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**

Christoph Kessler, IDA,
Linköpings universitet, 2007.

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 yearly ~ 60 Mrd. $
- **What error types can occur?**
  - Classification
- **Prevention, Diagnosis, Treatment**
  - Programming language concepts
  - Compiler, IDE, Run-time support
  - Other tools: Debugger, Verifier, ...

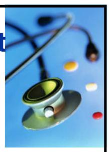
Classification of program errors (1)

- **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
  - **Semantic Error**
    - Static type error — Wrong parameter number or type; Downcast without run-time check
    - Undeclared variable
    - Use of uninitialized variable
    - Static overflow — Constant too large for target format
- **Compiler Errors** — Symbol table / constant table / string table / type table overflow

Classification of program errors (2)

- **Run-time errors** – usually not checkable statically
  - Memory access error — e.g.:
    - Array index error — Index out of bounds
    - 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
  - Ressource 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 [Fagan'76] → All error types
- Testing and Debugging → Run-time errors
- Runtime protection monitor → Access errors
- Trace Visualiser → Communication errors, Synchronisation errors

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 meaningfull 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

TDDB29 Compilers and Interpreters

TDDB44 Compiler Construction

Handling Syntactic Errors

in the lexical analyser and parser

Christoph Kessler, IDA,
Linköpings universitet, 2007.

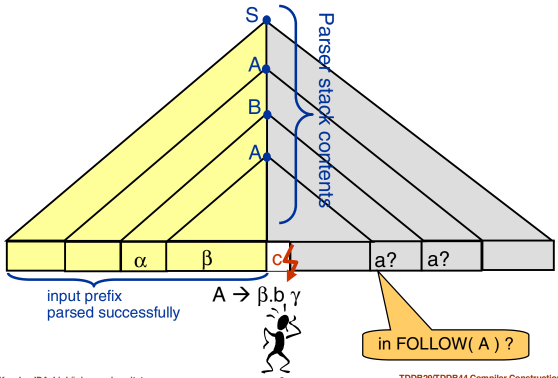
Syntax errors

- Discovered rarely by the lexical analyzer
  - E.g., ”unterminated string constant; identifier too long
- Mostly in the parser
- Usually local errors
  → should be handled locally by lexical analyser or parser
- LL and LR parsers have the **viable prefix property**,
  i.e. discover an error as soon as the substring being analysed together with the next input symbol does not form a viable prefix of the language.

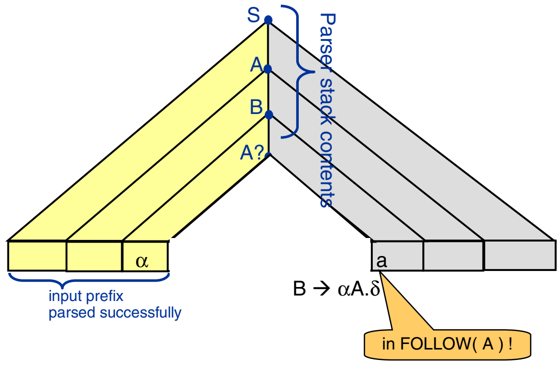
Methods for syntax error management:

- Panic mode (for LL parsing)
- Coding error entries in the ACTION table (for LR parsing)
- Error productions for ”typical” errors (LL and LR parsing)

Synchronization points for recovery after a syntax error



Panic mode recovery after a syntax error



Panic mode (for predictive (LL) parsing)

- A wrong token c was found for current production A → β . b γ
- 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

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

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 a 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

Example: LR Phrase-level Recovery

0. S' -> L |--
1. L -> L , E
2. | E
3. E -> a
4. | b

ACTION table:

state	|--	,	a	b
0	E1	E2	S4	S5
1	A	S2	E4	E4
2	E1	E3	S4	S5
3	R1	R1	E5	E5
4	R3	R3	E6	E6
5	R4	R4	E6	E6
6	R2	R2	E5	E5

GOTO table:

state	L	E
0	1	6
1	*	*
2	*	3
3	*	*
4	*	*
5	*	*
6	*	*

Error handling routines
triggered by new ACTION table error transitions:

E1: errmsg(”Found EOF where element expected”);
push state 3 = the GOTO target of finding (fictitious) E

E2: errmsg(”No leading comma”); read the comma away and stay in state 0

E3: errmsg(”Duplicate comma”); read the comma away and stay in state 2

E4: errmsg(”Missing comma between elements”);
push state 2 (pretend to have seen and shifted a comma)

