17 論理プログラミング

論理プログラミングは、数理論理の公式が計算の仕様として解釈できるという観察に基づいている。プログラミングのスタイルは命令型ではなく宣言型である。代わりに、入力データと出力データの関係を記述し、入力から出力を得る方法をコンピュータに考えさせる。これが成功する限り、論理プログラミングは格段に高い抽象度を提供し、それに対応してプログラムも非常に簡潔になるという利点がある。

論理プログラミングを特徴づける抽象化には、大きく2つある。一つ目は、おなじみのfor文やif文のような制御文が抽象化されていること。その代わりに「コンパイラ」は、プログラム全体に一様に適用される非常に強力な制御メカニズムを提供する。このメカニズムは、数理論理学における証明の概念に基づいている。ステップ・バイ・ステップのアルゴリズムの代わりに、プログラムは真であると仮定された論理式（公理）の集合であると考えられ、計算はプログラムの公理から式を証明する試みである。第2の抽象化は、代入文と明示的なポインタが使われなくなることである。代わりに、データ構造の構築と分解に、単一化と呼ばれる一般化されたパターンマッチングメカニズムが使われる。ユニフィケーションの実装では、データ構造の構成要素間に暗黙のポインタが作成されるが、プログラマーが目にするのはリスト、レコード、ツリーなどの抽象データ構造だけである。

純粋な”論理プログラミングについて議論した後、最初の、そして今でも非常にポピュラーで実用的な論理プログラミング言語であるPrologによって導入された妥協点について説明します。

17.1 純粋論理プログラミング

ある文字列が別の文字列の部分文字列であるかどうかをチェックし、そのような文字列が最初に出現した場合はその文字列へのポインタを返し、そのような文字列が出現しなかった場合はゼロを返すC関数を考えてみよう。これは、2つの入れ子ループと様々な境界条件をチェックする非自明なプログラムであり、まさに不明瞭なバグを持つ可能性が高い種類のプログラムである。可能な実装は以下の通りである：

C

```
/* Is s1 a substring of s2 ? */
char *substr(char *s1, char *s2)
```

より正確には、数式はプログラムの論理的帰結である。この概念を説明するために脱線するよりも、より直感的な証明の概念を使用します。

```
{
   char *p = s2; int len1 =        /* Used to index through s2 */
    strlen(s1) - 1; int len        /* Store lengths */
   2 = strlen(s2) - 1; int
   i ;                             /* Index through strings */

   while (*p != 0) {               /* Check for end of s2 */
      i = len1;
      if (p+i) > (s2+len2)         /* Too few characters */
         return NULL;
      while ( (i >= 0) && (*(p+i) == *(s1+i)) )
         i--;                      /* Match from end of s1 */
      if (i == -1) return p;       /* Found - matched all of s1 */
      else p++;                    /* Not found - continue loop */
   }
   return NULL;
}
```

文字列sが別の文字列tの部分文字列であることの意味を正確に定義してみよう。

$$
\begin{aligned}
&t \sqsubseteq t \\
&(t = t1 \parallel t2) \wedge (s \sqsubseteq t1) \Rightarrow (s \sqsubseteq t) \\
&(t = t1 \parallel t2) \wedge (s \sqsubseteq t2) \Rightarrow (s \sqsubseteq t)
\end{aligned}
$$

これらは、より原始的な連結の概念から部分文字列を定義する3つの論理式である。最初の式は、すべての文字列はそれ自身の部分文字列であるという基本的な事実を示している。次の2つの公式は、文字列tが2つの（より小さな）文字列t1とt2に分解され、（「∧」）sがこれらの構成要素の1つの部分文字列である場合、（「⇒」）sはtの部分文字列である、というものである。もちろん、これらは循環的な定義ではなく、再帰的な定義であることはご存じだろう。しかし、これが計算とどのような関係があるのだろうか？論理式を逆に考えてみよう：

$$
\begin{aligned}
&t \sqsubseteq t \\
&(s \sqsubseteq t) \Leftarrow (t = t1 \parallel t2) \wedge (s \sqsubseteq t1) \\
&(s \sqsubseteq t) \Leftarrow (t = t1 \parallel t2) \wedge (s \sqsubseteq t2)
\end{aligned}
$$

