cybozu labs

レジスタ割り当てアルゴリズム

第 6 回 自作 CPU を語る会 2025/10/18

@uchan_nos

自己紹介

- 名前：内田公太（X：@uchan_nos）
- 所属：サイボウズ・ラボ株式会社

- 著書：
  - 『ゼロからの OS 自作入門』
  - 『自作エミュレータで学ぶ x86 アーキテクチャ』
- その他：
  - 自作 CPU を語る会の運営の一人
  - osdev-jp 創始者の一人

この発表の目的

Compilers: Principles, Techniques, & Tools Second Edition（通称「ドラゴンブック」）で解説されているコンパイラのレジスタ割り当てアルゴリズムを紹介します。

皆さんが、高々 2 個のレジスタだけを使う簡易なコンパイラ実装を卒業し、複数のレジスタを有効活用するものを作るきっかけになることが目標です。

この発表で説明すること

- 式に対する最適なレジスタ割当て
  - Ershov（エルショフ）数を用いてレジスタを決める方法
- 基本ブロック内でのレジスタ割当て
  - 3 アドレスコードと基本ブロック
  - レジスタ記述子とアドレス記述子を用いる方法

説明しないこと

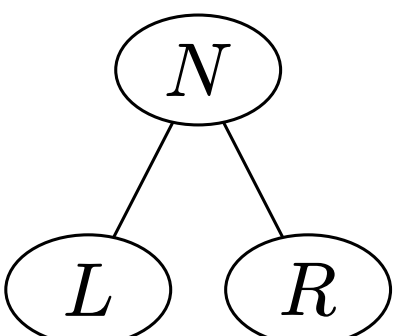
- ソースコードから 3 アドレスコードへ変換する方法
- 各種のデータフロー解析
- ブロックをまたがるグローバルなレジスタ割当て

式に対する最適なレジスタ割当て

愚直なレジスタ割当て

```
Gen(Reg, N): // ノードを評価し、結果をレジスタ Reg に格納
    If N is 変数:
        Ld(Reg, N)
    Else:
        Gen(Reg, N->L)
        tmp = GetFreeReg()
        Gen(tmp, N->R)
        Op(Reg, Reg, tmp)
        AddFreeReg(tmp)
```

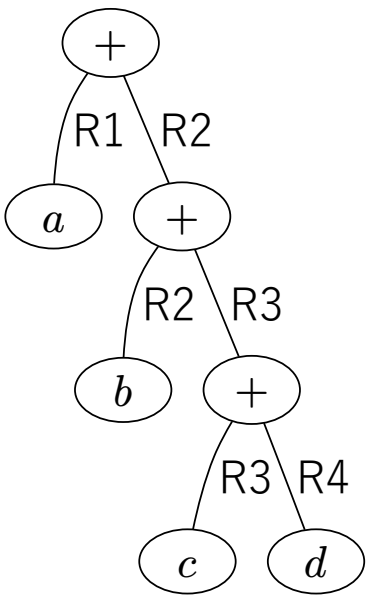
構文木



レジスタ消費が多い場合

a + (b + (c + d)) に対する出力例

```
LD R1, a
LD R2, b
LD R3, c
LD R4, d
ADD R3, R3, R4
ADD R2, R2, R3
ADD R1, R1, R2
```



