CMSC 22620 Spring 2009	Implementation of Computer Languages	Project 1 April 2, 2009

Translation to LambdaIR

Due: April 10, 2009

1 Introduction

The first project is to implement a translation from the typed, abstract-syntax tree representation produced by the compiler's front end to a normalized λ-calculus representation that we call `LambdaIR`. You will use the algorithm discussed in class. The datatypes that represent this IR are as follows:

```
datatype exp
  = LET of (var list * rhs * exp)
  | TRY of var * exp * exp
  | IF of var * exp * exp
  | CASE of var * (pat * exp) list * exp option
  | ESCAPE of var
  | APPLY of var * var list
  | RETURN of var list

and rhs
  = EXP of exp
  | FIX of lambda list
  | PRIM of primop * var list
  | UPDATE of var * var * var
  | CON of dcon * var list
  | LIT of literal

and lambda = FN of var * var list * exp

and pat = PAT of dcon * var list
```

2 Details

2.1 Pre-defined operators and identifiers

Most of the built-in operators, such as +, are translated to *primitive operations* (see the `Prim` structure). The one exception is string concatenation (^), which should be mapped to a call to the predefined function `stringConcat`. Note that since this function is not visible in the surface

language, it can have a more efficient calling convention. Specifically, it is not curried and takes its two arguments directly.

Array sssignment is another special case. The surface-language typing rule for assignment says that assignment has type `Unit`, but the `LambdaIR` assignment operator yields no results. Thus, an expression

```
a!i := x
```

produces

```
LET([], UPDATE(a, i, x),
LET([t], CON(UNIT, []),
  RETURN[t]))
```

The predefined functions (*e.g.*, `print`) are going to be translated to predefined variables in the `LambdaIR` representation. We will also be translating local AST variables to `LambdaIR` variables. To manage these translations, you will need to pass an environment as an extra argument to your translation functions (see the `TranslateEnv` module).

2.2 Non-local control-flow

To manage non-local control-flow, we add an explicit parameter to functions that represents the current **`try`** handler. When translating a function application, you will need to include the current handler as an additional argument (it should be the last argument).

2.3 Type abstraction and application

The `LambdaIR` representation is not typed, but we must keep the runtime effect of type abstraction in the resulting IR. We do so by using one-argument functions to represent type abstraction (the single argument is the current handler).

2.4 Smart constructors

The `LambdaIR` module defines functions for building `LambdaIR` terms. You should *always* use these functions to construct terms, as they ensure that certain data-structure invariants are preserved.

3 An example

Consider the following LangF program:

```
fun ['a] id (x : 'a) : 'a = x;
id [Integer -> Integer] (id [Integer])
```

It translates into the following LamdaIR term:

```
LET([id], FIX(id', [ex1],
  LET(t1, FIX(t2, [x, ex2], RETURN[x]), RETURN[t1])),
LET(t3, APPLY(id, [ex0]),
LET(t4, APPLY(id, [ex0]),
  APPLY(t3, t4))))
```

We assume that `ex0` is the handler for this code (the front-end will wrap the program in a global handler). Note also that the type applications are preserved.

4 Submission

Sources for Project 1 will be seeded into your gforge repositories. These sources include the front-end code, as well as support code for this assignment. You will complete the implementation of the `Translate` module (in `translate/translate.sml`) and commit your solution into the repository. You should not need to change any other file in the source code.

We will collect projects from the SVN repositories at 10pm on Friday, April 10. Please make sure that you have committed your final version before then.

Revision history

April 6, 2009 Added discussion about string concatenation and assignment.

April 2, 2009 Original version