Software Compiler Assignment on:

Exceptions and Indirect Addressing

The Software Compiler course exam is composed by two parts. One is a written test, the other is an homework, to be terminated before course last class. The written test contributes with the 40% to the whole grade, while the homework contributes with the remaining 60%. To pass the whole exam, you must get a pass grade from both the test and the homework.

During the lab classes, we show you the ACSE compiler and the associated assembler [2]. They must be used as starting point for your homework. During the last class, you must present your work, showing your edits, giving a brief demo, and responding at some questions given by course lecturers.

Sources of ACSE can be found on the course site [3]. It is a tarball of the ACSE mercurial [1] repository. You are required to version your code with mercurial. A copy of your edits must be submitted to course lectures in the form of a patch with respect to the version of ACSE you have used as starting point.

Assignment

You are required to modify both the front-end of the compiler and the back-end.

Front-end

The exception handling mechanism allows to decouple error handling code from normal code. Currently ACSE lacks support for exceptions.

Supporting exceptions with full semantic is an hard task, so the exception construct to implement is subjected to the following limitations:

- `try catch` cannot be nested
- `catch` is mandatory
- there must be only one `catch`
- `throw` does not take parameters

```
try {
   ...
   throw;
   ...
catch {
   ...
}
```

Figure 1: Exception construct example in ACSE

- `finally/catch(...)` is not supported

Figure 1 reports an example.

If a `throw` statement is found inside a `try` block, then control is immediately transferred to the associated `catch` block. A `throw` statement that is not inside a `try` block must generate an error – it does not matter if the error is generated at compile-time or at run-time.

Back-end

Peephole optimizations are optimizations applied later in the compilation process. Their goal is to identify groups of subsequent instructions that can be substituted with another, more efficient, instruction.

Given an assignment `a = exp`, currently the ACSE machine translates code in the following way:

- load values needed by `exp` in registers by means of `LOAD`
- compute the expression
- store expression result in `a` using `STORE`

You are required to implement peephole optimizations to limit the number of generated `LOAD/STORE`, substituting them with indirect addressing.

For example, given an assembly code like the one reported in Figure 2(a), the optimizer must detect that `R1` is first used to hold the result of the addition and then its content is stored in memory at location `L0`. The optimization is using a `MOVA` to move the address of label `L0` inside register `R1` and use indirect addressing on `R1` in the `ADD` instruction. Figure 2(b) shows the optimized code.

You are required to build a simple peephole optimizer for instruction sequences similar to the ones of the example. The optimizer must be a new ACSE tool, like `asm` that can be used as starting point. It must read an input text assembly file, optimize it, and write the result assembly on a new text file.

```
ADD R1 R0 R2
STORE R1 L0
```
(a) Source

```
MOVA R1 L0
ADD (R1) R0 R2
```
(b) Optimized

Figure 2: Example of peephole optimization

Creating and Applying Patches

The ACSE distribution is already provided as a mercurial repository. See [4] for the basic mercurial commands.

Assuming that your work has been committed with revision X, to generate a patch use the following command:

```
hg diff -r 0 -r X
```

Assuming the patch has been saved in the file `patch.diff`, use the following command to apply it to a freshly unpacked ACSE source tree:

```
hg import patch.diff -m "Applied patch"
```

The submitted project must successfully pass the above process, i.e. before submitting your patch check that it can be applied to a freshly unpacked ACSE source tree including compilation and testing process where applicable.

References

[1] Mercurial. http://mercurial.selenic.com, 2011.

[2] A. Di Biagio and G. Agosta. Advanced Compiler System for Education. http://compilergroup.elet.polimi.it, 2008.

[3] Formal Languages and Compilers Group. Software Compilers. http://compilergroup.elet.polimi.it, 2010.

[4] J. Spolsky. Hg Init: a Mercurial Tutorial. http://hginit.com, 2011.