平成 19 年度

東京大学大学院情報理工学系研究科

コンピュータ科学専攻

入学試験問題

専門科目 I

解答

東京大学理学部情報科学科 2007

2012 年 8 月 7 日

問題 1

(1)

テイラー展開は以下の式で与えられる。

$$f(x+ih) = f(x) + ihf'(x) + \frac{(ih)^2}{2}f''(x) + \frac{(ih)^3}{6}f^{(3)}(x)$$

...

両辺の係数を比較し、できるだけ同じになるようにする。-1 次の項を 0 にするために $a_{-1}+a_0+a_1=0$, 0 次の項を $f'(x)$ にするために $-a_{-1}+a_1=1$, 1 次の項を 0 にするために $a_{-1}+a_1=0$ とする。この 3 本の式より

$$a_{-1} = -\frac{1}{2}, a_0 = 0, a_1 = \frac{1}{2}$$

が決まり、2 次の項 $\frac{1}{6}h^2 f^{(3)}(x)$ が残る。

(2)

同様に、低い次数を消すように 5 本の独立な式を立てる。

$$b_{-2}+b_{-1}+b_0+b_1+b_2=0$$

$$-2b_{-2}-b_{-1}+b_1+2b_2=0$$

$$2b_{-2}+\frac{1}{2}b_{-1}+\frac{1}{2}b_1+2b_2=0$$

$$-\frac{8}{6}b_{-2}-\frac{1}{6}b_{-1}+\frac{1}{6}b_1+\frac{8}{6}b_2=0$$

$$\frac{2}{3}b_{-2}+\frac{1}{24}b_{-1}+\frac{1}{24}b_1+\frac{2}{3}b_2=0$$

これを解いて、$b_{-2}=\frac{1}{12}, b_{-1}=-\frac{2}{3}, b_0=0, b_1=\frac{2}{3}, b_2=-\frac{1}{12}$ となり、4 次の項 $\frac{1}{30}h^4 f^{(5)}(x)$ が残る。

(3)

同様に、h の次数ごとに係数比較。

$$d_{-1}+d_0+d_1=0$$

$$c_{-1}+c_0+c_1=-d_{-1}+d_1=1$$

$$-c_{-1}+c_1=\frac{1}{2}d_{-1}+\frac{1}{2}d_1$$

$$\frac{1}{2}c_{-1}+\frac{1}{2}c_1=-\frac{1}{6}d_{-1}+\frac{1}{6}d_1$$

$$-\frac{1}{6}c_{-1}+\frac{1}{6}c_1=\frac{1}{24}d_{-1}+\frac{1}{24}d_1$$

これを解いて、$c_{-1}=\frac{1}{6}, c_0=\frac{2}{3}, c_1=\frac{1}{6}, d_{-1}=-\frac{1}{2}, d_0=0, d_1=\frac{1}{2}$、4 次の項が残る。

(4)

よくわかりません助けてください。

問題 3

(1)

問題

次の論理式を充足する Herbrand 解釈を与えよ.

$$P(c) \land \neg P(f(c)) \land (\forall x.P(x) \supset P(f(f(x)))) \land (\forall x.P(f(f(x))) \supset P(x))$$

解答

H を Herbrand 領域とする。

$$\mathbf{H} = c, f(c), f(f(c)), ...$$

となる。ここで、c に対して f を n 回適用したものを $f^n(c)$ と表記することにする。

P の解釈 I(P):$\mathbf{H} \to \{\top, \bot\}$ として、以下のようなものを与える。

$$I(P)(f^n(c)) = \begin{cases} \top(\text{n が偶数}) \\ \bot(\text{n が奇数}) \end{cases}$$

この解釈は与論理式を充足する

(2)

問題

次の論理式を充足する Herbrand 解釈が存在しないことを説明せよ.

$$P(c) \land (\forall x.P(x) \supset P(f(x))) \land (\forall x.\exists y.P(x) \supset \neg P(y))$$

解答

(なんかかなり汚い回答になってしまった)

背理法を使う。この論理式を充足する Herbrand 解釈 I が存在したとする。

$P(c) \land (\forall x.P(x) \supset P(f(x)))$

の部分より、帰納的に全ての n に対して、$I(P)(f^n(c))$ は全て $\top$ でなければならない。($\because$ $P(c)$ の部分より n=0 のとき $\top$ となり、$(\forall x.P(x) \supset P(f(x)))$ の部分より $n \geq 0$ も $\top$ となる)

