Programming Assignment VI
Due date will be announced later

1   Introduction

In this assignment, you will extend *your* code generator for Cool from Programming Assignment V to handle exceptions. The appropriate mechanisms for exceptions have already been added to the reference lexer, parser, and semantic analyzer for you.

This assignment requires much less code than previous projects; we have implemented the code generation for exceptions in coolc in under 50 lines of C++ code. However, getting the code correct is challenging and requires careful thought before implementation.

You may work a group of one or two people for this assignment. The submit program will ask you to specify group members when you turn in your assignment

2   Files and Directories

We suggest that you work using the same CVS repository and working directory as for PA5. From your working directory, type the following command to get a copy of the additional files for PA6 (for both C++ and Java):

```
gmake -f ~cs164/assignments/PA6/Makefile source
```

The additional files for PA6 are as follows:

- example-PA6.cl
  This file should contain a test program of your own design. Test as many uses of exceptions as you can.

- README-PA6
  This file will contain the write-up for your assignment. It is critical that you explain design decisions, how your code is structured, and why you believe your design is a good one (i.e., why it leads to a correct and robust program). It is part of the assignment to explain things in text as well as to comment your code. Also, please follow the directions in this file to update your project files from PA5 with the *.PA6update files.

- finally_exception.cl
  This is an example program that uses **try-finally**.

- finally_exception.output
  This file gives the output of running spim on code generated from finally_exception.cl by coolc. You should consult below for a complete description of the semantics of **try-finally**.

3   Exceptions in Cool

Exceptions are a general mechanism for dealing with error conditions that arise in computation. It has many advantages over other methods for handling errors; for example, the programmer does not need to specify special return values to denote error, and exceptions can be propagated to the function caller in case the caller has more information about how to deal with the exception.

3.1   Syntax

We extend Cool with exceptions by adding the follow new constructs:

$$
\begin{aligned}
expr \quad ::= \quad & \dots \\
| \quad & \textbf{throw } expr \\
| \quad & \textbf{try } expr \textbf{ catch } ID => expr \\
| \quad & \textbf{try } expr \textbf{ finally } expr
\end{aligned}
$$

where **throw**, **try**, **catch**, and **finally** are four new keywords. In each of the new constructs, the last *expr* extends as far as possible (i.e., encompasses as many tokens as possible).

In Cool, the programmer can throw an exception where the value of the exception is the result of any Cool expression. The expression "**throw** *expr*" returns the value of *expr* to the closest dynamically enclosing **catch** block. The expression "**try** $expr_1$ **catch** $ID => expr_2$" evaluates expression $expr_1$, and if an exception is thrown in $expr_1$, then $expr_2$ is executed with *ID* bound to the value of the thrown exception. If no exception is thrown in $expr_1$, then $expr_2$ is not executed. An uncaught exception will be caught by the Cool runtime, in which case will abort the program.

Note that the form of **try-catch** you will implement is slightly different than in lecture. Here, the **catch** expression will handle exceptions of any type, rather than only handling exceptions with dynamic type that is a subtype of some given type.

The expression "**try** $expr_1$ **finally** $expr_2$" evaluates $expr_1$ and then evaluates $expr_2$ regardless of whether $expr_1$ throws an exception. If $expr_1$ throws an exception and $expr_2$ evaluates normally, then the exception must continue to propagate after $expr_2$ is executed. However, if an exception is thrown in $expr_2$ (regardless of the evaluation of $expr_1$), only the exception from $expr_2$ will continue to propagate. Finally, if both $expr_1$ and $expr_2$ evaluate normally, then the value for the entire expression is the value of $expr_2$ (i.e., it behaves exactly like "$\{expr_1; expr_2; \}$").

3.2   Static Semantics

We now extend our typing rules to accommodate these new constructs. Recall from Written Assignment 5 the following rules for **throw** and **try-catch**.

$$
\frac{O, M, C \vdash e : T_1}{O, M, C \vdash \textbf{throw } e : T_2} \qquad \text{[Throw]}
$$

$$
\frac{\begin{array}{c} O, M, C \vdash e_1 : T_1 \\ O[\text{Object}/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \textbf{try } e_1 \textbf{ catch } x => e_2 \ : T_1 \sqcup T_2} \qquad \text{[Try-Catch]}
$$

A **throw** causes the computation to make a non-local "jump", so the expression "**throw** *e*" never "returns" to the context in which it appears syntactically, so can be assigned any type. We must, however, check that *e* is well-typed.

The exception caught in a **try-catch** block can be of any type, so we conservatively assume that x has type Object. Evaluation of e_1 can either return normally, in which case the value of the whole **try-catch** expression is the value of e_1, or exceptionally, in which case the value of the whole expression is the value of e_2. Thus, the type of the **try-catch** construct is the least upper bound of T_1 and T_2.

