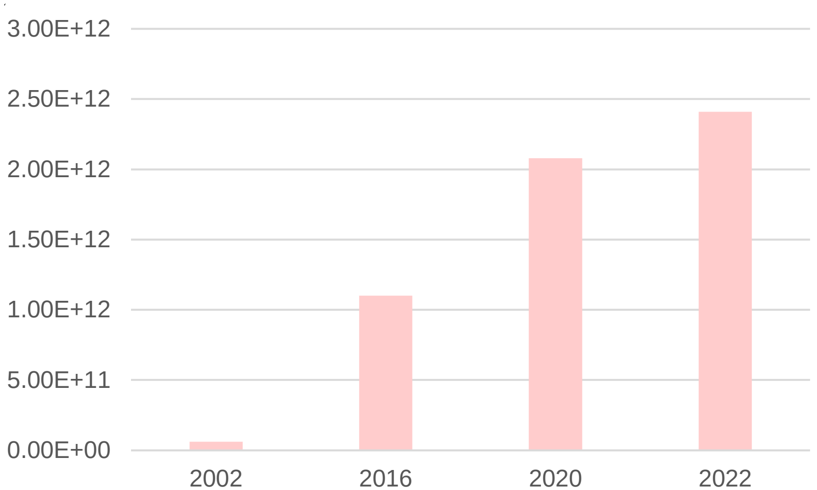
Error Management in Compilers and Run-time Systems

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

Adrian Pop, Martin Sjölund,
Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2023.

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 ~$60 billion yearly
  - 2016 – Jumped to ~$1.1 trillion
  - 2020 – Poor software quality cost US companies ~$2.08 trillion
  - 2022 – Software Quality Issues in the US cost ~$2.41 trillion
- **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**
    - 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 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	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
Synchronization 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 [Fagan'76] → all error types
- Testing, Debugging, Static Analysis → run-time errors
- Runtime protection monitor → access errors
- Trace Visualizer → communication errors, synchronization errors

Some Debugging Research at PELAB
(Needs a lot of compiler technology, integrated with compiler)

LINKÖPINGS UNIVERSITET

- High-Level Host-Target Embedded System Debugging
  - Peter Fritzson: Symbolic Debugging through Incremental Compilation in an Integrated Environment. The *Journal of Systems and Software* 3, 285-294, (1983)

- Semi-automatic debugging – automatic bug localization by automatic comparison with a specification /or using oracle
  - Peter Fritzson, Nahid Shahmehri, Mariam Kamkar, Tibor Gyimothy: Generalized Algorithmic Debugging and Testing. In *ACM LOPLAS - Letters of Programming Languages and Systems*, Vol 1, No 4, Dec 1992.
  - Henrik Nilsson, Peter Fritzson: Declarative Algorithmic Debugging for Lazy Functional Languages. In *Journal of Functional Programming*, 4(3):337 - 370, July 1994.

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)
  - Adrian Pop and Peter Fritzson. An Eclipse-based Integrated Environment for Developing Executable Structural Operational Semantics Specifications. *Electronic Notes in Theoretical Computer Science* (ENTCS), Vol 175, pp 71–75. ISSN:1571-0661. May 2007.
  - Adrian Pop (June 5, 2008). Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages. Linköping Studies in Science and Technology, Dissertation No. 1183.
  - Martin Sjölund. *Tools for Understanding, Debugging, and Simulation Performance Improvement of Equation-Based Models*. Licentiate thesis No 1592, Linköping University, Department of Computer and Information Science, April 2013
  - Adrian Pop, Martin Sjölund, Adeel Ashgar, Peter Fritzson, and Francesco Casella. Integrated Debugging of Modelica Models. *Modeling, Identification and Control*, 35(2):93-107, 2014

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

TDDD55 Compilers and Interpreters

TDDB44 Compiler Construction

Handling Syntactic Errors

in the lexical analyser and parser

Adrian Pop, Martin Sjölund,
Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2023.

Local or Global Errors

- Lexical errors (local - usually)
- 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.

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; Global vs Local Correction

- Example. From PL/1 (where "=" is also used for assigment).

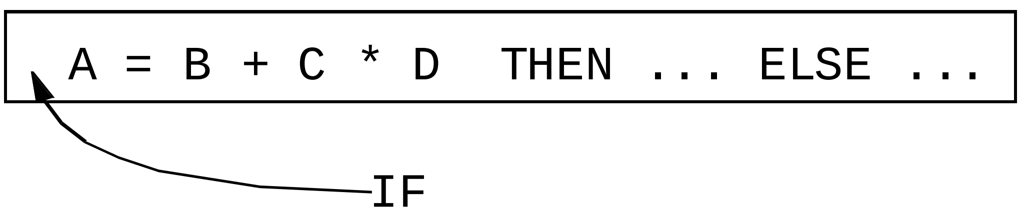
```
A = B + C * D THEN . . . ELSE . . .
```

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.



