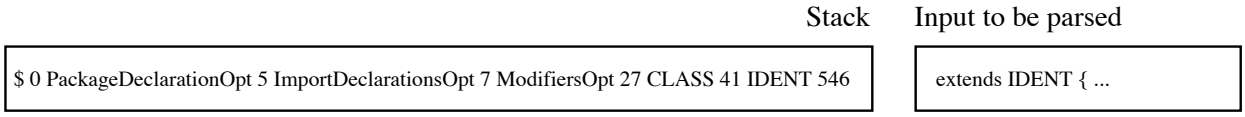
Appendix 1 An Algorithm for Performing Error Detection When Parsing Computer Programs

Introduction

When performing bottom up parsing, we have a stack indicating the constructs already parsed, and the input remaining to be parsed. These together constitute a right sentential form.

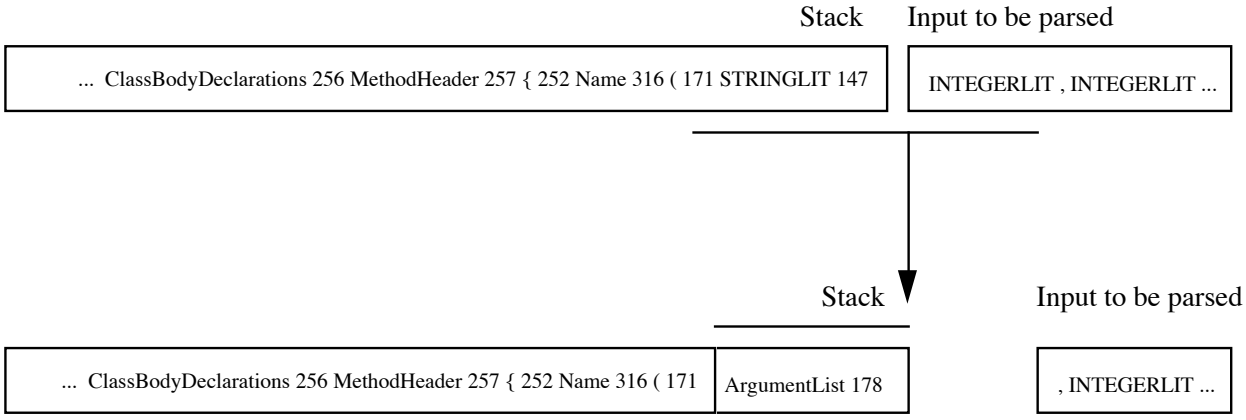


Most parsers assume that the error occurs somewhere close to the point at which the error was first detected. If an error occurs, many general purpose bottom up parsing algorithms attempt to recover from the error by deleting a top portion of the stack, and an initial portion of the remaining input, and replacing it by a nonterminal, which is pushed on to the stack. The assumption is that the error occurs within a construct corresponding to the nonterminal, that bridges the gap between the previously correct input and the following correct input.

For example, if we had the following fragment of a Java program

```
public void paint( Graphics g ) {
     g.drawString( "Hello World!"  30, 30 );
     }
```

with a missing "," between the string literal "`"Hello World!"`", and the integer literal "`30`", then we might push the string literal onto the stack, detect an error, then replace the string literal on the stack and integer literal current token to the nonterminal "`ArgumentList`". Other algorithms might be more aggressive, and reduce everything from after the left parenthesis up to the right parenthesis to "`ArgumentListOpt`".



Different error recovery algorithms have different ways of deciding which portion of the stack and input (essentially the"handle" for the rule enclosing the error) are reduced to a nonterminal. For example, in the yacc parser generator system, explicit additional grammar rules have to be included that use the special nonterminal "`error`", that can match anything. Essentially the stack is cut back to the topmost state that can shift "`error`", and input is consumed until the current token can then be shifted onto the stack. The yacc algorithm is notoriously bad at accidentally getting on the wrong track. The grammar designer must be exceptionally sparing in their use of grammar rule involving the "`error`" nonterminal.

Other error recovery algorithms are more explicit about the manner in which the error is corrected. They involve insertion and deletion of tokens to convert the program into something that is syntactically valid. Such algorithms still satisfy the same general model, and end up reducing the top portion of the stack and the following input containing the error to a nonterminal that bridges the gap between the previously correct input and the following correct input.

A question that has to be asked is whether there is really a need to take into account the previous context when dealing with errors. The real error may well be a long way before the error was detected. For example the error in the following Java fragment is probably the missing left brace, but the error is not detected until the "`else`".

```
if ( a < b )
     min = a;
     max = b;
     }
else {
     min = b;
     max = a;
     }
```

Many spurious error messages can be generated as a result of this.

