CSCI-P523/423 Project One: Integers, Expressions and Variables (oh my!)

Carl Factora — Jeremy Siek

November 17, 2015

The `S0` Language

For this project, our compiler will be targetting the `S0` langauge, which we include below.

```
S0 ::= Int | Sym | (+ S0 S0) | (- S0 S0) | (- S0) | (read) |
       (let ([Sym S0]) S0)
```

In this document, we include the Backus-Naur Form (BNF) of each intermediate language (IL) and brief descriptions of the respective *compiler passes* each IL is handled by. This week, we encounter the `C0` and `Pseudo-x86` ILs as we compile down to x86 Assembly.

Scheme to C

1. `programify : S0 -> S0`

The preliminary pass, `programify`, wraps each `S0` expression with a `program` tag.

```
(+ 5 5) => (program () (+ 5 5))
```

The `programify` pass transforms `S0` into the intended input language of `uniquify`.

```
Program ::= (program S0* S0)
S0      ::= Int | Sym | (+ S0 S0) | (- S0 S0) | (- S0) |
            (read) | (let ([Sym S0]) S0)
```

2. `uniquify : Env -> S0 -> S0`

The `uniquify` pass is responsible for handling this week's binding construct: `let`. Here, we use Racket's `gensym` to generate unique names for each let-bound variable. For example, the `s0_8.scm` test case:

```
(program ()
  (let ([x 20])
    (+ (let ([x 22]) x) x)))
```

is transformed into:

```
(program ()
  (let ([g30 20])
    (+ (let ([g31 22]) g31) g30)))
```

Due to the behavior of `gensym`, your `uniquify` pass may generate different names than the one above (viz. `g30,g31`). This is normal and should be expected.

For simplicity, we recommend using an environment to store and associate the names of generated variables to their respective names in the original program.

3. `flatten : Bool -> S0 -> C0`

In `flatten`, we transform the `S0` language into the `C0` IL. This pass should generate an expression from the grammar below.

```
Atomic    ::= Int | Sym
Expr      ::= Atomic | (prim op Atomic*)
Statement ::= (assign Sym Expr) | (return Atomic)
Program   ::= (program Sym* Statement*)
```

Essentially, this pass is responsible for removing complex (i.e., nested) calls to primitive operators (e.g. `+, -, *`). This pass also converts `let` into a series of `assign` and `return` expressions, while lifting the names of all assigned variables. For example, if we return to our earlier example (after `uniquify`):

```
(program ()
  (let ([g30 20])
    (+ (let ([g31 22]) g31) g30)))
```

is transformed into:

```
(program (g30 g31)
  (assign g30 20)
  (assign g31 22)
  (return (+ g31 g30)))
```

To clarify, this pass *flattens* complex operation calls. For example, in our ongoing example, the call to `+` contained a call to a `let`. Thus, `flatten` converts this expression into an expression containing only `Atomic` values. To keep this pass simple, we recommend passing around a boolean value that determines whether a given subexpression is required to be simple.

This pass is also responsible for *lifting* every variable that appears in the *left-hand side* (LHS) of each generated call to `assign` (e.g. `(assign Sym Expr)`) into the symbol list of the main `program` wrapper (viz. `g30,g31`).

Pseudo-x86 to x86

4. `select-instructions : C0 -> Pseudo-x86`

In this pass, we transform `C0` into the pre-x86 IL `Pseudo-x86` (Px86). This IL is contained within this following grammar.

```
Atomic  ::= (int Int) | (var Sym) | (register Reg)
Expr    ::= (add Atomic Atomic) | (sub Atomic Atomic) | (imul Atomic Atomic) |
            (mov Atomic Atomic)
Program ::= (program Sym* Expr*)
```

This pass is not responsible for handling *register allocation*. Instead, this pass is simply responsible for introducing the Px86 IL. Returing to our running example:

```
(program (g30 g31)
  (assign g30 20)
  (assign g31 22)
  (return (+ g31 g30)))
```

is tranformed into:

```
(program (g30 g31)
  (mov (int 20) (var g30))
  (mov (int 22) (var g31))
  (mov (var g31) (register rax))
  (add (var g30) (register rax)))
```

5. `assign-instructions : Homes -> Pseudo-x86 -> Pseudo-x86`

6. `insert-spill-code : Pseudo-x86 -> x86`

7. `print-x86 : x86 -> String`