Error Management in Compilers and Run-time Systems

Classification of program errors
- Handling static errors in the compiler
- Handling run-time errors by the run-time system
  - Exception concept and implementation

Program Errors ...
- A major part of the total cost of software projects is due to testing and debugging.
- US-Study 2002: Software errors cost the US economy ~60e9 $ yearly

What error types can occur?
- Classification
- Prevention, Diagnosis, Treatment
  - Programming language concepts
  - Compiler, IDE, Run-time support
  - Other tools: Debugger, Verifier, ...

Classification of Program Errors (1)
- Design-Time Errors (not considered here)
  - Algorithmic errors: e.g., forgotten special case; non-terminating program
  - Numeric errors: Accumulation of rounding errors
  - Contract violation: Violating required invariants
- Static Errors
  - Syntax Error: forgotten semicolon, misspelled keyword, e.g. BEGNI (BEGIN)
  - Semantic Error: Wrong parameter number or type; Downcast without run-time check
- Compiler Runtime Errors
  - Symbol table / constant table / string table / type table overflow

Classification of Program Errors (2)
- Execution Run-time errors – usually not checkable statically
  - Memory access error: e.g.,
    - Array index error
    - Pointer error: Dereferenced NULL pointer
  - Arithmetic error: Division by 0; Overflow
  - I/O – error: unexpected end of file, write to non-opened file
  - Communication error: Wrong receiver, wrong type
  - Synchronisation error: Data “race”, deadlock
  - Resource exhaustion: Stack / heap overflow, time account exhausted
- Remark: There are further types of errors, and combinations.

Error Prevention, Diagnosis, Treatment
- Programming language concepts
  - Type safety → static type errors
  - Exception concept → run-time errors
  - Automatic memory mgmt → memory leaks, pointer errors
- Compiler frontend → syntax errors, static semantic errors
- Program verifier → Contract violation
- Code Inspection [Fagan76] → All error types
- Testing and Debugging → Run-time errors
- Runtime protection monitor → Access errors
- Trace Visualiser → Communication errors, Synchronisation errors

Some Debugging Research at PELAB
(Needs a lot of compiler technology, integrated with compiler)
- High-Level Host-Target Embedded System Debugging
  - Semic-automatic debugging – automatic bug localization by automatic comparison with a specification for using oracle
More Debugging Research at PELAB
(Needs a lot of compiler technology, integrated with compiler)

- Debugging of very high level languages: specification languages (RML), equation-based languages (Modelica)


The Task of the Compiler…

- Discover errors
- Report errors
- Restart parsing after errors, automatic recovery
- Correct errors on-the-fly if possible

Requirements on error management in the compiler

- Correct and meaningful error messages
- All static program errors (as defined by language) must be found
- Not to introduce any new errors
- Suppress code generation if error encountered

Handling Syntactic Errors
in the lexical analyser and parser

Local or Global Errors

- Lexical errors (local)
- Syntactic errors (local)
- Semantic errors (can be global)

Lexical and syntactic errors are local, i.e. you do not go backwards and forwards in the parse stack or in the token sequence to fix the error. The error is fixed where it occurs, locally.

Example; Global vs Local Correction

- Syntax errors are discovered (by the parser) when we cannot go from one configuration to another as decided by the stack contents and input plus parse tables (applies to bottom-up).
- LL- and LR-parsers have a valid prefix property i.e. discover the error when the substring being analyzed together with the next symbol do not form a prefix of the language.
- LL- and LR-parsers discover errors as early as a left-to-right parser can.
- Syntax errors rarely discovered by the lexical analyzer
  - E.g., “unterminated string constant; identifier too long, illegal identifier: 55ES
- Example. From PL/1 (where “=” is also used for assignment).
  A = B + C * D THEN . . . ELSE . . .

The error is discovered here, but the real error is here. “IF” is missing.

Two kinds of methods:
- Methods that assume a valid prefix (called phrase level in ASU).
- Methods that do not assume a valid prefix, but are based on a (mostly) valid prefix, are called global correction in ASU.

When is a Syntax Error Discovered?