故に、これは $(\forall x.\exists y.P(x) \supset \neg P(y))$ の部分を充足しない。

よって、背理法より、この論理式を充足する Herbrand 解釈 I は存在しない。

解説

Herbrand の定理はあくまで Skolem 化を終えたあとの話なので、ここでは使えません。

(3)

問題

(2) の式を Skolem 関数を導入して、充足する Herbrand 解釈を与えよ

問題

与論理式を冠頭形にする。

$$\forall x.\exists y.\forall z. \quad P(c) \wedge (P(z) \supset P(f(z))) \wedge (P(x) \supset \neg P(y))$$

これを Skolem 化する (Skolem 関数:g(x))

$$\forall x.\forall z. \quad P(c) \wedge (P(z) \supset P(f(z))) \wedge (P(x) \supset \neg P(g(x)))$$

すると、Herbrand 領域 $\mathbf{H}$ は、c に f と g を適用していった形の全体となる。
これに対して、次のような Herbrand 解釈はこの論理式を充足する。

$$I(P)(t) = \begin{cases} \top(t = f^n(c)(c \geq 0) \text{ の形をしている場合}) \\ \bot(\text{それ以外、すなわち f,g を合成した関数 h に対して、t=h(g(c)) の形をしている場合}) \end{cases}$$

問題 4

(1)

多くのプログラミング言語では、関数呼び出しの際に、メモリのスタック領域に少なくとも次のものが置かれる。

- 呼び出す関数に渡す引数
- 呼び出し元の関数の、呼び出し時の局所変数
- 戻りアドレス

そのため、固定サイズのメモリ領域で関数呼び出しを実現可能にするためには、以下のどちらかの関数呼び出しの規約が必要となる。

- 呼び出し元の局所変数や戻りアドレスを退避する必要がないように、末尾呼び出しのみを許す
- 関数の呼び出しの深さに上限を定める

また、どちらについても、引数の数に上限を定める必要がある。(さらに、関数呼び出しとは関係がないが、関数の局所変数の数の上限も定める必要がある。)

(2)

関数の入れ子がない場合、関数の呼び出しがあるたびにフレームを作成し、その中に局所変数などを入れ処理を行い、呼び出し先から戻る際にフレームを破棄すれば良い。

高階関数がなく関数の入れ子がある場合、関数 f 内で参照される変数 v が f の「外側」の関数で定義されたものであるときに、v の値は f のフレーム内にはない。そのため、入れ子がない場合の方法では、入れ子がある場合に一部の変数を参照することができない。

ディスプレイ法は、関数の入れ子がない場合の手法に加え、その時点で参照可能な関各数のフレームへのポインタを保 有する「ディスプレイ」と呼ばれるデータ構造を定義し、関数呼び出しがなされる度にディスプレイのポインタを更新していくことで、変数の値を適切に参照できるようにした方法である。

具体的には、ディスプレイ法の動作は次のようなものである。

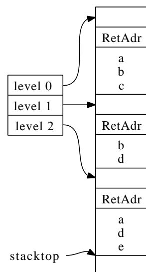
- ネストレベル i の関数を呼ぶとき、
  - ディスプレイ中のレベル i のポインタを新しいフレーム内に退避
  - 新しいフレームへのポインタをディスプレイのレベル i にセット
- ネストレベル i の関数から帰るとき
  - トップフレームに退避してあったポインタをディスプレイのレベル i にセット

例として、高階関数がなく関数の入れ子がある言語で書かれた次のコードを考える。関数のネストレベルは、それぞれ p が 0、q が 1、r が 2 である。

```
1 p() {
2   int a, b, c;
3   q() {
4     int b, d;
5     r() {
```

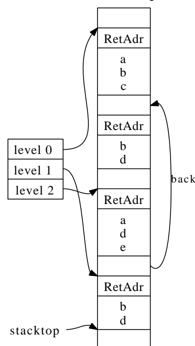
```
 6        int a, d, e;
 7        a = c+b;          /* *2* */
 8        q();              /* *3* */
 9      }                   /* *5* */
10      r();                /* *1* */
11      c = a;              /* *4* */
12    }
13    a = b+c;
14    q();
15  }
```

