CS 444
Assignment 1
Group 7: bcymet, jm3lee, tkim

January 31, 2006

Table of Contents

1.0 Design Documentation

1.1 Design Overview

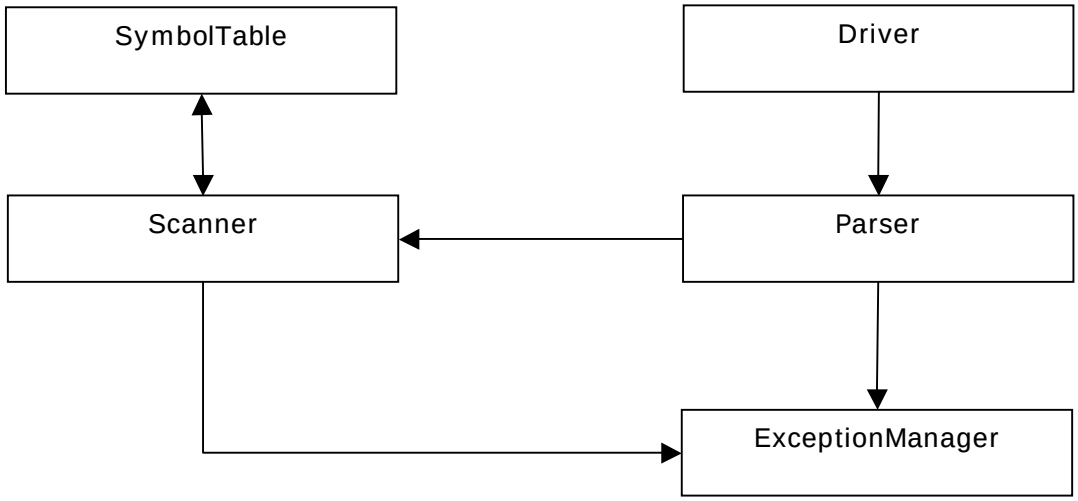
From a high level, our compiler is split up into several modules: Scanner, Parser, ExceptionManager, and SymbolTable (Note that this is sufficient for assignment 1. Future components will be incorporated as we progress our compiler).

With a module design, future revisions of any component can simply replace the old version. Testing was also very effective as we were able to unit test each component before we merged them all together. Each of the components communicates with each other through a well defined API.

1.2 Control Flow

The parser calls `nextToken()` on the scanner object. The scanner will then consume input from the source file and create a token by calling `CreateToken()` on the `SymbolTable` object. The parser will then use this token and determine if it will shift, reduce, or shift and reduce the token based on the Parse table generated by `ilalr`.

Data Flow



1.3 Grammar

LALR(1) was chosen to implement Ada/CS. [1] lays out the advantages and disadvantages of simple LR (SLR) and canonical LR (CLR) clearly.

	Advantage	Disadvantage
SLR	Simple and easy to implement	Weakest recognition of input ability among LR(k) grammars
CLR	The most powerful	Very complicated to

	recognition of input	implement

LALR grammars are the middle ground between SLR and CLR. Since we were not given a tool to generate a CLR parse table, we chose LALR over SLR for better recognition capability, and in our experience, it simplified our grammar due to LALR(1)'s look-ahead capability.

Our current grammar does not produce conflicts in the parse table when using ilalr, but slr fails to process the grammar with shift/reduce conflicts. This is due to the fact that SLR does not have look-ahead capability.

LALR(1) grammar is also easy to prototype. There are various tools available that process LALR(1) grammar, notably bison on Solaris systems. Whenever `ilalr` reported conflicts in the grammar, we were able to find more concise information in bison output.

2.0 Class Descriptions

The following pages will give a description of the classes used in this program, along with an overview of how the class was implemented. Also, the public API for each component is shown in detail. However, it should be noted that there are more methods in the API for some of the classes but have been omitted for this assignment because they are not used. Future revisions of this document will include the omitted methods.