Now, consider the typing rule for **try-finally**.

$$
\frac{\begin{array}{c} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \textbf{try } e_1 \textbf{ finally } e_2 \ : T_2} \qquad \text{[Try-Finally]}
$$

Since e_2 is always evaluated after evaluating e_1, the type of "**try** e_1 **finally** e_2 " is the type of e_2.

3.3   Dynamic Semantics

In this section, we extend the operational semantics of Cool to accommodate the additional syntactic forms for exceptions. To do this, we need to define a notion that distinguishes between a normal return and an exceptional return. Thus, we define a generalized value g in terms of conventional Cool values v as follows:

$$
\begin{array}{rlll}
g & ::= & Norm(v) & \text{a normal return with value } v \\
  & | & Exc(v) & \text{an exceptional return with value } v
\end{array}
$$

Recall that every Cool value is an object, written

$$
\begin{array}{rll}
v & ::= & X(a_1 = l_1, a_2 = l_2, \ldots, a_n = l_n) \quad \text{an object of dynamic type } X \text{ with attributes} \\
  &   & \hspace{4.5cm} a_1, a_2, \ldots, a_n \text{ at locations } l_1, l_2, \ldots, l_n
\end{array}
$$

We now redefine our evaluation judgment in the following manner to incorporate these generalized values:

$$
so, S, E \vdash e : g, S'
$$

This new judgment is read as follows: "in the context where *self* is the object *so*, the store is S, and the environment is E, the expression e evaluates to generalized value g and yields the new store S'".

For every rule we had before, we modify the rule to accommodate both normal return and exceptional return. For example, the evaluation rule for arithmetic under normal execution is as follows:

$$
\frac{
\begin{array}{l}
so, S_1, E \vdash e_1 : Norm(Int(i_1)), S_2 \\
so, S_2, E \vdash e_2 : Norm(Int(i_2)), S_3 \\
op \in \{*, +, -, /\} \\
v = Int(i_1 \; op \; i_2)
\end{array}
}{
so, S_1, E \vdash e_1 \; op \; e_2 : Norm(v), S_3
}
\qquad [\text{Arith}]
$$

However, in the case that e_1 or e_2 returns exceptionally, we need to propagate the exception as follows:

$$
\frac{
\begin{array}{l}
so, S_1, E \vdash e_1 : Exc(v_1), S_2 \\
op \in \{*, +, -, /\}
\end{array}
}{
so, S_1, E \vdash e_1 \; op \; e_2 : Exc(v_1), S_2
}
\qquad [\text{Arith-Exc1}]
$$

$$
\frac{
\begin{array}{l}
so, S_1, E \vdash e_1 : Norm(v_1), S_2 \\
so, S_2, E \vdash e_2 : Exc(v_2), S_3 \\
op \in \{*, +, -, /\}
\end{array}
}{
so, S_1, E \vdash e_1 \; op \; e_2 : Exc(v_2), S_3
}
\qquad [\text{Arith-Exc2}]
$$

Note that in [Arith-Exc1], the evaluation of e_2 is aborted. Also, notice that the store is threaded through all expressions, which means that all side-effects are propagated.

For **throw**, there are two cases to consider depending on whether the expression returns normally or exceptionally. If the expression returns normally with object v, then v is value of the thrown exception.

$$
\frac{
so, S_1, E \vdash e : Norm(v), S_2
}{
so, S_1, E \vdash \textbf{throw } e : Exc(v), S_2
}
\qquad [\text{Throw}]
$$

Otherwise, if the expression returns exceptionally, that exception should still be propagated.

$$\frac{so, S_1, E \vdash e : Exc(v), S_2}{so, S_1, E \vdash \textbf{throw}\ e : Exc(v), S_2} \qquad \text{[Throw-Exc]}$$

Note that an exceptional value is always returned.

For **try-catch**, the operational semantics are as follows.

$$\frac{so, S_1, E \vdash e_1 : Norm(v_1), S_2}{so, S_1, E \vdash \textbf{try}\ e_1\ \textbf{catch}\ Id => e_2\ : Norm(v_1), S_2} \qquad \text{[Try-Catch]}$$

If e_1 return normally, then the **try-catch** simply returns that value. Otherwise, if e_1 returns exceptionally, then the **catch** expression is evaluated.

$$\frac{\begin{array}{l} so, S_1, E \vdash e_1 : Exc(v_1), S_2 \\ l_1 = newloc(S_2) \\ S_3 = S_2[v_1/l_1] \\ E' = E[l_1/Id] \\ so, S_3, E' \vdash e_2 : g_2, S_4 \end{array}}{so, S_1, E \vdash \textbf{try}\ e_1\ \textbf{catch}\ Id => e_2\ : g_2, S_4} \qquad \text{[Try-Catch-Exc]}$$

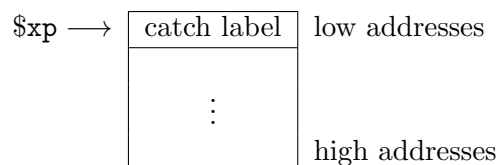Note that g_2 can be either a normal or an exceptional value.