There is some advantage in "bridging the gap" for syntax errors, in that it allows an almost complete parse tree to be built, so that the rest of the compiler can continue performing semantic analysis. However, after syntax errors, such as unmatched braces, the compiler's notion of scope is often wrong, and the number of spurious semantic errors is often excessive. Continuing the semantic analysis may be necessary if parsing a language for which semantic analysis feeds back to the lexical analyser. For example in C it is necessary to determine whether an identifier is declared as a type or not, to know how to parse text such as "`a * b`". However, even if the parser only creates fragments of the parse tree after detection of an error, it is probably still possible to perform semantic analysis of these fragments. If the parser creates several alternative fragments, the semantic analysis can be performed on each alternative, so long as the information is not kept globally. Also, as the compilation process becomes automated, language designers are moving towards demanding that the language has a clearly defined LALR(1) syntax, without the need to relate it to semantic analysis.

I can see little justification for the requirement that the parser "bridge the gap" and essentially convert the program into something that is syntactically valid, when errors are detected in the program. There is no guarantee that the cause of the error is close to the point of detection. The requirement that a program be essentially converted into a valid program has major problems if the parser gets on the wrong track, or the programmer writes valid code out of context. For example, the formal definition of Pascal requires that variables be declared before procedures and functions, but many compilers are more relaxed about the ordering, and permit the intermingling of different kinds of declarations. If the programmer is used to using such a compiler, or is attempting to compile a program written for such a compiler, then a parser that tries to "correct" a syntactically invalid program might attempt to convert declarations that are out of order into something completely different. Almost all error messages will be spurious.

I believe a much better approach is for the parser to ignore the previous context after detecting an error. When an error is detected, the parser should print out an error message. The error message could be a plain statement of "syntax error", or it could use the information on the stack and the current token, to generate a more informative error message. For

example, a more informative error message could list such things as the set of items for the top of stack, the symbols on the stack, possible tokens that would allow the parse to continue, and the current token. After displaying an error message, the parser could parse ahead, with all possible alternative parses from the point of error, until it runs out of alternatives, or reaches the end of the input. If it runs out of alternatives, it should generate another error message, and again parse ahead, with all possible alternative parses from the point of error. The parser effectively splits the program up into maximal sized fragments of syntactically valid input, and marks the end of each fragment by an error message. There is no need for the grammar designer to add in special rules to deal with syntax errors. Moreover the whole system is independent of the specific grammar being parsed, so such a system could easily be inserted into any table driven bottom up parser. Moreover, such a system seems to be remarkably simple. It is fundamental to its design that it will neither generate much in the way of spurious error messages, nor fail to point out errors after the first error message. The only real questions seem to be how to find the longest fragment of syntactically valid input after the point of error, and whether such an algorithm can execute in reasonable space and time.

The parsing algorithm

The parsing algorithm is based on a bottom up parsing technique, such as LALR(1) parsing, used by yacc, Java CUP, etc.

At each stage, we have a set of partial stacks. Initially, this set contains a single stack, namely the stack with the start state pushed on. So long as no error occurs, the set of partial stacks contains a single element, corresponding to the conventional stack.

After a syntax error, the set of partial stacks can and often does contain several stacks. Since we do not have a previous context, these stacks are “partial stacks”. We know the states on the top portion of the stack, but not the bottom portion of the stack.

```
public void parse() {
     StackSet stackSet = new StackSet();
     stackSet.addElement( StateStack.create( start_state() ) );
     while ( true ) {
          int currentToken = scan();
          StackSet newStackSet = performAction( stackSet, currentToken );
          if ( newStackSet == null ) // Accept
               return;
          else if ( newStackSet.size() == 0 ) { // Error
               printErrorMessage( stackSet , currentToken );
               newStackSet.addTerminalStackSet( currentToken );
               }
          stackSet = newStackSet;
          }
     }
```

If the set of partial stacks becomes empty, there is no alternative parse, and we have a syntax error. We print an error message, and generate all possible partial stacks with a single state that could occur after shifting the current token on to the stack.

```
// In the class StackSet
     public void addTerminalStackSet( int terminalID ) {
          for ( int stateID = 0; stateID < table.size(); stateID++ ) {
               Action action = table.getAction( stateID, terminalID );
               switch ( action.actionType() ) {
                    case Action.SHIFT:
```

```
                    addElement( StateStack.create( action.shiftState() ) );
                    break;
               }
          }
     }
```