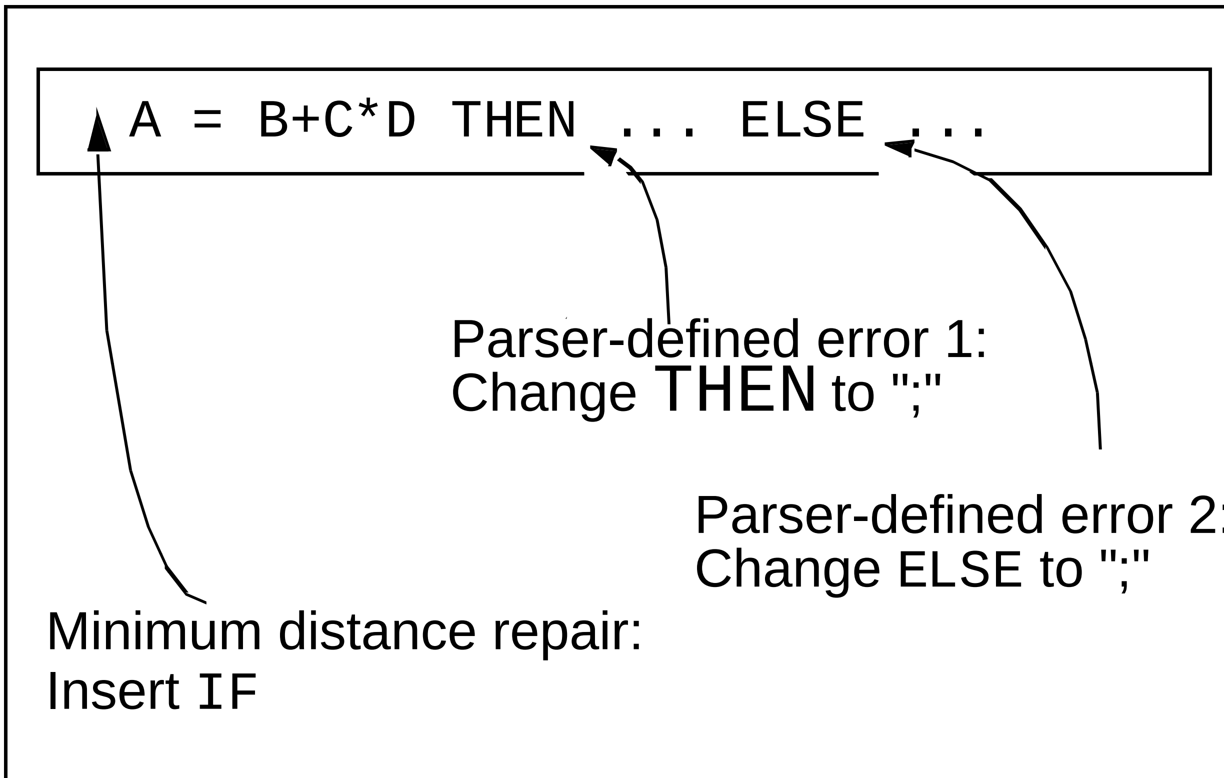
Inserting `IF` is a *minimum distance repair*.

- 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. $w \notin L(G)$.