- Syntax errors are discovered (by the parser) when we cannot go from one configuration to another as decided by the stack contents and input plus parse tables (applies to bottom-up).
- LL- and LR-parsers have a valid prefix property i.e. discover the error when the substring being analyzed together with the next symbol do not form a prefix of the language.
- LL- and LR-parsers discover errors as early as a left-to-right parser can.
- Syntax errors rarely discovered by the lexical analyzer
  - E.g., “unterminated string constant; identifier too long, illegal identifier: 55ES
- Example. From PL/1 (where “=” is also used for assignment).
  A = B + C * D THEN . . . ELSE . . .

The error is discovered here, but the real error is here. “IF” is missing.

Two kinds of methods:
- Methods that assume a valid prefix (called phrase level in ASU).
- Methods that do not assume a valid prefix, but are based on a (mostly) valid prefix, are called global correction in ASU.
Minimum Distance Error Correction

- Definition: The least number of operations (such as removal, inserting or replacing) which are needed to transform a string with syntax errors to a string without errors, is called the minimum distance (Hamming distance) between the strings.
- Example. Correct the string below using this principle.

\[ A = B + C \times D \text{ THEN ... ELSE ...} \]

Inserting \( :) \) is a minimum distance repair.

- This principle leads to a high level of inefficiency as you have to try all possibilities and choose the one with the least distance!

Parser-Defined Error Correction

- More efficient!
- Let \( G \) be a CFG and \( w = xty \) an incorrect string, i.e. \( w \notin L(G) \).

If \( x \) is a valid prefix while \( xt \) is not a valid prefix, \( t \) is called a parser-defined error.

Parser-defined error 1:
Change THEN to ;
\[ A = B + C \times D \text{ THEN ... ELSE ...} \]
Minimum distance repair:
Insert IF

Parser-defined error 2:
Change ELSE to ;

Some Methods for Syntax Error Management

- Panic mode (for LL parsing/recursive descent, or LR parsing)
- Coding error entries in the ACTION table (for LR parsing)
- Error productions for "typical" errors (LL and LR parsing)
- Language-independent methods
  - Continuation method, Röchrich (1980)
  - Automatic error recovery, Burke & Fisher (1982)

Panic Mode Recovery after a Syntax Error

- A wrong token \( c \) was found for current production \( A \rightarrow \beta \cdot b \gamma \)
- Skip input tokens until either
  - parsing can continue (find \( b \), or
  - a synchronizing token is found for the current production (e.g. \( \{ \), \}, while, if, ; ...)
    - tokens in FOLLOW(A) for current LHS nonterminal A
      - then pop A and continue
    - tokens in FOLLOW(B) for some LHS nonterminal B on the stack below A
      - then pop the stack until and including B, and continue
    - tokens in FIRST(A)
      - Then resume parsing by the matching production for A
- Further details: [ALSU06] 4.4.5

Systematic, easy to implement
- Does not require extra memory
- Much input can be removed
- Semantic information on stack is lost if popped for error recovery

Synchronization Points for Recovery after a Syntax Error

- input prefix parsed successfully
- \( A \rightarrow \beta \cdot b \gamma \) in FOLLOW(A) ?
Error Productions
- For "typical beginner"'s syntax errors
  - E.g. by former Pascal programmers changing to C
- Define "fake" productions that "allow" the error idiom:
  - E.g., `<id>` := `<expr>` similarly to `<id>` := `<expr>`
- Error message: "Syntax error in line 123, v := 17 should read v = 17?"

- very good error messages
- can easily repair the error
- difficult to foresee all such error idioms
- increases grammar size and thereby parser size

Define "fake" productions that "allow" the error idiom:
1. Construct a continuation `u`, `u` takes into consideration that `a` <expr> similarly to <id>
By looking in the tables you know what is allowed in `a`
E.g. by former Pascal programmers changing to C
2. Remove input symbols until an <expr> is reached
3. Insert parts of `u` after `x`, `E` takes care of the context
Error handling is generated automatically

Language-Independent Error Management Methods - "Röhrich Continuation Method"
- All information about a language is in the parse tables.
- By looking in the tables you know what is allowed in a configuration.
- Error handling is generated automatically
- Input: `w` rest of token sequence valid prefix parser-defined error

Röhrich Continuation Method (Cont.)
1. Construct a continuation `u`, `u` ∈ S*, and `w` = xu ∈ L(G).
2. Remove input symbols until an important symbol is found (anchor, beacon) e.g. WHILE, IF, REPEAT, begin etc.
   - In this case: then is removed as BEGIN is the anchor symbol.
3. Insert parts of `u` after `x`, and provide an error message.
   - "DO" expected instead of "THEN"
- "Röhrich Continuation Method"
  - + Language-independent
  - + Efficient
  - A valid prefix can not cause an error.
  - Much input can be thrown away.