Performing a shift-reduce parse with a set of partial stacks is not so different from performing it for a single stack. For each stack in the old set, we create a new stack by performing the action indicated by the top of stack state, and current token. If the action is to shift or accept, the action is clear. If the action is error, we discard the stack. If the action is to reduce by a rule, there are two situations to consider. We have to pop as many states off the stack as there are symbols on the right hand side of the rule, and push on a state corresponding to the left hand side of the rule. If the stack has more states on it than we need to pop off, there is no problem. The normal action can be performed. If there are insufficient states to pop off (because the construct starts at or before our last detected error), we delete the stack, and generate all possible partial stacks with a single state that could occur after shifting the left hand side onto the stack.

```
public StackSet performAction( StackSet stackSet, int currentToken ) {
     StackSet newStackSet = new StackSet();
     for ( int i = 0; i < stackSet.size(); i++ ) {
          StateStack stack = stackSet.elementAt( i );
          StateStack newStack;
          Action action = table.getAction( stack.stateID(), currentToken );
          Rule rule;
          switch ( action.actionType() ) {
               case Action.SHIFT:
                    newStackSet.addElement(
                         stack.push( action.shiftState() ) );
                    break;
               case Action.REDUCE:
                    rule = table.getRule( action.ruleID() );
                    newStack = stack.pop( rule.rhsLength() );
                    if ( newStack == null ) {
                         stackSet.addNonterminalStackSet( rule.lhs() );
                         }
                    else {
                         int shiftState =
                              table.getReduce(
                                   newStack.stateID(),
                                   rule.lhs() ).shiftState();
                         stackSet.addElement( newStack.push( shiftState ) );
                         }
                    break;
               case Action.ACCEPT:
                    return null;
               case Action.ERROR:
                    break;
               }
          }
     return newStackSet;
     }

// In the class StackSet
     public void addNonterminalStackSet( int nonterminalID ) {
          for ( int stateID = 0; stateID < table.size(); stateID++ ) {
               Action action = table.getReduce( stateID, nonterminalID );
```

```
                switch ( action.actionType() ) {
                     case Action.SHIFT:
                          addElement( StateStack.create( action.shiftState() ) );
                          break;
                     }
                }
          }
```

Some examples

Suppose we have a syntax error in a Java program, followed by the input

```
     while ( a < 10 )
          a = a+1;
     System.out.println( "a = " + a );
     }
}
```

The parser prints an error message, and creates the three alternative partial stacks corresponding to the three states that can result by shifting "`while`".

```
WHILE 330
WHILE 433
WHILE 471
```

These three states correspond to an ordinary "`while`" statement not nested inside the "then" part of an "`if`" statement, a "`while`" statement nested inside the "then" part of an "`if`" statement, and a "`do ... while`" statement.

```
{ WhileStatement → WHILE . LEFT Expression RIGHT Statement }
```

```
{ WhileStatementNoShortIf → WHILE . LEFT Expression RIGHT StatementNoShortIf ,
     WhileStatement → WHILE . LEFT Expression RIGHT Statement }
```

```
{ DoStatement → DO Statement WHILE . LEFT Expression RIGHT SEMICOLON }
```

After parsing up to the beginning of the assignment statement, the third alternative drops out.

```
WHILE 330 LEFT 365 Expression 366 RIGHT 367 IDENT 333
WHILE 433 LEFT 461 Expression 462 RIGHT 463 IDENT 439
```

Eventually the sub-statement is shifted onto the stack.

```
WHILE 330 LEFT 365 Expression 366 RIGHT 367 StatementExpression 286 SEMICOLON 484
WHILE 433 LEFT 461 Expression 462 RIGHT 463 StatementExpression 286 SEMICOLON 484
```

The whole "`while`" statement is reduced to "`BlockStatement`", then "`BlockStatements`", and the identifier is shifted onto the stack.

```
BlockStatements 486 IDENT 333
BlockStatements 280 IDENT 333
BlockStatements 383 IDENT 333
BlockStatements 356 IDENT 333
```

It is worth noting that the number of alternatives increases at this point. The reason for this is that there are four different contexts in which "`BlockStatements`" can occur, namely

```
{ Block → LEFTCURLY BlockStatements . RIGHTCURLY }
{ ConstructorBody → LEFTCURLY BlockStatements . RIGHTCURLY }
{ ConstructorBody → LEFTCURLY ExplicitConstructorInvocation BlockStatements .
RIGHTCURLY }
{ SwitchBlockStatementGroup → SwitchLabels BlockStatements . }
```

The parsing continues, shifting the method invocation onto the stack, reducing it to "StatementExpression", shifting the semicolon onto the stack, reducing "StatementExpression ;" to "BlockStatement", then "BlockStatements BlockStatement" to "BlockStatements", etc., shifting on the right brace, and obtaining

```
BlockStatements 486 RIGHTCURLY 488
BlockStatements 280 RIGHTCURLY 485
BlockStatements 383 RIGHTCURLY 385
SwitchBlockStatementGroups 352 RIGHTCURLY 354
```

After shifting on the final right brace, we get

```
ClassBodyDeclarationsOpt 260 RIGHTCURLY 265
BlockStatements 486 RIGHTCURLY 488
BlockStatements 280 RIGHTCURLY 485
BlockStatements 383 RIGHTCURLY 385
SwitchBlockStatementGroups 352 RIGHTCURLY 354
```

The current token is now end of file, and the first alternative manages to reduce down an accept state, so there are no more error messages.