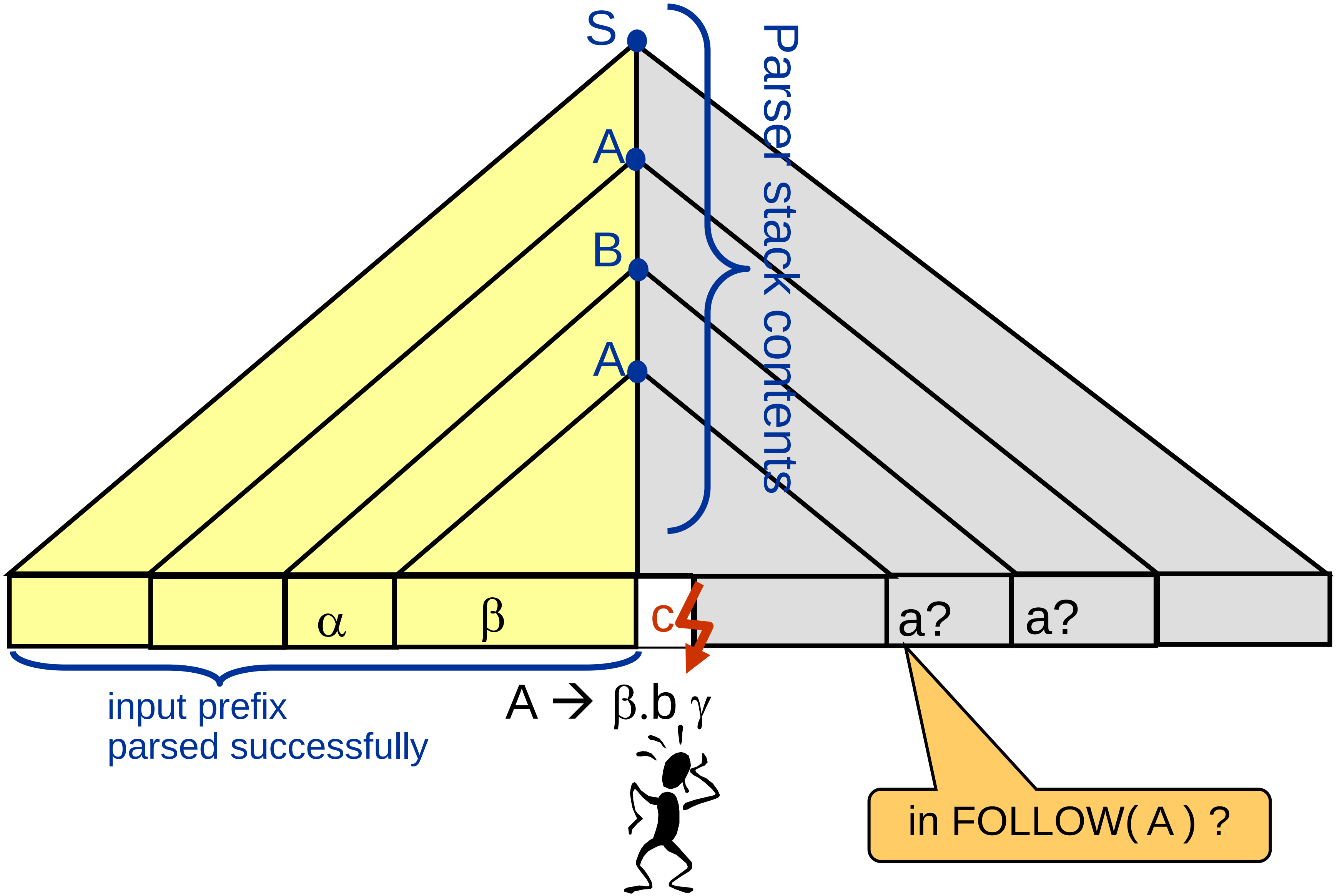
  If *x* is a valid prefix while *xt* is not a valid prefix, *t* is called a parser defined error.



