プログラミング言語処理

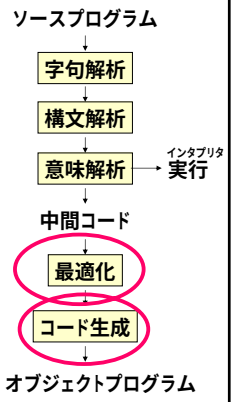
第８回（平成１５年度１１月１１日）

最適化について

筑波大学　佐藤三久

言語処理系の基本構成

- 意味解析(semantics analysis): 構文木の意味を解析する。コンパイラでは、この段階で内部的なコード、中間コードに変換する。
- 最適化(code optimization): 中間コードを変形して、効率のよいプログラムに変換する。
- コード生成(code generation): 内部コードをオブジェクトプログラムの言語に変換し、出力する。例えば、ここで、中間コードよりターゲットの計算機のアセンブリ言語に変換する。



コード最適化 とは

- 最適化とは、効率のよい目的プログラムを生成することである。
- 「効率のよい」 とは？
  - なるべくサイズの小さいコードを生成するのも「効率のよい」と意味にもなる。コードを小さくするためには、なるべく小さい命令コードですむスタックマシン（大抵、１バイトで表現されるためバイトコードとも呼ばれる）にし、それを仮想スタックマシンで実行する方法もこの一つである。
  - しかし、一般的には「効率のよい」とは、速いコード、すなわち実行時間が短いコードのことをいう。
  - また、「最適化」といっているが、「最適」は事実上、不可能であるため、ここではなるべく効率を改善するという意味である。

実行時間を短くする

- 行時間を短くするには、大別して以下のことを考える
  - 命令の実行回数を減らす。より早い命令（もしくは命令の組み合わせ）を使う。
  - メモリ階層を効率的に使う。
  - 並列度の高い命令を使う。

最適化：命令数を減らす

- 命令の数を減らしても必ずしも、速いとはいえない。
  - RISCマシンでは大抵の命令は一定のサイクル（１サイクル？）で実行されるために命令数を減らすことは実行時間の短縮になるが、CISCマシンではそうはいえない。
  - 命令数を減らす最適化は、基本的には無駄な計算を取り除くことである。例えば、ループ内で同じ計算している場合には、これをループの外で計算することによって、大幅に命令数を減らすことができる。

最適化：メモリ階層をうまく使う

- 現在のマイクロプロセッサはレジスタ、キャッシュ（１次、２次）、メモリ、そしてディスクというメモリ階層をもっている。
  - 特にコンパイラではレジスタを効率的に使うことは重要な最適化になっている。
  - もっと進んだコンパイラでは、ループの実行順序を入れ替えて、なるべくキャッシュを効率的に使う最適化を行うものもある。

最適化：命令の並列度を上げる

- プロセッサの中で命令は並列に実行されるのが普通である。
  - 現在のマイクロプロセッサではスーパースケラ（SuperScalar）機構があるが、この機構を効率的に使うために命令をいれ変える。
  - また、数値計算を効率的に行うベクトルマシンに対しては、この機構を利用するコードを生成するが、これも並列度を持つ命令を使う最適化の一つである。

ループ最適化

- コンパイラで最も重要な最適化は、ループに関する最適化である。
- このループ最適化は上に挙げた最適化の組み合わせで行われる。
  - 多くのプログラムで、比較的小さい部分が実行時間のほとんどを占めるといわれている。つまり、数個のループを最適化することで実行時間を多くを改善することができる。

コンパイラでできないこと

- コンパイラでできない（できることもある？）最適化は、アルゴリズムの最適化である。
  - 例えば、ソートする部分をバブルソートからもっとよいquickソートにするだけで大幅に性能が改善する。しかし、バブルソートから自動的に quickソートに変換することはコンパイラではできない。
- コンパイラは、ひどいアルゴリズムを救うことはできない。このような最適化はプログラムの本質の部分であり、プログラマの力量が問われるところでもある。

命令の実行回数を減らす最適化

1. １度計算した結果を再利用する。（共通部分式の削除）
2. コンパイル時に実行できるものは実行（計算）しておく。（定数の畳込み）
3. 命令をより、実行頻度が低い部分に移す。（ループ最適化：ループ不変式の削除）
4. 実行回数を減らすように、プログラムを変換する。（ループ変換）
5. 式の性質を利用して、実行を省略する。（帰納変数の削除、演算子の強さの低減）
6. 冗長な命令を取り除く。（死んだコードの削除、複写の削除）
7. 特殊化する。（手続き呼び出しの展開、判定の展開）

共通部分式の削除
(common sub-expression elimination)

- b+cに関して、(1)が先に実行されていて、(1)から(2)の間にb+cが変わらない時、(2)のb+cは、aに置き換えることができる。これを、*共通部分式の削除*という。

```
a=b+c;     (1)
 ....
x = (b+c)*e; (2)
```

⇒

```
a=b+c;     (1)
 ....
x = a*e; (2)
```


定数の畳込み(constant folding)
定数伝播(constant propagation)

- (1)をa=7にしてしまう最適化を、*定数の畳み込み*と呼ぶ。
- 共通部分式の削除と同様に、(1)の後に(2)が実行され、(1)から(2)の間にaの値がかわらなければ、b=14にしてしまうことができる。この最適化を、*定数伝播*と呼ぶ。

```
a=3+4   (1)
....
b = a*2  (2)
```

⇒

```
a=7   (1)
....
b = a*2  (2)
```

⇒

```
a=7   (1)
....
b = 14  (2)
```

ループ不変式の削除 (loop invariant motion)

