Generating LR Syntax Error Messages from Examples

Written by: Clinton L. Jeffery
Summarized by: Seonho Kim,
Kevin Myers,
and Michael Narayan

Outline

- Problem Overview
- Solution Overview
- LR Grammars
- Standard Approaches
- Merr + Example Errors
- Analysis
- Summary

The Problem

- Generating error messages for LR parsers is currently a difficult and error laden process
- Adding error productions to the grammar makes the grammar difficult to read, and can hinder error recovery
- Manually adding error messages based on parse state is error prone, and needs to be updated after every grammatical change

---

The Solution

- Use examples of erroneous code fragments, and associated errors messages to generate error messages within the parser
- Have the parser generator perform the tedious analysis of associating error messages with illegal grammar constructs

LR Grammars

- A subset of Context Free Grammars (CFGs)
- Effective for parsing nearly all practical programming languages
- Performs a left to right scan of the input (L), and produces a rightmost derivation of the parse (R)

Types of LR

- There are a number of different classes of LR grammars, with varying levels of power and associated difficulties in parsing
- The three main classes of LR grammars are SLR, LALR, and full LR, in increasing order of power, with a corresponding increase in storage requirements

Operation of a LR Parser

- A LR Parser has two main units, a stack which contains the current state that the parser is in, and a list of input symbols that have yet to be parsed
- Based on the state at the top of the stack, and the current input symbol, the parser than performs one of two actions, a shift or a reduce

Shifting

- On a shift, the parser removes an input symbol from the input list, and then pushes it and a corresponding state onto the stack
- In practical compilers, the input symbol itself is often left off the stack, but it’s type can be inferred from the state if necessary

Reducing

- On a reduce, the parser takes a right hand side of a grammar rule, and reduces it to it's corresponding left hand side
- This reduction pops of some number of (state, input) pairs corresponding to the number of tokens on the right hand side of the rule, and then pushes on the left hand side symbol and an associated state

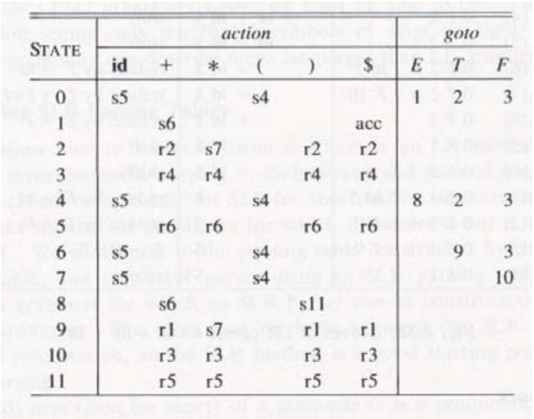
LR tables

- A LR parser is based upon a table that tells it what to do for a given state at the top of the stack and a current input token
- There are a number of rules that can be used to construct these tables, though it would be extremely tedious to do so by hand
- In practice these tables can be automatically generated using tools such as YACC, ANTLR

Example Grammar

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

Example LR Table



STATE	*action*						*goto*		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Interpreting the Table

- The entries for each (state, input) pairs tell what should be done given the pair
- A shift has the state to go to after the shift, and a reduce has the rule number to reduce by
- After a reduction, the grammar symbol from the left hand side of the production is pushed onto the stack, and the associated state from the Goto table is also pushed

Example Parse

Stack	Input	Action
0	id + id $	Shift 5
0id5	+ id $	Reduce 6
0F3	+ id $	Reduce 4
0T2	+ id $	Reduce 2
0E1	+ id $	Shift 6
0E1+6	id $	Shift 5
0E1+6id5	$	Reduce 6
…		

Possible Errors

- When parsing an input sentence; if the sentence is not accepted by the language, then an error will be detected
- This error will occur if there is no entry for the current (state, input) pair
- When an error occurs, it is necessary for the compiler to inform the user, hopefully with enough information for the user to fix the problem

Error Messages

- When an error is detected, an automatically generated parser (e.g. YACC) knows where in the input it is currently processing, as well as the state it is currently in
- Typically, without any guidance it will inform the user that it ran into an unexpected token at the current line

Detailed Error Messages

- Default error messages are often cryptic and may only be helpful to experienced programmers
  - helloworld.c : 1 : parse error before ‘}’ token
- Can a more descriptive error message be provided?