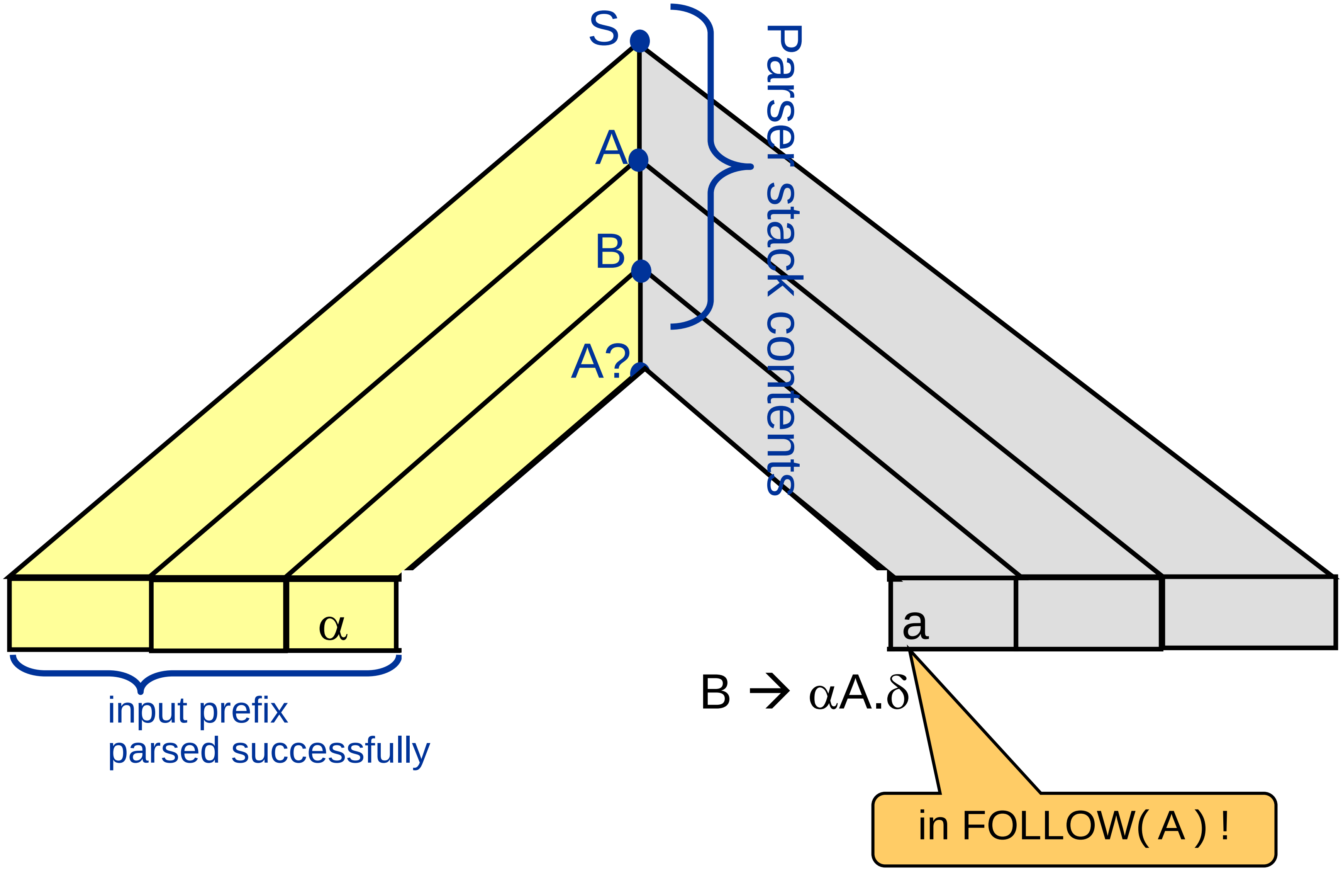
Some Methods for Syntax Error Management

- 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, LR, Any parsers)
- Language-independent methods
  - Continuation method, Röchrich (1980)
  - Automatic error recovery, Burke & Fisher (1982)

Synchronization Points for Recovery after a Syntax Error



Panic Mode Recovery after a Syntax Error



Panic mode (for predictive (LL) parsing)

LINKÖPINGS UNIVERSITET

- A wrong token c was found for current production $A \rightarrow \beta \,.\, b\, \gamma$
- 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

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 , M
2. | M
3. M -> 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	M
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) M

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 → α

  error: fictitious token, reserved keyword in Yacc

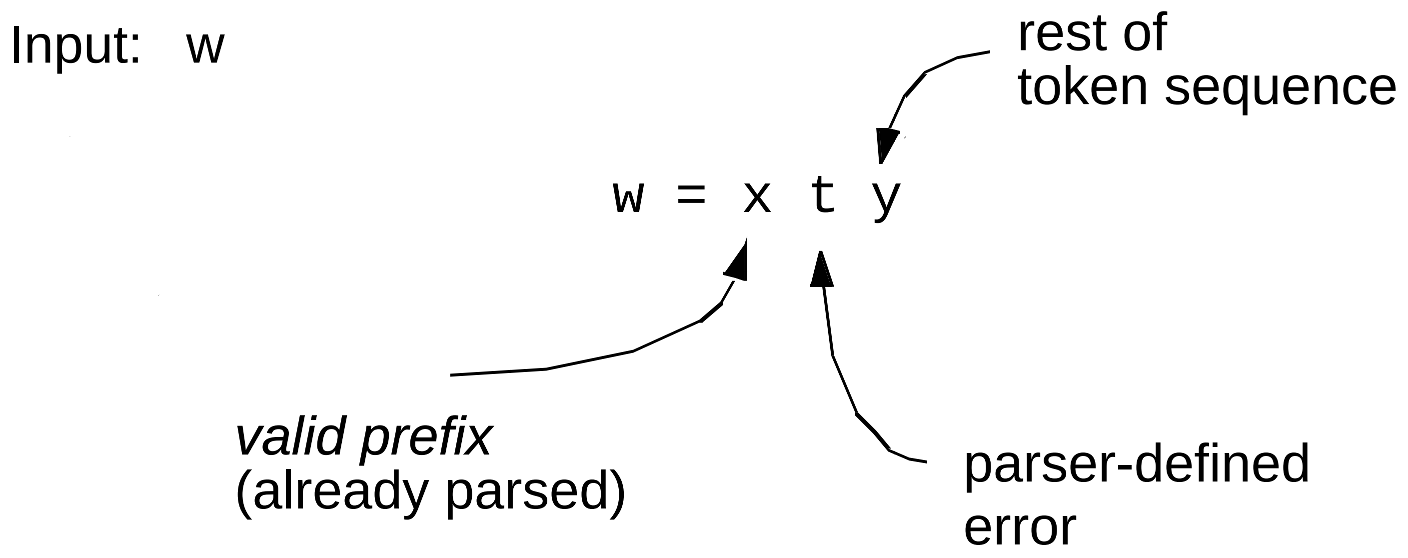
  - Example: <stmt> ::= **error** <id> := <expr>