論理式はその方法を正確に教えてくれる。まずsがおそらくtと同じかどうかをチェックし、同じでなければtを2つの部分文字列t1とt2に分解し、（再帰的に）sがどちらかの部分文字列であるかどうかをチェックする。sがtの部分文字列であることを示す分解シーケンスが見つかるま で、tのすべての可能な分解に対してこのチェックを行う。

s v tが真であるためには何が真でなければならないか？さて、もしそうなら：

$$(t = t1 \parallel t2) \wedge (s \sqsubseteq t1)$$

が真であれば、問題は解決する。しかし、ある部分文字列t1に対してs ∨ t1が真であるためには、次のようになる：

$$(t1 = t11 \parallel t12) \wedge (s \sqsubseteq t11)$$

or:

$$(t1 = t11 \parallel t12) \wedge (s \sqsubseteq t12)$$

は真でなければならない。この再帰的推論は、s ∨ sのような無条件に真となる事実に到達するまで続けられる。

For example, is "wor" ⊑ "Hello world"? "Hello world" can be decomposed in twelve different ways:

""	"Hello world"
"H"	"ello world"
"He"	"llo world"
...	...
"Hello wor"	"ld"
"Hello worl"	"d"
"Hello world"	""

可能なすべての分解について、「wor 」が2つの文字列のどちらかの部分文字列であるかどうかをチェックしなければならない。もちろん、そのような分解（ヌル文字列を構成要素の1つとするものを除く）はそれぞれ、さらなる分解の集合をもたらす。しかし最終的には、文字列を「 」と 「wor 」に分解し、「wor 」が 「wor 」の部分文字列であると結論づけることに成功する。実際、この問題を解く方法は1つではない。なぜなら、同じ構成要素をいくつかの異なる方法で得ることができ、「wor 」と「」という対称的な分解を得ることさえできるからである。

上記の計算を手作業で行うのはかなり面倒だが、これはコンピュータが得意とするタイプの繰り返し作業である。論理プログラムを実行するには、論理式の集合（プログラム）と次のような目標が必要である：

$$\text{"wor"} \sqsubseteq \text{"Hello world"} \ ?$$

推論エンジンは、問題が解かれるまで、ある式から別の式への論理的推論を行うので、#は推論エンジンと呼ばれるソフトウェアシステムに提出される。推論エンジンは、ゴールとなる式が、公理、つまり真であると仮定されたプログラムの式から証明できるかどうかをチェックする。答えはイエスかノーで、論理プログラミングでは成功か失敗と呼ばれる。例えば、「wro 」は 「Hello world 」の部分文字列ではないといったように、ゴールがプログラムから導かれないことが失敗の原因であることもあれば、プログラムの公式のひとつを省略した場合など、プログラムが正しくないことが失敗の原因であることもある。C言語のwhileループが決して終了しないのと同じようにである。

論理プログラミングの基本的な概念は以下の通り：

- プログラムは宣言的であり、数理論理の公式のみから構成される。
- 同じ述語（例えば 「v」）に対する公式の各セットは、（おそらく再帰的な）手続きとして解釈される。

- 特定の計算はゴールを提出することで定義される。ゴールとは、プログラムから証明できるかどうかをチェックする式である。
- コンパイラは推論エンジンであり、プログラムからゴールの証明の可能性を 探索する。

このように、すべての論理プログラムには、公式の集合としての読み方と、計算の仕様としての読み方の二重の読み方がある。ある意味、論理プログラムは最小限のプログラムなのです。ソフトウェア工学では、プログラムを実装しようとする前に、その意味を正確に特定するように教えられる。仕様がプログラムであれば、それ以上することはなく、何千人ものプログラマーは一握りの論理学者で置き換えることができる。論理プログラミングが些細なことではないのは、純粋な論理では実用的なプログラミングに十分な効率が得られないためであり、そのため論理の科学的理論からプログラミングへの工学的応用に至るまで、明確な段階を踏まなければならない。

論理プログラミングには「制御文」が存在しない。なぜなら制御構造はすべてのプログラムに一様であり、数式を証明するための探索から構成されるからである。問題の解を探索することは、もちろん新しいことではない。新しいのは、計算問題の解を論理証明の一般的な枠組みで探索することを提案したことである。論理学が論理プログラミングになったのは、式の構造と証明の探索方法を限定することで、論理宣言の単純さを保ちつつ、効率的な方法で問題の解を探索できることが発見されたからである。この方法を説明する前に、論理プログラミングにおけるデータの扱い方について説明しなければならない。