ここで、最初に関数 r() が呼ばれる (*1*) が、r() が呼ばれた時のスタック及びディスプレイは次のような状態になっている。



ここで、例えば*2*で b を参照するとき、スタックトップから b を探していき、最初に見つかる q で宣言された b が使われる。

関数 r() 内では関数 q() が呼ばれる (*3*) が、q() が呼ばれた時のスタックとディスプレイは次のような状態になっている。



ここで、例えば*4*で a を参照するとき、同様にスタックトップから a を探していくが、back ポインタによって、(トップの)q のフレームの次に探すのは p のフレームとなるので、p 内で宣言された a が使われることになる。

そして、*3*で呼ばれた関数 q から帰ってくる (*5*) と、スタックとディスプレイは*1*と同じ状態に戻る。

(3)

例えば、次のプログラムを考える。

```
1  let f x =
2    let g y = x + y in
3      g in
4  let g3 = f 3 in        (* *1* *)
5    g3 5                 (* *2* *)
```

この言語は高階関数がある言語で書かれているとするので、この言語では関数が (何らかの形で) 値として扱えるとする。

ここで、*1*において f が x=3 として呼ばれ、「外側の中で最も内側にある関数で定義された x」と引数 y とを足す関数が新たに定義され、g3 にはそのアドレスが入る。このとき、*1*での f の呼び出しから戻ると、f のフレームは破棄されてしまうため、*2*で g を呼び出して演算を行う際に x の値は意図しないものになりかねない。

(4)

関数が自由変数を持つことを許さず、内側の関数が外側の関数で宣言された変数にアクセスすることを、外側の関数が内側の関数への引数として渡すことによってのみ可能とすることで (2) の実現法が使用できる。

例えば先ほどの例をこの制限に合わせて書き直すと次のようになる。

```
1 let f x =
2   let g x y = x + y in
3     g x in
4 let g3 = f 3 in        (* *1* *)
5   g3 5                 (* *2* *)
```

この場合、*1*で let g x y = x + y in g 3 という関数が定義され、(この例の場合言語がカリー化をサポートしている必要があるが、) それへのポインタが g3 に代入される。このときは (3) の場合のように値が別の関数内での値に依存することがないため、(2) の方法で関数呼び出しが実現できる。

問題 5

(1)

$\frac{\sum_{k=1}^{n} T_i}{T_p+\max(T_i)}$

最長の時間がかかるステージの実行時間にパイプラインレジスタの遅延時間を足したものが理想的なクロックサイクル時間となる。パイプライン化では各ステージが完全に並列実行でき、スループットのみが性能に影響を与える場合に最も高い性能向上比が得られ、そのときの性能向上比はもとの実行時間をクロックサイクル時間で割ったものとなる。

(2)

構造ハザード

原因 資源の競合に起因するハザード。異なるステージで同じ資源に対する要求があると起きる。レジスタの読み込み 2 ポート書き込み 1 ポートの場合、パイプライン化されていないプロセッサではレジスタポートは 2 あればよかったが、パイプライン化されたプロセッサでは読み込みと書き込みを同時に実行しようとすると競合が起きハザードが発生する。

速度低下を減少させる方法 資源を多重化する。ハードウエア量を増やす物理的に多重化する方法と、ハードウエア量は増やさないが資源を利用する時間をずらす時間的に多重化する方法がある。

制御ハザード

原因 分岐命令などがある場合、その結果が後のステージにならないとわからないために発生する。結果を待たないと次に実行すべき命令が定まらないためその後の実行を停止しなくてはならない。

速度低下を減少させる方法 分岐予測により、実行結果を待たずに後続の命令の処理を続けることができる。また、分岐命令を条件実行命令にすることで制御ハザードをデータハザードに転嫁することができる。同時マルチスレッディングを行えば予測ミスやハザードによるストール中に別スレッドの命令実行を行うことができ、ハードウエアの利用率を上げることができる。

(3)

WaW 命令完了が OoO なので同じオペランドに書き込む命令の順番が保たれず、あとに間違った値が残ってしまう危険がある。

WaR オペランドからの読み込みのあとに書き込まなければならないのに命令発行が OoO なので書き込み後の値を間違って読んでしまう危険がある。

RaW オペランドへの書き込みのあとに読み込まなければならないのに命令発行が OoO なので古い値を間違って読んでしまう危険がある。

(4)

