図7: 平均を求めるプログラム

```
fee()
{
   int   s, x, i;
   float xx, yy;
   s = 0;
   x = 0;
   i = 0;
   read(s);
   read(i);
   if (i != 0) {
     xx = s / i;
     yy = s * i;
     write(xx);
   }
}
```

図8: 単純な計算のプログラム

```
xx = s / i;
```

という上で解析したプログラムと全く同じ文があるが、前提条件が満たされていないために平均を求めるプランとはマッチしない。

4 考察

4.1 プランの柔軟性

本システムではテンプレートにマッチするプランさえ存在すれば、パターン・マッチングによってそのプランを検出できるが、いちいちすべてのソースコードのパターンにマッチするプランを記述するのでは、プラン・データベースには膨大な数のプランが登録されることになり、マッチングにかかる時間、及びプランを登録する手間を考えると、あまりプランの利点がなくなると考えられる。この問題をある程度解決するのがゴールである。

```
foo2()
{
   int   s, x, i;
   float xx;

   s = 0;
   x = 0;
   i = 0;
   do {
     read(x);
     if (x != 99999) {
       s = s + x;
       i = i + 1;
     }
   } while (x == 99999);
   if (i != 0) {
     xx = s / i;
     write(xx);
   }
}
```

図 9: インプリメントの異なる平均を求めるプログラム

既出の average-plan にしても、前提条件としてはゴール count と sum が、満たされるべきだと言明しているだけで、その具体的な実現方法に関しては述べていない。またゴール input にしても、入力を行う処理の存在を言明しているだけで、ここにはいろいろな実現方法があると思われる。ここにプラン記述の柔軟性が生まれる。

図9は、やはり平均を求めるプログラムであるが、ループの部分の実現方法が異なる。例えば新たにこのループの記述をデータベースに登録する場合、変更点はゴール read-and-process の instance スロットにこのプラン (例えば do-while-process-read-plan とする) を加え、実際にこのプランを記述するだけで、他のゴール及びプランには影響を与えない。このプランの柔軟性のために、少数のプランで目的は同じだが実現方法の異なるプログラムを解析することが可能となる。

4.2 段階的詳細化との関係

段階的詳細化は現在最も広く使われている設計法の一つである。この段階的詳細化を具体化し、抽象度の高い要求仕様とプログラムの中間レベルを埋めようとしたものに PDL(Program Development Language)([3]) があり、また同様の詳細化とプログラミングとを同時に行ってしまうシステムとして D. E. Knuth の WEB([5]) がある。例えば PDL の場合、

x と y を交換する

という仕様は

```
t := x
x := y
y := t
```

の列に詳細化される。PDL の場合この詳細化はすべて紙上で行われるが、WEB の場合は計算機上で作成したこのような文書から、自動的にプログラムが作成される。しかしどちらの場合もこれらの詳細過程の再利用は考えていないので、その度毎に詳細化を行うことになる。

この4.2という抽象的な要求と、具体化された4.2との関係は、ゴールとプランの関係に類似している。つまり本稿で述べたようなプログラミング・プランのデータベース化は、段階的詳細化過程の再利用へと発展する可能性を持っていると考えられる。現在、この問題についてハイパーテキストとの関連も含めて研究中である。

4.3 本システムの問題点

現在のシステムが複雑なプログラムを解析できない原因の 1 部は Parser に起因するものであるが、主たる原因はプログラミング言語自体の柔軟性によるものである。ループ 1 つにしても多くの書式が存在し、それぞれに対応するプランを記述しなくてはならない。前述のプランの柔軟性によって汎用のプログラムに対応できるかは不明だが、限定された領域におけるプログラミングには対応可能だと思われる。

本システムではプログラマの意図を解析するという目的に対し、文脈に相当する部分に注目し、単語自体が持つ意味を無視した。しかし、現実にはプログラマは制限された中でも識別子等に意味のある名前を付けようとしている。実際、プログラムの形はほとんど同一でも、異なる意味を持つプログラムが存在する。本当に意図を理解しようとするならば、各単語の持つ意味をも考慮にいれる必要があろう。

5 結論

本稿では、プログラミング・プランと呼ばれるテンプレート型の知識構造がプログラムの作成・読解に関係しており、このプランの理解がプログラムの理解につながることを述べ、プログラム中のプランを抽出するシステムを試作した。さらにプランと段階的詳細化との関係、及びシステムの問題点について考察を行なった。今後は更にプログラミング・プランについての研究を進め、実際のシステムへの応用を考えていきたい。

参考文献

[1] 石井, 広瀬, 小池. プラン・ゴール理論を用いたプログラム再利用支援システム, 第 36 回全国大会論

文集, pp.965-966, 1988.