The evaluation rule(s) for **try-finally** is/are left as an exercise that you will answer in README-PA6.

4   Code Generation

You will implement the long jump scheme to implement exceptions. In this approach, for every **try-catch**, some state information is pushed onto the stack (called an *exception frame*) so that it can be restored when execution passes to the **catch** block to handle an exception. A special *exception register* points to the most recent exception frame, with each exception frame containing a pointer to the previous exception frame in a linked-list fashion. For convenience, we let $xp denote the exception register (note that this is not an actual register name on the MIPS architecture). To be compatible with the runtime system and Coolaid, you should use $t9 as the exception register for your code generator. On a **throw**, the value of the exception is put in $a0, the exception register is read, and control is passed to the **catch** block specified in the exception frame. Note that the exception is returned to the **catch** block in $a0. The **catch** block is responsible for first restoring the state from the exception frame. The exception frame is then popped by restoring the exception pointer to point to the previous exception frame.

4.1   Exception Frame

The exception frame must store (either implicitly or explicitly) state information in the evaluation of the **try** block before the occurrence of the exception to evaluate the **catch** block to handle the exception. To be compatible with the top-level exception handler in the Cool runtime system, the layout of the exception frame should be as follows:

$xp $\longrightarrow$ | catch label | low addresses

⋮

high addresses

The first word of the exception frame contains the address of the label of the closest dynamically enclosing **catch** block. The catch label will indicate where to jump to on a **throw**. Thus, the final two instructions of the code generation for a **throw** is something like

```
lw    $t1 0($t9)
jr    $t1
```

It is up to you how to layout the rest of the exception frame. In general, the exception frame saves the state information that needs restored to evaluate the **catch** block. Minimally, you will need to be able to restore the stack pointer, the frame pointer, and a pointer to the previous exception frame.

4.2   Runtime Support

In order to handle uncaught exceptions, we have installed a top-level exception frame in the trap handler before your Cool program is invoked. This exception frame is laid out according to the layout described in the previous section so that the code generation of **throw** following this layout will jump to a special "**catch**" block provided by the runtime system. This runtime **catch** block prints the message "Uncaught Exception" and aborts the Cool program. You can look in the `trap.handler` to see how this implemented. Look for the labels `__exception` and `__exception_handler`.

4.3   Try-Finally

Note that code generation for **try-finally** is not enabled in coolc. You should first think carefully about the expected behavior of a **try-finally** expression and an implementation strategy before coding.

5   Testing and Debugging

As with Programming Assignment V, you will need a scanner, parser, and semantic analyzer that handles exceptions to test your code generator. You may extend your own components or the components from coolc. By default, the coolc components are used. To change this behavior, replace the lexer, parser, and/or semant executable (which are symbolic links in your project directory) with your own scanner/parser. Even if you use your own components, it is wise to test your code generator with the coolc scanner, parser, and semantic analyzer at least once because we will grade your project using coolc's version of the other phases.

We have included a number of test programs using the exception constructs. They can be found in `~cs164/examples/`. All exception programs are named `*_exception.cl`. Coolaid is able to verify programs with exceptions with the exception frame layout and code generation for **throw** given above. Try running your compiler on these examples and verifying the result in Coolaid. If you use Coolaid, please help us by providing feedback about it in the README-PA6 (as in Programming Assignment V). You may also use spim/xspim to running code generated from your compiler on these examples. These examples are by no means comprehensive. You should also develop your own test cases.

6   Grading

To minimize the effect of incorrect solutions for Programming Assignment V, the test cases that we will use to grade this project will not require code generation for `new SELF_TYPE` nor for static dispatch.

7   Final Submission

Make sure to complete the following items before submitting to avoid any penalties. Then, submit your project by typing "`submit PA6`".

☐ Include your write-up in README-PA6. Please include your feedback about Coolaid in README-PA6.

☐ Include your test cases that test your code generator with exceptions in example-PA6.cl.

☐ Make sure all your code for the code generator with exceptions is in

 – cool-tree.h, cgen.h, cgen.cc, cgen_supp.cc, and emit.h for the C++ version; or
 – cool-tree.java, CgenClassTable.java, CgenNode.java, CgenSupport.java, Bool-Const.java, IntSymbol.java, StringSymbol.java, TreeConstants.java, and additional .java files you may have added for the Java version.

☐ Be sure to answer 'yes' to the submission prompt for files that contain your code.