レジスタリネーミングによりレジスタに対する WaW ハザード・WaR ハザードを除去することができる。リオーダバッファなどのアウトオブオーダの命令完了をインオーダ化するコミット機構とロードバイパシングなどの技術によりメモリに対するハザードによる速度低下を減少させることができる。RaW ハザードは命令ウインドウを大きく取るなど引き出すことのできる並列性を高めることで速度低下を減少させることができる。値予測により投機的に実行することもできる。

(5)

問題　OoO では ABC という並びの命令が ACB と実行されることがある。このとき AC が完了した直後に割り込みが入った場合、割り込み処理後どの地点から復帰したらよいかわからないという問題がある。また、Store 命令や IO 命令の実行をどの時点で行うかも割り込みの実現方法に影響する。

解決方法　リオーダバッファを用いることで命令コミットがインオーダとなる。どの命令からリオーダバッファに残っているか記録しておけば、割り込みが入った時点でバッファをフラッシュし、記録してあったアドレスから命令実行を再開することで割り込みからの復帰を正しく行うことができる。また、リオーダバッファから追い出される時点で Store 命令や IO 命令を実行することで適切な順序を保つことができる。

専門科目 II

問題 3

(1)

$$L = \{u^n v^n | n \geq 1\}$$

(2)

$$A \to uAv$$
$$A \to \varepsilon$$

(3)

背理法で示す．L を受理する決定性有限オートマトンが存在すると仮定して，それを $M = (Q, \Sigma, \delta, q_0, F)$ とする．また，$N = |Q|$ とする．このとき，記号列 $w = u^{2N}v^{2N}$ を考える．$w \in L$ であるから，M による w の受理計算 $q_0, q_1, ..., q_{2N}, p_1, p_2, ..., p_{2N}$ が存在する．ところが，状態数は全部で $|Q|$ 個しか無いので，状態 $q_0, q_1, ..., q_{2N}$ には重複がある．すなわち，ある $0 \leq i < j \leq N$ が存在して，$q_i = q_j$ となっている．そこで，上の受理計算において，状態 q_i から q_j の間をスキップした計算 $q_0, q_1, ..., q_i, q_{j+1}, ..., q_{2N}, p_1, p_2, ..., p_{2N}$ を考えると，この計算より記号列 $u^{2N-(j-i)}v^{2N}$ が受理されている．ところがこの記号列は L に明らかに含まれないため，これは矛盾である．よって言語 L は正則でない．

(4)

背理法で示す.L' が文脈自由言語と仮定すると, 文脈自由言語の pumping lemma より, この時 L' にのみ依存する定数 N が存在して,$|z| \geq N$ を満たす言語 L' の記号列 z に対して, 次の 3 つの条件を満たす z の分解 $z = uvwxy$ が存在する.

1. $|vx| \geq 1$
2. $|vwx| \leq N$
3. 全ての $n \geq 0$ に対して $uv^nwx^ny \in L$ である.

ここで,$z = a^Nb^Nc^Nd^N \in L'$ とする. 条件から,$|vwx| \leq N$ であるから,vwx は 3 種類以上の文字を同時に含むことが出来ない.

(i)

$vx \in \{a\}^*$ のとき.$n = 0$ とすれば $uwx \in L'$ となり,$n = 1$ とすると $uvwxy \in L'$ となる. しかし $|vx| \geq 1$ であるので,$uwy \in L'$ に現れている a の個数は N より少なくなっている. したがって $uwy \notin L'$ となり, 矛盾が生じる.

(ii)

$vx \in \{b\}^*$ または $vx \in \{c\}^*$ のとき. 上と同様に矛盾.

(iii)

vx が 2 種類の文字を含むとき. 上と同様に矛盾.

(i),(ii),(iii) より, 言語 L' は文脈自由言語でないから, 言語 L' は線形文法によって生成されない.

問題 3

(1)

〇が必勝戦略を持つノードは 1,2 である.1 は□が次の手を打てないため,2 は 1 への辺がある為である. □が必勝戦略を持つノードは 5,6 である.5 は〇が次の手を打てないため,6 は 5 への辺がある為である.

