CSE 504 Project

E-- Code generation and Optimization

1 Event Matching

Event matching should be implemented using one of the direct DFA construction techniques we have discussed — either the one based on the concept of derivatives discussed in class, or the technique described in your textbook. Of course, neither of these algorithms handle event parameters. In this project, you have one of the following options:

- **Option A:** ignore event arguments, thus handling just regular expressions. In this case, there are unlikely to be any special event-related optimizations, so your optimizations will likely be traditional compiler optimizations such as common-subexpression elimination.
- **Option B:** handle event arguments and the conditions on them. In this case, you need a more general algorithm. You can refer to Section 5 of the paper available at `http://seclab.cs.sunysb.edu/seclab/pubs/usenix99.pdf`. Note that there are some notational differences: the paper uses ";", "||" and "$*$" in the places of ":", "\/" and "**". The paper uses the term "REE" to refer to these event patterns, and NEFA to the automata constructed for matching these patterns.

  Note that when you choose option B, you can think of optimizations that are specific to event matching and choose to focus on these optimizations instead of (or in addition to) traditional compiler optimizations.

In either case, events will be input using the IN instruction. You can assume that, for the purposes of this project, all event names will have only a single character, so it will be easy to input event names using the IN instruction that returns just a single byte. (This limits us to a maximum of 53 events.) With option A, each IN operation will return the next event in the input stream, whereas with option B, you will need to be able to read event arguments as well. For this project, we will restrict ourselves to integer and floating point event arguments. Since you have the declaration of events, which specifies the number and types of event arguments, you should be able to use an appropriate number of IN operations to input event arguments and convert them into integers or floating point numbers. For instance, given the following declaration

```
event a(int x1, float x2)
```

the input

```
aK\0\0\0\0\0\0AaP\0\0\0\0\0@A
```

will denote a sequence of two events `a(75,11.0),a(80,12.0)`. (Note that `\0` denotes a null character.) Note that the input representation reflects how integers and floating point numbers are represented internally. For instance, an integer is represented using 4-bytes, with the first byte representing the least significant byte of the integer. A similar observation applies to floating point numbers, except that their internal representation is a bit more complex. The above event information can be read using the following icode:

```
IN R100 // Read event name
....    // Decide how many parameters to read, and their types
INI R101 // Read the integer parameter
INF F101 // Read the floating point parameter
```

Note that `IN` behaves differently from `INI` and `INF` on errors: `IN` will return a negative value to denote input errors or en-of-file. The other two instructions will simply abort the program with an error message.