CSC3F002EP: Compilers | Project

Exceptions

Starts: `2024-11-28`

1 SYNOPSIS

Exceptions are a programming construct designed to handle errors or unexpected events that arise during program execution. Rather than causing the program to crash or behave unpredictably, exceptions provide a structured mechanism for detecting, responding to, and recovering from such situations. They serve as a signal that something unusual or problematic has occurred, such as invalid input, file access failure, or division by zero. By allowing error-handling logic to be separated from the main program logic, exceptions enhance code clarity and maintainability. Additionally, exceptions can propagate up the call stack, enabling higher levels of the program to determine how to handle them if they are not addressed locally. This mechanism also facilitates graceful recovery, allowing programs to execute predefined corrective actions or cleanup tasks, such as closing files or releasing resources, ensuring robust and reliable behavior.

The key aspects of exceptions handling are:

Raising/throwing exceptions: when an error or unexpected condition occurs, the program raises (or throws) an exception. This is typically done using a specific language construct (e.g., `raise`).

Catching/handling exceptions: exception handling involves specifying how a program should respond when an exception is raised. This is usually done using constructs like `try` and `except`, where code that may raise an exception is placed in a `try` block, and appropriate handling logic is provided in a `except` block.

Propagation: if an exception is not handled at the level where it occurs, it propagates up the call stack until it is caught by an appropriate handler or results in the program's termination.

In this project, we will add exception handling capabilities to the BX programming language framework. To achieve this, the project introduces enhancements across multiple components of the compilation pipeline:

Parser: the parser will be augmented to recognize and validate syntax for declaring exceptions and specifying exception handling constructs (e.g., `try`, `catch` & `throw`). (See Section 2)

Type-Checker: the type-checker will be extended to analyze exception usage for correctness and consistency. It will verify that exceptions are declared, handled, or propagated in a way that adheres to the language's type system and runtime safety guarantees. (See Section 3)

Maximal-Munch: the maximal munch (lowering) algorithm will be adapted to incorporate support for the exception mechanism. Instead of extending the TAC, all exception handling functionality will be encoded directly using standard TAC instructions. (See Section 4)

The following is an example of a BX program that utilizes exceptions:

```
exception DivByZero;

proc div(x : int, y : int) : int raises DivByZero {
  if (y = 0) {
    raise DivByZero;
  }
  return x / y;
}

proc main() {
  try {
    var x = div(5, 0) : int;
  } except DivByZero {
    print(42);
  }
}
```

2 EXTENDING THE BX LANGUAGE

The BX language in this project is a strict superset of that of lab 4. You are free to start from the reference implementation.

2.1 *Exception declaration*

In BX, exceptions must be explicitly declared with a specific name, as they are not anonymous. The syntax for declaring an exception is defined as follows:

$\langle\text{decl}\rangle ::= \cdots \mid \langle\text{exception}\rangle$

$\langle\text{exception}\rangle ::=$ `'exception'` `IDENT` `';'`

Here, the identifier specifies the name of the exception.

2.2 *Raising an exception*

The BX language incorporates the `raise` construct into its statement definitions to support raising exceptions, specified as follows:

$\langle\text{stmt}\rangle ::= \cdots \mid$ `'raise'` `IDENT` `';'`

Here, the identifier represents the name of the exception that is being raised.

2.3 *Exception handling*

The exception handling in BX extends the grammar with a new ⟨tryexcept⟩ statement construct, allowing a block of code to be executed with corresponding exception-handling logic. A ⟨tryexcept⟩ statement begins with a `try` block, which contains the code that might raise exceptions. This is followed by one or more ⟨catch⟩ clauses, where each clause specifies how a particular exception is handled.

The catch clauses are introduced by the keyword `except`, followed by the identifier of the exception to be caught and a block of code that will execute when that specific exception is raised. The identifier denotes the name of the exception being handled, and the block provides the handling logic.

```
⟨stmt⟩ ::= ··· | ⟨tryexcept⟩
⟨tryexcept⟩ ::= ··· | 'try' ⟨block⟩ ⟨catch⟩+
⟨catch⟩ ::= 'except' IDENT ⟨block⟩
```

2.4 *Declaring exceptions escaping a procedure*

In BX, it is mandatory for a procedure to annotate the exceptions that can escape its scope using the ⟨raises⟩ clause. This clause serves as part of the procedure's declaration, following the optional return type, and explicitly lists the exceptions that the procedure may propagate during its execution.