17.2 Unification

ホーン節とは、素式の接続(「and」)であり、単一の素式である結論を含意する式のこと：

$$(s \sqsubseteq t) \Leftarrow (t = t1 \parallel t2) \wedge (s \sqsubseteq t1)$$

論理プログラミングは、数式をホーン節に制限することで、表現力と効率的な推論の間でちょうどよいバランスが得られるという観察に基づいている。t v tのような事実は、何にも暗示されない結論であり、つまり常に真である。結論は式の先頭とも呼ばれ、この逆形で書かれると式の最初に現れるからである。

論理プログラムの計算を開始するには、ゴールが与えられる：

$$\text{"wor"} \sqsubseteq \text{"Hello world"} \ ?$$

推論エンジンはゴールと数式の結論を一致させようとする。この場合、即座にマッチする： 「wor 「は変数sにマッチし、」Hello world ”は変数tにマッ チする。この場合、変数に対する項(この場合は定数)の置換が定義される：

$$\text{"wor"} \sqsubseteq \text{"Hello world"} \Leftarrow (\text{"Hello world"} = t1 \parallel t2) \wedge (\text{"wor"} \sqsubseteq t1)$$

Applying backwards reasoning, we have to show that:

$$(\text{"Hello world"} = t1 \parallel t2) \wedge (\text{"wor"} \sqsubseteq t1)$$

論理プログラムの計算を開始するには、次のようなゴールが与えられ る。ここで、もちろん、多くのマッチの可能性があり、これが検索につながる。例えば、推論エンジンはt1が 「He 」を、t2が 「llo world 」を指すようにすることができる。

我々は 「指し示す 」と言ったか？パターンマッチングを使えば、文字列の任意の部分へのポインタを、明示的な表記なしに作ることができる！データ構造の構成要素へのポインターは推論エンジンによって自動的に（そして正しく）作成され、維持されるため、ポインターの危険性に関する注意はもはや当てはまらない。論理プログラミングは、制御構造だけでなく、動的データを割り当てたり操作したりするための明示的なポインタの必要性も抽象化する。

論理式からプログラミング言語を作るためには、1つだけ欠けている概念がある。上の例では、（変数を含まない）ゴールが真かどうかを尋ねているだけで、計算の結果はゴールが真か偽かを示す単なる成功か失敗である。ゴールに変数が含まれている場合、変数に特定の代入が行われれば、計算が成功する可能性がある。数学的論理では、これは実存的量化子を用いて表現される：

$$\exists s(s \sqsubseteq \text{"Hello world"})$$

この式が真となるような、sに置き換えられる値が存在する場合、 # これは真となる。ゴールとして表現する：

$$s \sqsubseteq \text{"Hello world"}$$

もちろん、これは 「Hello world 」がsにも代入されている場合にのみ可能である：

$$[\text{"Hello world"} \rightarrow s]$$

推論エンジンに探索を続けるように頼んだとしよう。ゴールは2つ目の式の先頭にもマッチする：

$$(\text{"Hello world"} = t1 \parallel t2) \wedge (s \sqsubseteq t1)$$

Hello world ”の可能な分解の1つを選ぶことで、推論エンジンは数式を得る：

$$(\text{"Hello world"} = \text{"Hello w"} \parallel \text{"orld"}) \wedge (s \sqsubseteq \text{"Hello w"})$$

But $s \sqsubseteq \text{"Hello w"}$ matches $t \sqsubseteq t$ under the substitution:

$$[\text{"Hello w"} \rightarrow s]$$

ゴールに別の答えを与える。

一般化されたパターンマッチングは単一化と呼ばれ、変数とデータ構造（またはデータ構造の構成要素）の間のリンクを維持することができる。変数がインスタンス化されると、つまり変数が具体的な値を受け取ると、その値は即座にすべての接続された変数に伝搬される。計算が正常に終了すると、ゴール内の変数に行われた置換が答えを形成する。