(2)

〇が必勝戦略を持つノードは 1,2,3,4,9 である.3,4 はそれらのノードのみが続く無限ゲームとなるため,9 は 4 への辺がある為である. □が必勝戦略を持つノードは 5,6 である.

(3)

〇が必勝戦略を持つノードは 1,2,3,4,9 である. □が必勝戦略を持つノードは 5,6,7,8 である.8 は 5 へと至る辺か*を無限回含む無限ゲームのどちらかの選択肢しかたいため,7 は 8 への辺がある為である.

問題 4

(1)

A^2 の 1 行目を求める。計算がとても面倒だが、(4) の答えがだいたい推測できる。

$$P_2 = \frac{1}{256}(103P_0 + 51Q_0 + 51R_0 + 51S_0)$$

(2)

固有値 1 に対応する固有ベクトル $\lambda_1 = (1,1,1,1)^T$, 固有値 $\frac{1}{4}$ に対応する 2 つの独立な固有ベクトル $\lambda_2 = (0,1,1,-2)^T, \lambda_3 = (0,1,-2,1)^T$, 固有値 $\frac{1}{16}$ に対応する固有ベクトル $\lambda_4 = (3,-2,-2,-2)^T$ がたしかに存在する。■

(3)

代入してみると、$Ap = \lambda_1 av_1 + \lambda_2 bv_2 + \lambda_3 cv_3 + \lambda_4 dv_4$ となる。

$$A^n p = \lambda_1^n av_1 + \lambda_2^n bv_2 + \lambda_3^n cv_3 + \lambda_4^n dv_4$$

(4)

上式より、$|\lambda_k| < 1$ ならば 分割を繰り返すと i 乗は 0 に収束する。$\lambda_1^i = 1$ の項だけが残り、$\lim_{i\to\infty} A^i p = a(1,1,1,1)^T$ ■

(5)

上で定義した固有ベクトルを用いて $(P_0, Q_0, R_0, S_0)^T$ を表す。

$$P_0 = a + 3d$$

$$Q_0 = a + b + c - 2d$$

$$R_0 = a + b - 2c - 2d$$

$$S_0 = a - 2b + c - 2d$$

$$a = \frac{2P_0 + Q_0 + R_0 + S_0}{5}$$

$$b = \frac{Q_0 - R_0}{3}$$

$$c = \frac{Q_0 - S_0}{3}$$

$$d = \frac{3P_0 - Q_0 - R_0 - S_0}{5}$$

よって、P_i は av_1 の 1 行目 $\frac{2P_0+Q_0+R_0+S_0}{5}$ に収束する。

問題 5

(1)

LRU(Least Recently Used) アルゴリズムとは使われてから最も長い時間が経ったものをスワップアウトするアルゴリズムである。

参照するページ	0	2	3	1	0	2	4	0	2	3	1	4
最後の参照からの経過時間順	0	2	3	1	0	2	4	0	2	3	1	4
		0	2	3	1	0	2	4	0	2	3	1
			0	2	3	1	0	2	4	0	2	3
				0	2	3	1	1	1	4	0	2
							3	3	3	1	4	0

というわけで数えると、

$$F(m) = \begin{cases} F(1) = 12 \\ F(2) = 12 \\ F(3) = 10 \\ F(4) = 8 \\ F(m) = 5 \quad (m \geq 5) \end{cases}$$

(2)

FIFO アルゴリズムとは主記憶に読み込まれてから最も時間の経ったページをスワップアウトするアルゴリズムである。

数えると、$F(3) = 9, F(4) = 10$ より $m = 3$ で $F(m) < F(m+1)$ となる。

(3)

各ページエントリに reference bit を設け、ページにメモリアクセスがあったときそのエントリの reference bit を 1 にする。

(4)

- 全ページエントリの reference bit の初期値は 0 とする。
- 定期的に、あるいはページの置き換えが発生したときに全ページエントリの reference bit を 0 にする。
- ページの置き換えが必要になったときは reference bit が 0 であるページから置き換え対象を選ぶ。

reference bit が 0 であるということは一定の間アクセスがなかったということなので、これにより擬似的に LRU アルゴリズムを実装することができる。[*1]

[*1] オペレーティングシステムの授業のプリントのうち右上の日付が 2009/6/27 のものの p.9 と、実例として x86 アーキテクチャの仕様に参考にするとこんな感じになるはず。