5.1 能書き

本章では ML^4 言語から MIPS アセンブリへのコンパイラの設計について解説する．コンパイラは，**??**章で言及した通り，ソース言語のプログラムを同じ振る舞いをするターゲット言語のプログラムに変換するソフトウェアである．本章で設計するコンパイラでは，ソース言語が ML^4，ターゲット言語が MIPS アセンブリということになる．生成された MIPS アセンブリを世の中にあるアセンブラで実行可能バイナリにさらに変換することにより，ML^4 プログラムを MIPS アーキテクチャの計算機やシミュレータで動かすことが可能になる．[1]

一般的にコンパイラはソースプログラムを一度にターゲット言語に変換するのではなく，その間に幾つかの言語を挟んで，徐々にターゲット言語への変換を行う．間に挟まれるこれらの言語を**中間言語** (*intermediate language*) と呼ぶ．このような設計の利点は

- 徐々に中間言語の抽象度を下げることができ，各変換がわかりやすくなる．[2]
- 新しい言語を設計したときに，中間言語を再利用することができる．すなわち，中間言語 I を作り，I からアセンブリへの変換を作ってしまえば，将来別のプログラミング言語 L のコンパイラを作る際に，コンパイラ全体を実装する必要はなく，L から I への変換を実装するだけでよい．

等がある．

本章で設計するコンパイラでは，二つの中間言語を置く．一つ目は ML^4 プログラムで明示されていない式の評価順序等の情報を明示した関数型言語，もう一つは MIPS アセンブリにより近い命令形言語である．名前があったほうが教科書を書きやすいので，前者を言語 $\mathcal{C}$，後者を言語 $\mathcal{V}$ と呼ぶことにしよう．また，ターゲット言語である MIPS アセンブリを言語 $\mathcal{A}$ と呼ぶことにする．すると，本章で作るコンパイラの概略は図 **??**に示すとおりとなる．[3]

[流れ図を書く．] ←

5.2 ソース言語

中身の説明に入る前に，ソース言語を再確認しよう．ソース言語は**??**章で定義した ML^4 のうち，関数定義はトップレベルのみで行うことに制限した言語である．[4]構文は以下のように定義される．

---

[1] 本章では MIPS アセンブリの知識を（できるだけ）仮定せずに読めるように書いたつもりである．コンパイラの最後のフェーズではさすがに MIPS アセンブリの知識が必要になるんだけれども．

[2] ここで抽象度とは，言語機能のリッチさと思ってもらえればよい．高階関数やオブジェクト指向や○○指向やらがたくさん入った言語で書かれたプログラムを一気にアセンブリに落とすよりは，徐々にアセンブリに近づけていく方が変換が分かりやすいし実装も容易ということである．

[3] なお，本章で解説するコンパイラは東北大学の住井英二郎氏の「美しい日本の ML コンパイラ」(通称 MinCaml コンパイラ) [**?**] からつまみ食いをしたものになっている．MinCaml は OCaml さえ読めればとても分かりやすいミニコンパイラになっているので，できればそっちも読んでほしい．

[4] この制限は関数閉包を作らなくともプログラムを実行できるために講義の時間の都合上設けている制限である．実行時に関数閉包が作られうるようなプログラムをアセンブリ言語に落とすには，**クロージャ変換** (*closure conversion*) と呼ばれるプログラム変換を途中で行うなどして，関数がトップレベルで定義される形にプログラムを変換する必要がある．この変換を講義中で扱う時間がないので，ソース言語の方を制限しちゃうのであ

$$
\begin{array}{rcl}
P & ::= & (\{d_1, \ldots, d_n\}, e) \\
d & ::= & \mathsf{let\ rec}\ f = \mathsf{fun}\ x \to e \\
e & ::= & x \mid n \mid \mathsf{true} \mid \mathsf{false} \mid e_1\ op\ e_2 \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid e_1\ e_2 \\
op & ::= & + \mid * \mid <
\end{array}
$$

ここで，P，d, e，op はそれぞれプログラム，再帰関数定義，式，二項演算子を表す**メタ変数** $(metavariable)$ とする．また，x, y, f, g を変数を表すメタ変数とする．前の章でも述べたが，メタ変数とは，言語のある要素を表すものとしてあらかじめ定められた記号である．例えば上記の BNF では P は「プログラム」を表すメタ変数として，e は「式」を表すメタ変数として定めておくわけである．このようにすると，いちいち「e_1 と e_2 はそれぞれ式であり...」と断ることなく，単に式 $e_1 + e_2$ と書いただけである形の式を表すことができるわけである．

この制限された言語では，プログラム（P）は関数定義の集合（$\{d_1, \ldots, d_n\}$）とプログラム開始時に評価されるメインの式（e）とからなる．各関数定義は，相互再帰が可能であるものとする．また，各関数定義の中で定義されている関数の名前は他のどの場所でも現れない名前であるものとする．式の構文からは let rec 式と fun 式が除かれていることに注意されたい．

> **メタ変数と変数** 「メタ」とはギリシア語に語源を持つ接頭語である．プログラミング言語の文脈では「メタ○○」で「対象に言及するための○○」を表すことが多い．例えば「e は式を表すメタ変数である」とは「e は式に言及するための変数である」という意味である．
>
> プログラム中の変数を表すメタ変数 x と変数 x の違いに注意すること．let 式は let $x = e_1$ in e_2 の形をしているが，ここでの x はメタ変数であるから，let x = true in false も let y = true in false もこの形に当てはまることになる．もし e の定義が
>
> $$
> \begin{array}{rcl}
> \ldots & & \\
> e & ::= & \cdots \mid \mathsf{let\ x} = e_1\ \mathsf{in}\ e_2 \mid \ldots \\
> \ldots & &
> \end{array}
> $$
>
> と，let 式の x の部分がメタ変数でないプログラム変数の x と書いてあったとすると，後者の let 式は（定義しようとしている変数が x ではないので）式の構文には当てはまらないことになる．

5.3 言語 $\mathcal{C}$

これからソースプログラムをいくつかの中間言語を経由して MIPS アセンブリまで変換する．ソース言語と MIPS アセンブリの大きな差異の一つは，ソース言語においては，式を評

る．これではほとんど C 言語と変わらないのであるが，講義ができないとわしが怒られるので仕方ないのである．興味のある者は上記の住井コンパイラ [] を参照のこと．また，実験 4 にてこの制限のない ML^4 コンパイラの実装をする機会を設ける予定である．

価したときにどのような順序でどのような計算が起こるかが必ずしも明示されていない，すなわち式の評価でどのような**制御** $(control)$ が行われるかが明示されていないということがある．例えば，式 ((x + 1) * 2) + (3 + 1) を考えよう．この式を評価する際には

1. 式 x の値を取り出し，
2. その値に 1 を加え，
3. さらに 2 を加え，その値を覚えておき，
4. 3 + 1 を評価して 4 を得て，
5. 4 を覚えておいた値に加える

という計算が起こるはずである.[5] MIPS アセンブリでは，このような計算順序に関する情報（もう少し正確に言えば，各計算ステップで得られた値が次にどのような計算で用いられるのかに関する情報）を逐一指定しなければならない．したがって，ソース言語を MIPS アセンブリに変換する過程では，計算順序に関する情報を明示化する必要がある．元の式が仮に以下のように書いてあれば，計算順序が明示化されている感じがしないだろうか．

```
let t1 = x + 1 in
let t2 = t1 * 2 in
let t3 = 3 + 1 in
let t4 = t2 + t3 in
  t4
```

このプログラムでは，元のプログラム中のすべての部分式の評価結果に let で何らかの変数が束縛されており，かつ各部分式の評価結果がその後どのような計算でどのように用いられるかが明示されている．

言語 $\mathcal{C}$ は，すべての部分式の評価結果に何らかの変数が束縛されることを強制した関数型言語である．文法は以下の通りである．

$$
\begin{array}{rcl}
P & ::= & (\{d_1, \ldots, d_n\}, e) \\
d & ::= & \mathsf{let\ rec}\ f = \mathsf{fun}\ x \to e \\
v & ::= & x \mid n \mid \mathsf{true} \mid \mathsf{false} \\
e & ::= & x \mid n \mid \mathsf{true} \mid \mathsf{false} \mid v_1\ op\ v_2 \mid \mathsf{if}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid x_1\ x_2 \\
op & ::= & + \mid * \mid <
\end{array}
$$

メタ変数 v は変数，整数定数，真偽定数を表すメタ変数である．式 e の文法がソース言語のそれと少し変わっており，二項演算子の引数，条件式のガード部分，関数適用式の関数部分と引数部分には（式ではなく）変数か値しか取れないようになっている．これにより，先に

[5] この説明では $e_1 + e_2$ や $e_1 * e_2$ という式の評価が「e_1 が初めに評価され，e_2 が続いて評価され，最後にそれぞれの評価結果を用いて足し算や掛け算が行われる」と定義されていることを仮定している．式の評価をどのように定義するか（あるいは定義しないか）は言語による．

