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

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

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

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

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 $\mathcal{I}$.