中間レポート1

1029289895 尾崎翔太

2018/10/18

1　課題 2

1.1　プログラムの設計方針

この課題は至極単純なので, 設計方針は資料に書いてある通りに実装することである.

1.2　プログラムの説明

1.2.1　lexer.mll

Loop 式のための”loop”と Recur 式のための”recur”を予約語に追加した. また, 組のために”,”と”.”も認識できるようにした.

1.2.2　parser.mly

与えられた BNF をそのまま実装しただけで, 特に説明することはない.

1.3　感想

これは実験 3 をやっていれば難しいところは一つもなく, コードする量も少なかったので, 肩慣らしだなと思った.

2　課題 4

2.1　プログラムの設計方針

「末尾位置ならば･･･」という文言があるので, 今処理しようとしているノードが末尾位置にあるかどうかを表す変数を用意した. 後は木を巡回して RecurExp を見つけた時に末尾位置にあるかどうかを見れば良い.

2.2　プログラムの説明

syntax.ml の 25〜50 行目に recur_check 関数がある. 実体は 26〜48 行目の body_loop 関数で, この関数は exp に加えて, 今末尾位置にいるかどうかを表す bool 型の is_end を引数に取る.　一番トップレベルは末尾位置ではないので, 最初 body_loop は false で呼び出される. RecurExp を探すので, exp が子にないノードは何もしない. また, exp でない子に対しても何もしない. exp を子に持つノードは, 末尾位置の定義に従ってその子が末尾位置にあるかどうかを判断して, is_end を適切な値にして body_loop 関数を呼び出す. そし

て, RecurExp を処理する際に is_end が false ならばエラーを吐くようになっている.

2.3 感想

末尾位置の定義を見て素直に実装するとこうなると思う. is_end を使い回すことで if 式は RecurExp の場合にしか現れず, 全体的にすっきりした.

3 課題 5

3.1 プログラムの設計方針

はじめは素直に実装していたのだが, 無駄な Let 束縛を無くさないと後の課題の中間表現が読みにくいことに気づいたので, そうすることにした. この部分に関しては, 資料の付録の通りに実装することにした. しかし, それで上手く行ったのは IfExp だけで, BinOp と AppExp については試行錯誤することになった. また, 無駄な Let 束縛を無くしたことで, 変換前後でプログラムの意味が変わってしまう場合があったので, それを防ぐために前処理を行うことにした. この前処理は norm_exp の中で行うこともできなくはなかったが, 正規形にするという本質から外れているのでそうしなかった. 結果として, 変換に 2 パスかかってしまったがご容赦願いたい.

3.2 プログラムの説明

3.2.1 前処理部

normal.ml の 103〜127 行目の rename 関数と 130〜158 行目の preprocess 関数で行う. rename 関数は exp と id を受け取って, exp 中の id をすべて同じフレッシュな id に置き換える. このフレッシュな id は元の id の頭に"$"を付けるようにした fresh_id 関数で作る. また, 一度リネームしたものをもう一度リネームすると"$"と数字がどんどん付いて行くので, それを防ぐために 105 行目で先頭と最高尾の文字を削ってから fresh_id をしている. preprocess 関数は exp と束縛された id を保持する id_list を引数に取る. LetExp あるいは LetRecExp を見つけると, それが束縛しようとしている id が id_list に含まれているかどうかを調べて, そうならこの LetExp あるいは LetRecExp に rename 関数を適用して新しくなった id を id_list に追加する. そうでないなら, そのまま id を id_list に追加する. 全体としては 226 行目で preprocess 関数を呼び出すことで前処理を行う.

3.2.2 正規化部

normal.ml の 96〜100 行目のｖｋ関数は資料の $\mathcal{V}_k$ を表しており, そのまま実装してある. 正規化の本体は 163〜220 行目の norm_exp 関数である. ｆは資料の付録の value を cexp に変えただけといった使い方をしている. ただ, 上から渡された f をそのまま下に渡すかどうかは判断しなければならない. 無駄な Let 束縛を無くすことは IfExp では自然な実装になっている. BinOp, AppExp, TupleExp は無駄な Let 束縛を無くしたい部分が二箇所あるので, 片方はｖｋ関数なしで処理して, もう片方はその結果を利用してｖｋ関数を用いて処理している.

3.3 感想

急に難易度が上がったと感じた. プログラムの説明においてｆの使い方が微妙に曖昧なのは, きちんと理解して書いたというよりは試行錯誤の結果こうすればいいとわかったからである. 無駄な Let 束縛の排除については, BinOp, AppExp, TupleExp は苦労したし, コードも気持ち悪い感じになってしまった. とはいえｖｋ関数にまかせてしまうと, その中で作られたフレッシュな id を得ようとすると大変なのでこういうやり方ですることにした. それと, テストがないのが辛いと感じた. 先へどんどん進めたくなって, 資料と同じようなコードを吐けばそれでいいやという気持ちになり, 変なケースを考えられない. 正直今のプログラムが本当にきちんと動いているかは余り自信がない.