挙げた式 ((x + 1) * 2) + (3 + 1) のような，引数部分に変数でも値でもない式 3+1 を取ることはできないようになっており，すべての部分式に名前をつけ，評価順序を明示することを強制している．

5.4 ソース言語から$\mathcal{C}$への変換$\mathcal{I}$

ソース言語で書かれたプログラムを，そのプログラムと同等の振る舞いを持つ$\mathcal{C}$のプログラムに翻訳する変換$\mathcal{I}$を図 5.1 に示す．

関数$\mathcal{I}$は，式の構造に従って$\mathcal{I}$を再帰的に適用し，かつ各部分式について他とカブらない変数（すなわち，fresh な変数）を生成してその部分式に束縛している．それぞれのケースの右辺が言語$\mathcal{C}$の構文に添っていることを各自確認されたい．例えば，$\mathcal{I}$によって生成されたプログラムは，二項演算子 *op* の引数に必ず変数をとっている．

変換$\mathcal{I}$ではさらに式に現れるすべての束縛変数を fresh な変数名に置き換えることを行っている．これにより，異なる束縛変数が異なる名前を持つようになり，以降の変換の定義がシンプルになる．例えば，let x = 3 in let x = x + 1 in x は（それと等価な）let t1 = 3 in let t2 = t1 + 1 in t1 という式に変換される.[6]各束縛変数がどのような fresh な変数に付け替えられたかを記録しておくために，変換$\mathcal{I}$は写像δを持ち運ぶように定義してある．

[6]生成された fresh な変数が順に t1，t2 であると仮定した．

（以下の定義において，$x, x_1, x_2, t_{f_1}, \ldots, t_{f_n}, t_1$ は fresh な識別子である．また，δ は識別子から識別子への部分関数である．）

Definition of $\mathcal{I}_\delta(e)$

$$
\begin{array}{rcl}
\mathcal{I}_\delta(x) & = & \delta(x) \\
\mathcal{I}_\delta(n) & = & n \\
\mathcal{I}_\delta(\mathsf{true}) & = & \mathsf{true} \\
\mathcal{I}_\delta(\mathsf{false}) & = & \mathsf{false} \\
\mathcal{I}_\delta(e_1\ op\ e_2) & = & \mathsf{let}\ x_1 = \mathcal{I}_\delta(e_1)\ \mathsf{in} \\
 & & \mathsf{let}\ x_2 = \mathcal{I}_\delta(e_2)\ \mathsf{in} \\
 & & \quad x_1\ op\ x_2 \\
\mathcal{I}_\delta(\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & \mathsf{let}\ x = \mathcal{I}_\delta(e)\ \mathsf{in} \\
 & & \quad \mathsf{if}\ x\ \mathsf{then}\ \mathcal{I}_\delta(e_1)\ \mathsf{else}\ \mathcal{I}_\delta(e_2) \\
\mathcal{I}_\delta(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \mathsf{let}\ t_1 = \mathcal{I}_\delta(e_1)\ \mathsf{in}\ \mathcal{I}_{\delta[x \mapsto t_1]}(e_2) \\
\mathcal{I}_\delta(e_1\ e_2) & = & \mathsf{let}\ x_1 = \mathcal{I}_\delta(e_1)\ \mathsf{in} \\
 & & \mathsf{let}\ x_2 = \mathcal{I}_\delta(e_2)\ \mathsf{in} \\
 & & \quad x_1\ x_2
\end{array}
$$

Definition of $\mathcal{I}_\delta(d)$

$$
\mathcal{I}_\delta(\mathsf{let\ rec}\ f = \mathsf{fun}\ x \to e) = \mathsf{let\ rec}\ \delta(f) = \mathsf{fun}\ t_1 \to \mathcal{I}_{\delta[x \mapsto t_1]}(e)
$$

Definition of $\mathcal{I}(P)$

$$
\begin{array}{rcl}
\mathcal{I}((\{d_1, \ldots, d_n\}, e)) & = & (\{\mathcal{I}_\delta(d_1), \ldots, \mathcal{I}_\delta(d_n)\}, \mathcal{I}_\delta(e)) \\
 & \text{where} & \{f_1, \ldots, f_n\} = d_1, \ldots, d_n \text{ で定義されている関数名} \\
 & & \delta = \{f_1 \mapsto t_{f_1}, \ldots, f_n \mapsto t_{f_n}\}
\end{array}
$$

図 5.1: ソース言語から $\mathcal{C}$ への変換関数 $\mathcal{I}$.

5.5 仮想マシンコードの生成

$\mathcal{C}$ から直接にアセンブリを生成することもできるのだが，アセンブリ言語は書かれている命令を順番に実行していく**命令形言語** (*imperative language*) なので，関数型言語である $\mathcal{C}$ とはギャップがまだ大きい．そこで，$\mathcal{C}$ とアセンブリ言語の間に仮想マシン言語 $\mathcal{V}$[1] という中間言語を挟むことにする．[2]

$\mathcal{V}$ の定義を示す前に，$\mathcal{V}$ がどんな感じの言語かを見てみよう．以下のプログラムは，言語 $\mathcal{V}$ で書かれた，3 に 1 を加えるプログラムである．

$$
\begin{array}{ll}
l_f: & \\
 & \quad \mathbf{local}(4) \leftarrow \mathbf{param}(1) \\
 & \quad \mathbf{local}(0) \leftarrow \mathit{add}(\mathbf{local}(4), \mathbf{imm}(1)) \\
 & \quad \mathbf{returnlocal}(0) \\
l_{main}: & \\
 & \quad \mathbf{local}(0) \leftarrow \mathbf{call}\ \mathbf{labimm}(l_f)(\mathbf{imm}(3))
\end{array}
$$

プログラムは命令の列である．プログラム中の l_f や l_{main} は**ラベル** (*label*) と呼ばれる識別子で，プログラム中の位置を表している．ラベルは処理のジャンプ先を指定する際に用いられる．例えば，プログラム中の $\ldots \leftarrow \mathbf{call}\ l_f(\ldots)$ 命令は関数呼び出しをするために l_f に処理を移す命令である．

このプログラム中の各命令の動作を順番に見てみよう．ラベル l_f から始まる部分にかかれている命令は以下のとおりである．

$\mathbf{local}(4) \leftarrow \mathbf{param}(1)$: ラベル l_f から始まる関数の第一引数の内容を，$\mathbf{local}(4)$ で指される記憶領域に格納する．アセンブリ言語において関数呼び出しを実装するには，呼び出された関数のローカルな記憶領域（この記憶領域のことを**フレーム** (*frame*) と呼ぶ）をどのように確保するか，その記憶領域をどのように使うか，引数や返り値をどのように受け渡しするかを決定する必要がある．これらの決まりごとを**呼び出し規約** (*calling convention*) という．言語 $\mathcal{V}$ においては，関数に渡された引数は $\mathbf{param}(1), \mathbf{param}(2), \ldots$ で参照し，ローカルな変数の格納先は $\mathbf{local}(0), \mathbf{local}(4), \ldots$ のようにローカルな記憶領域内部の場所を表す名前を付けておくことにより，あとでアセンブリ生成を行う際に呼び出し規約を完全に決められるようにしてある．

$\mathbf{local}(0) \leftarrow \mathit{add}(\mathbf{local}(4), \mathbf{imm}(1))$: フレーム中で $\mathbf{local}(4)$ という名前で指される領域に格納されている値（すなわち前の命令でセットされた関数の第一引数）と整数値 1 とを加算して $\mathbf{local}(0)$ に格納する．$\mathbf{imm}(n)$ は整数定数 n を表すオペランドで，imm というオペランド名はアセンブリ言語で命令語中に直接現れる定数を表す**即値** (*immediate*) に由来する．

---

[1] 「仮想マシン言語」という名前は，命令形の中間言語の名前として本書で便宜的に使っている名前である．このような中間言語に相当する言語は多くのコンパイラやコンパイラの教科書で用いられているが，その名前は様々である．

[2] ソース言語とターゲット言語の間にどのような中間言語を挟むかはコンパイラを作る上で重要なデザインチョイスである．本書では $\mathcal{C}$ と $\mathcal{V}$ を中間言語として挟むが，より多くの中間言語を挟むコンパイラもある．

return : $\mathbf{local}(0)$ に格納されている値を関数の返り値として返す.

l_{main} はプログラムが起動されたときに実行が始まるプログラム中の箇所を指すラベルであり，命令 $\mathbf{local}(0) \leftarrow \mathbf{call}\ \mathbf{labimm}(l_f)(\mathbf{imm}(3))$ が書いてある．この命令は l_f から始まる命令列を関数と思って引数 $\mathbf{imm}(3)$ で呼び出し，返り値を記憶領域 $\mathbf{local}(0)$ に格納する．

$\mathcal{V}$ は以下の BNF で定義される言語である.

$$
\begin{array}{rcl}
op & ::= & \mathbf{param}(n) \mid \mathbf{local}(ofs) \mid \mathbf{labimm}(l) \mid \mathbf{imm}(n) \\
i & ::= & \mathbf{local}(ofs) \leftarrow op \mid \mathbf{local}(ofs) \leftarrow op(op_1, op_2) \mid l: \mid \mathbf{if}\ op\ \mathbf{then\ goto}\ l \\
 & \mid & \mathbf{goto}\ l \mid \mathbf{local}(ofs) \leftarrow \mathbf{call}\ op_0(op_1, \ldots, op_n) \mid \mathbf{return}(op) \\
d & ::= & \langle l \,\|\, i_1 \ldots i_m \,\|\, n \rangle \\
P & ::= & \langle d_1 \ldots d_m \,\|\, i_1 \ldots i_n \,\|\, k \rangle
\end{array}
$$

l はラベル名を表すメタ変数，ofs は整数値である．プログラムは**命令** (*instruction*) の列である．各命令は，命令の種類と命令の引数（**オペランド** (*operand*)）によってどのように動作するかが決まる．言語 $\mathcal{V}$ のオペランドは値の記憶領域か定数値を表す情報で，具体的には以下のいずれかである.