ゴールのすべての引数において、単一化は対称的であることに注意。ゴールが次のようなものだとする：

$$\text{"Hello world"} \sqsubseteq t$$

これはつまり Hello world ”はある文字列の部分文字列か？答えは 「Hello world」、あるいは 「Hello worldxxx 」となる。論理プログラムは方向性がなく、「前方」または「後方」に実行することができ、複数の解を生成する。

このように論理プログラミングは、論理的推論を探索する統一的な制御構造と、データを構成・分解する統一的なデータ構造化メカニズムに基づいている。論理プログラミングにおける「文」はただ一つ、すなわち単一化を使って素式と式を一致させる推論ステップだけである、というのが正しい。プログラミング言語がこれよりずっとシンプルになるとは考えにくく、何十もの規則や規定を含むかさばるマニュアルを勉強する必要もない！もちろん、この言語のたったひとつの「文」は非常に強力で、論理プログラムを書くのに熟達するまでにはかなりの練習が必要だ。

17.3 Prolog

計算と探索のルール

Prologは論理的なプログラミング言語であり、簡潔で効率的なプログ ラムを書くために実際に使うことができる。Prologは、これまで述べてきた理想的な論理プログラミングの概念に対して、2つの妥協をしています。1つ目は、この言語が効率的に実装できるように、論理式に特定の探索規則と計算順序規則を定義することである。もう一つは、非論理述語を含めることである。非論理述語とは、論理的な意味を持たず、その代わりに入力や出力といった副次的な効果に使われる素式である。

Prologプログラムは、ホーン節(1つの素式は、0個以上の他の式の連接によって暗示される)の集合で構成される。同じ先頭を持つ各ホーン節の集合は手続きと呼ばれる：

```
substring(T, T).
substring(S, T) :- concat(T, T1, T2), substring(S, T1).
substring(S, T) :- concat(T, T1, T2), substring(S, T2).
```

The sign ":–" denotes implication, and variables must begin with upper-case letters. Given a goal:

```
?- substring("wor", "Hello world").
```

もし統一に成功すれば、ゴールは一連の素式（ゴールとも呼ばれる）に置き換え られる：

```
?- concat("Hello world", T1, T2), substring("wor", T1).
```

推論エンジンは、解を見つける試みを続けるために、そのうちの1つを 選ばなければならない。Prologの計算規則では、推論エンジンは常に一番左の素式を選ぶ。

この例では、concatはsubstringを再帰的に呼び出す前に選択されることが計算規則で要求されている。

そして、推論エンジンは、単一化を試みるために、そのうちの1つを 選ばなければならない。Prologの検索ルールは、数式がプログラムテキストに現れる順番で試行されることを指定する。ゴール式とsubstring手続きの式をマッチさせようとするとき、検索ルールは、まずsubstring(T,T)という事実を選択し、次にsubstring(S,T1)という2番目の式を選択し、これらが失敗した場合のみ、substring(S,T2)という3番目の式を選択することを要求する。

これらの一見任意の要求の理由は、PrologをCやAdaのようなスタッ ク・アーキテクチャで実装することを可能にするためです。計算は、バックトラックによって行われます。上の例では

```
?– concat("Hello world", T1, T2), substring("wor", T1).
```

計算が置換を伴う連結を証明したと仮定する：

$$["H" \rightarrow t1, "ello\ world" \rightarrow t2]$$

ここで、substring(「wor」, 「H」)を証明しようとするが、明らかに失敗する。計算をバックトラックし、別の置換でconcatの証明を見つけようとする。バックトラックの際、substring(「wor」, 「H」)の計算に必要なデータはすべて捨てることができる。このように、Prologの計算ルールは、効率的なスタックの実装に自然にマップされる。

Prologプログラムの効率をさらに向上させるために、Prolog言語には、推論エンジンに潜在的な解空間の一部を探索しないように指示するカット（「！」と表記）と呼ばれる機能が含まれています。カットされた」可能解がないようにするのは、プログラマーの責任である。例えば、演算子で区切られた2つの項として定義される算術式を解析しようとしているとする：

```
expression(T1, OP, T2) :– term(T1), operator(OP), !, term(T2).

演算子('+').
演算子('-').
演算子('*').
演算子('／').
```

