

言語処理系分科会

第4回 - 関数, 参照型(配列)



semiexp



今回の内容


- 関数の表現, 呼び出し
- 参照型
 - たとえば配列
- もう少しまともな GC (mark-sweep garbage collector)

関数の表現

- 関数といえども、実体は AST で表せる
- ただの「Stmt のリスト」じゃ少し不十分
 - 関数は引数をとる
- 関数を AST で表すのに必要なものは,
 - 関数の実体 (Stmt のリスト)
 - 引数 (名) のリスト
 - (デフォルト引数に対応する場合は、デフォルト引数の値 Expr)
- ちなみに型付き言語だと、戻り値や引数の型も持たせないといけない

関数呼び出し

- 関数内では、ローカル変数が存在
- 呼び出す際には,
 1. 新たに local environment を作成し
 2. その local environment を持って、関数の AST 評価を行い
 3. 呼び出しが終わったら、local environment を破棄する
- 変数呼び出しの Expr を扱うときには、少し注意が必要
 - 変数が global, local どちらにあっても探せるようにしないといけない
 - 両方に出てきたら？
 - そもそも、型付き言語に explicit な「変数宣言」はあるのか？



戻り値

- 前回の continue とかと同様の発想
- Stmt の戻り値に状態を覚えさせる
- ただし, 今回は「状態」に加えて「戻り値 (あれば)」も覚えさせる必要がある



引数

- local environment があれば, 引数はすごい簡単
- その environment に, (引数名 -> 引数の値) を登録してしまうだけ

Native function

- 独自言語の内部だけで完結させるのは無理！
 - 入出力とか入出力とか入出力とか
 - primitive 型についての処理 (文字列の長さとか) も無理そう
 - 速度が必要なところだけ native 関数を書くという方法もある
- 関数呼び出しで場合分けする
 - Native 関数だったら、関数リストには関数ポインタを登録しておき、そのポインタを C++ で呼び出す
 - さもなくば、関数リストには「関数定義の AST ノード」を登録しておき、前のページで言ったような方法で呼び出す

参照型

- 変数には、普通の型以外に、参照型がある
- 普通の型は、その変数に直接データが結びついている
- 参照型は、それ自体データを持たず、データがある場所を指し示している
 - 要はポインタ
- 例えば配列が参照型なら、
 - `a = {1, 2, 3}`
`b = a`
`b[0] = 4`
 - とやると、`a` も変更される

参照型の実装

- 基本的には、変数の型に「参照型」を追加するだけ
- 参照型の場合は、内容は「実体へのポインタ」になる
- 変数に参照型を代入するときには、単に「実体へのポインタ」だけをコピーする

参照型の例：配列

- 配列は大量のデータを持っていたら大変
 - 例えば、引数渡しで全部コピーしたら大変
- そういうものは参照型にすると便利
- 実装は、さっき書いた参照型の実装法とあまり変わらない
- 「配列」という特別な参照型があるとする

左辺値

- $a = a + 2$
- ここで、左辺と右辺の a の意味は違う
- 右辺の場合は、今 a に入っている値の意味
- 左辺の場合は、現在の a の内容には興味がなく、 a という入れ物の意味
- 今までは、代入の左辺にあるものは「変数 1 個」だったので、いくらでもごまかせた
- $p[q[a+1]]$ みたいな式が出てくると、ごまかせなくなってくる

左辺値

- 左辺の「値」は、それが数値型だろうと、見かけ上参照型となる
- 代入の実現の方法
 1. $\phi = \psi$ の形の式が出てきたら、左辺を「左辺値」として特別に処理し、値を保持している変数のアドレス or (C++ 上の) 参照を取得する
 2. Expr を評価する関数で、左辺値と右辺値を区別せず扱う
 - 普通、左辺値は問題なく右辺値になれる
 - 値が左辺値として有効かどうか覚えておく
 - あるいは、値が「参照型」であれば左辺値としてしまってもよい？

参照の闇

- 参照が出てくると、もっと面倒なことが発生する
- $a = \{1, 2, 3\}$, $b = \{4, 5, 6\}$
 $a[0] = b$, $b[0] = a$
- 循環参照が起きる
- すると、参照カウント方式では自然に解放されなくなる
-> mark-sweep garbage collector



mark-sweep garbage collector

- 参照カウントはやめる
- 「適当なタイミング」で、すべてのオブジェクトについて、それが使用中かどうかを判定し、もはや使用していない場合は破棄する
- 使用中の判定は,
 1. グローバル、ローカルの変数に入っているものは使用中
 2. 式の評価中に一時的に保持される参照が指しているものも、使用中
 3. 使用中のものが参照しているものは使用中
 4. 上の3つのルールで使用中にならなかったものは使用中でない

mark-sweep garbage collector

- 参照カウントはやめる
- 「適当なタイミング」で、すべてのオブジェクトについて、それが使用中かどうかを判定し、もはや使用していない場合は破棄する
- 使用中の判定は、
 1. グローバル、ローカルの変数に入っているものは使用中
 2. 式の評価中に一時的に保持される参照が指しているものも、使用中 ←たいへん
 3. 使用中のものが参照しているものは使用中
 4. 上の3つのルールで使用中にならなかったものは使用中でない

一時的参照？

1. C/C++ のローカル変数をスキャンして、オブジェクトっぽいものがあつたらそこからもマークを行う
 - Ruby はこの方法を使ってるらしい
2. 一時的に保持する参照は、ローカル変数に隠しておかず、どこか (独自のスタック) に保存する
 - そんな微妙なタイミングで GC かけるなー > < ?
 - `x = "result: " + naniyara_memory_wo_takusan_tsukau_shori()`
 - こういう場合は、関数呼び出し中に GC が必要になる可能性が高い