$\mathbf{param}(n)$: 関数に渡された n 番目の引数の格納場所を表す.

$\mathbf{local}(ofs)$: 現在のフレームのうち,「基準となるアドレス」から ofs バイト目のアドレスを表す.[3]

$\mathbf{imm}(n)$: 整数定数 n を表す.

$\mathbf{labimm}(l)$: ラベル名 l を表す定数を表す．関数呼び出しを行う際に使用する.

では，各命令の意味を説明しよう．以下の説明で「op の値」という表現を用いることがある．これは，op が $\mathbf{param}(n)$ であれば n 番目の引数として渡された値を，$\mathbf{local}(ofs)$ であればフレーム中の場所 ofs に格納されている値を，$\mathbf{imm}(n)$ であれば整数値 n を，それぞれ表す.

$\mathbf{local}(ofs) \leftarrow op$: op の値をフレーム中の $\mathbf{local}(ofs)$ の指す記憶領域に格納する.

$\mathbf{local}(ofs) \leftarrow op(op_1, op_2)$: op_1 と op_2 の値を op で計算して，フレーム中の $\mathbf{local}(ofs)$ の場所に格納する.

$l:$: プログラム中のラベル名 l で指される場所を表す.

if op **then goto** l : op の値が 0 でなければ l に制御を移す．そうでなければ何もしない.

goto l : l に制御を移す.

[3]後述のフレームの内部構造のところでもう少し詳しく説明する.

$\mathbf{local}(ofs) \leftarrow \mathbf{call}\ op(op_1, \ldots, op_n)$: $op_1, \ldots, op_n$ の値を引数として op に格納されているラベルから始まる命令列を関数として呼び出す．関数が返ったら，返り値を $\mathbf{local}(ofs)$ に格納する．

$\mathbf{return}(op)$: op に格納されている値を現在実行中の関数の返り値として返す．

関数定義 $\langle l \,\|\, i_1 \ldots i_m \,\|\, n \rangle$ は，関数のラベル名と，その関数本体の命令列と，関数内で使われるローカル変数に必要な記憶領域のサイズ n からなる．この記憶領域サイズは，後のコード生成フェーズで使用される．プログラムは $\langle d_1 \ldots d_m \,\|\, i_1 \ldots i_n \,\|\, k \rangle$ の形をしており，関数定義の列 $d_1 \ldots d_m$ と，メインのプログラムに対応する命令列 $i_1 \ldots i_n$ と，メインのプログラム内で使われるローカル変数のための記憶領域のサイズ k からなる．

$\mathcal{C}$ から $\mathcal{V}$ への変換を図 5.1 に示す．**変換の定義を簡潔に保つために，変換対象の $\mathcal{C}$ プログラムではすべての束縛変数が一意な名前にあらかじめ変換されているものとする．**例えば，$\mathsf{let}\ x = 1\ \mathsf{in}\ \mathsf{let}\ x = 2\ \mathsf{in}\ x$ というプログラムは $\mathsf{let}\ x_1 = 1\ \mathsf{in}\ \mathsf{let}\ x_2 = 2\ \mathsf{in}\ x_2$ というプログラムにあらかじめ変換がなされているものとする．実際に，先に示した変換 $\mathcal{I}$ はすべての束縛変数が一意な名前を持つように変換を行っている．

式 e の変換 $\mathcal{VT}_{\delta,tgt}(e)$ は e の他に変数からオペランドへの部分関数 δ とオペランド tgt を引数として取り，「変数の記憶領域が δ に書いてあると仮定して e を評価した結果を tgt に格納する」仮想マシンコードを生成する．各ケースの説明は以下の通りである．

$\mathcal{VT}_{\delta,tgt}(x)$: x の評価結果を tgt に格納するコードを生成する必要がある．x が格納されている場所は $\delta(x)$ なので，$tgt \leftarrow \delta(x)$ を生成すればよい．

$\mathcal{VT}_{\delta,tgt}(n)$: n の評価結果を tgt に格納するコードを生成するので，$tgt \leftarrow \mathbf{imm}(n)$ を生成すればよい．

$\mathcal{VT}_{\delta,tgt}(\mathsf{true}), \mathcal{VT}_{\delta,tgt}(\mathsf{false})$: 考え方は $\mathcal{VT}_{\delta,tgt}(n)$ と全く同じだが，true は整数定数 1 で，false は整数定数 0 でエンコードしていることに注意．

$\mathcal{VT}_{\delta,tgt}(x_1\ op\ x_2)$: x_1, x_2 を格納している場所はそれぞれ $\delta(x_1), \delta(x_2)$ なので，これらを op で計算して $\mathbf{local}(tgt)$ に格納するコード $tgt \leftarrow op(\delta(x_1), \delta(x_2))$ を生成している．

$\mathcal{VT}_{\delta,tgt}(\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)$: $\mathbf{if}\ \delta(x)\ \mathbf{then\ goto}\ l_1$ 命令で x の値が格納されている $\delta(x)$ に非ゼロの値が入っていれば（すなわち x が true であれば）l_1 にジャンプする．もしここで値がゼロであれば（すなわち x が false であれば）その後ろがそのまま実行されるので，e_2 を評価するコード $\mathcal{VT}_{\delta,tgt}(e_2)$ を書いておき，その後ラベル l_1 のコードを飛び越せるように $\mathbf{goto}\ l_2$ を書いておく．ラベル l_1 以降には $\mathcal{VT}_{\delta,tgt}(e_1)$ で e_1 を評価するコードが書いてある．

$\mathcal{VT}_{\delta,tgt}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$:まず初めに e_1 を評価して $\delta(x)$ に格納するコード $\mathcal{VT}_{\delta,\delta(x)}(e_1)$ を置く．その後，e_2 の評価結果を tgt に格納するコード $\mathcal{VT}_{\delta,tgt}(e_2)$ を置く．

Definition of $\mathcal{VT}_{\delta,tgt}(e)$ （δ は識別子からオペランドへの部分関数，tgt は $\mathbf{local}(n)$ の形をしたローカル領域のアドレスである．また，以下の定義中 l_1 と l_2 は fresh なラベル名である．）

$$
\begin{array}{l}
\mathcal{VT}_{\delta,tgt}(x) = tgt \leftarrow \delta(x) \\
\mathcal{VT}_{\delta,tgt}(n) = tgt \leftarrow \mathbf{imm}(n) \\
\mathcal{VT}_{\delta,tgt}(\mathsf{true}) = tgt \leftarrow \mathbf{imm}(1) \\
\mathcal{VT}_{\delta,tgt}(\mathsf{false}) = tgt \leftarrow \mathbf{imm}(0) \\
\mathcal{VT}_{\delta,tgt}(x_1\ op\ x_2) = tgt \leftarrow op(\delta(x_1), \delta(x_2)) \\
\mathcal{VT}_{\delta,tgt}(\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) = \begin{array}{l} \mathbf{if}\ \delta(x)\ \mathbf{then\ goto}\ l_1 \\ \quad \mathcal{VT}_{\delta,tgt}(e_2) \\ \quad \mathbf{goto}\ l_2 \\ l_1: \\ \quad \mathcal{VT}_{\delta,tgt}(e_1) \\ l_2: \end{array} \\
\mathcal{VT}_{\delta,tgt}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) = \begin{array}{l} \mathcal{VT}_{\delta,\delta(x)}(e_1) \\ \mathcal{VT}_{\delta,tgt}(e_2) \end{array} \\
\mathcal{VT}_{\delta,tgt}(x_1\ x_2) = tgt \leftarrow \mathbf{call}\ \delta(x_1)(\delta(x_2))
\end{array}
$$