機械語は以下 3 種を仮定

- LD reg, mem
- ST mem, reg
- OP reg, reg, reg

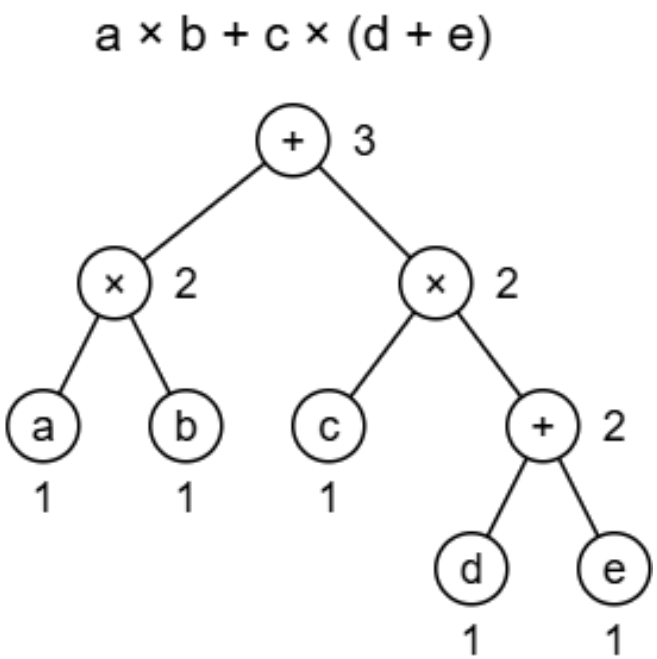
4 個のレジスタを消費（本来は 2 個で足りる）

Ershov 数

そのノードを評価するのに必要なレジスタの数

求め方

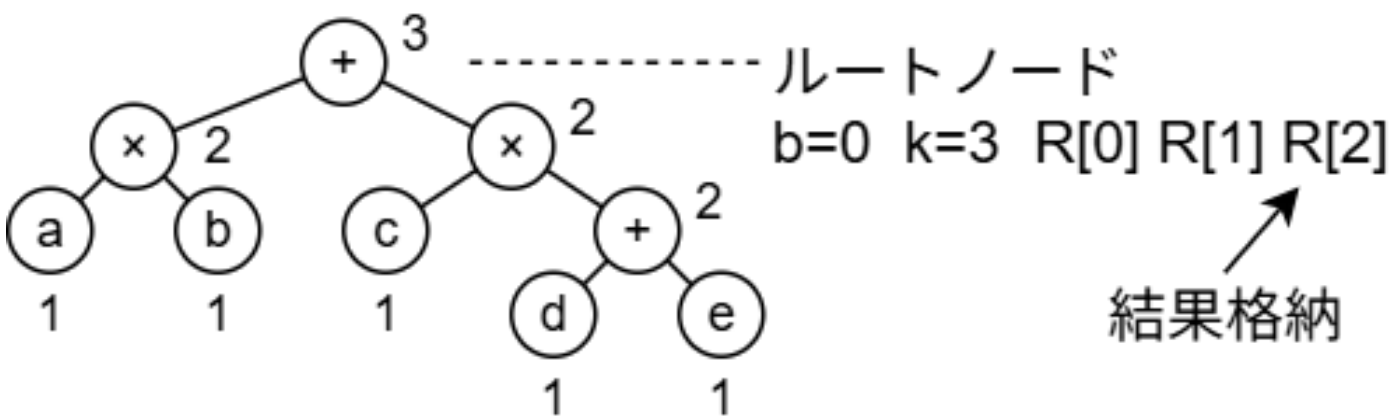
1. すべての葉のラベルを「1」とする
2. 1 つの子だけ持つ中間ノードのラベルは、
   子と同じ数値とする
3. 2 つの子を持つ中間ノードのラベルは
   1. 子の数値が等しい→ +1
   2. 子の数値が異なる→大きい方と同じ



