ゼミ: Appel, Modern Compiler Implementation in ML

久野　靖*

1998.9.10〜

□ 本読みゼミ…目的は?

- 英語を読む訓練 (だから自分のパート以外もちゃんと読んでくること!)
- まとまった内容を体系的に学ぶ (cf.　新しい内容→論文を読む)
- 内容をネタに議論?雑談?　する
- 旧交を暖める?　新しい知合いを作る?

1　本について

□ Appel, Modern Implementation in ML, Cambridge, 1998.

- たまにはコンパイラ本というリクエスト
- ML のお勉強も兼ねて
- 読んで見たら、とても面白かったので (Java 版とかは評判悪いらしいが…)
- コンパイラを学んだことのない人→やはり 1 度は学ばなくては
- ある人→昔学んだことが「もはや古い」ことを思い知るため

2　各自の Duty について

□ 担当部分を紹介する

□ 資料は必ず用意する

- 全翻訳しようとする人がいるがそれはダメ

□ OHP はできれば用意する

- まあ皆様忙しいので…

*筑波大学大学院経営システム科学専攻

□ 教科書なのでコンパイラの課題がついてる

- できればやって見て頂きたいですが…

□ 重要! 分からないことは分からせてから来る

- 質問はメールでいつでもください

3 Preface

□ コンパイラの作り方はこの 10 年で大きく変わった。

- 新しい言語: OOL、関数型、GC。
- 新しいマシン: 多数のレジスタ、遅いメモリ、スケジューリング、キャッシュ。

□ この本では ML で動く実例を示す。

□ コンパイラプロジェクト

- よくある「おもちゃコンパイラ」ではない。まっとうな実装。
- 各章ごとにその章の内容を反映した課題。

□ 課題に必要なコード: 本の WWW ページから。

- ML 処理系やツール等もここからたどれる。

□ 謝辞…

4 Introduction

□ 本書のテーマ: プログラム言語を実行コードに変換するためのアルゴリズムとデータ構造 (要するにコンパイラ)

□ 各章ごとにコンパイラの各部分を取り上げて行く

□ 実例として Tiger という言語を使用。

- Algol 系の、簡潔だがおもちゃでない言語。
- 入れ子スコープ、ヒープ。
- functional Tiger とか OO Tiger とかもあり (後で出て来る)

□ コンパイラの各モジュール間のインタフェースも重要な問題

- ここではそれを厳密に示すため ML を採用。
- ML は汎用で多くの用途に向いているが、コンパイラには極めて向いている。楽しい。
- ML を知らない人は別途勉強してね。

4.1 モジュールとインタフェース

□ 大きなシステム→モジュール分けが重要。

- 図 1.1: 典型的なコンパイラのモジュール分け。

□ モジュール分けすることで部品の再利用が可能に。

- --- 言語を取り換える、マシンを取り換える、等。

□ モジュール間のインタフェースは私 (著者) が定める。

- 本当は自分で試行錯誤した方がためになるけど (時間がないから)。

□ Part I では表 1.2 に従って各モジュールを学んで行く。

- コンパイラによってはいくつかのモジュールを「フェーズ」にまとめている。
- 本書のコンパイラはできるだけ簡単に、しかし重要なことは省略しない限度は守って簡単にしている。

4.2 ツールとソフトウェア

□ 現代のコンパイラにとって最も役立つツール:

- 文脈自由文法 (context-free grammer, CFG) --- 構文解析に
- 正規表現 (regular expression) --- 字句解析に

□ これらを使ったツール: Yacc、Lex。その ML 版あり。

- 本書の例題テストに使っている SML-NJ に付属。本書の WWW ページから。

□ その他課題に必要なソースコードも本書の WWW ページから。

4.3 データ構造とツリー言語

□ コンパイラ中の重要なデータ構造の多くはプログラムの「中間表現」(intermediate representation, IR)

- そして IR は多くの場合、木構造。
- この木構造をちょうど言語と同様に書き記すことができる。

□ 例として、図 1.3 の文法を持つ言語を考える。

- ;、:=、print、e1 op e2。

□ プログラム例:

```
a := 5+3; b := (print(a, a-1), 10*a); print(b)
```

- これをどうやって扱う? ソースコードのままだと結構面倒。
- 図 1.4 のように木構造で表す
- それには図 1.5 のような ML の型定義を使えばよい。

4.4 ML プログラムの構造化

□ コンパイラは大きいプログラム。その各モジュールをどうインタフェースするか、きちんとしないと混乱状態に。

- 1. 各モジュールを structure とする。
- 2. open 宣言は混乱の元なので排除。そうする代わりに structure abbreviation を使う。

4.5 例題: 一本道インタプリタ

□ ループも分岐もない言語のプログラムのインタプリタを作る。

- 環境、抽象構文木、木の上の再帰。ML のウォームアップ。
- ただし解析は面倒なので、いきなり ML のデータ構造として与える。

```
type id = string
datatype binop = Plus | Minus | Times | Div
datatype stm = CompoundStm of stm * stm
             | AssignStm of id * exp
             | PrintStm of exp list
     and exp = IdExp of id
             | NumExp of int
             | OpExp of exp * binop * exp
             | EseqExp of stm * exp
val prog =
 CompoundStm(AssignStm("a",OpExp(NumExp 5, Plus, NumExp 3)),
  CompoundStm(AssignStm("b",
      EseqExp(PrintStm[IdExp"a",OpExp(IdExp"a", Minus,NumExp 1)],
           OpExp(NumExp 10, Times, IdExp"a"))),
   PrintStm[IdExp "b"]))
```

□ 副作用のないインタプリタを書く

- 表示意味論や属性文法のよい入門
- コンパイラでも有用な手法。コンパイラは「プログラムが何をするか」に関わるプログラムだから。

□ だから、ref や配列や代入は使わないこと。

- 1. 関数 maxargs : stm → int --- print 文の引数個数の最大を求める
- 2. 関数 interp : stm → unit --- 副作用を持たないインタプリタ

□ 後者の例題では、interpStm と interpExp が相互に呼び合う。

- 変数の値を表すテーブル (id × int のリスト) を用意。
- interpStm : stm × table → table
- update: table × id × int → table (単にリストの先頭に追加でよい)
- lookup: table × id → int
- interpExp: exp × table → int × table