Definition of $\mathcal{VT}_{\delta}(d)$

$$
\begin{array}{rcl}
\mathcal{VT}_{\delta}(\mathsf{let\ rec}\ f\ = \mathsf{fun}\ x \to e) & = & \left(l \parallel \begin{array}{l} \mathcal{VT}_{\delta\cup\delta_1\cup\delta_2,\mathbf{local}(0)}(e) \\ \mathbf{return}(\mathbf{local}(0)) \end{array} \parallel 4n+4 \right) \\
\text{where} \quad \delta_1 & = & \{x \mapsto \mathbf{param}(1)\} \\
\{x_1, \ldots, x_n\} & = & e \text{ 中に出現する変数の集合} \\
\delta_2 & = & \{x_1 \mapsto \mathbf{local}(4), x_2 \mapsto \mathbf{local}(8), \ldots, x_n \mapsto \mathbf{local}(4n)\} \\
\mathbf{labimm}(l) & = & \delta(f)
\end{array}
$$

Definition of $\mathcal{VT}(P)$

$$
\begin{array}{rcl}
\mathcal{VT}((\{d_1, \ldots, d_n\}, e)) & = & \left(\begin{array}{l} \mathcal{VT}_{\delta}(d_1) \\ \ldots \\ \mathcal{VT}_{\delta}(d_n) \end{array} \parallel \begin{array}{l} \mathcal{VT}_{\delta\cup\delta',\mathbf{local}(0)}(e) \\ \mathbf{return}(\mathbf{local}(0)) \end{array} \parallel 4m+4 \right) \\
\text{where} & & \{f_1, \ldots, f_n\} = d_1, \ldots, d_n \text{ で定義されている関数名の集合} \\
& & \{x_1, \ldots, x_m\} = e \text{ 中の変数の集合} \\
& & \delta = \{f_1 \mapsto \mathbf{labimm}(f_1), \ldots, f_n \mapsto \mathbf{labimm}(f_n)\} \\
& & \delta' = \{x_1 \mapsto \mathbf{local}(4), x_2 \mapsto \mathbf{local}(8), \ldots, x_m \mapsto \mathbf{local}(4m)\}
\end{array}
$$

図 5.1: $\mathcal{C}$ から $\mathcal{V}$ への変換 $\mathcal{VT}$.

$\mathcal{VT}_{\delta,tgt}(x_1\ x_2)$ ：関数呼び出しを行い，その返り値を tgt に格納するコード $tgt \leftarrow \mathbf{call}\ \delta(x_1)(\delta(x_2))$ を生成する．ジャンプ先のラベルは $\delta(x_1)$ に格納されている．また，$\delta(x_2)$ に引数が格納されている．

関数定義 d の仮想マシンコード生成を行う変換 $\mathcal{VT}_\delta(d)$ は，d 以外に δ を引数にとる．δ はトップレベルで定義されている関数名を受け取って，それを対応するコードが書かれているラベルオペランド $\mathbf{labimm}(l)$ に写像する．$\mathcal{VT}_\delta(\mathsf{let\ rec}\ f\ =\mathsf{fun}\ x \to e)$ は，その後関数本体 e を評価するコード $\mathcal{VT}_{\delta\cup\delta_1\cup\delta_2,\mathbf{local}(0)}(e)$ を生成する．$\delta\cup\delta_1\cup\delta_2$ は δ を以下の二つの写像で拡張したものである．

δ_1 ：仮引数名 x から $\mathbf{param}(1)$ への写像．

δ_2 ：e 中に現れるすべての変数からそれぞれ固有の記憶領域 $\mathbf{local}(i)$ への写像．[4] ここでは，すべての値が 4 バイトで表現できるものとして，各変数に 4 バイトの記憶領域を割り当て，δ_2 を $\{x_1 \mapsto \mathbf{local}(4), x_2 \mapsto \mathbf{local}(8), \ldots, x_n \mapsto \mathbf{local}(4n)\}$ としている．

末尾に e の評価結果（$\mathbf{local}(0)$ に格納されている）を $\mathbf{return}(\mathbf{local}(0))$ で返す．この関数で必要とされるローカルな記憶領域のサイズは $4n$ である．

プログラム $(\{d_1, \ldots, d_n\}, e)$ の変換においては，まず各 d_i の変換結果 $\mathcal{VT}_\delta(d_i)$ を生成する．δ は各 d_i で定義されている関数名 f_i からラベル名 $\mathbf{labimm}(f_i)$ への写像である．その後メインの式である e を評価するコードを生成すればよい．このコードの先頭にはラベル l_{main} : を生成している．e を評価する際に，e 中の変数のための記憶領域を割り当てる必要があるが，これは上記の関数定義の仮想マシンコード生成と同じ考え方である．

[4] 変換 $\mathcal{I}$ においてすべての束縛変数の名前を一意になるように付け替えたのがここで地味に効いている．

5.6 アセンブリ生成

この節では，前の節の説明に従って生成した仮想マシンコードを入力とし，現実の計算機アーキテクチャの一つである MIPS のアセンブリコードを生成する方法について説明する．説明にあたり，MIPS に関する基本的な知識は仮定する．たとえば：

```
li $v0, 1
li $v1, 2
add $v0, $v0, $v1
li $v1, 3
mul $v0, $v0, $v1
```

は，$(1+2)\times 3$ を計算し，その結果を v0 レジスタに格納する命令列であることが理解でき，また：

```
    .data
ARR:
    .word 1, 2, 3, 4, 5
    .text
    la $v2, ARR
    lw $v0, 1($v2)
    mul $v1, $v0, $v0
    sw $v1, 3($v2)
```

は，メモリ中の ARR ラベルの付けられた番地から順に並んでいる長さ 5 の 32 ビット整数配列の中から，2 番目の値を取り出し，その値の二乗を同じ配列の 4 番目に上書きする，ということが理解できるものとする．もしこれらを理解できないようであれば，[?] を読みなおして復習すること．

5.6.1 MIPS の呼出し規約

現実のハードウェアが備えている物理レジスタの個数は有限である．そのため，関数（とくに再帰関数）を用いるプログラムではプログラム中のすべての計算を物理レジスタだけで実行することは困難であり，一般に，何らかの方法に則ってプログラム中の各変数の値を格納する場所をメモリ中のどこかへ確保する必要がある．

たとえば，次のような ML[4] のプログラム[1]：

```
let rec f a = a + 1
and g b = f b
in g 0
```

[1]読みやすさのため，ML[4] のシンタックスではなく，同等の言語機能による OCaml のシンタックスを使って説明する．

を素朴に実行するだけであれば，プログラム全体で,「関数 f の引数 a を格納する場所」「関数 f を呼び出した結果（返り値）を格納する場所」「関数 g の引数 b を格納する場所」「関数 g の返り値を格納する場所」の計 4 ワード分を確保すれば十分である．

しかし，そのような素朴な方法だと，たとえば：

```
let rec fact = fun n -¿ if n > 0 then n * fact (n + (-1)) else 1
in fact 10
```

のような再帰関数 fact においては，複数の異なる (再帰) 呼出しの引数や返り値の格納場所が同じ領域を使うことになり，実行がおかしくなってしまう．具体的には，関数呼出し fact 9 の直後には，その返り値に対し 10 を掛ける必要があるが，関数呼出し fact 10 の引数である値 10 は，fact の引数を格納するただ一つの場所に置かれていたため，すでに値 9 で上書きされ失われてしまっている（正確には，さらに $8, 7, \ldots, 0$ で上書きされているが，ともかく 10 という値はもはや存在しない）．

結局のところ，関数定義単位で必要な領域のサイズを求め，その総和分を確保するだけでは不十分であり，関数呼出し毎に異なる場所を確保する必要があることが分かる．そこで，関数機能を持つプログラミング言語（関数型言語に限らず，たとえば C 言語なども含む）の処理系では，通常，呼出し/リターンの制御構造に対応できるよう，各関数呼出しの実行に必要となるメモリ領域を管理するためのデータ構造として，LIFO (last-in first-out) で管理するスタックを用いる．また，このスタックは，呼出し側が関数呼出しの次に実行するべき命令のアドレス（**リターンアドレス** (*return address*) と呼ばれる）を保存しておくためにも用いられる（詳細は後述）．

原理的には，ここまでに説明したように各関数の実行で必要となるすべての引数・返り値・局所変数（ならびにリターンアドレス）のための場所をスタック上に必ず確保することにすれば，再帰関数を含むようなプログラムであっても正しく実行できるが，それだけだと，プログラム実行中に計算されるすべての値をメモリ領域にいちいち格納するため，レジスタを有効活用できておらず実行性能はあまり良くない．たとえば，返り値を格納する場所をスタック上のある場所に定めてしまうと,（とくにレジスタで計算を行う RISC 系のハードウェアにおいては）呼び出された側が最後にスタックのその場所に返り値をストアしたすぐ後に，呼出し側が同じ場所からレジスタへロードするということが頻繁に起こる．この実行オーバヘッドは，たとえば「関数呼出しの返り値は必ず v0 レジスタを使って受け渡しする」と決めておけば避けることができる．まったく同様に，引数の受け渡しについてもレジスタの使い方に関し何らかの約束事を決めておけば，実行効率を良くすることができる．