2.1 Scanner

The scanner is passive. It consumes input from the file, invokes the Symbol Table to create tokens and queues tokens onto the output queue. The scanner will tokenize the entire file at once and queue all the tokens into a token list. We decided to tokenize the entire file at once as opposed to creating new tokens as the parser needs them because our error recovery scheme in the parser needs to be able see all the tokens in the file.

If a lexical error is encountered, an error token is generated and placed on the output queue. By using an error token, we communicate with the Parser that a lexical error has occurred and the Parser can determine what action to take.

When the end of the file is encountered, the scanner will create an `EOF` token and add it to the output queue. This will let the parser know that there is no more input and tokenization has stopped.

To tokenize the file the scanner uses a state machine defined in `Transition.h` and `Transition.cc`. See Appendix 4.1 for the transition table.

The scanner will take each line and run it through the state machine until an error state is reached. When the error state is reached the last non error state is checked to see if the machine has reached a final state. If a final state had been reached then the scanner will look at the character that caused the error and check to see if it would be a valid character after the token that was found at the error state. If that character is valid, most likely a delimiter, the type of token is decided by calling `stateType` and then the token is created and added to the token queue. If an error is found the type of error is printed out as well as the location, line number and column number as to where the error occurs. The line is printed out and the column where the error occurs indicated by a ^ character.

When an error occurs, the scanner will skip over input until a delimiting character has been found. The scanner will output an `ERROR` token and then continue to tokenize the rest of the input.

2.1.1 API

```
Scanner(
   char* inputFileName,
   SymbolTable& SymbolTable,
   ExceptionManager& ExceptionManager)
```

This is the main constructor for the Scanner class.

```
Token* NextToken()
```

This method will return the next token in the token stream.

2.2 Parser

The parser is the active part in this assignment. At each stage, it will invoke the Scanner object and ask for a token. While it consumes the token, it will consult the grammar to verify that each production can be reduced and report errors if the input is invalid.

The key concept in our parser design is to separate various parser's states and functions. In this document, we define a "parser state object" as a collection of the following objects.

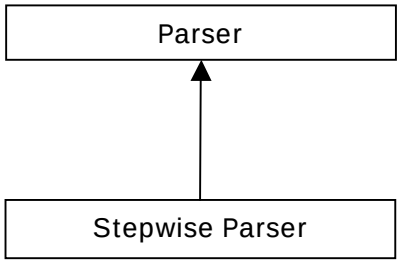
1. Parse stack
2. State stack
3. Token stream

The parser's functions are

1. Simple error recovery
2. Parsing token stream contained in a parser state object.

This design provides us with great flexibility. Temporary parser states can be constructed on the fly to incorporate various error recovery strategies. A generalized parser state interface also allows us to compose parser states in any way we wish.

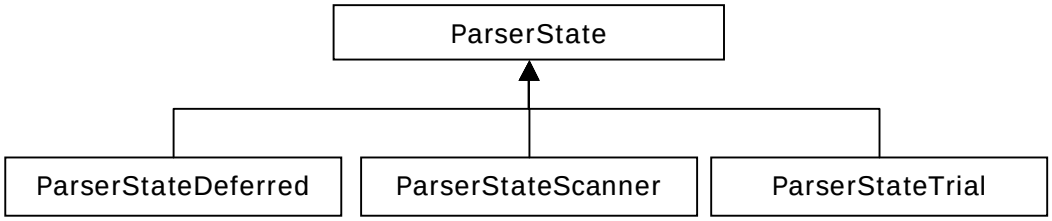
2.2.1 Parser Class Hierarchy



`Parser`: Main LR Parsing Engine with simple error recovery scheme outlined in [3].

`StepwiseParser`: This class implements a simplified parser except that only performs one operation (shift, reduce or shift-reduce) at a time.

