Structure

1. Exception Handling
2. Exceptions in LOOP
3. Task: TRY CATCH and THROW statement
4. Bonus Task: Advanced exception handling
Exception Handling

• Run time events (occurring rarely) that interrupt the normal control flow, but ...

• May be caught, and handled so that the program can continue.

• JAVA, C#: try ... catch ... finally ... throw
Example for Exceptions(LOOP)

CLASS Main IS
  METHOD main IS x: Integer; BEGIN
    TRY
      x := helper * 1 + 0;
      WRITE x;
    CATCH 'b' DO
      WRITE 'Y'; WRITE 'e'; WRITE 's'; WRITE '\n';
    END TRY
  END METHOD
  METHOD helper : Integer IS x, y, z: Integer;
  BEGIN
    READ x;
    IF x='a' THEN THROW x+1;
    ELSE RETURN x; END IF
  END METHOD
END CLASS

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Yes</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>c</td>
</tr>
</tbody>
</table>

(actually, the example is bad)
Implementing Exceptions


Method 1: Dynamic Registration
- Maintains exception frame on the stack at runtime that contains information about stack unravelling and exception handling.
- Proposal for LOOP

Method 2: Table-driven Approach
- Generate tables at compile-time, which are indexed by the instruction counter at run time in order to do stack unravelling and exception handling.
- State of the art for C++ compilers.
TRY CATCH Syntax (LOOP)

\[
\text{statements} ::= \{ \text{statement} \}
\]

\[
\text{statement} ::= \text{READ memberaccess ';'}
\mid \text{WRITE expression ';'}
\mid \text{THROW expression ';'}
\mid \text{TRY}
\]

\[
\text{TRY}
\]

\[
\text{statements}
\]

\[
\text{CATCH (number | character) DO}
\]

\[
\text{statements}
\]

\[
\text{END TRY}
\]

\[
\mid \ldots
\]
Exception Frames in LOOP

• Global storage address `_exception` points to the actual exception frame

• The frame contains the code address of the exception handler, and a reference to the next exception frame

<table>
<thead>
<tr>
<th>Address</th>
<th>Storage / Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>_exception</td>
<td>304 (actual exception frame)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>201</td>
<td>next exception frame</td>
</tr>
<tr>
<td>202</td>
<td>Code address exception handler 1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>304</td>
<td>201 (next exception frame)</td>
</tr>
<tr>
<td>305</td>
<td>Code address exception handler 2</td>
</tr>
</tbody>
</table>
Using Exception Frames in LOOP

Principle:

• Reconstructing a previous state
  – TRY stores the state
  – In case of an exception, it is reconstructed
  – Stack contents is reconstructed to the state when state was stored (reduction).

• After reconstruction, control flow is continued
  – At another place (CATCH)
Task: TRY CATCH THROW

• Extend lexical, syntax, context analysis
• Initial exception frame refers to final exception handler
  – Output: ABORT <character>
• Storage cell _exception points to initial exception handler
• THROW computes exception number and follows the innermost exception frame
• Entry / exit of TRY block generates / removes an exception frame
• CATCH block checks the exception thrown in order to handle or propagate it
• RETURN needs to be adapted

10%
• Error lists
• CATCH alternatives
• Predefined exceptions
  – Division by zero throws 0
  – NULL pointer access throws 1
• Context analysis

statements ::= { statement }
statement ::= ...
  | THROW expression ';
  | TRY
    statements
    CATCH ( number | character)
    {',' (number | character)} DO
    statements
    {CATCH ( number | character)
    {',' (number | character)} DO
    statements}
END TRY
  | ...

5%