以上のようなスタックとレジスタの使い方に関する約束事は，レジスタの種類や個数，関数の呼出し/リターンに用いることのできるジャンプ命令の詳細な振舞いなどに依存するため，通常，計算機アーキテクチャ毎に定められている．関数呼出しに関するそのような約束事は，一般に**呼出し規約** (*calling convention*) と呼ばれる．

なお，本講義で作成する ML[4] コンパイラは MIPS アセンブリをターゲットとしているため，本来であれば MIPS の呼出し規約に厳密に従うべきところだが，説明を簡単にするため，少し簡略化した独自の呼出し規約を用いている．より本格的なプログラミング言語や言語処

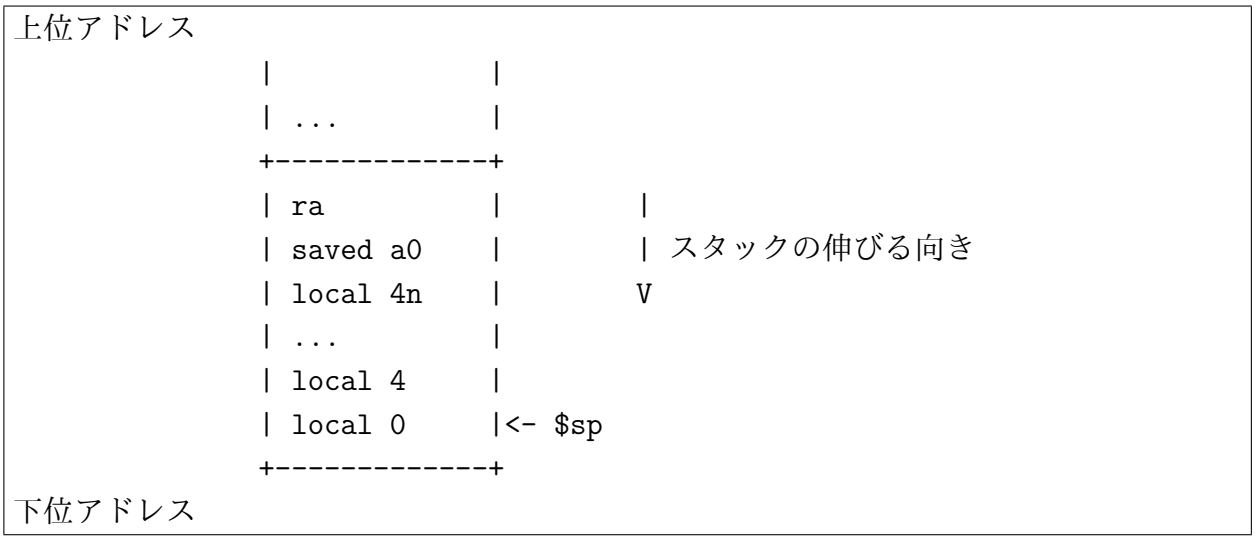


図 5.1: フレームの中身.

理系との互換性を気にする場合には，この資料に書かれている説明を理解した後に各自でさらに勉強して欲しい（「MIPS calling convention」などと検索すれば，たくさんの情報を得られる）.

ML^4 コンパイラにおける呼出し規約　まず，関数の返り値については先程触れたように，v0 レジスタを介して受け渡すことにする．また，ML^4 言語におけるすべての関数は引数を 1 つしか受け取らないため，その唯一の引数は a0 レジスタを介して受け渡すことにする.

各関数を実行するために必要なスタック上の連続領域（**フレーム** (*frame*) と呼ぶ）は，図 5.1 に示すように，スタックの一番上（メモリアドレスは下位の方）に確保される.

フレーム中に含まれる各データの説明は以下のとおりである.

リターンアドレス（ra）: このフレームを使っている関数を呼び出した関数が，その関数呼出しの次に実行する命令のアドレスを退避するための場所．さらに別の関数を呼ぶことがなければ必ずしも退避する必要はないが，簡単化のため，必ずフレームの一番上に退避することにしている.

退避された a0 レジスタ（saved a0）: このフレームを使っている関数が実引数を a0 レジスタにセットして他の関数を呼び出す際，すでに a0 レジスタに入っている自分自身のパラメータを退避するための場所．関数呼出し以降にパラメータを使わないのであれば必ずしも退避する必要はない（が，次節のコード生成では，簡単化のため必ず退避することにしている）.

局所変数（local 0〜4n）: 関数本体で let により束縛する局所変数の値を格納するための場所．$\mathcal{VT}$ で $\mathbf{local}(0)$ を関数からの返り値を格納する場所として使用したので，それに則って，局所変数の個数が N 個ならば局所変数を格納するための領域が $4N + 4$ バイトになっている.

なお，sp レジスタ（スタックポインタ）は，常にスタック最上部（フレームの一番下）を指すようにする．したがって，i 番目 $(1 \leq i \leq N)$ の局所変数が格納されている領域はアドレス$sp+$4i となる．また，saved a0 のアドレスは$sp+$4$N+4$，リターンアドレスのアドレスは$sp+$4$N+8$ となる．

以下は，上記のスタックレイアウトに基づいて関数呼出しを行う手順の概要である（詳細な手順については次節を参照）．

1. 呼出し側は，a0 レジスタの値を（自身の）saved a0 に退避してから，関数呼出しの実引数を a0 レジスタにセットする．
2. jal あるいは jalr 命令によって呼び出される関数の先頭へジャンプする．呼出し側の次の命令のアドレスは ra レジスタにセットされる．
3. （以下，呼び出された関数側で実行）sp レジスタを必要なだけ下げる．
4. ra レジスタに入っているリターンアドレスを，スタックの所定の位置に退避する．
5. 関数本体を実行し，求まった返り値を v0 レジスタにセットする．
6. スタック中のリターンアドレスをレジスタに戻す．
7. 3 で下げたのと同じ分だけ sp レジスタを上げることで，スタックからフレームを取り除く．
8. jr 命令によって 6 で取り出したリターンアドレスへリターンする．
9. （呼出し側で実行）v0 レジスタから返り値を取り出し，さらに，退避しておいた saved a0 を a0 レジスタに戻す．

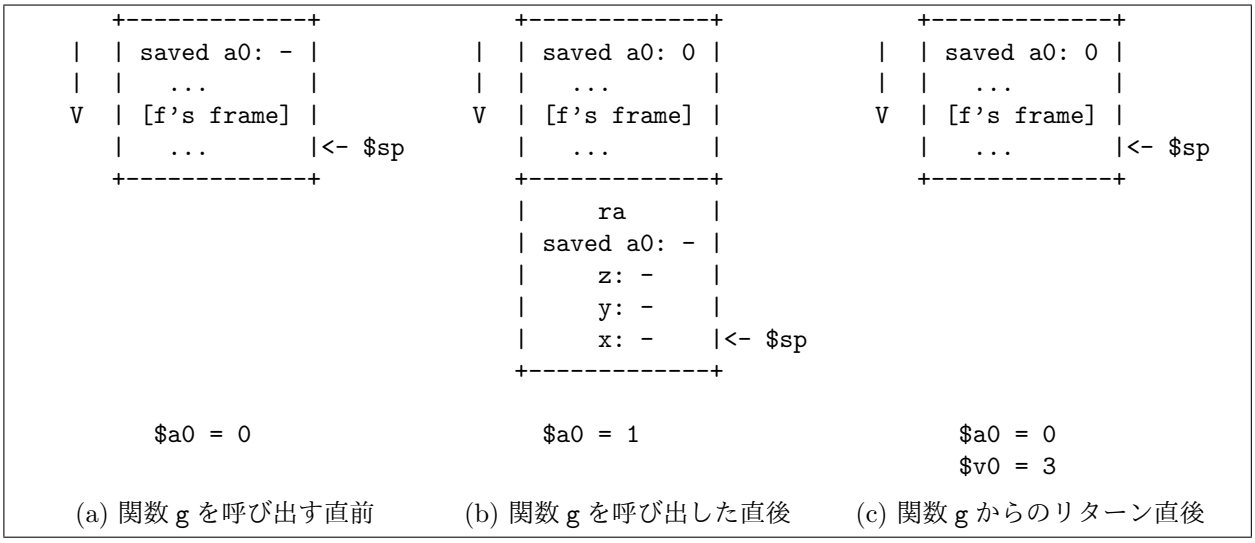
最後に，以下の簡単な OCaml コード：

```
let rec f a = g (a+1)
and g b = let x = b + b in
          let y = x * x in
          let z = y - 1 in
            z
in f 0
```

の実行中，関数 f から関数 g を呼び出す前後のスタックの状態を図 5.2 に示す．

現実の呼出し規約に関する余談

一般のプログラミング言語では，多引数関数を定義できたり，引数の数が固定ではない（可変長引数と呼ばれる）関数を定義できたりする．また，関数本体の実行中に使用するメモリサイズを正確に求めることが難しいこともある（たとえ