Ershov 数を使ったコード生成

準備: レジスタに通し番号を振る：R[0], R[1], ……

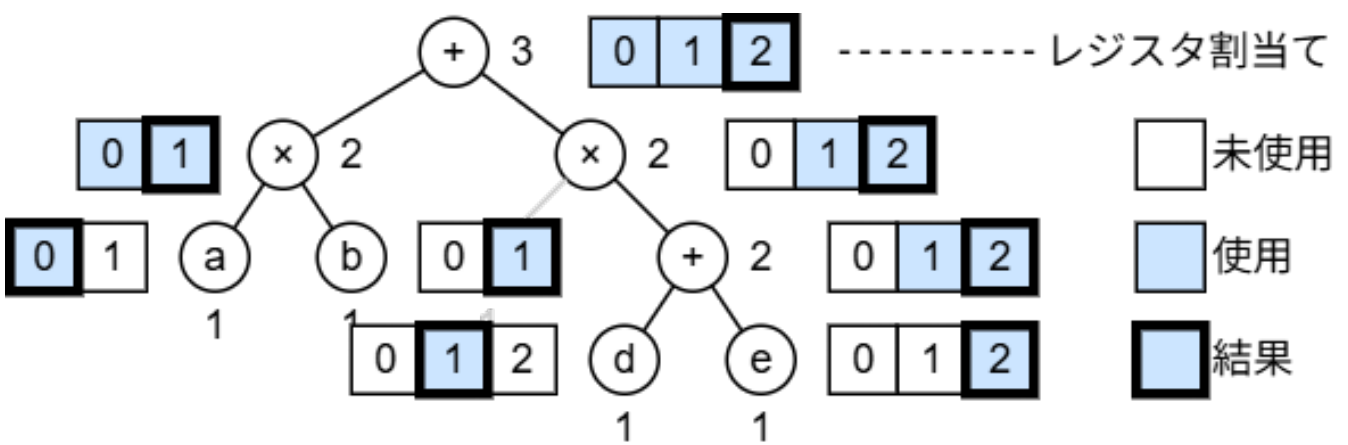
- ベース値 b=0 でルートから開始し、再帰的に実行
- ラベル k を持つノードのコード生成は k 個のレジスタを使う
  - ベース値 b から k 個分：R[b], R[b+1], …… R[b+k-1]
  - 評価結果は常に最後のレジスタ R[b+k-1] に格納される



2 つの子のラベルが等しい場合

評価対象ノードのラベルを k とすると、子のラベルは k-1 である

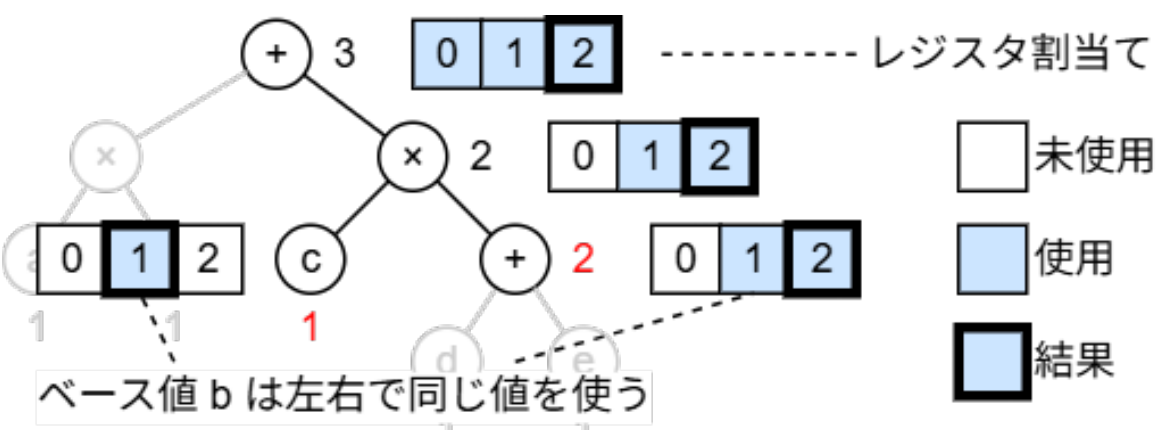
1. ベース値を b+1 として右辺のコードを再帰的に生成 → R[b+k-1]
2. ベース値を b として左辺のコードを再帰的に生成 → R[b+k-2]
3. OP R[b+k-1], R[b+k-2], R[b+k-1] を発行