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 → **. 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"); }

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



Röhrich Continuation Method (Cont.)

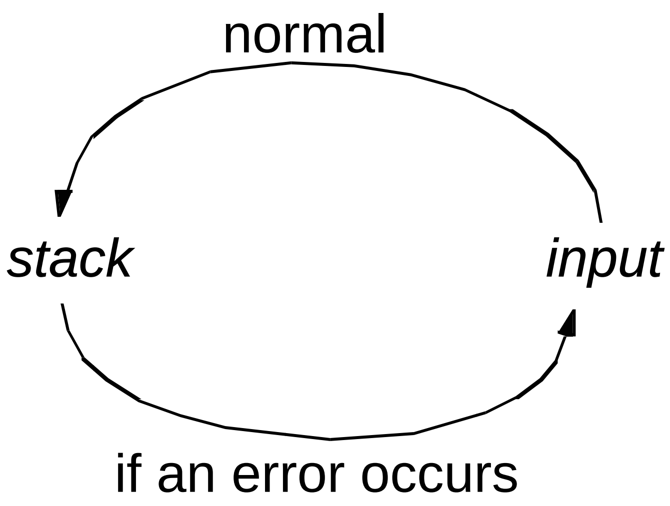
- 1. Construct a continuation u, $u \in S^*$, and $w' = xu \in L(G)$.
- 2. Remove input symbols until an *important* symbol is found (*anchor, beacon*) e.g. WHILE, IF, REPEAT, BEGIN etc.
  - In this case: **then** is removed as **BEGIN** is the anchor symbol.
- 3. Insert parts of u after x, and provide 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.

```
program foo;
begin
    while a > b then begin
                     ↑
                     Parser-defined error
    end
end;
```

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.



Automatic Error Recovery, Burke & Fisher (2)

- 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
    (BEGNI $\rightarrow$ BEGIN)

Automatic Error Recovery, Burke & Fisher (3)

- **Phase 2**:
  Scope Recovery

  Insertion of several tokens to switch off *open scope*.

Opener	*Closer*	
PROGRAM	BEGIN	END.
	.	
PROCEDURE	BEGIN	END;
	;	
BEGIN	END	
(	)	
[	]	
REPEAT	UNTIL	*identifier*;
	UNTIL	*identifier*
ARRAY	OF	*identifier*;
	OF	*identifier*

Automatic Error Recovery, Burke & Fisher (2)

- **Phase 3**: *Secondary recovery*
  - Similar to *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 (by ’’backing up’’ the stack)
  - - Consumes some time and memory.

Example Test Program for Error Recovery

```
PROGRRAM scoptest(input, output);
CONST mxi dlen = 10
VAR a,b,c;d :INTEGER;
    arr10 : ARRAY [1..mxidlen]  ;

   PROCEDURE foo(VAR k:INTEGER) : BOOLEAN;
   VAR i, : INTEGER;
   BEGIN )* foo *)
      REPEAT
         a:= (a + c);
         IF (a > b) THEN a:= b ; ELSE b:=a;

  PROCEDURE fie(VAR i,j:INTEGER);
  BEGIN (* fie *)
     a = a + 1;
  END (* fie *);


  A := B + C;
END.
```

