Error Handling/Recovery

1

Developing a Parser for CPRL
Version 2: Error Handling/Recovery

- When the compiler is integrated with an editor or as part of integrated development environment (IDE), it might be acceptable to stop compilation after detecting the first error and pass control to the editor.
- In general, even if integrated with an editor, a compiler should try to detect and report as many errors as possible.

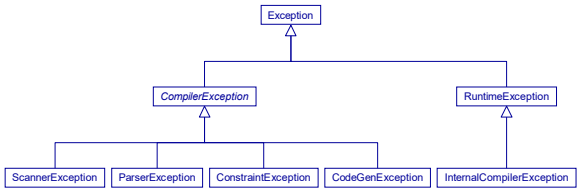
2

Types of Compilation Errors

- Syntax errors – based on the grammar; e.g., invalid or missing tokens such as missing semicolons or using "=" instead of "`:=`" for assignment.
- Scope errors – violation of language scope rules; e.g., declaring two identifiers with the same name within the same scope.
- Type errors – violation of language type rules; e.g., the expression following an "`if`" does not have type `Boolean`.
- Miscellaneous errors – other errors not categorized above; e.g. functions may not have `var` parameters.

3

The CPRL Exception Hierarchy



Kotlin does not have checked exceptions, but all four subclasses of `CompilerException` are classified as checked exceptions in Java. Class `InternalCompilerException` is an unchecked exception in Java.

4

Checked Versus Unchecked Exceptions

- In Java, any exception that derives from class `Error` or class `RuntimeException` is called an *unchecked* exception.
- All other exceptions are called *checked* exceptions.
- Two special situations for Java (but not Kotlin): If a call is made to a method that throws a checked exception or if a checked exception is explicitly thrown, then an enclosing block **must** either handle the exception locally or else the enclosing method must declare the exception as part of its exception specification list.
- Unchecked exceptions **may** be declared in the exception specification list or handled, but doing so is not required.

5

Error Handling Versus Error Recovery

- **Error Handling** – Finding errors and reporting them to the user.
- **Error Recovery** – Compiler attempts to resynchronize its state and possibly the state of the input token stream so that compilation can continue normally.
- The purpose of error recovery is to find as many errors as possible in a single compilation, with the goal of reporting every error exactly one time.
- Effective error recovery is extremely difficult. Any error reported after the first one should be considered suspect by the programmer.

6

Object `ErrorHandler`

- Kotlin object (not a class).
- Handles the reporting of errors
- Exits compilation after a fixed number of errors have been reported
- Implements the singleton pattern; i.e., there is only one instance of `ErrorHandler`.

7

Two Key Methods in Object `ErrorHandler`

```
/**
 * Returns true if errors have been reported by the
 * error handler.
 */
fun errorsExist() : Boolean

/**
 * Reports the error.  Stops compilation if the maximum
 * number of errors have been reported.
 */
fun reportError(e : CompilerException)
```


8

General Approach to Error Handling

- Enclose the parsing code for each rule with a `try/catch` block.
- When errors are detected, control transfers to the `catch` block.
- The `catch` block will
  - report the error by calling appropriate methods in the error handler
  - skip tokens until it finds one in the follow set of the nonterminal defined by the rule
  - return from the parsing method for this rule

9

Method `recover()`

Method `recover()` implements error recovery by skipping tokens until it finds one in the follow set of the nonterminal defined by the rule.

```
/**
 * Advance the scanner until the current symbol is one of the
 * symbols in the specified array of follows.
 */
private fun recover(followers : Array<Symbol>)
  {
    scanner.advanceTo(followers)
  }
```


10

Example: Error Handling/Recovery

- Consider the rule for a `varDecl`:
  ```
  varDecl = "var" identifiers ":" typeName ";" .
  ```
- In CPRL, the follow set for a `varDecl` is
  { “const”, “var”, “type”, “procedure”, “function”, “begin” }

11

Example: Error Handling/Recovery (continued)

```
// varDecl = "var" identifiers ":" typeName ";" .
fun parseVarDecl()
  {
    try
      {
        match(Symbol.varRW)
        val identifiers = parseIdentifiers()
        match(Symbol.colon)
        parseTypeName()
        match(Symbol.semicolon)

        for (identifier in identifiers)
            idTable.add(identifier, IdType.variableId)
      }
```

(continued on next slide)