- ループ内で変わらない式を *ループ不変式*で、これをみつけ移動すること*(code motion)*を*ループ不変式の削除*という。
  - このためには、ループ内で、代入されていなく、かつ関数呼び出しがあった場合そこでも代入されていないことを確かめる必要がある。
- ループのなかで、bとcの値が変わらなければ、a=b*cは、ループの外に出しておくことできる。

```
for(i=0; i < 10; i++){
    ....
    a=b*c;
    ...
}
```

⇒

```
a=b*c;
for(i=0; i < 10; i++){
    ....
    ...
}
```


帰納変数の削除(reduction variable elimination) 演算子の強さの低減(strength reduction)

- ループを制御するiのような変数を*ループ変数*（*loop variable*）と呼ぶ。ループ内に、i=i+cしか代入がない変数を*基本帰納変数(basic induction variable)*という。
  - ループ変数は、基本帰納変数である。
- この基本帰納変数に対し、*帰納変数(reduction variable)*とは、基本帰納変数もしくは、変数に代入されるたびにiの線形の関数になる変数である。
- k*10という乗算を加算というより簡単な演算に変形しているが、これを演算子の *強さの低減(strength reduction)* と呼ぶ。
  - 一般に乗算よりも加算は速く実行できるので、効率的になる。この最適化は帰納変数 x に対し、線形の計算 a*x+b に適用できる最適化である。

```
for(i=0; i < 10; i++){
    ....
    k = 10*i+123;
    ...
}
```

⇒

```
k=123;
for(i=0; i < 10; i++){
    ....
    k = k+10;
    ...
}
```


帰納変数の削除(reduction variable elimination) 演算子の強さの低減(strength reduction)

- C言語では配列を使った計算について、余計なアドレス計算を削除する最適化が行われれることがある。多次元配列などについては、そのままコード生成すると乗算が必要となるが、この最適化で加算のみで計算されるようになる。

```
for(i = 0; i < 10; i++){
    ....
    x = a[i];
    ...
}
```

⇒

```
p = a;
for(i = 0; i < 10; i++){
    ....
    x = *p;
    ...
    p++;
}
```


演算子の強さの低減(strength reduction)

- 演算子の強さの軽減とは、一般的にはx*2をx+xとしたり、x**2(べき乗) をx*xとしたり、より簡単な演算に置き換えることをいう。

ループ展開(loop unrolling)

- 複数の繰り返しをほどく最適化
- これにより、条件判定が半分になるほか、ループ内のレジスタの割り当てが効率的になることがある。
- これを刻み幅を2にして、２回ずつ展開

```
for(i = 0; i < 100; i++){
   a[i]= ...;
}
```

⇓

```
for(i = 0; i < 100; i+=2){
   a[i]=....;
   a[i+1]=...
}
```


ループ融合(loop fusion)

- ２つのループを一緒にする最適化
- ループ間でデータの依存がないことを確認しなくてはならない

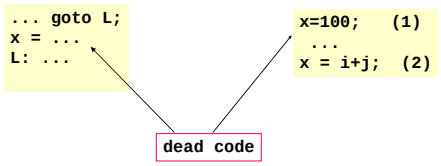
```
for(i = 0; i < 10; i++){
     .... a[i] = ...; ...
}
 for(i = 0; i < 10; i++){
     .... b[i] = ...; ...
}
```

⇒

```
for(i = 0; i < 10; i++){
     ... a[i] = ...; ...
     ... b[i] = ...; ...
}
```

死んだ命令の削除(dead code elimination)

- 不要な命令をdead codeという
- (1)の後に(2)が実行され、(1)から(2)の間にxが使われなければ、(1)は不要な命令であり、削除できる。



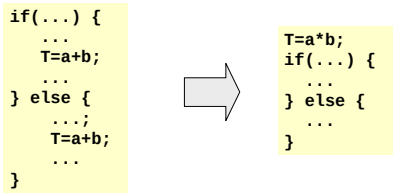
複写の伝播(copy propagation)

- (1)の後に(2)が実行され、aもbも代入されなければ、(2)は、c=b+dにすることができる

```
a= b; (1)
...
c=a+d; (2)
```

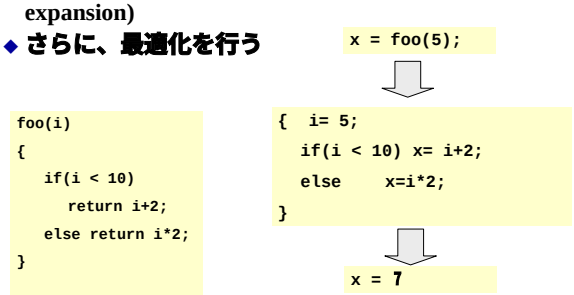

コードの巻き上げ(code hosting)

- 分岐先で同じ計算をする場合、コードを移動することを *コードの巻き上げ(code hosting)*という



手続き呼び出しの特殊化

- 関数を適用して展開してしまう（in-line expansion)
- さらに、最適化を行う



式の性質の利用

- 式の性質を利用して計算を省略できる。
  x*1はx,
  y+0は、y

最終課題

以下の２つの課題のどちらかを選択し、レポートを提出すること。

- 課題8-1：これまで説明したtiny Cのコンパイラでは大域変数や配列宣言を処理していない。配列宣言と配列参照を処理できるように拡張して、8 queenのプログラムを書き、コンパイル、実行しなさい。
  ヒント：
  - まずは適当なプログラムを作ってみて、-sのオプションを付けてコンパイルして、どのようなコードに変換されるかを調べること。
  - Cの大域的な宣言 int a[10]は、.comm a,40, 32のようにコンパイルされている
  - 適当な中間コードを加えて、それに対するgenFuncCodeのルーチンを書けばよい。
- 課題8-2：これまで説明したtiny Cのコンパイラに対して、自分なりの拡張をして、その拡張を用いたプログラムを実行しなさい。