(a) 関数 g を呼び出す直前　(b) 関数 g を呼び出した直後　(c) 関数 g からのリターン直後

図 5.2: ML^4 のスタックレイアウト

ば，スタック上に任意サイズのメモリ領域を確保できる C 言語の `alloca` 関数を使用した場合など）．そのような場合，物理レジスタに収まりきらない引数をスタックに置く必要がある．それに加え，実行中に位置が一定ではない `sp` レジスタからの相対アドレスを用いてスタック中のパラメータおよび局所変数へアクセスしようとすると，コンパイル時に求める必要のある相対アドレスの計算が複雑になる．

そこで，現実のコンパイラにおいては，`sp` レジスタとは別に，関数フレームの `sp` とは反対側（典型的には `ra` の入っているあたり）を常に指し続ける `fp` レジスタを別に用意しておき，パラメータや局所変数へのアクセスには `fp` レジスタからの相対アドレスを用いるのが一般的である．ただし，リターンアドレスと同様に，呼出し側の `fp` レジスタの値もスタック中に退避する手間がさらに必要となる．

5.6.2 アセンブリ生成

以上を踏まえて，$\mathcal{V}$ のプログラムを MIPS アセンブリに変換する関数 Asm の定義を図 5.3 に示す．なお，以下の説明では MIPS アセンブリ中の命令について逐一説明はしないので，必要であればリファレンス [] を参照されたい．

オペランドの変換は $Asm_r(op)$ で行う．この変換では「op に格納されている値をレジスタ r にロードする」MIPS アセンブリが生成される．各ケースの説明は以下の通りである．

$Asm_r(\mathbf{param}(1))$: $\mathbf{param}(1)$（関数の第一引数）をレジスタ r にロードする．現在の $\mathcal{V}$ では一引数関数のみが定義できるため $\mathbf{param}(1)$ についてのみ定義されている．関数の引数は関数呼び出し規約からレジスタ `$a0` に格納されているので，この内容を r にロードするために `move` 命令を用いている．

（$[\![op]\!]$ は $\mathcal{V}$ の op に対応する MIPS の命令，$[\![l]\!]$ はラベル名 l を MIPS アセンブリ内のラベルとして解釈できるようにした表現である．ただし，$[\![l_{main}]\!] = \texttt{main}$ とする．）

Definition of $Asm_r(op)$

$$
\begin{array}{rcl}
Asm_r(\mathbf{param}(1)) & = & \texttt{move}\ \ r, \$\texttt{a0} \\
Asm_r(\mathbf{local}(n)) & = & \texttt{lw}\ \ r, n(\$\texttt{sp}) \\
Asm_r(\mathbf{labimm}(l)) & = & \texttt{la}\ \ r, [\![l]\!] \\
Asm_r(\mathbf{imm}(n)) & = & \texttt{li}\ \ r, n
\end{array}
$$

Definition of $Asm_n(i)$

$$
\begin{array}{rcl}
Asm_n(\mathbf{local}(ofs) \leftarrow op) & = & Asm_{\$\texttt{t0}}(op) \\
 & & \texttt{sw}\ \ \$\texttt{t0}, ofs(\$\texttt{sp}) \\
Asm_n(\mathbf{local}(ofs) \leftarrow op(op_1, op_2)) & = & Asm_{\$\texttt{t0}}(op_1) \\
 & & Asm_{\$\texttt{t1}}(op_2) \\
 & & [\![op]\!]\ \ \$\texttt{t0}, \$\texttt{t0}, \$\texttt{t1} \\
 & & \texttt{sw}\ \ \$\texttt{t0}, ofs(\$\texttt{sp}) \\
Asm_n(l :)) & = & [\![l]\!]: \\
Asm_n(\mathbf{if}\ op\ \mathbf{then\ goto}\ l) & = & Asm_{\$\texttt{t0}}(op) \\
 & & \texttt{bgtz}\ \ \$\texttt{t0}, [\![l]\!] \\
Asm_n(\mathbf{goto}\ l) & = & \texttt{j}\ \ [\![l]\!] \\
Asm_n(\mathbf{local}(ofs) \leftarrow \mathbf{call}\ op_f(op_1)) & = & Save_{n+4}(\$\texttt{a0}) \\
 & & Asm_{\$\texttt{a0}}(op_1) \\
 & & Asm_{\$\texttt{t0}}(op_f) \\
 & & \texttt{jalr}\ \ \$\texttt{ra}, \$\texttt{t0} \\
 & & \texttt{sw}\ \ \$\texttt{v0}, ofs(\$\texttt{sp}) \\
 & & Restore_{n+4}(\$\texttt{a0}) \\
Asm_n(\mathbf{return}(op)) & = & Asm_{\$\texttt{v0}}(op) \\
 & & Epilogue(n) \\
 & & \texttt{jr}\ \ \$\texttt{ra}
\end{array}
$$

Definition of $Asm(d)$

$$
\begin{array}{rcl}
Asm((l \mid i_1 \ldots i_m \mid n)) & = & [\![l]\!]: \\
 & & Prologue(n) \\
 & & Asm_n(i_1) \\
 & & \ldots \\
 & & Asm_n(i_m)
\end{array}
$$

Definition of $Asm(P)$

$$
\begin{array}{rcl}
Asm((d_1 \ldots d_m \mid i_1 \ldots i_n \mid k)) & = & \texttt{.text} \\
 & & \texttt{.globl main} \\
 & & Asm(d_1) \\
 & & \ldots \\
 & & Asm(d_m) \\
 & & [\![l_{main}]\!]: \\
 & & Prologue(k) \\
 & & Asm_k(i_1) \\
 & & \ldots \\
 & & Asm_k(i_n)
\end{array}
$$

図 5.3: アセンブリ生成の定義.

（$[\![op]\!]$ は $\mathcal{V}$ の op に対応する MIPS の命令名である.）

Definition of $Prologue(n)$

$$Prologue(n) \quad = \quad \begin{array}{l} \texttt{addiu} \quad \texttt{\$sp}, \texttt{\$sp}, -n-8 \\ Save_{n+8}(\texttt{\$ra}) \end{array}$$

Definition of $Epilogue(n)$

$$Epilogue(n) \quad = \quad \begin{array}{l} Restore_{n+8}(\texttt{\$ra}) \\ \texttt{addiu} \quad \texttt{\$sp}, \texttt{\$sp}, n+8 \end{array}$$

Definition of $Save_n(r)$

$$Save_n(r) \quad = \quad \texttt{sw} \quad r, n(\texttt{\$sp})$$

Definition of $Restore_n(r)$

$$Restore_n(r) \quad = \quad \texttt{lw} \quad r, n(\texttt{\$sp})$$

図 5.4: アセンブリ生成用補助関数の定義.

$Asm_r(\mathbf{local}(n))$: $\mathbf{local}(n)$ に格納されている値をレジスタ r にロードする．$\mathbf{local}(n)$ は関数呼び出し規約から $n(\texttt{\$sp})$ に格納されているので，これをレジスタ r にロードするために `lw` 命令を用いる.

$Asm_r(\mathbf{labimm}(l))$: コード中のラベル l のアドレスを r にロードする.[2]このために `la` 命令を用いている．$[\![l]\!]$ はラベル l を MIPS 内で解釈できる記号に変換したものである.

$Asm_r(\mathbf{imm}(n))$: 整数定数 n をレジスタ r にロードする．これは `li` 命令を用いて実装することができる.

命令の変換を行う関数 $Asm_n(i)$ は，$\mathcal{V}$ の命令 i を，局所変数用に n バイトを使う関数の内部にあると仮定して実行する MIPS の命令列を生成する．この n はフレーム内に格納されている値にアクセスする際に，そのアドレスの $\texttt{\$sp}$ からのオフセットを計算するために用いられる．各ケースの説明は以下の通りである.

$Asm_n(\mathbf{local}(ofs) \leftarrow op)$: この命令は「op に格納されている値を $\mathbf{local}(ofs)$ に格納する」ように動作する．そのためにまず op の値を求め，一時レジスタ $\texttt{\$t0}$ に格納する命令を生成し（$Asm_{\texttt{\$t0}}(op)$）その後 $\texttt{\$t0}$ に格納されているアドレスからレジスタ r に値をロードする命令 $\texttt{sw} \quad \texttt{\$t0}, ofs(\texttt{\$sp})$ を生成する.

$\mathbf{local}(ofs) \leftarrow op(op_1, op_2)$: op_1 に格納されている値をレジスタ $\texttt{\$t0}$ に（$Asm_{\texttt{\$t0}}(op_1)$），$op_2$ に格納されている値をレジスタ $\texttt{\$t1}$ に（$Asm_{\texttt{\$t0}}(op_1)$）それぞれロードする．その上で，レジスタ $\texttt{\$t0}$ の値とレジスタ $\texttt{\$t1}$ の値を引数として演算子 op によって計算し，その結

[2]このようにレジスタにコード中のアドレスをロードすることで，コード中の「場所」を値として保持することが可能となる．これは高階関数の実装で必要になる.