そして、ゴールはexpression(n,'+',27)である。明らかに、nと27はどちらも項であり、'+'は演算子の1つなので、ゴールは成功する。しかし、ゴールがexpression(n,'+','>')であった場合、カットがなければ計算は以下のように進む：

```
n is a term
'+' matches operator('+')
'>' is not a term
'+' does not match operator('–')
'+' does not match operator('*')
'+' does not match operator('/')
```

推論エンジンは、演算子(OP)を満足させるために、異なるマッチが項(T2)を も満足させることを期待して、後戻りしてさまざまな方法を試す。もちろん、プログラマーはそれが絶望的であることを知っている。カットが渡された後に失敗が起こると、カットは式の式全体を失敗させる。もちろん、カットはPrologを宣言的論理プログラミングの理想からさらに遠ざけるが、効率を向上させるために実際に広く使われている。

非論理式

本当に実用的であるために、Prologは論理プログラミングとは全く関係のない機能を含んでいます。定義によれば、出力文は、プログラムの外の環境に対してのみ効果があるため、計算に対して論理的な意味を持ちません。とはいえ、ファイルを開いたり、画面に文字を表示したりするプログラムを書く場合には、出力文は必要です。

Prologが純粋な論理プログラミングから逸脱しているもう一つの領域は、数値計算です。論理で足し算を定義することは確かに可能である：

$$N+0=N$$
$$N+s(M)=s(K) \Leftarrow N+M=K$$

0は数字のゼロであり、s(N)はNに成功する数字なので、例えばs(s(0))は数字の3である。(1)数字にゼロを足したものは数字そのものであり、(2)NにMの後継を足したものはN+Mの後継である。明らかに、555+777の論理版を書くのは非常に面倒であり、実行するのも非効率的である。

Prologには初歩的な式がある：

```
Var is Expression
```

式が評価され、この値で新しい変数Varが作成される。この変数の値は二度と代入されることはなく、後の初等式の引数 として使われるだけである。

Expression evaluation and assignment to a newly created variable can be used to simulate for-loops:

```
loop(0).
loop(N) :-
    proc,
    N1 is N − 1,
    loop(N1).
```

以下のゴールはprocを10回実行する：

```
?− loop(10).
```

引数はインデックスとして使われる変数である。最初の式は再帰の基本ケースであり、インデックスがゼロの場合、それ以上何もすることはない。

そうでなければ、手続きprocが実行され、新しい変数N1が作成され、N-1に設定され、loopの再帰呼び出しの引数として使用される。Unificationは、loopの2番目の式を使用するたびに新しい変数を作成する。これはスタック上で行えるので、それほど非効率的ではありません。また、多くのPrologコンパイラは、再帰呼び出しがプロシージャの中で最後に実行される文である場合、再帰を通常の反復に置き換える方法である、末尾再帰最適化を行うことができます。

isの使用が非論理的である理由は、対称的ではないからです：

```
28 is V1 * V2
```

or even:

```
28 is V1 * 7
```

ここで、V1とV2はインスタンス化されていない変数、つまり値が代入されていない変数である。これには算術の意味的な知識（整数の因数分解や除算の方法）が必要になるが、 統一では純粋に構文的なマッチングが行われる。

Prologデータベース

Prologプログラムが含むことのできる数式の数に制限はない。特に、含めることのできる事実の数に制限はないので、 Prologの事実の集合は、データベースのテーブルの機能を果たすことが できる：

```
customer(1, "Jonathan").   /* customer(Cust_ID, Name) */
customer(2, "Marilyn").
customer(3, "Robert").

salesperson(101, "Sharon"). /* salesperson(Sales_ID, Name) */
salesperson(102, "Betty").
salesperson(103, "Martin").

order(103, 3, 「Jaguar」).     /* order(Sales_ID, Cust_ID, Article) */
 order(101, 1, 「Volvo」).
 order(102, 2, 「Volvo」).
 order(103, 1, 「Buick」).
```

普通のPrologのゴールは、データベースのクエリとして解釈できます。例えば

```
?- salesperson(Sales_ID, "Sharon"),          /* Sharon's ID */
   order(Sales_ID, Cust_ID, "Volvo"),        /* Order for Volvo */
   customer(Cust_ID, Name).                  /* Customer of this order */
```

