Assignment 15

Contents

1. Background

In this pass we complete our compiler (yeah!) by replacing `verify-scheme` with a Scheme parser. The parser recognizes a richer language that allows:

- unquoted fixnum, boolean, and empty-list constants;
- variables to be arbitrary symbols;
- `lambda`, `let`, and `letrec` bodies to be implicit `begin` expressions;
- one-armed `if` expressions;
- `or` and `and`;
- calls to the primitive `not`.

In the final source language, a variable may be bound by multiple `lambda`, `let`, and `letrec` expressions, with its scope determined via the usual Scheme rules. No names are reserved, so a keyword or primitive name loses its special meaning within the scope of a `lambda`, `let`, or `letrec` binding of the name. For example,

```
(let ([if (lambda (x y z) z)]) (if #t 0 1))
```

is a valid program that evaluates to 1.

2. Scheme Subset 15

Here is a grammar for the subset of Scheme we'll be handling this week:

Program ⟶ *Expr*
Expr ⟶ *constant*
| *var*
| `(quote` *datum*`)`
| `(if` *Expr Expr*`)`
| `(if` *Expr Expr Expr*`)`
| `(and` *Expr*`*)`
| `(or` *Expr*`*)`
| `(begin` *Expr*`*` *Expr*`)`
| `(lambda (`*uvar*`*)` *Expr*`+)`
| `(let ([`*var Expr*`]*)` *Expr*`+)`
| `(letrec ([`*var Expr*`]*)` *Expr*`+)`
| `(set!` *uvar Expr*`)`
| `(`*prim Expr*`*)`
| `(`*Expr Expr*`*)`

where:

- *constant* is `#t`, `#f`, `()`, or a fixnum;
- fixnum is an exact integer;
- *datum* is a constant, pair of *datum*s, or vector of *datum*s; and
- *var* is an arbitrary symbol.

The set of primitives now includes the one-argument `not` operator but otherwise does not change. The constraints on fixnums do not change.

3. Things to do

For this assignment, we have two things to do:

- replace `verify-scheme` with a new pass, `parse-scheme`
- make sure the entire compiler is working.

You must hand in the code for each of your passes, i.e., `parse-scheme` and all of other passes required to compile source programs to assembly-language programs. Since your passes may differ from ours, the file you submit should also set the `compiler-passes` parameter appropriately to allow us to run your compiler.

3.1. `parse-scheme`

This pass has several tasks to perform:

- verify that the syntax of the input program is correct;
- verify that there are no unbound variables;
- convert all variables to unique variables, handling the shadowing of identifiers (other variables, keyword names, and primitive names) correctly;
- convert unquoted constants into quoted constants;
- verify that each constant and quoted datum is well formed, with each fixnum in the fixnum range;

- rewrite `not` calls, `and` expressions, and `or` expressions in terms of the other language expressions.

In broad terms, this pass must recognize programs in the source language, reject all others, and for programs in the source language, replace them with programs in the source language of Assignment 14.

The pass should expand `not` calls, `and` expressions, and `or` expressions according to the following specifications:

`(not` e`)` $\rightarrow$ `(if` e `'#f '#t)`

`(and)` $\rightarrow$ `'#t`
`(and` e`)` $\rightarrow$ e
`(and` e_1 e_2 e_3 `...)` $\rightarrow$
`(if` e_1 `(and` e_2 e_3 `...) '#f)`

`(or)` $\rightarrow$ `'#f`
`(or` e`)` $\rightarrow$ e
`(or` e_1 e_2 e_3 `...)` $\rightarrow$
`(let ([`t e_1`]) (if` t t `(or` e_2 e_3 `...)))`

where t is a fresh unique variable and the nested calls to `and` and `or` are expanded recursively.

You might be tempted to special-case `not` calls appearing in the test part of an `if` expression as follows:

`(if (not` e_1`)` e_2 e_3`)` $\rightarrow$ `(if` $e1$ e_3 e_2`)`

but this should not generate any better assembly code if your `optimize-jumps` pass is working properly.

In our implementation, we pass our `Expr` helper an environment, represented as an association list, that maps identifiers to action procedures. Each action procedure accepts an environment plus an expression and transform its input expression appropriately. The initial environment contains bindings for each keyword and primitive name. For example, our `and` action procedure matches the syntax of the input expression and converts it into nested `if` expressions following the expansion given above.

```
(lambda (env x)
  (match x
    [(and ,[(Expr env) -> x*] ...)
     (if (null? x*)
         '(quote #t)
         (let f ([x* x*])
           (let ([x (car x*)] [x* (cdr x*)])
             (if (null? x*)
                 x
                 `(if ,x ,(f x*) (quote #f))))))]
    [,x (error who "invalid Expr ~s" x)]))
```

When a variable binding is seen, an association from the variable to a generic "variable" action procedure is added to the front of the association list representing the environment, effectively shadowing any existing binding.

You may, however, structure your code however you like.

4. Boilerplate and Run-time Code

The boilerplate code and run-time code do not change.

5. Testing

A small set of invalid and valid tests for this assignment have been posted in tests15.ss. You should make sure that your compiler passes work at least on this set of tests.

Make sure you test the invalid tests as well as the valid tests this week, since your `parse-scheme` pass is responsible for rejecting invalid inputs.

6. Coding Hints

Before starting, study the output of the online compiler for several examples.