果を $t0 にロード（$[\![op]\!]$ $t0, $t0, $t1）する．定義を簡潔にするために，演算子 op に対応する MIPS の命令を $[\![op]\!]$ で表し，具体的に使わなければならない命令を $[\![-]\!]$ の定義の中に押し込めている．(例えば $[\![+]\!] = \texttt{addu}, [\![-]\!] = \texttt{mulou}$ とすればよい.）最後にレジスタ $t0 の値を $\mathbf{local}(ofs)$ にストア（sw $t0, ofs($sp)）している．ofs バイト目のローカル変数のアドレスが $sp + ofs であることに注意せよ．

l : : ラベル $[\![l]\!]$ を生成（$[\![l]\!]$:）している．ここで $[\![l]\!]$ はラベル l を MIPS アセンブリ内でラベルとして解釈できる識別子に変換したものである．この変換は一対一対応でさえあればどのように定義しても良いが，メインのプログラムを表すラベル l_{main} は，MIPS アセンブリ内のエントリポイント（プログラムの実行時に最初に制御が移される場所）を表す main というラベル名に変換する必要がある．

$Asm_n(\mathbf{if}\ op\ \mathbf{then\ goto}\ l)$: まず op に格納されている値をレジスタ $t0 に格納する．その上で，レジスタ $t0 が true を表す非ゼロ値であれば $[\![l]\!]$ にジャンプ（bgtz $t0, $[\![l]\!]$）する．

$Asm_n(\mathbf{goto}\ l)$: 無条件でラベル $[\![l]\!]$ にジャンプ（j $[\![l]\!]$）する．

$Asm_n(\mathbf{local}(ofs) \leftarrow \mathbf{call}\ op_f(op_1))$: 関数呼び出しを行う際には，関数呼び出し規約に従ってレジスタの内容を退避・復帰したり，引数をセットしたり，返り値を取得したりしなければならない．今回のコンパイラにおいては，関数の呼び出し側では，3 ページで説明した通り，(1) レジスタ a0 の値の退避，(2) レジスタ v0 に格納されている返り値の取得，(3) 退避しておいたレジスタ a0 の値の復帰を行う必要がある．レジスタ値の退避・復帰を行う命令列は他のケースでも使用するので，それぞれテンプレ化して図 5.4 に「アドレス $sp $+ n$ にレジスタ r の内容を退避する命令列 $Save_n(r)$」と「アドレス $sp $+ n$ に退避したレジスタ r の内容を復帰する命令列 $Restore_n(r)$」として定義してある．$Save$ と $Restore$ を使うと，関数呼び出し前に実行されるべき命令列は以下の通りとなる．

1. レジスタ $a0 をメモリ上のアドレス $sp $+ n + 4$ に退避 ($Save_{n+4}$($a0)) する．$a0 は今から行う関数呼び出しのための実引数で上書きされるからである．
2. op_1 に格納されている実引数を $a0 にロードする．
3. op_f に格納されているラベル (=コード上のアドレス) を $t0 にロードする．
4. jalr 命令を使って $t0 に格納されているラベルにジャンプする．jalr 命令の第一引数 $ra には，ジャンプ先からリターンするときに帰ってくるべきコード上のアドレス (=この命令の次の行) がセットされる[3]．

この次の行からは，この後呼び出された関数が実行されリターンした後に実行されるべき命令列が書いてある．

[3]なので，$ra はこの命令の実行前にどこかに退避されていなければならないが，これは関数定義のアセンブリ生成のところで説明する．

1. レジスタ $v0 に格納されているはずの（関数呼び出し規約を参照のこと）リターンされた値を **local**(ofs)，すなわち $sp + ofs に sw 命令を使ってストアする．
2. $sp + $n + 4$ に呼び出し前に退避しておいたレジスタ $a0 の内容を復帰させる ($Restore_{n+4}$($a0))．

以上の命令列が実際に正しく関数呼び出しを実行することを確認するためには，関数呼出し時の命令のみではなく，リターン命令 (Asm_n(**return**(op))) や関数定義側でどのような命令列が生成されるかも確認することがある．前者についてはすぐ，後者については後で $Asm(d)$ の定義を説明する際にそれぞれ説明する．

return(op) : op に格納されている値を呼び出し側に返さなければならない．関数呼び出し規約によれば，関数がリターンする前には以下の処理を行う必要がある: (1) 返り値をレジスタ $v0 にロード, (2) 関数の先頭でフレーム内に退避しておいた $ra の値を復帰, (3) $sp レジスタの値を先頭で下げた分だけ上げる (すなわち，現在のフレームに使っていたスタック上の領域を解放する), (4) jr 命令を用いて $ra に格納されたアドレスにリターンする．具体的には以下の命令列が生成される:

1. op に格納されている値を返り値を格納すべきレジスタ ($v0) にロード ($Asm_{\$v0}(op)$) する．
2. $ra の値の復帰とフレームの解放を行う．定義中では $Epilogue(n)$ でこの処理を行う命令を生成している．$Epilogue(n)$ はローカルな記憶領域のサイズが n のフレームを持つ関数呼び出しのリターン前の処理を行う命令列で，退避しておいた $ra の復帰 $Restore_{n+8}$($ra) と，$sp の値の更新を行う．
3. jr 命令で復帰した $ra にリターンする．

関数定義 ($l \mid i_1 \ldots i_m \mid n$) に対応する命令列 $Asm((l \mid i_1 \ldots i_m \mid n))$ は以下のアセンブリを生成する．

1. この関数のラベルを生成する ($[\![l]\!]$:)．
2. 関数本体の先頭で行わなければならない処理を行う命令列を生成する．関数呼び出し規約によれば以下の処理を行う必要がある:
   (a) レジスタ $sp の値を更新して今から使うフレームを確保する．
   (b) レジスタ $ra の値をフレーム内の所定の位置に退避する．

   以上の処理を行うための命令列を $Prologue(n)$ として図 5.4 に定義している．ここで n はこの関数内で使用する局所変数のための記憶領域のサイズである．$Prologue(n)$ は初めにフレームのサイズ分 $sp の値を addiu 命令を用いて減らす．フレームのサイズは，(局所変数用領域)+($ra 退避先用の領域 4 バイト)+($a0 退避先用の領域 4 バイト) なので $n + 8$ である．その後，$ra をフレーム内の所定の場所に退避している．

最後に，プログラム $(d_1 \ldots d_m \mid i_1 \ldots i_n \mid k)$ のアセンブリ生成の定義を説明しよう．まず先頭で，以降にかかれている情報がプログラムである旨を示す `.text` ディレクティブを生成している．その次の行には，アセンブリ中の `main` というラベル名がグローバルなラベル，すなわち外部から見える名前であることが宣言されている．その後 d_1 から d_m までのアセンブリを順番に生成した後に，$[\![l_{main}]\!]$: に続いて，メインのプログラムを実行するためのフレームの確保を $Prologue(k)$ で行い (k はフレームのサイズ)，命令列 $i_1 \ldots i_n$ に対応するアセンブリを生成する．

5.7　簡単な最適化

コンパイラは生成されるターゲットプログラムの効率を改善するために**最適化** (*optimization*)[1]と呼ばれるプログラム変換を行う．言語 $\mathcal{C}$ の上で簡単な最適化をやってみよう．

5.7.1　無駄な束縛の除去

無駄な束縛の除去 (*elimination of redundant bindings*) は，その後使われることがなく，除去してもプログラムの意味を変えないとわかっている束縛を除去する変換である．例えば，プログラム let x = 3 in 4 は，x を 3 に束縛して 4 を返すが，束縛された x はその後一切使われないので，この束縛は**無駄** (*redundant*) である．これを束縛を行わない（多くの場合より効率のよい）プログラム 4 に変換するのが，無駄な束縛の除去である．

この変換は（というか，一般に多くの最適化は）何度か繰り返すことでより効率のよいプログラムを得ることが可能となる．例えば，プログラム let x = 3 in let y = x + 2 in 4 において，束縛変数 x は y の束縛先の計算を行う式 x+2 で使われているので無駄ではないが，束縛変数 y はその後使われていないので無駄である．そこで，無駄な y の束縛を除去すると，このプログラムは let x = 3 in 4 になるが，このプログラムにおいては x の束縛が無駄であるから，再度無駄な束縛を除去することにより，4 を得ることができる．

無駄な束縛の除去は図 5.1 に示す変換によって定義することができる．ここで $\mathbf{FV}(e)$ は e 中に現れる自由変数の集合である．この変換のキモは let $x = e_1$ in e_2 の e_2 中で束縛されている変数 x や let rec f = fun $x \to e_1$ in e_2 の e_2 中で束縛されている変数 f が無駄かどうかを判定する部分である．前者については x が e_2 に自由変数として現れておらず，かつ e_1 が関数適用の形をしていなければ，x の束縛を無駄であるとして除去する．前者の自由変数に関する条件は，x がその後使われないための十分条件になっている．後者の条件は e_1 が無限ループする式だった場合にプログラムの意味を変えないための条件である．例えば，関数定義 $d_1 :=$ let rec f x = f x を含むプログラム $(\{d_1\}, \text{let x = f 3 in 4})$ を考えよう．このプログラムは必ず無限ループする．もし後者の条件がなければ，x の束縛が（x が式 4 に自由に現れていないために）無駄だとして除去されてしまい，$(\{d_1\}, 4)$ となって，無限ループしないプログラムになってしまう．最適化はあくまでプログラムの効率を上げるために行う変換なので，プログラムの意味を変えるのはマズい．これを防ぐために，x の束縛先が，無限ループに陥る可能性のある式である関数適用である場合は，束縛を除去しないようにしている．[2][*AI:* e_1 が *if* の場合があるので，このままではちょっとまずいと思います．] ←