The ⟨raises⟩ clause begins with the keyword raises, followed by one or more exception identifiers, separated by commas. Each identifier corresponds to a specific exception that the procedure can raise. This ensures that any code calling the procedure is aware of the potential exceptions and can handle them appropriately.

```
⟨procdecl⟩ ::= 'def' IDENT '(' (⟨param⟩ (',' ⟨param⟩)*)? ')' (':' ⟨ty⟩)? ⟨raises⟩? ⟨block⟩
⟨raises⟩ ::= 'raises' IDENT (',' IDENT)*
```

3 EXTENDING THE TYPE CHECKER

To handle all the exception constructs discussed, the type-checker must be extended to perform specific validations:

Validate raise statements: the type-checker must ensure that every `raise` statement refers to a valid exception, i.e. check that the identifier used in the `raise` statement corresponds to an exception that has been explicitly declared within the compilation unit, and that exceptions raised in a `raise` statement are either handled within the same scope or declared as escaping through the procedure's `raises`.

Validate try and except blocks: For the `try` and `except` constructs, the type-checker should: i) ensure that every `except` clause specifies an exception identifier that has been declared in the compilation unit, ii) verify that each `except` block uniquely matches a declared exception and does not repeat the same exception within the same `try` construct, and iii) check that the code within each block (`try` or `except`) type-checks independently and adheres to the BX language's standard typing rules. Moreover, unreachable catch blocks must be flagged as errors, ensuring that each `except` clause corresponds to a meaningful exception.

Validate procedure declarations with the raises clause : for procedures, the type-checker should: i) ensure that every exception listed in the `raises` clause is a declared exception, ii) verify that every `raise` statement within the procedure refers to an exception declared in the procedure's `raises` clause or is caught within the procedure itself, and iii) check that procedures without a `raises` clause do not allow any exceptions to propagate beyond their scope. Note that the `main` function is a bit special in the sense that it cannot have a `raises` clause.

Validate procedure calls and exception propagation : when a procedure that includes a `raises` clause is called, the type-checker must ensure that all exceptions listed in the called procedure's `raises` clause are either caught within the caller's code or declared in the `raises` clause of the calling procedure.

4 EXTENDING THE MAXIMAL-MUNCH ALGORITHM

Here is how exceptions will be encoded in TAC and handled during the maximal munch process:

- Each exception is represented as a unique non-zero machine word derived from its name, for instance, by using a cryptographically secure hash function to minimize the risk of collisions. (In Python, you can use the `hashlib.sha256` function)
- We introduce a global TAC variable, named `@exception`, that stores the currently active exception, identified by its encoded value. If no exception is active, this variable is set to `0`.
- During maximal munch, upon entering a try block, two fresh labels are generated and pushed onto a dedicated stack, similarly to the mechanism used for while loops for the `continue` and `break` statements. The first label marks the location of the exception handler for the `try` block, and the second label indicates the position immediately following the entire `try`/`except` construct. These labels will be removed from the stack right after the maximal munching of the `try`/`except` is completed.
- If the `try` block executes completely without raising an exception, the program skips over the exception handler and jumps directly to the label indicating the position after the `try`/`except` block.
- When an exception is raised, the global variable `@exception` is updated with the corresponding exception code. Execution then jumps to the exception handler of the innermost try block, if one exists.
- The exception handler is implemented as a series of `if` statements that compare the value of `@exception` against the exception codes handled by the block. If a match is found, the value of `@exception` is reset to `0` and the corresponding handler is executed. After the handler finishes, normal execution resumes, starting with the instruction immediately following the `try`/`except` block.
- If no matching handler is found in the current block, execution jumps to the exception handler of the next outer try block, if one exists (in the case of nested try statements).
- If no (more) exception handlers are available, the exception is considered to have escaped the procedure. In this case, the procedure performs a normal return. However, upon completion of a function call, the caller must check whether the global variable `@exception` is set to `0`. If it is not, this indicates that an exception has escaped the called procedure. The caller must then handle the exception by following the same exception-handling mechanism described above.

- If an exception is raised within an exception handler, the new exception overrides the current one. The global variable `@exception` is updated with the new exception's code, and the handling process continues as if the new exception were raised normally.

This approach to handling exceptions is admittedly naive and introduces significant time penalties, primarily due to the reliance on global variables, repeated comparisons in exception handlers, and frequent control flow jumps. These inefficiencies can lead to slower execution, especially in programs with deeply nested try blocks or frequent exceptions. However, to keep the project manageable and focused on the core concepts of exception handling, we have deliberately chosen this straightforward method. While not optimized for performance, it provides a clear and simple mechanism for implementing exception constructs within the language.

5 FINAL DELIVERABLE

The final deliverable for this project is your compiler, implemented as a Python script named `bxc.py`. The submission must include all necessary dependencies for the compiler to function correctly.

As usual, the compiler should take as input the filename of a BX program, specified as `$name.bx`, and generate the corresponding assembly code, saving it to a file named `$name.s`.

If additional dependencies are required for the compiler to operate, you may submit your work as a zip file named `bxc.zip`, containing `bxc.py` along with all the dependencies needed to run the compiler seamlessly.

You may work alone, or you may work in groups of size **2**. In the latter case, your submission *must* contain a file called `GROUP.txt` that contains the names (login) of the group members.