If we delete the "+" in

```
System.out.println( "a = " + a );
```

then we generate a second error at this point.

We shift the identifier onto the stack, generating 15 new stacks.

```
IDENT 12
IDENT 16
IDENT 546
IDENT 43
IDENT 103
IDENT 77
IDENT 95
IDENT 241
IDENT 243
IDENT 333
IDENT 272
IDENT 481
IDENT 476
IDENT 371
IDENT 439
```

because there are lots of different contexts in which identifiers can occur.

The number of alternatives rises to 22 on processing the right parenthesis, then drops down to 6, before the first alternative is accepted.

```
ClassBodyDeclarationsOpt 260 RIGHTCURLY 265
BlockStatements 486 RIGHTCURLY 488
BlockStatements 280 RIGHTCURLY 485
BlockStatements 383 RIGHTCURLY 385
SwitchBlockStatementGroups 352 RIGHTCURLY 354
VariableInitializers 494 RIGHTCURLY 497
```

Potential problems with algorithmic complexity

When a syntax error is detected, the number of stacks created is equal to the size of the set of states { T | there exists a state S such that goto(S, currentToken) == T }. For a standard grammar for Java, this is at most 25. The worst cases are

```
 5 COLON
 5 DOT
 6 COMMA
 6 LEFTCURLY
10 RIGHTSQ
13 LEFTSQ
15 IDENT
15 RIGHTCURLY
22 LEFT
24 RIGHT
25 SEMICOLON
```

For example, most of the cases for a semicolon occur because all simple statements and declarations terminate with a semicolon. These alternatives rapidly disappear, after a reduction is performed, and the next token is shifted onto the stack. For example, if a semicolon is followed by "`while`", 25 alternatives are created, but there are only 6 alternatives after shifting the "`while`".

Additional stacks can be generated by a reduction that pops off all states on a partial stack. The number of stacks created is equal to the size of the set of states { T | there exists a state S such that goto(S, leftHandSide) == T }. For a standard grammar for Java, this is at most 20. The worst cases are

```
 4 AdditiveExpression
 4 BlockStatements
 4 PrimitiveType
 4 Type
 5 ModifiersOpt
 5 ShiftExpression
 6 StatementNoShortIf
 7 Statement
 8 ArgumentListOpt
 8 Block
11 Name
11 UnaryExpression
20 Expression
```

Since there could be reductions by rules for several nonterminals, there could be more than 20 stacks generated, but the total number of new stacks cannot possibly be more than the total number of states, a constant for a given grammar. Hence the total number of stacks generated after parsing n tokens after an error is potentially of order O(n). In the worst case, for some grammars, this order can be achieved. For example, consider the grammar

```
S→ X Y
X→ a X b | ε
Y→ b Y c | ε
```

This grammar generates text of the form $a^m b^m b^n c^n$, for some m >= 0, n >= 0. Suppose we have a syntax error, then a sequence of "b"s. It is impossible to tell how many of the "b"s belong to "X" and how many belong to "Y", so we generate n + 1 alternative stacks, after shifting n "b"s. For example, when n is 5, we have:

```
b 4 b 4 b 4 b 4 b 4
X 3 b 4 b 4 b 4 b 4
X 3 b 4 b 4 b 4
X 3 b 4 b 4
X 3 b 4
X 9 b 10
```

In fact, for my sample Java grammar, by parsing tens of millions of randomly generating input sequences, I have been unable to get more than the 25 stacks created after shifting a semicolon. While I have not been able to characterise the conditions necessary for the number of stacks to be bounded, it does appear that there is no problem with algorithmic complexity for realistic grammars. This is not surprising, because as the parser consumes more tokens beyond the error, and more information becomes available, we would expect the parser to become more certain about what it is parsing. Thus we would expect the number of alternatives to decrease, rather than increase.

When a syntax error occurs in the body of a method or constructor declaration in a Java program, the parser quickly settles down to about 6 alternative stacks. After processing the start of a method or constructor declaration, a single alternative stack remains. Hence the algorithm seems to have little problems with complexity for realistic grammars.