2 つの子のラベルが異なる場合

評価対象ノードのラベルを k とすると、子のラベルは k と m<k

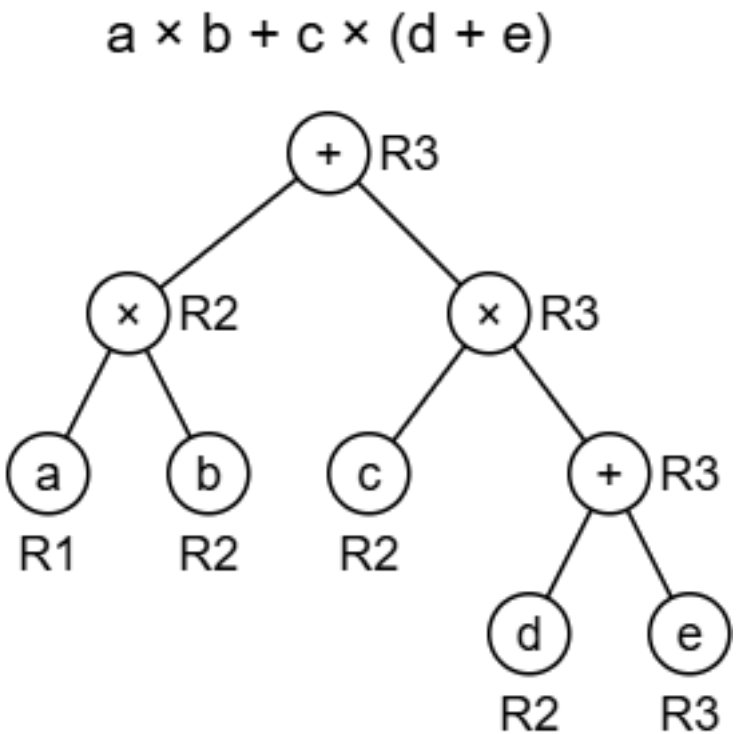
1. ベース値を b として大きな子のコードを再帰的に生成
2. ベース値を b として小さな子のコードを再帰的に生成
3. OP R[b+k-1], R[b+m-1], R[b+k-1]（左辺の方が小さい場合）あるいは
   OP R[b+k-1], R[b+k-1], R[b+m-1]（右辺の方が小さい場合）を発行



コード生成例

a*b + c*(d + e) に対するコード生成の例

```
LD R3, e
LD R2, d
ADD R3, R2, R3
LD R2, c
MUL R3, R2, R3
LD R2, b
LD R1, a
MUL R2, R1, R2
ADD R3, R2, R3
```



基本ブロック内でのレジスタ割当て

3 アドレス命令

x = y **op** z

- x, y, z は名前か、定数か、コンパイラが生成したテンポラリ変数。
- 右辺に高々 1 つの演算子しか現れない

特殊な場合

- x [y] = z （配列書き込み）
- x = y [z] （配列読み出し）
- ifFalse x goto L
- ifTrue x goto L
- goto L
- x = y

3 アドレスコード

3 アドレス命令の列

- ソースコードを変換して得る
- 代表的な中間表現の 1 つ

例：

```
t = a - b
u = t + c
a = d
```

- t, u はコンパイラが生成したテンポラリ変数

基本ブロック

3 アドレスコードを区分けしたもの

- 必ずブロック先頭の命令から実行が始まる（ブロックの途中に飛び込んでくる分岐命令が無い）
- 途中で停止や分岐せずにブロックの最後の命令に達する

イメージ：

```
基本ブロック 1
if（基本ブロック 2）{
   基本ブロック 3
}
基本ブロック 4
```

コード生成に用いるデータ構造

レジスタ記述子（RD） ぞれぞれのレジスタに 1 つずつ

- そのレジスタに乗っている変数の名前を追跡
- 通常は 1 つのレジスタには高々 1 つの変数が乗る
- 代入 x=y のとき 1 つのレジスタに x と y が乗る（後述）

アドレス記述子（AD） それぞれの変数に 1 つずつ

- その変数の最新値が書かれているすべての場所を追跡
- レジスタ、メモリ、スタック

関数 getReg(I)

- 3 アドレス命令 I に関連付くメモリ位置それぞれに対してレジスタを選ぶ関数
- getReg は基本ブロック内の全変数についてのレジスタ記述子とアドレス記述子を利用
- ブロック出口で生存している変数などのデータフロー情報を使っても良い

3 アドレス命令に対する機械語生成

1. getReg(x=y+z) が x, y, z のためのレジスタ Rx, Ry, Rz を選ぶ
2. y が RD[Ry] になければ LD Ry, y' を発行
   - y' は AD[y] に含まれるメモリ位置のいずれか
3. z についても同様に。
4. ADD Rx, Ry, Rz を発行