Automatic Error Recovery, Burke & Fisher (2)
(PLDI Conference 1982)
- Takes into consideration that a valid prefix can be error-prone. Can also recover/correct such errors.
- Problem: you have to "back up"/undo the stack
- This works if information is still in the stack but this is not always the case!
Remember that information is popped from the stack at reductions.

Error Entries in the ACTION table (LR)
- Empty fields in the ACTION table (no transition in GOTO graph when seeing a token) correspond to syntax errors.
- LR Panic-mode recovery: Scan down the stack until a state `s` with a goto on a particular nonterminal `A` is found such that one of the next input symbols is in FOLLOW(`A`). Then push the state `GOTO(s, A)` and resume parsing from `a`.
- Eliminates the erroneous phrase (subexpr., stmt., block) completely.
- LR Phrase-level recovery: For typical error cases (e.g. semicolon before `else` in Pascal) define a special error transition with pointer to an error handling routine, called if the error is encountered
  - See example and [ALSU06] 4.8.3 for details
- Can provide very good error messages
- Difficult to foresee all possible cases
- Much coding
- Modifying the grammar means recoding the error entries
Phase 3 is called if phase 1 and 2 did not succeed in consuming some time and memory.

BEGIN (* fie *)
a = a + 1;
(* a so *)
Malformed declaration
END.

PROCEDURE fie(VAR i,j:INTEGER);
BEGIN)
END (* fie *);

Procedures cannot have types

Error spelling

Replaced '=' with a

PROCEDURE foo(VAR k:INTEGER) : BOOLEAN;

Similar to Phase 1

Inserted keyword end

A := B + C;

Deleted ','

Better!

The algorithm has three phases:

• 1. Simple error recovery
• 2. Scope recovery
• 3. Secondary recovery

Phase 1: Simple Error Recovery (a so-called token error)

• Removal of a token
• Insertion of a token
• Replace a token with something else
• Merging: Concatenate two adjacent tokens.
• Error spelling (BEGIN → BEGIN)

Phase 2: Scope Recovery

Insertion of several tokens to switch off open scope.

Example Test Program for Error Recovery

PROGRAM scoptest(input,output);
REPEAT
- 1. Simple error recovery
- 2. Scope recovery
- 3. Secondary recovery

OPEN tokens to switch on panic mode.
Phase 3 is called if phase 1 and 2 did not succeed in putting the parser back on track.

Summary "Automatic error recovery", Burke & Fisher

• Language-independent, general
• Provides very good error messages
• Able to make modifications to the parse stack
• Consumes some time and memory.

Automatic Error Recovery, Burke & Fisher (2)

Automatic Error Recovery, Burke & Fisher (3)

Program closer and closer

Example Test Program for Error Recovery

PROGRAM closer(input,output);
BEGIN
END.

PROGRAM closer(input,output);
BEGIN
END.

PROGRAM closer(input,output);
BEGIN
END.

PROGRAM closer(input,output);
BEGIN
END.

Summary "Automatic error recovery", Burke & Fisher

• Language-independent, general
• Provides very good error messages
• Able to make modifications to the parse stack
• Consumes some time and memory.

Automatic Error Recovery, Burke & Fisher (2)

Automatic Error Recovery, Burke & Fisher (3)

Example Test Program for Error Recovery

PROGRAM closer(input,output);
BEGIN
END.

PROGRAM closer(input,output);
BEGIN
END.

PROGRAM closer(input,output);
BEGIN
END.

PROGRAM closer(input,output);
BEGIN
END.

Error Messages from Old Hedrick Pascal - Bad!

Error Messages from Old Sun Pascal - Better!
Error Messages from Burke & Fisher’s “Automatic Error Recovery” – Best!