Error Messages from Old Hedrick Pascal - Bad!

```
 1   PROGRRAM scoptest(input, output);
P*  1**        ^         **************^
1.^:  "BEGIN" expected
2.^:  ":=" expected


    3   CONST mxi dlen = 10
P*  1**     ^        ^  **
1.^:  "END" expected
2.^:  "=" expected
2.^:  Identifier not declared


    5   VAR a,b,c;d :INTEGER;
P*  1**   ^      ^
1.^:  ";" expected
2.^:  Can't have that here (or something
extra or missing before)
2.^:  ":" expected


    7       arr10 : ARRAY [1..mxidlen]  ;
P*  1**                               ^^  ^
1.^:  Identifier not declared
2.^:  Incompatible subrange types
3.^:  "OF" expected
```

```
10      PROCEDURE foo(VAR k:INTEGER) :
BOOLEAN;
P*  1**
^********
1.^:  Can't have that here (or something
extra or missing before)


   12       VAR i, : INTEGER;
P*  1**           ^
1.^:  Identifier expected


   14       BEGIN )* foo *)
P*  1**          ^*******
1.^:  Can't have that here (or something
extra or missing before)


   20           IF (a > b) THEN a:= b ;
ELSE b:=a;
P*  1**
^*****
1.^:  ELSE not within an IF-THEN (extra
";","END",etc. before it?)


   22     PROCEDURE fie(VAR i,j:INTEGER);
P*  1**           ^
```

Error Messages from Old Sun Pascal - Better!

```
 1  PROGRAM scoptest(input,output);
e ------^--- Inserted '['
E -------------------------------------^---
Expected ']'

     3  CONST mxi dlen = 10
e ---------------^--- Deleted identifier

     5  VAR a,b,c;d :INTEGER;
e ------^--- Inserted ';'
e ---------------^--- Replaced ';' with a
','

     7      arr10 : ARRAY [1..mxidlen] ;
E -------------------------------------^-
Expected keyword of
E -------------------------------------^-
Inserted identifier

   PROCEDURE foo(VAR k:INTEGER) : BOOLEAN;
E--------- Procedures cannot have types

    12     VAR i, : INTEGER;
E --------------^--- Deleted ','
```

```
 14      BEGIN )* foo *)
E ---------------^--- Malformed statement
    20            IF (a > b) THEN a:= b ;
ELSE b:=a;
e -------------------------------------^---
Deleted ';'
before keyword else

    22    PROCEDURE fie(VAR i,j:INTEGER);
E --------^--- Expected keyword until
E --------^--- Expected keyword end
E --------^--- Inserted keyword end
matching begin on line 14
e --------^--- Inserted ';'
    26       a = a + 1;
e ------------^--- Replaced '=' with a
keyword (null)

    32    A := B + C;
e --------^--- Inserted keyword (null)
    34  END.
E ------^--- Malformed declaration
E ------^--- Unrecoverable syntax error -
QUIT
```

Error Messages from Burke & Fisher's "Automatic Error Recovery" – Best!

LINKÖPINGS UNIVERSITET

```
 1  PROGRRAM scoptest(input,output);
      ^^^^^^^^
*** Lexical Error: Reserved word "PROGRAM"
misspelled

   3  CONST mxi dlen = 10
            ^^^ ^^^
*** Lexical Error: "MXIDLEN" expected
instead of "MXI"  "DLEN"

   3  CONST mxi dlen = 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: Unexpected ";" , ignored

  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;
              ^
```

Handling Semantic Errors

in the compiler front end

Adrian Pop, Martin Sjölund,
Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2023.

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 (some type inference algorithms exist)
- Usually handled ad-hoc in the semantic actions / frontend code

Exception handling

Concept and Implementation

Adrian Pop, Martin Sjölund,
Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2023.

Exception Concept

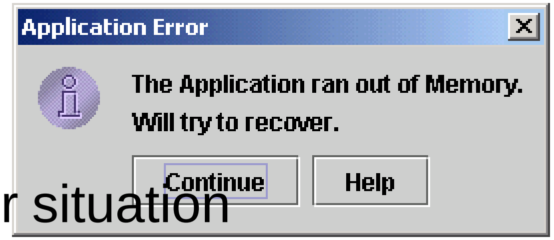
- PL/I (IBM) ca. 1965: **ON** *condition* …
- J. B. Goodenough, POPL'1975 and *Comm. ACM* Dec. 1975
- Supported in many modern programming languages
  - CLU, Ada, Modula-3, ML, C++, Java, C#, MetaModelica

- **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 issue, 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