このタイプのデータベースは、リレーショナルデータベースとして 知られています。

「シャロンは誰にボルボを売ったか？もし、問い合わせが成功すれば、変数Nameは、その顧客の一人の名前を代入する。そうでなければ、シャロンはボルボを売らなかったと結論づけられる。

洗練されたデータベースクエリは、単純なPrologゴールになります。例えば 「シャロンとマーティンの両方から車を売られた顧客はいるか？

```
?- salesperson(ID1, "Sharon"),      /* Sharon's ID */
   salesperson(ID2, "Martin"),      /* Martin's ID */
   order(ID1, Cust_ID, _),          /* Sharon's customer ID */
   order(ID2, Cust_ID, _).          /* Martin's customer ID */
```

変数Cust IDは、2つの初歩的な式に共通なので、ゴールは、同じ 顧客がそれぞれの販売員に注文した場合のみ、真になります。

同様に、retract(F)は事実Fを削除します。ファクトのリストの実装は非常に効率的であり、数千のエントリーの テーブルに対するクエリに簡単に答えることができます。しかし、テーブルが数万項目からなる場合、より洗練された検索 アルゴリズムが必要になります。また、あなたのデータベースがノンプログラマーを対象としている場合、適切なユーザーインターフェイスが必要であり、Prologの実装は、この場合に適切なプログラミング言語であるかもしれませんし、そうでないかもしれません。

「これは専門家によるものではない 」ことを強調しておくことが重要である。つまり、新しい言語やデータベースの概念が導入されたわけではなく、これは普通のPrologプログラミングです。どんなプログラマーでも、ファクトを列挙するだけで、小さなデータベースを作成することができます。

Dynamic databases

もし、Prologプログラムの始めに、全ての事実のセットが存在すれば、クエリは完全に宣言的です。しかし、Prolog言語には、推論中にデータベースを修正するための非論 理的な機能がある。初等式assert(F)は、論理式としては常に真ですが、副次的な効果として、事実にFをデータベースに追加します。同様に、retract(F)は、事実にFを削除します：

```
? - assert(order(102, 2, 「Peugeot」)),／* Betty sells a car *
／ assert(order(103, 1, 「BMW」)), ／* Martin sells a car *／
assert(order(102、 1, 「Toyota」)), ／* Betty sells a car *／
assert(order(102, 3, 「Fiat」)), ／* Betty sells a car *／ ret
ract(salesperson(101, 「Sharon」)). シャロンを解雇する！*／
```

データベースの修正は、Prologの代入文をシミュレートするために使用することができます。ファクトcount(0)がプログラム中に存在すると仮定します：

```
increment :-
   retract(count(N)),        /* Erase old value */
```

```
N1 is N + 1,            新しい値を持つ新しい変数 *／
assert(count(N1)).      /* Restore new value */
```

つの素式のうち、論理式になるものは1つもない！

代入は計算の状態を記録するために使用されることを思い出してください。従って、代入をシミュレートするための代替手段は、すべての状態変数を式の追加引数として運ぶことです。実際には、Prologプログラムでは、動的データベースを実装するためや、 プログラムの可読性を向上させるために、非論理的なデータベース 操作を使用することができます。

Sorting in Prolog

論理プログラムの記述的ビューと手続き的ビューの関係の例として、 Prologのソートプログラムについて説明します。ここでは、整数のリストのソートに限定します。表記法 [Head | Tail]は、最初の要素をHead、残りの要素をTailとするリストである。[は空リストを表す。

なぜなら、リストL2がリストL1のソートされたバージョンであるということが何を意味するのかを記述すればよいからである：

```
sort(L1, L2) :- permutation(L1, L2), ordered(L2).
```

ここで、本体の数式は次のように定義される：

```
permutation([], []).
permutation(L, [X | Tail]) :-
    append(Left_Part, [X | Right_Part], L),
    append(Left_Part, Right_Part, Short_List),
    permutation(Short_List, Tail).

ordered([]).
ordered([Single]).
ordered([First, Second | Tail]) :-
    First =< Second,
    ordered([Second | Tail]).
```

L2 は L1 の全要素の並べ替え（順列）で構成される：

- 空リストは空リストの順列である。空でないリストの順列は、リストを要素Xと2つの部分Left PartとRight Partに分割し、2つの部分を連結した順列の先頭にXを付加したものである。例えば

  ```
  permutation([7,2,9,3], [9|Tail])
  ```

  if Tail is a permutation of [7,2,3].

