Introduction

Need to report useful error messages when parsing.

Consider parsing C++.

Parser Error Handling:

- Detecting, reporting, and recovering from parsing errors.
- Problem: inner details of grammars are complicated.
  Grammar often bears little resemblance to end-language user perceives.

Parser Exception Handling: History, Philosophy and Design

Terence Parr
Parr Research Corp
San Fransisco, CA
parrt@parr-research.com

Russell W. Quong
Purdue University
W. Lafayette, IN 47906
quong@ecn.purdue.edu

- Introduction
- History
- Concepts
- Issues

History of Error facilities in ANTLR

(As I recall it at a doggy 2:15 AM).

- TP: Automatic, simple “found X but expected Y” messages.
  Often not terribly clear. Limited usefulness.
- TP: Improved messages/recovery via error classes.
  Improved messages. Simple but good recovery heuristic. Automatic use of FIRST FOLLOW sets. In retrospect, quite nifty.
- TP: Still looking for better mechanisms.
- RQ: Makes passing mention to C++ exception handling.
- (Later) TP: (working with NeXT) Considers throw/catch paradigm. Hears about C++ exceptions. Likes idea. Tries to implement it.
- RQ & TP: Numerous discussions follow on implementation, semantics, syntax.
- TP: Does preliminary implemenation. Goes nuts. Devises notion of local exception handlers for labelled items and specific alternatives.

Error reporting wish list

* = poor $\Longrightarrow$ **** = great;

Feature	$LR()$	1992 ANTLR	P E H
Parsing context is available	*	*	***
Simple to use	*	**	****
Error handling separate from grammar	**	**	****
Automatic error messages	*	**	***
Industrial power if desired	*	**	****
Error recovery	**	**	***
Works with C and C++	***	***	***
Does not interfere with predicates	NA	NA	***

Exception = a parsing error. Has state: [type, location, details] of error.

Handler = code to process/recover an exception of a specific type.

Handler stack = runtime stack of handlers.

- *Throw* an exception of type T_E, which is then *caught* by a handler.
- If a rule r has a handler H_E, push H_E onto the handler-stack when parsing r. Pop H_E when done with r.
- Search handler stack. Handler nearest the top of the stack compatible with T_E gets the nod.
- If recursive calls, might have recursive handlers. Handler can (i) execute an action and/or (ii) re-throw an exception to a higher-up handler. Can throw an exception of a different type (!)

Exception Handling

- ANTLR = recursive-descent parsers
- Parser exception similar to handling an error in recursive code.
- Can detect errors deep in the bowels of a grammar.
  Conceptually want to unwind the function-call stack until reaching a high-level rule.

Note: Getting context for LR parsers is much harder: table driven, do not know where we are.

Parser Exception Handling (PEH)

Consider parsing typedefs in C/C++.

Specfied type can be complicated (e.g, function pointers).

Bowels = grammar for type declaration code.

```
typedef
  -> "typedef" t_label:type-decl name ´;´
   | "typedef" type-decl-fn-ptr ´;´
   ;
   exception
     catch MismatchedToken :
       << print "Syntax error in typedef"; >>
   exception [t_label]
     catch MismatchedToken :
       << print "Syntax error in type spec for typedef"; >>
```

Recursive-descent code. Rule r has a handler H_E.

```
void r () {
  push H_E onto handler stack;
  normal recursive-descent code
       ⋮
  pop H_E from handler stack;
}
```

Issues

- Can we get an exception in a syntactic predicate? No.
- Can we get an exception in a semantic predicate? No.
- Do we distinguish between syntactic and semantic errors? Yes.
- Do we distinguish between semantic disambiguation and validation errors? No.
- Can we use C++ exceptions to implement ANTLR exceptions? YES! But alas, “No,” not if we generate C code. Also, C++ exception not available yet.
- What if parser has C++ class objects as local variables, whose destructors must be called? Ugh. (Lots of discussion). Will unwind the call stack “manually” via return values.

Want good default PEH mechanism

- Default handlers supplied for all built-in exception types.
- Sits at bottom of handler stack, but . . .
- Default handler is called immediately; does not unwind the recursive call stack, otherwise unwind to start rule (blown out of the water).
- Must search handler stack before unwinding function stack. Ugh.