E5: errmsg(”Missing comma”); reduce + push state 1 as if seeing the comma

E6: errmsg(”Missing comma”); reduce + push state 3 as if seeing the comma

Error productions in Yacc

- Extend grammar with error productions of the form
  A ::= **`error`** α
  which correspond to most common errors $A \rightarrow \alpha$
  `error`: fictitious token, reserved keyword in Yacc
  - Example: <stmt> ::= **`error`** <id> := <expr>

Panic mode for LR parsing

- When an error occurs:
  - Pop stack elements until the state on top of the stack has an item of the form [$A \rightarrow$. **`error`** α] in its item set
  - Shift **`error`** in as a token
  - If α is ε, reduce using semantic action for this rule:
    A ::= **`error`** ε { **printf**(”Error: …”); }
  - Otherwise, skip tokens until a string derivable from α is found, and reduce for this rule:
    A ::= **`error`** α { **printf**(”Error, continued from α”); }
- **Example**: A ::= **`error`** ; { **printf**(”Error, continued from semicolon”); }

Handling Semantic Errors

in the compiler front end

Christoph Kessler, IDA,
Linköpings universitet, 2007.

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
- Usually handled ad-hoc in the semantic actions / frontend code

Exception handling

Concept and Implementation

Christoph Kessler, IDA,
Linköpings universitet, 2007.

Exception Concept

- PL/I (IBM) ca. 1965: **ON** *condition …*
- J. B. Goodenough, POPL'1975 und *Comm. ACM* Dec. 1975
- Supported in many modern programming languages
  - CLU, Ada, Modula-3, ML, C++, Java, C#
- **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
  - Run-time object defi … 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

Exception Example (in Java)

```
public class class1 {
   public static void main ( String[] args ) {
        try {
           System.out.println("Hello, " + args[0] );
        }
        catch (ArrayIndexOutOfBoundsException e ) {
           System.out.println("Please provide an argument! " + e);
        }
        System.out.println("Goodbye");
   }
}
```

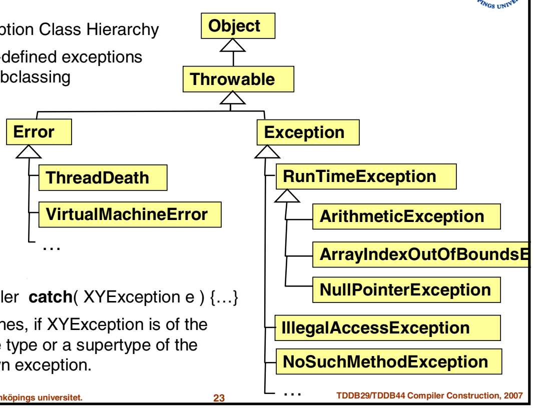
```
% java class1 Christoph
Hello, Christoph
Goodbye
% java class1
Please provide an argument! java.lang.ArrayIndexOutOfBoundsException: 0
Goodbye
% java class1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
at class1.main(class1.java:4)
```


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?

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.

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 und 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

Implementation

```
void bar(…) {
  try { … }
  catch(E1 e) {…}
  catch(E2 e) {…}
  …
}
```

	-> catch(E1)
	-> catch(E2)
fp(bar):	AR(bar)
	-> catch(E2)
	-> catch(…)
fp(foo):	AR(foo)
main	AR(main)

Simple solution:

- Stack of handlers
- When entering a monitored block (**try** {…}):
  - Push all its handlers (**catch**(…) {…})
- When an exception occurs:
  - Pop topmost handler and start (test of exception type).
    If it does not match, re-throw and repeat.
    (If the last handler in current methode did not match either, pop also the method's activation record → exit method.)
- If leaving the try-block normally: pop its handlers

☺ simple

☹ Overhead (push/pop) also if no exception occurs

More efficient solution:

- Compiler generates table of pairs (try-block, matching handler)
  - When exception occurs: find try-block by binary search (PC)

Exceptions: Summary, Literature

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

M. Scott: *Programming Language Pragmatics*. Morgan Kaufmann, 2000. Section 8.5 about Exception Handling.

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

J. Goodenough: Exception Handling: Issues and a proposed notation. *Communications of the ACM*, Dec. 1975

B. Ryder, M. Soffa: Influences on the Design of Exception Handling, 2003