- 要素が0個または1個のリストは順序付けされている。最初の2つの要素が順序付きであり、最初の要素以外からなるリスト も順序付きである場合、リストは順序付きである。

手続き上、これは世界で最も効率的なソートプログラムではない！これは、ソートされたリストが見つかるまで、数字のリストのすべての並べ替えを試す（生成する）だけである。しかし，前章でMLで解いた挿入ソートのような，より効率的なソートアルゴリズ ムの記述版を書くのは簡単です．

```
insertion_sort([], []).
insertion_sort([Head | Tail], List) :-
    insertion_sort(Tail, Tail_1),
    insert_element(Head, Tail_1, List).

insert_element(X, [], [X]).
insert_element(X, [Head | Tail], [X, Head | Tail]) :-
    X =< Head.
insert_element(X, [Head | Tail], [Head | Tail_1]) :-
    insert_element(X, Tail, Tail_1).
```

このプログラムは、無目的な探索をすることなく、サブリストを直接操作してソートを行うので、手続き的には非常に効率的です。関数型プログラミングのように、インデックスやforloop、明示的なポイン タは存在せず、このアルゴリズムは他のオブジェクトのソートにも即座に 一般化されます。

Typing and failure

Prologは静的に型チェックされません。残念なことに、型エラーに対するProlog推論エンジンの反応は、プログラマーに深刻な問題を引き起こす可能性があります。あるリストの長さを計算する手続きを書いたとしましょう：

```
length([], 0).              /* Length of empty list is 0 */
length([Head | Tail], N) :- /* Length of list is */
    length(Tail, N1),       /*    length of Tail */
    N1 is N+1.              /*    plus 1 */
```

なぜなら，lengthの定義に追加のマッチング式が含まれている可能性が # あるからである：

```
?- length(5, Len).
```

長さの定義に追加のマッチング式が含まれる可能性は確かにあるので、 # これは違法ではありません。

推論エンジンの反応は、単にlengthの呼び出しに失敗するというものです。lengthは他の式pの中で呼び出され、lengthの失敗がpの失敗を引き起こし（これも予期していなかった）、呼び出しの連鎖をさかのぼっていく。この結果、制御不能なバックトラックが発生し、

最終的には明確な理由もなく元のゴールが失敗することになる。このようなバグを見つけるのは、エラーが診断されるまで段階的に呼び出しをトレースしていく非常に難しい問題である。

このため、Prologのいくつかの方言は型付きであり、引数が整数かリ スト、あるいはプログラマーが定義した型のいずれかであることを宣言す る必要があります。型付けされたPrologでは、上記の呼び出しはコンパイルエラーとなる。このような方言を評価する場合、通常のトレードオフが適用されます： コンパイル時にエラーを検出するのとは対照的に、柔軟性が低下します。

17.4 高度な論理プログラミングの概念

Prologの成功に基づき、他の論理プログラミング言語が提案されてきた。多くの言語が、論理プログラミングの利点を、オブジェクト指向プログラミングや関数型プログラミングなどの他のプログラミングパラダイムと組み合わせることを試みてきた。おそらく最も集中的な努力は、論理プログラミングに固有の並行性を利用する試みに費やされてきた。論理プログラムは数式の列から構成される：

$$
\begin{aligned}
&t \sqsubseteq t \\
&(t = t1 \parallel t2) \wedge (s \sqsubseteq t1) \Rightarrow (s \sqsubseteq t) \\
&(t = t1 \parallel t2) \wedge (s \sqsubseteq t2) \Rightarrow (s \sqsubseteq t)
\end{aligned}
$$

というのも、次のような理由からである。 # そして、計算中のどの時点でも、推論エンジンは次のようなゴールの列を減らそうとしている：

$$\ldots \wedge (\text{"Hel"} \parallel t1) \wedge \ldots \wedge (t1 \parallel \text{"orld"}) \wedge \ldots$$