1  PROGRAM scoptest(input, output);
   ^^^^^^^^^
   *** Lexical Error: Reserved word “PROGRAM” misspelled

3  CONST mxdlen = 10
   ^^ ^^^
   *** Lexical Error: “MXIDLEN” expected instead of “MXI” “DLEN”

3  CONST mxdlen = 10
   ^^
   *** Syntax Error: “;” expected after this token

5  VAR a, b, c; d: INTEGER;
   ^
   *** Syntax Error: “,” expected instead of “;”

7      arr10 : ARRAY [1..mxidlen];
   ^
   *** Syntax Error: “OF IDENTIFIER” inserted to match “ARRAY”

10     PROCEDURE foo(VAR k: INTEGER): BOOLEAN;
   ^^^^^^^^^
   *** Syntax Error: “FUNCTION” expected instead of “PROCEDURE”

12     VAR i, : INTEGER;
   ^
   *** Syntax Error: “IDENTIFIER” expected before this token

14     BEGIN )* foo )* 
   <-------
   *** Syntax Error: Unexpected input

20           IF (a > b) THEN a:= b; ELSE b:=a;
   ^
   *** Syntax Error: “UNTIL IDENTIFIER” inserted to match “REPEAT”
   *** Syntax Error: “END” inserted to match “BEGIN”

26       a = a + 1;
   ^

Semantic Errors

- Can be global (needs not be tied to a specific code location or nesting level)
- Do not affect the parsing progress
- Usually hard to recover automatically
  - May e.g. automatically declare an undeclared identifier with a default type (int) in the current local scope – but this may lead to further semantic errors later
  - May e.g. automatically insert a missing type conversion
  - May e.g. try to derive the type of a variable which is not declared (exist type inference algorithms)
- Usually handled ad-hoc in the semantic actions / frontend code

Exception handling

Concept and Implementation

Exception Concept

- PL/I (IBM) ca. 1965: ON condition …
- Supported in many modern programming languages
  - CLU, Ada, Modula-3, ML, C++, Java, C#, Modelica

Overview:
- Terminology: Error vs. Exception
- Exception Propagation
- Checked vs. Unchecked Exceptions
- Implementation

Exception Concept

2 sorts of run-time errors:
- Error: cannot be handled by application program – terminate execution
- Exception: may be handled by the program itself
  - Triggered (thrown) by run-time system when recognizing a run-time error, or by the program itself
  - Message (signal) to the program
  - Run-time object defining an uncommon or error situation
    - has a type (Exception class)
    - May have parameters, e.g. a string with clear-text error message
    - Also user-defined exceptions e.g. for boundary cases
- Exception Handler:
  - Contains a code block for treatment
  - is statically associated with the monitored code block, which it replaces in the case of an exception
When Does a Handler "match"?

- Exception Class Hierarchy
- User-defined exceptions by subclassing
- Handler `catch` (XYException e) {...} matches, if XYException is of the same type or a supertype of the thrown exception.

Implementation

```
void bar(...) { 
  try { 
    catch(E1 e) {...} 
    catch(E2 e) {...} 
  } 
  System.out.println("Hello, " + e);
  System.out.println("Please provide an argument! "+ e);
  System.out.println("Goodbye");
}
```

Propagating Exceptions

- If an exception is not handled in the current method, program control returns from the method and triggers the same exception to the caller. This schema will repeat until either:
  - a matching handler is found, or
  - main() is left (then error message and program termination).
- Optional finally-block will always be executed, though.
  - E.g. for releasing of allocated resources or held locks

To be determined:
- When does a handler match?
- How can we guarantee statically that a certain exception is eventually handled within the program?
- Implementation?

Checked and Unchecked Exceptions

- Checked Exception: must be
  - Treated in a method, or
  - Explicitly declared in method declaration as propagated exception: `void writeEntry(...)` throws `IOException {...}`
- Unchecked Exception: will be propagated implicitly
  - In Java: All Exceptions are checked, except RunTimeException and its subtypes.

- Checked Exceptions:
  - Encapsulation
  - Consistency can be checked statically
  - become part of the contract of the method's class/interface
  - suitable for component systems, e.g. CORBA (→ TDDC18)
  - Extensibility

Exceptions: Summary, Literature

- Exceptions
  - Well-proven concept for treatment of run-time errors
  - Efficiently implementable
  - Suitable for component based software development

Section 8.5 about Exception Handling.

J. Goodenough: Structured Exception Handling. ACM POPL, Jan. 1975