代入文 x = y に対する特別処理

1. getReg(x=y) が x と y に対して同じレジスタ Rx=Ry を返す
2. RD[Ry] が y を含んでいなければ LD Ry, y を発行
3. RD[Ry] に x を加える

記述子の更新例

以下のコード例に対し、各命令により RD と AD がどう変化するかを示す。

```
t = a - b
u = t + c
a = d
```

- t, u はブロックローカルのテンポラリ変数
- a, b, c, d はブロック出口で生存する変数

記述子の更新例

3 アドレス命令　RD　AD

TAI	機械語命令		R1	R2	R3		a	b	c	d	t	u
初期状態							a	b	c	d		
t = a - b	LD R1, a		a				a,R1	b	c	d		
	LD R2, b		a	b			a,R1	b,R2	c	d		
	SUB R2, R1, R2		a	t			a,R1	b	c	d	R2	
u = t + c	LD R3, c		a	t	c		a,R1	b	c,R3	d	R2	
	ADD R3, R2, R3		a	t	u		a,R1	b	c	d	R2	R3
a = d	LD R2, d		a	d,a	u		R2	b	c	d		R3

この時点で変数 a の最新値はレジスタにしかない

手法の比較

ソースコード

```
t = a-b+2*(a-c)
a = d
d = t
```

3 アドレスコード

```
_1 = a - b
_2 = a - c
_3 = 2 * _2
t = _1 + _3
a = d
d = t
```

RD & AD

```
LD R0, a
LD R1, b
SUB R2, R0, R1
LD R3, c
SUB R1, R0, R3
LI R0, 2
MUL R3, R0, R1
ADD R0, R2, R3
LD R1, d
ST a, R1
ST d, R0
```

Ershov 数

```
LD R3, c
LD R2, a
SUB R3, R2, R3
LI R2, 2
MUL R3, R2, R3
LD R2, b
LD R1, a
SUB R2, R1, R2
ADD R3, R2, R3
LD R1, d
ST a, R1
ST d, R3
```

BuntanPC

```
LD a
LD b
SUB
PUSH 2
LD a
LD c
SUB
MUL
ADD
LD d
ST a
ST d
```

変数を使う度にロードが必要

まとめ

- Ershov 数は、式木の葉のラベルを 1 とし、そこから再帰的に計算
- Ershov 数 k とベース値 b を使ってコードを生成
- レジスタ記述子はレジスタにどの変数が乗っているかを追跡
- アドレス記述子は変数の最新値がどこにあるかを追跡
- getReg(I) で選んだレジスタについて記述子を更新しながらコードを生成

おまけ（質問があれば見せる）

- getReg の設計

getReg の設計

getReg(x=y+z) で y に対応するレジスタ Ry の決定

1. y が何らかのレジスタに乗っているならそのレジスタ
   - この場合ロード命令は発行しない
2. y がレジスタに乗っておらず、空きレジスタがあるなら、それ
3. y がレジスタに乗っておらず、空きレジスタもない場合が難しい
   - とにかく何らかのレジスタ R を選ぶ
   - R を安全に再利用できるようにする

R を安全に再利用できるようにする方法

変数 v∈RD[R] とする

1. AD[v] を確認し、R 以外にもあるなら OK
   - v の最新値が R にしかない場合、保存が必要かも
2. v が x（I の左辺）であり、x が I の右辺に来ないなら OK
3. v が以降で利用されないなら OK
4. OK でない場合、ST v, R を発行（**spill**）

R は複数の変数を同時に表すことがあるので、全変数について上記を繰り返す。

R の「スコア」：発行が必要なストア命令の数

x に対するレジスタ Rx の決定

getReg(x=y+z) で x に対応するレジスタ Rx の決定は、y と大部分は同じ。差分は以下。

1. x だけを持つレジスタは常に Rx として利用可能
   - これは x=y+z の y や z が x でも成り立つ
   - なぜなら、仮定している機械語が Rdst = Rsrc を許すから
2. y が以降で使われず、Ry が y だけを持つなら Rx として使える
   - Rz についても同じ事が言える

代入命令 x=y の場合

- まず上記のアルゴリズムで Ry を選ぶ
- 常に Rx = Ry とする