Prolog言語は、各ゴールを左から右に順次計算しますが、ゴールを同時 に評価することも可能です。これは、ゴールの式を接続する接続詞のため、and-parallelismと呼ばれる。ゴールとプログラム式の先頭をマッチさせるとき、Prolog言語は各式をテキスト順に順次試行するが、同時に試行することも可能である。各ゴールは、最初の式または2番目の式にマッチしなければならないので、これはor-parallelismと呼ばれる。

並行論理言語の定義と実装は難しい。and-parallelismの難しさは同期に起因する。1つの変数が2つの異なるゴールに現れる場合、例題のt1のように、実際にその変数をインスタンス化（書き込む）できるのは1つのゴールだけであり、インスタンス化が完了する前に他のゴールがその変数を読み取るのをブロックしなければならない。解が見つかったら、他のプロセスが検索を終了できるように、その事実を他の プロセスに伝えるための何らかの準備が必要である。

また、関数型プログラミングと論理プログラミングの統合にも多くの努力が払われてきた。なぜなら、関数と論理の数学には非常に密接な関係があるからだ：

$$y = f(x_1, \ldots, x_n)$$

is equivalent to the truth of the logical formula:

$$\forall x_1 \cdots \forall x_n \exists! y (y = f(x_1, \ldots, x_n))$$

ここで、∃！yは「一意なyが存在する」ことを意味する）。この2つのプログラミング概念の主な違いは以下の通りである：

1. 論理プログラミングは、関数型プログラミングで使われる(一方向の)パターンマッチングよりも強力な(双方向の)単一化を使う。

2. 2.関数型プログラムは一方向性であり、すべての引数が与えられると、プログラ ムは値を返す。論理プログラムでは、ゴールのどの引数も未指定のままにしておくことができ、単一化はそれらを答えでインスタンス化する役割を担う。

3. 論理プログラミングは、自動的に答えを探す推論エンジンに基づいている。

4. 関数型プログラミングでは、関数や型がデータとして使用できる一流のオブジェクトであるため、より高いレベルのオブジェクトを自然に扱うことができる。

5. 同様に、関数型プログラミング言語の高階機能はモジュールに自然に一般化されるのに対し、論理型プログラミング言語は「フラット」である傾向がある。

論理プログラミングにおける新しい研究分野は、純粋に構文的な単一化から、意味情報を含むようにマッチングを拡張することである。例えば、ゴールが$4 < x < 8$を指定し、式の先頭が$6 \leqq x < 10$を指定する場合、$6 \leqq x < 8$、つまり$x = 6$または$x = 7$を解くことができる。マッチングに意味情報を含む言語は制約論理プログラミング言語と呼ばれる。制約論理言語は、基礎となる方程式を解くための効率的なアルゴリズムに基づいていなければならない。

これらの進歩は、論理型言語の抽象度と効率性の両方を向上させる大きな可能性を示している。

17.5 Exercises

1. 足し算の論理的定義を使って$3 + 4$を計算せよ。

2. ループプログラムがゴールloop(-1)で呼び出された場合、何が起こるか？このプログラムはどのように修正できますか?

3. 言語特有の計算ルールのために終了しないPrologプログラムを書け。探索ルールのために終了しないプログラムを書きなさい。

4. Prologのルールは、解を深さ優先で探索することを要求している。なぜなら、左端の式が置換された後も繰り返し選択されるからである。左から右へ順次数式を選択し、他の数式がすべて選択されたときだけ一番左の数式に戻ることによって、幅優先探索を行うことも可能である。このルールは計算の成功にどのような影響を与えるだろうか？

5. 以下のクエリに対するPrologゴールを書きなさい：

シャロンが販売し、ベティが販売しなかった 車種はあるか? もしそうなら、それはどんな車種か?

6. 6.定義済みのProlog式findallを研究し、それが以下の問い合わせをどの ように解決できるかを示せ：

マーティンはシャロンより多くの車を売ったか？

マーティンはシャロンより多くの車を売ったか？

7. リストを追加するPrologプログラムを書き、MLプログラムと比較しなさい。Prologプログラムを異なる「方向 」で走らせてみて下さい。

8. 型の不一致を検出してエラーメッセージを書くように Prologプロシージャを修正するには？

9. どのようなタイプの論理プログラムがand-parallelismの恩恵を受け、 どのようなタイプがor-parallelismの恩恵を受けるか？