[2] 石井, 広瀬, 小池. ソフトウェア・カルチャーとプログラム再利用, 第37回全国大会 (投稿中), 1988.

[3] P. Grogono, S. H. Nelson. *Problem Solving and Computer Programming.* Addison-Wesley, 1982.

[4] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*, Morgan Kaufmann, 1986.

[5] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2), 1984.

[6] M. Minsky. A Framework for Representing Knowledge. In P.H.Winston, editor, *The Psychology of Computer Vision*, Mcgraw-Hill, 1975.

[7] R. C. Schank, C. K. Riesbeck. *Inside Computer Understanding.* Lawrence Erlbaum Associates, 1981.

[8] E. Soloway. Learning To Program = Learning To Construct Mechanisms And Explanations. *Comm. of ACM*, 29(9), 1986.

[9] 特集：自動プログラミング. 情報処理, 28(10), 1987.

[10] D. E. Barstow. Domain-Specific Automatic Programming. *IEEE Trans. on Software Eng.*, 11(11), 1985.

[11] G. Booch. *Software Engineering with Ada*, The Benjamin/Cummings Publishing, 1983.

[12] A. Goldberg, D. Robson. *Smalltalk-80: The Language and its Implementation.* Xerox, 1983.

Appendix

解析例

```
> (process-plan "foo.c")
Process start !!!
    <<< Input sentence is >>>
(= S 0)
Can not find suitable plan
   <<< Input sentence is >>>
(= X 0)
Can not find suitable plan
   <<< Input sentence is >>>
(= I 0)
Can not find suitable plan
   <<< Input sentence is >>>
(READ X)
Suitable plan is READ-PLAN
   <<< Input sentence is >>>
(WHILE (!= X 99999)
       (= S (+ S X))
       (= I (+ I 1))
       (READ X))
Suitable plan is WHILE-PROCESS-READ-PLAN

   <<< Input sentence is >>>
(= S (+ S X))
Suitable plan is TOTAL-RUNNING-PLAN
   <<< Input sentence is >>>
(= I (+ I 1))
Suitable plan is COUNTER-PLAN
   <<< Input sentence is >>>
(READ X)
Suitable plan is READ-PLAN
   <<< Input sentence is >>>
(IF (!= I 0)
    (= XX (/ S I))
    (WRITE XX))
Suitable plan is SENTINEL-SKIP-GUARD
   <<< Input sentence is >>>
(= XX (/ S I))
Suitable plan is AVERAGE-PLAN
   <<< Input sentence is >>>
(WRITE XX)
Suitable plan is WRITE-PLAN
Process end !!!
 << Interpretation-tree >>
NIL
    ((SUM INIT 22))
    NIL
    ((COUNT INIT 19))
    ((READ-AND-PROCESS INIT 26)
     (INPUT MAIN 6))
    ((READ-AND-PROCESS MAIN 26))
        ((AVERAGE PREGOAL 15)
         (SUM MAIN 22))
        ((AVERAGE PREGOAL 15)
         (COUNT MAIN 19))
        ((INPUT MAIN 6))
    ((SENTINEL-GUARD MAIN 13))
        ((AVERAGE MAIN 15))
        ((OUTPUT MAIN 6))
NIL
```

処理開始

入力文は s = 0

この文から活性化されるプランが見つからない。

read-plan がマッチする。

while-process-read-plan にマッチする。 while 文は次にブロックの中が調べられる。
この文は total-running-plan の main 文にマッチしたので、init パートをも調べたところ、以前に s = 0 という初期化文が存在するので、最終的に total-running-plan であると判断する。

同様に counter-plan である。

前半に counter-plan、及び total-running-plan がすでに確認されているのでこの文は単なる割り算ではなく、平均を求めるプランの一部であると判断する。

プロセス終了

解析木の出力。各リストの最後の数値は評価関数の値である。