12

Example: Error Handling/Recovery (continued)

```
catch (e: ParserException)
  {
    ErrorHandler.reportError(e);
    val followers = arrayOf(
        Symbol.constRW,     Symbol.varRW,      Symbol.typeRW,
        Symbol.procedureRW, Symbol.functionRW, Symbol.beginRW)
    recover(followers)
  }
}
```


13

Shared Follow Sets

- When several nonterminals have the same follow set, it is convenient to declare the array of "followers" once as a static field and then reference it as needed.
- CPRL Example:

```
/** Symbols that can follow an initial declaration. */
private val initialDeclFollowers = arrayOf(
    Symbol.constRW,     Symbol.varRW,      Symbol.typeRW,
    Symbol.procedureRW, Symbol.functionRW, Symbol.beginRW)
```

- The array `initialDeclFollowers` can be used for error recovery when parsing a `constDecl`, a `varDecl`, or an `arrayTypeDecl`.

14

Method `parseVarDecl()` (reimplemented)

```
// varDecl = "var" identifiers ":" typeName ";" .
fun parseVarDecl()
  {
    try
      {
        ...
      }
    catch (e : ParserException)
      {
        ErrorHandler. reportError(e)
        recover(initialDeclFollowers)
      }
  }
```

Note use of shared follow set.

15

Error Recovery for `parseStatement()`

- Method `parseStatement()` handles the rule

```
statement = assignmentStmt | ifStmt | loopStmt | exitStmt
          | readStmt | writeStmt | writelnStmt
          | procedureCallStmt | returnStmt .
```

- Error recovery for `parseStatement()` requires special care when the symbol is an identifier since an identifier can not only start a statement but can also appear elsewhere in the statement.
- Consider, for example, an assignment statement or a procedure call statement. If we advance to an identifier, we could be in the middle of a statement rather than at the start of the next statement.

16

Error Recovery for `parseStatement()` (continued)

- Since the most common identifier-related error is to declare or reference an identifier incorrectly, we will assume that this is the case and advance to the next semicolon before implementing error recovery.
- The goal is that by advancing to the next semicolon, we hopefully move the scanner to the end of the erroneous statement.

17

Error Recovery for `parseStatement()` (continued)

```
try
  {
    ...
  }
catch (e: ParserException)
  {
    ErrorHandler.reportError(e)
    scanner.advanceTo(Symbol.semicolon)
    recover(stmtFollowers)
  }
```


18

Implementing Error Recovery

- Not all methods will need a `try/catch` block for error recovery at this stage of parser development.
- Example

```
fun parseInitialDecls()
  {
    while (scanner.symbol.isInitialDeclStarter())
        parseInitialDecl()
  }
```

- Method `match()` throws a `ParserException` when an error is detected – does not implement error recovery.
- Any parsing method that calls `match()` will need a `try/catch` block for error recovery.

19

Implementing Error Recovery (continued)

- Additionally, method `parseVariableExpr()` and the `add()` method of class `IdTable` can throw a `ParserException`.

`match()`, `add()`, and `parseVariableExpr()` are the only three methods that throw a `ParserException` back to the caller, so any method that calls one of these three methods will need to have a try/catch block.

20

Additional Error Recovery Strategies

- After reporting the error, replace the token with one that might be allowed at that point in the parsing process.
- Examples
  - Replace "=" by ":=" when parsing an assignment statement in a CPRL compiler.
  - Replace "=" by "==" when expecting a relational operator in a Java compiler.

21

Additional Error Recovery Strategies (continued)

- After reporting the error, insert a new token in front of the one that generated the error.
- Examples
  - When parsing an exit statement, after matching "`exit`", if the symbol encountered is in the first set of expression, insert "`when`" and continue parsing.
  - When parsing an expression, if ")" is expected but "`;`" is encountered as the next symbol, insert ")" with the expectation that the "`;`" will likely terminate a statement.

22

Example: Additional Recovery Strategies (in method `parseAssignmentStmt()`)

```
...
// match(Symbol.assign);
try
  {
    match(Symbol.assign)
  }
catch (e : ParserException)
  {
    if (scanner.symbol == Symbol.equals)
      {
        ErrorHandler.reportError(e)
        matchCurrentSymbol()   // treat "=" as ":="
                               // in this context
      }
    else
        throw e
  }
...
```

Instead of simply calling `match(Symbol.assign)`, use a nested `try/catch` block that treats "=" as ":="

23