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

% java class1

Please provide an argument! java.lang.ArrayIndexOutOfBoundsException

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0

Goodbye

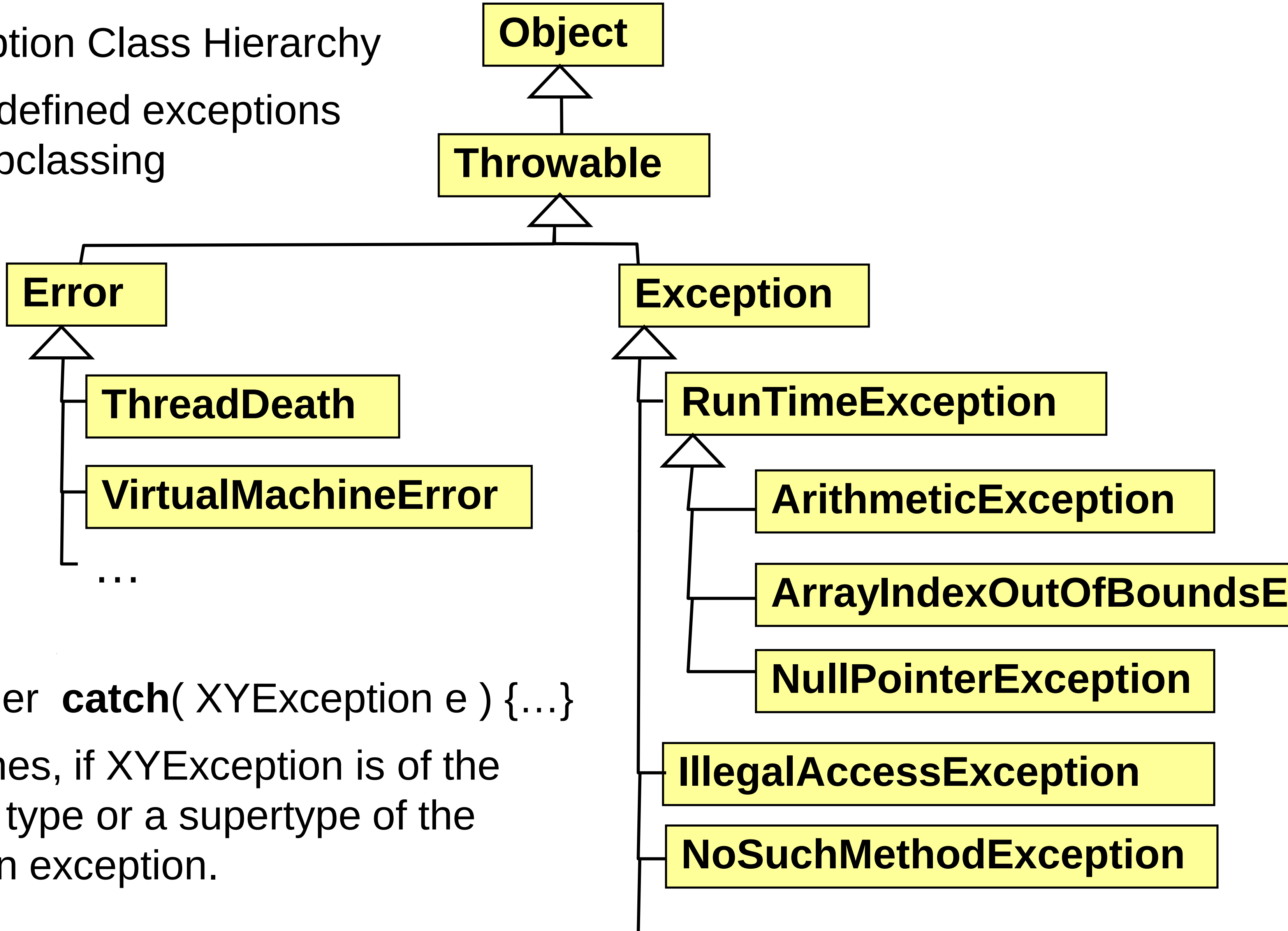
at class1.main(class1.java:4)

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 { … }

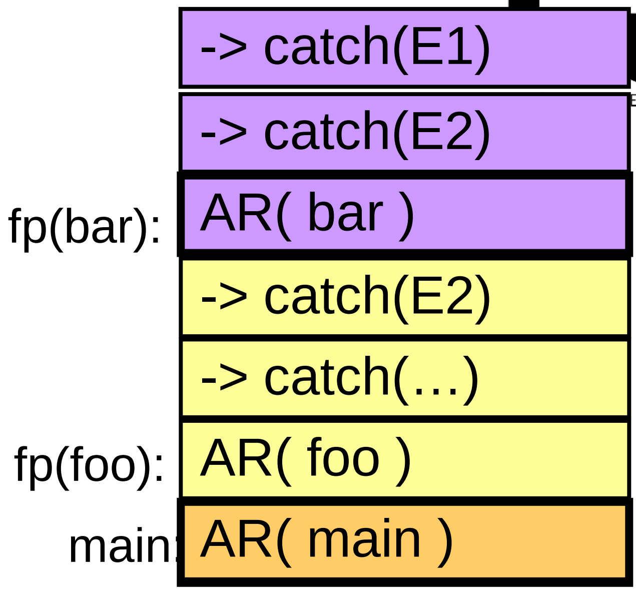
- **Unchecked Exception**: will be propagated implicitly

