Tiger book **輪講資料** 2008-06-05

5 章 Semantic Analysis の第 1 回。4 つのセクションのうち、今日は最初のふたつを終えたい。

- 5.1 Symbol Table
- 5.2 Bindings for the Tiger compiler
- 5.3 Type-checking expressions
- 5.4 Type-checking declarations

5 Semantic Analysis (**意味解析**)

意味解析フェーズは……

- 変数定義と、それら変数の使用 (use) を結びつける
- それぞれの式が正しい型を持っているかをチェック
- 抽象構文を、より単純な（機械語生成に適した）表現形式に変換する。

5.1 **シンボルテーブル**

シンボルテーブル (環境とも呼ばれる) ： 識別子 → 型・場所 の mapping (写像、対応関係)

型、変数、および、関数の宣言が処理されるにしたがい、それらの識別子はシンボルテーブル上で「意味」に結びつけられていく。

識別子の「使用」 (use; 定義以外の出現) に出くわしたら、シンボルテーブルが検索される。

以下、5.1 節で述べられているトピックたち：

- 変数のスコープ (p.103 下〜 p.104)
- 内側の変数による外側にある変数の隠蔽、環境追加 (+) の順序 (可換でない)。

シンボルテーブルの実装については、命令型スタイルと関数型スタイルがある。

シンボルテーブルをどっちのスタイルで実装するかというのは、コンパイル対象の言語やコンパイラを実装する言語が関数型か命令型かとかオブジェクト指向かなどとは関係ないからね！

Multiple symbol tables

アクティブな環境がいちどに複数あるような言語もある。モジュール、クラス、レコードなどがそれぞれのシンボルテーブルを持つような場合。

Figure 5.1 の例は、よくにたプログラム片の ML 版と Java 版だが、ML と Java では変数の scope が異なるので、E, N, D の環境がことなる。

$\sigma = \{M \mapsto \sigma_7\}$ なのは、どちらも同じ。

N をコンパイルするときの（N の中から見える）環境は、ML 版では $\sigma_0 + \sigma_2$ であるのに対し、Java 版では $\sigma 7$ である。

効率の良い命令型シンボルテーブル

大きなプログラムには大量の相異なる識別子があるだろうから、シンボルテーブルの検索は速くなければ。

ハッシュテーブルを使うのがいいよということで、まずハッシュのおさらい。

ハッシュ関数とハッシュテーブル

ハッシュ関数：ここでは、文字列 $\mapsto$ 整数値 (0 以上 SIZE 未満)

文字列は SIZE 種類よりたくさん存在し得るので、異なる文字列に同じハッシュ値がわりあたることもある (ハッシュの conflict)。(単射でなくてもいい。全射……の方がいいと思うけど、必須じゃないか。)

なので、ハッシュテーブル（配列ですね）の各要素はリンクリストになってます。

（ここで、絵をかこう）

相異なる文字列をひとつのハッシュ値に割り当てるための機構（上記リンクリスト）を用いて、環境の連結も実現できる。

（ここでも絵を書く。`insert / pop`）

この方式を製品レベルまで強化するためのやり方はいくつかある。練習 5.1 をみよ。

効率の良い関数型シンボルテーブル

こんどは関数型にしてみよう。つまり、$\sigma' = \sigma + \{a \mapsto \tau\}$ を得る際に、σ も破壊せずにそのまま使える状態で残しておくやり方を考える。

前述のハッシュでやると、Figure 5.3 のようになる。連結の度にハッシュテーブル（ハッシュキー個数と同じ大きさの配列）をコピーしなければならない。これは大変効率（スペース効率）が悪い。

Binary Tree を使うようにすると、このような大きな領域のコピーは発生しなくていいよ！

Baranced Tree を維持するようにすると、検索に必要な計算量を log(n) にできる。これは、永続データ構造 (persistent data structure) の例になっている。赤黒木を用いると、branced tree を維持できて、計算量 log(n) を実現できる（練習 1.1c と page 286 を参照)。

Tiger コンパイラにおけるシンボル

事ある毎に文字列の比較をするのは高コストで嫌ですね。

文字列をシンボルに変換してしまいましょう。シンボルから文字列の復元ができるようになっていれば、いいよね。

シンボルに求められる性質はなにか：

- 一致・不一致を判定するための比較がとても高速 に行える。(もちろん、元の文字列の比較結果と同じ結果を得る)
- シンボルからハッシュ値を得るのがとても高速に行える。
- 大小比較をとても高速に行える。これは、Binary Tree を作りたいときに必要となる。

シンボルの実装は整数でいいだろう（シンボルから元の文字列を復元するためだけに、string メンバをもつけど)。ただし、異なる文字列は、かならず異なる整数（シンボル）にならなければならない。これまでに出現した相異なる文字列の個数を数えておけば、それを使えるでしょう。

疑問：p.110 最後で、「Symbol module は、単純なインタフェースによって、破壊的更新・副作用の汚さを利用者から隠す」とあるけど、なのこれ？利用者からも、Symbol module が内部状態を持つことは見えるよね？見えるけど、めんどくさいのを肩代りしてくれてて、ありがたいですねという話かな。

IMPERATIVE-STYLE SYMBOL TABLES

正直なところ、なにが書いてあるのかよくわからん。とりあえず、とばす。なんなら次回までの宿題に。

5.2 Tiger コンパイラむけ BINDINGS

これまで、シンボルテーブルの役割・作り方について見てきましたが、じゃあ、シンボルテーブルに格納すべき内容ってどんなのでしょう？つまり、束縛 (binding) ってなに？

Tiger は、ふたつの互いに分離された名前空間を持ちます。ひとつは型のため、もうひとつは関数と変数のため。

原始型は int と string で、全ての型は原始型か、もしくは複合型（他の原始型もしくは複合型を組み合わせたもの＝レコード型）です。

レコード型には、追加情報として各フィールドの名前と型が含まれる。

別々に定義されて、結果的におなじ構造をもつことになったレコード型の扱いについて。同じ構造なら同じ型であると解釈する言語もあるでしょう。そういった場合、型の一致判定は各フィールドについて再帰的なチェックになる。

Tiger は、そういったことはしない。ただし、別名を定義することはできるため、「名前が違うように見えるけど同じ型」という状況は生じる。

環境

言語仕様（名前空間がわかれていること）から、環境がふたつ必要なことがわかる。

（次で行いたい）型チェックのためには、変数エントリ中身は ty (型)、関数エントリの中身は formals (引数の型) と result (返り値の型) だけがあればいい。他に必要になれば、あとで足そう。