Exercise 5.7.1 [⋆⋆]　図 5.1 の無駄な束縛の除去は，無駄な関数定義を除去することはできない．例えば，$d_1 :=$ let rec $fx = gx$ で $d_2 :=$ let rec $gx = fx$ であるとき，プログラム

[1]「最適化」という言葉は情報科学の分野では様々な意味を持つので注意が必要である．実数上の関数を与えられた制約の下で数値的な手法を用いて最小化または最大化する手法を研究する分野を**数値最適化** (*numerical optimization*)，離散値上の関数を与えられた制約の下で最小化または最大化する手法を研究する**組み合わせ最適化** (*combinatorial optimization*) 等があるが，これらをコンパイラで行われる最適化と混同しないこと．(もちろん，組み合わせ最適化や数値最適化を用いてコンパイラでの最適化を行うことはありうる．)

[2] $\mathcal{C}$ では，ある式の評価が無限ループに陥るためには，関数適用を行わなければならない．(多分．)

Definition of $\mathcal{R}(e)$

$$
\begin{array}{rcl}
\mathcal{R}(x) & = & x \\
\mathcal{R}(n) & = & n \\
\mathcal{R}(\mathsf{true}) & = & \mathsf{true} \\
\mathcal{R}(\mathsf{false}) & = & \mathsf{false} \\
\mathcal{R}(x_1\ op\ x_2) & = & x_1\ op\ x_2 \\
\mathcal{R}(\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & \mathsf{if}\ x\ \mathsf{then}\ \mathcal{R}(e_1)\ \mathsf{else}\ \mathcal{R}(e_2) \\
\mathcal{R}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \begin{cases} \mathsf{let}\ x = \mathcal{R}(e_1)\ \mathsf{in}\ \mathcal{R}(e_2) & (\text{if } x \in \mathbf{FV}(e_2) \text{ or if } e_1 = x_1\ x_2) \\ \mathcal{R}(e_2) & (\text{otherwise}) \end{cases} \\
\mathcal{R}(x_1\ x_2) & = & x_1\ x_2
\end{array}
$$

Definition of $\mathcal{R}(d)$

$$\mathcal{R}(\mathsf{let\ rec}\ f = \mathsf{fun}\ x \to e) \quad = \quad \mathsf{let\ rec}\ f = \mathsf{fun}\ x \to \mathcal{R}(e)$$

Definition of $\mathcal{R}(P)$

$$\mathcal{R}(\{d_1, \ldots, d_n\}, e) \quad = \quad (\{\mathcal{R}(d_1), \ldots, \mathcal{R}(d_n)\}, \mathcal{R}(e))$$

図 5.1: $\mathcal{C}$ 上で無駄な束縛の除去を行う変換 $\mathcal{R}$.

Definition of $\mathcal{P}_\delta(e)$ （δ は識別子から定数への部分関数である.）

$$
\begin{array}{rcl}
\mathcal{P}_\delta(x) & = & \begin{cases} \delta(x) & (\text{if } x \in \mathbf{dom}(\delta)) \\ x & (\text{otherwise}) \end{cases} \\
\mathcal{P}_\delta(n) & = & n \\
\mathcal{P}_\delta(\mathsf{true}) & = & \mathsf{true} \\
\mathcal{P}_\delta(\mathsf{false}) & = & \mathsf{false} \\
\mathcal{P}_\delta(x_1\ op\ x_2) & = & \mathcal{P}_\delta(x_1)\ op\ \mathcal{P}_\delta(x_2) \\
\mathcal{P}_\delta(\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & \mathsf{if}\ \mathcal{P}_\delta(x)\ \mathsf{then}\ \mathcal{P}_\delta(e_1)\ \mathsf{else}\ \mathcal{P}_\delta(e_2) \\
\mathcal{P}_\delta(\mathsf{let}\ x = y\ \mathsf{in}\ e) & = & \mathcal{P}_{\delta[x \mapsto y]}(e) \\
\mathcal{P}_\delta(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) & = & \mathsf{let}\ x = \mathcal{P}_\delta(e_1)\ \mathsf{in}\ \mathcal{P}_\delta(e_2) \quad (\text{where } e_1 \text{ is not a variable}) \\
\mathcal{P}_\delta(x_1\ x_2) & = & \mathcal{P}_\delta(x_1)\ \mathcal{P}_\delta(x_2)
\end{array}
$$

Definition of $\mathcal{P}(d)$

$$
\mathcal{P}(\mathsf{let\ rec}\ f\ = \mathsf{fun}\ x \to e) \quad = \quad \mathsf{let\ rec}\ f\ = \mathsf{fun}\ x \to \mathcal{P}_\emptyset(e)
$$

Definition of $\mathcal{P}(P)$

$$
\mathcal{P}((\{d_1, \ldots, d_n\}, e)) \quad = \quad (\{\mathcal{P}(d_1), \ldots, \mathcal{P}(d_n)\}, \mathcal{P}_\emptyset(e))
$$

図 5.2: $\mathcal{C}$ 上でコピー伝播を行う変換 $\mathcal{P}$.

$(\{d_1, d_2\}, 3)$ においてはどちらの関数定義も無駄であるから，$(\emptyset, 3)$ にしてしまえば良いはずである．無駄な関数定義を除去できるように $\mathcal{R}$ の定義を書き換えよ．$\mathcal{R}(\{d_1, \ldots, d_n\}, e)$ の定義を書き換えればよいが，無駄な関数定義をできるだけたくさん除去できるように（しかし無駄でない関数定義は除去しないように）定義すること.

5.7.2 コピー伝播

コピー伝播 (*copy propagation*) は，let x=y in e を $[y/x]e$ に置き換える変換である．ただし，$[y/x]e$ は e 中の x を y に置き換えた式を表す．これによって x の束縛が無くなるので，効率が良くなることが期待される.

図 5.2 がコピー伝播を行う変換 $\mathcal{P}$ である．変換の途中で式 $\mathsf{let}\ x = y\ \mathsf{in}\ e$ を見つけると，$\mathcal{P}$ は写像 δ に x の束縛式が y であることを記録して e を変換する．変数 x を変換する際には，δ の定義域に x が含まれるかどうかを確認して，含まれているならば $\delta(x)$ に，含まれていないならば x に変換する．これにより，$\mathsf{let}\ x = y\ \mathsf{in}\ e$ を $[y/x]e$ に変換することができる.

Exercise 5.7.2 [$\star$] **定数畳み込み** (*constant folding*) を行う変換を定義せよ．定数畳み込み

とは，実行時の評価結果が分かっている値を計算してしまう変換である．例えば，let $x = 3$ in let $y = 4$ in $x + y$ は 7 に畳み込むことができる．

Exercise 5.7.3 [$\star\star$] インライン化 (*inlining*) を行う変換を定義せよ．インライン化とは関数呼び出しを呼び出されている関数の本体で置き換える変換である．例えば，$d_1 :=$ let rec $f =$ fun $x \to x + 2$ であるときに，プログラム $(\{d_1\}, f\ 5)$ をインライン化すると $(\{d_1\},$ let $t_1 = 5 + 2$ in$t_1)$ のようになる．プログラムが再帰呼び出しを含む場合には，無制限にインライン化を行うとプログラム変換が止まらなくなるので，そのようなことが無いように何らかの制限を加える必要がある．(例えばインライン化する深さを制限する，再帰を含む場合はインライン化を行わない等.)

5.8 扱っていないトピック

ここまで解説したコンパイラは，あくまでソース言語からアセンブリ言語に中間言語をはさみながら変換が可能であることを見せるためのものであり，実際に「使える」コンパイラにするにはかなりのギャップがある．今年度は時間の都合でこの「使えない」コンパイラを説明するだけで終わってしまうのだが，o 概ね以下のトピックを実装するともう少し使えるコンパイラになる.[3]興味のある人は自習してほしい．また，このうちの幾つかのトピックは，実験 4 で「コンパイラ」を選択すると体験することができる予定である．

（ざっと解説を書こうとしたが，中途半端な解説を挙げるよりはキーワードを挙げておく方が役に立ちそうなので，とりあえずキーワードだけ挙げておく.）

5.8.1 仮想マシンコードにおける最適化

フロー解析

定数畳み込み

無駄な定義の除去

生存変数解析

レジスタ割り当て

5.8.2 クロージャ変換

5.8.3 命令選択

5.8.4 その他のトピック

[3]来年度はもう少しこの講義にこれらの内容を取り込みたい．