- In Java: All Exceptions are checked,
  except RunTimeException 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

Implementation

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



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 method 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

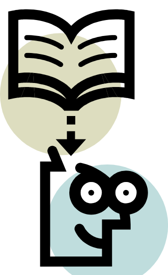
More efficient solution:

- 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

Adrian Pop, Kristian Stavåker, and Peter Fritzson. Exception Handling for Modelica. In Proceedings of the 6th International Modelica Conference (Modelica'2008), Bielefeld, Germany, March.3-4, 2008

Interpreters

Adrian Pop, Martin Sjölund,
Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2023.

Direct Interpretation

- Given the program source code and the run-time input
- Interpret the source code directly,
  i.e. parse and simulate it, statement by statement
  (syntax-directed interpretation)
  - UNIX shells (command line interpreter)
  - Early interpreters for BASIC, LISP, APL
- Symbol table
  - contains also storage for run-time values of program variables
- Full information about source-level program entities
  - Good for debugging
- Very slow
  - But ok for small scripts

Hybrid Compiler/Interpreter Scenario

Step 1:

- Translate the source program to an internal form
  - E.g. quadruples, postfix, abstract syntax tree
- Or to instructions for an abstract machine
  - E.g. P-code for Pascal and Modula-2, Diana for Ada, JVM bytecode for Java, CIL for C#/.NET

Step 2:

- Execute the interpreter
  - given the internal form / abstract machine program
  - simulate the abstract machine step by step

☺ More efficient than direct interpretation, but

☹ still much slower than compiled code, typ. by a factor ~10 to ~100

☺ Still portable – intermediate form is not processor specific

☺ ☹ Source code cannot be reconstructed completely from intermediate form

☺ Can be stored compactly

☺ Easy to write an interpreter (virtual machine)

Example: JVM Bytecode

- Instructions for the **JVM (Java Virtual Machine)**, an abstract stack machine
  - Executes .class or .jar files (loaded when first referenced)
    - Heap of loaded classes (program text and static data)
  - Program counter PC
  - Bytecode instructions (postfix order) have 1 byte opcode with 0 or 1 operand
    - span 1 or more bytes, depending on operand size
  - Run-time stack: Frame pointer fp, Stack pointer sp

☺ Could even be implemented in hardware (e.g. Sun MAJC)

JVM Bytecode Interpretation

JVM Instruction (examples)	Interpretation (by C code)	Stack top before	Stack top afterwards
iconst_0	Stack[sp++] = 0; PC++; // code needs 1 byte	() = don't care	(I) = int-value
istore *v*	Stack[fp + *v*] = Stack[--sp]; PC += 2; // needs 2 bytes	(I)	()
iload *v*	Stack[sp++] = Stack[fp + *v*]; PC += 2;	()	(I)
iadd	Stack[sp-1] = Stack[sp] + Stack[sp-1]; sp--; PC++;	(I, I)	(I)
goto *a*	PC = *a*;	()	()
ifeq *a*	if (Stack[sp--] == 0) PC = *a*; else PC += 3;	(I)	()

Just-In-Time (JIT) Compiling

- A.k.a. **dynamic translation**
- Program execution starts in interpreter as before
- Whenever control flow enters a new **unit** of bytecode (unit could be e.g. a class file, a function, a loop, or a basic block):
  - Do not interpret it, but call the JIT compiler that translates it to target code and replaces the unit with a branch to the new target code
- JIT compiling overhead → delay at run-time
  - paid once per unit (if code can be kept in memory)
  - pays often only off if translated code is executed several times (e.g., a loop body)
    - Can also be done lazily: Interpret the unit when executed for the first time. When re-entering the unit, JIT-compile.
    - Or pre-compile/pre-JIT to native code ahead of time
  - Trade-off:
    JIT-generated code quality vs. JIT compiler speed (run-time delay)

Just-In-Time (JIT) Compiling (cont.)

- Typically, performance boost by at least one order of magnitude
- Still somewhat slower,
  but may even be faster than statically compiled code in some cases
  - Can use on-line information from performance counters (e.g. #cache misses) for dynamic re-optimization and memory re-layout
- Example for Java: Sun JDK HotSpot JVM;
  for C#: .NET CLR, NGEN

Thank you!

- Any questions?
- Next week
  - L14 – Compiler frameworks & Bootstrapping
  - TDDB44 & TDDD55
    - Last Seminar: Exam preparation