Assignment 4: SSA and Memory

15-411: Compiler Design
Josiah Boning (jboning@andrew) and Ryan Pearl (jboning@andrew)

Due: Tuesday, October 25, 2010 (1:30 pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution or a printout of a typeset solution at the beginning of lecture on Tuesday, October 25. Please read the late policy for written assignments on the course web page. If you decide not to typeset your answers, make sure the text and pictures are legible and clear.

Problem 3: Exceptions (25 points)

Exceptions are a common language feature not present in C or C0 (though you probably recognize from the language you are using for your compiler—SML, Haskell, OCaml, and Java all support exceptions). Suppose we want to extend C0 with exceptions, so that the programmer can declare new flavours of exception, raise exceptions, and handle them. Our version of exceptions will not be very powerful, as they will not carry any data with them (aside from what flavour of exception was raised).

The semantics of our exceptions will be that when an exception is raised, it is handled by the nearest enclosing `try ... handle (...) ...` block. The exception is assigned to the identifier "argument" of the handle clause, which is available to the body of the handle clause. The syntax is as in the example on the next page.

(a) Describe how you would handle parsing the syntax given here.

(b) Describe any additions or changes to the static semantics (elaboration rules and typing rules) that would be needed for exceptions.

(c) Describe how you would compile exceptions. Be sure you discuss all of the operations that need to be supported.

```
exception Small;
exception Big;

int hailstone(int x) {
    // check argument
    if (x < 1) {
        raise Small;
    }
    if (x > 9000) {
        // IT'S---too big, give up
        raise Big;
    }

    // recursive computation
    if (x == 1) {
        return 0;
    }
    if (x % 2 == 0) {
        return 1 + hailstone(x / 2);
    }
    return 1 + hailstone(3 * x + 1);
}

int main()
{
    int i = readint();

    while (true) {
        try {
            return hailstone(i);
        } handle (e) {
            if (e == Small) {
                // whoops, bad argument. try a bigger problem instead.
                i += 5;
            }
            if (e == Big) {
                // ooof. try a smaller problem instead.
                i /= 2;
            }
        }
    }

    return 0;
}
```

Problem 2: Bitfields (10 points)

To get the most value out of their memory when using structs, programmers may want to use data sizes that aren't the standard word sizes. Most processors don't support moving arbitrary numbers of bits around, but the same effect can be achieved by storing multiple small fields into one memory word. This introduces some extra complications in using this data, but a compiler can automatically handle this.

(a) Suppose you want to store a 12 bit field named x and a 20 bit field named y in a 32 bit memory word with offset z in the struct s. Write down abstract assembly for both reading and writing x, using a variable r as the source/destination and whatever temps you need.

(b) What problem would this technique face in a concurrent environment? Give a detailed example.

Problem 3: Tuples (30 points)

Tuples are a language feature common in functional programming languages. Like exceptions, you probably recognize these from the language you are using for your compiler—SML, Haskell, and OCaml all feature tuples (though Java does not). Suppose we wanted to give C0 programmers the ability to declare tuples using SML-like types, assign values to them, pass them as parameters to functions, and use them as the return values. While individual tuple elements can be read, it is not permissible to assign only to a single element. Additionally, the types used to construct a tuple type must all be small types, so a tuple of an int and a struct is invalid. We will use the @ symbol to join multiple types into a tuple type, and array-like bracket syntax to reference the 0-indexed elements of a tuple. As an example, the following code would return 11 from main.

```
int@int inc_tuple (int@int x)
{
    return (x[0] + 1, x[1] + 1);
}

int main()
{
    int@int y = (0, 5);
    int@bool@bool *z = alloc(int@bool@bool);
    y = inc_tuple(y);
    *z = (5, true, true);
    return *z[0] + y[1];
}
```

(a) Describe how you would handle parsing the tuple syntax given here.

(b) Describe any additions or changes to the static semantics rules that would be needed to handle tuples.

(c) Describe how you would compile tuples. Be sure you discuss all of the operations that need to be supported.