Generating Descriptive Messages

- Currently there are two broadly used methods for generating error messages when using an automatically generated parser
- Error productions can be added so that for any (state, input) pair there is always a production, with this production indicating a type of error
- The compiler designer can manually assign error messages to different (state, input) pairs

---

Adding Error Productions

- When an error production is used the production explicitly instructs the parser generator to call an error routine
  - Lbrace : ‘{‘ | { error_code=MISSING_LBRACE; } error;
  - Lbrace : ‘{‘ | { yyerror(“Missing left brace”); yyerrok; };
- This effectively produces readable error messages, however it suffers from a number of problems

---

Problems with Error Productions

- Clutters the grammar, making it much more difficult to read and determine the syntax of the language
- Makes error recovery difficult, as the extra rules get in the way of the error recovery
- Easy to introduce Reduce-Reduce conflicts – unacceptable in LR parsing

Manually Assigning Error Messages

- The compiler designer modifies the generated parse so that when an error is detected, the (state, input) pair is used to lookup an error message
- If an error message has been associated with that (state, input) pair it is printed, otherwise a default one is used
- This provides the ability to print descriptive error messages, while avoiding disturbing the grammar

Manually Assigning Error Messages

- This introduces its own difficulties
- It is tedious for the compiler writer to make these manual associations
- Even slight changes to the grammar will change the state associations, forcing the compiler writer to reassign error messages to states

Solution Requirements

- Must be able to produce useful error messages
- Must be less error prone
- Must not complicate the grammar itself
- Must be capable of updating itself with grammar changes

Proposed Solution

- Given erroneous code fragments and associated errors, automatically associate errors with different parse states
- Must be able to produce useful error messages
  - Can be as descriptive as the example writer can be
  - Provides a clear association between error messages and cause

Proposed Solution

- Must be less error prone
  - The association between messages and states is automated to alleviate the tedious nature of the task
- Must not complicate the grammar itself
  - Leaves the original grammar unaltered
- Must be capable updating itself with grammar changes
  - Most erroneous code fragments can port between grammar changes

---

Merr

- The author has created Merr as an extension to YACC
- Merr takes as input the YACC generated parser and a set of possible error / error message pairs
- Merr automatically generates the code to associate an erroneous state with an error message

---

Example Errors

- int main{} ::: parenthesis or semi-colon missing
- int x y; ::: missing comma in variable list
- char () { } ::: function name expected
- int a[] = {1, 2; ::: unclosed initializer
- procedure main() {

        := 3

  end

  ::: assignment missing its left operand

Associating Messages with Errors

- Merr first creates a parser which merely outputs the parse state and input token whenever an error is detected
- This parser is run against all of the example errors, thus recording what state the parser was in for each of the errors
- A production parser is then created which uses this information to create a table associating each error message with a (state, input) pair

Merr + Grammar Changes

- Simple changes in the input grammar can dramatically alter the corresponding LR table
- Every time the grammar is changed, Merr must be rerun with all the examples

Merr Performance

- Successfully used during the development of the Unicon programming language

Creating Example Errors

- Attempt to create a comprehensive set of error examples initially
- Create and add example errors as the need arises – building into a robust set
- Once these example errors are created, they may be continually used as the grammar changes

Advantages

- Merr can be easily integrated into any YACC compliant LR parser
- Concept is easily distributed to all LR parser generators
- Provides the same power found in the traditional methods of error messaging for LR parsers

Advantages

- Simplifies the efficient creation of compilers
- Merr doesn’t require the compiler designer to modify the grammar
- Automatically updates itself with grammar changes
  - Suitable for research languages where languages are frequently changed

---

Problems

- No facilities are provided to help determine the coverage of error messages
- Abstracting the error generation process from the actual implementation makes it conceptually easier to create error messages, however, by removing the designer for the implementation it reduces the amount of insight the designer can gain into adequate error coverage

---

Problems

- If the grammar changes in such a way that formerly valid programs are no longer valid, error fragments may need to be changed to reflect the new syntax to avoid incorrectly associating the provided message with the incorrect error states
  - Error fragments are often brief and will often be unaffected, or be easily fixed

---

Problems

- Many popular production level languages are well defined and static (e.g. C)
  - The benefits of this process are wasted on these languages where highly effective and efficient compilers have already been made
- Author admits Merr is the ‘lowest common denominator’

Possible Improvements

- Assist in creating more complete error coverage
  - Automatically generate error fragments for the designer to annotate/associate
  - Associate production rules/specific non terminals with error messages (e.g. ANTLR)

Alternate Approaches

- Good error recovery can provide more descriptive messages than simply error detection
  - Compiler can provide possible correct code fragments
- Automated production of error message through AI techniques
- Integrate compiler phases (e.g. ANTLR)

Summary

- The author proposes a method where error messages can be automatically associated with errors given example code fragments

- Allows error messages to be written completely separately from the language recognizer

Summary

- Avoids traditional pitfalls

- Merr provides an effective solution to some of the problems facing the designers of rapidly developing languages

References

- Jeffery, Clinton (2003). Generating LR Syntax Error Messages from Examples. *ACM Transactions on Programming Languages and Systems,* 25(5).
- Sebesta, Robert. <u>Programming Languages</u>, 2003.
- Aho et al. <u>Compilers: Principles, Techniques and Tools</u>, 1986.