2.2.2 ParserState Class Hierarchy



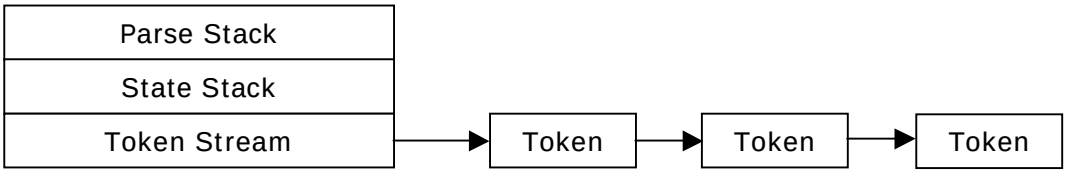
`ParserState`: This is a base class for ParserStateDeferred, ParserStateTrial and ParserStateScanner.

`ParserStateDeferred`: This class implements deferred parse stack, state stack, and deferred tokens queue as described in [2].

`ParserStateTrial`: This class implements parser state to be used in the trials in the simple error recovery algorithm.

`ParserStateScanner`: This class implements the main parser state.

2.2.3 Parser State Organization



2.2.4 API

2.2.4.1 Parser

```
Parser(
   Scanner& scanner,
   SymbolTable& symbolTable,
   ExceptionManager& exceptionManager)
```

The main constructor for the Parser class. It accepts a reference to the Scanner, SymbolTable, and ExceptionManager.

```
int MainLoop()
```

This is the main entry point for the program. Invoking this method will start the parsing of the specified input file.

2.2.4.2 ParserState

```
ParserState()
virtual ~ParserState()
```

The constructor and destructor for ParserState

```
Int      GetCurrentState()
void     PushState(int state)
int      PopState()
```

These methods are used to manipulate state stack.

```
virtual Token* PeekToken()
virtual void   PrependToken(Token*)
virtual Token* NextToken()
```

These methods are used to manipulate token stream. These are defined as pure virtual functions so that the subclasses must define them according to their needs.

```
virtual void         PushToken(Token*)
virtual Token*       PopToken()
```

These methods are used to manipulate parse stack. These are defined as pure virtual functions so that the subclasses must define them according to their needs.

```
const vector<Token*>&      GetParseStack()
const vector<int>&         GetStateStack()
```

These methods are used to access parse and state stack directly. const reference is chosen to prevent direct internal parser state modification. They are used to print debug outputs as well as in constructing ParserStateTrial from various components.

2.2.4.3 ParserStateDeferred

```
bool           IsQueueFull()
string         SerializeQueue()
const deque<Token*>& GetDeferredList()
unsigned int   GetQueueSize()
```

Deferred queue accessors.

```
void InsertTokenBefore(
   Token* token,
   int deferredTokenIndex)
```

```
void SubstituteTokenAt(
   Token* token,
   int deferredTokenIndex)

void DeleteTokenAt(
   Token* token,
   int deferredTokenIndex)

Token* GetDeferredTokenAt(
   int deferredTokenIndex)
```

These are methods used to modify the deferred queue. They are used when performing repair actions during error recovery phase.

2.2.4.4 ParserStateScanner

```
void ResetParseStack(const vector<Token*>& newStack)
void ResetStateStack(const vector<int>& newStack)
```

These methods are used to reset the parser state. They are used in error recovery phase to re-initialize the main parser state with corrected state.

2.2.4.5 ParserStateTrial

```
void Reset(TokenList *, ParserStateDeferred *)
```

This method is used to re-initialize trial state. This is used in error recovery phase.

2.2.5 Error Recovery

We also implemented the Simple Error Recovery scheme outlined in [3]. The deferred parse stack is implemented in `ParserStateDeferred`. Our experiments have shown that `K = 3` is sufficient to handle errors correctly where `K` is the number of deferred tokens. Burke and Fisher outline two strategies to implement simple error recovery.

1. Hold trials at each token on the parse stack, each deferred token and the error token.
2. Hold trials only at each deferred token and the error token.

We took the second approach. Since the error token is at the beginning of the token stream, we decided that it was plausible to only hold the trials at the most recently shifted three terminal and the error tokens. The statistical data from [3] indicates that this provides suitable error recovery with a minor pitfall. Any errors that require more than one token to correct the input token stream will not be fixed under this scheme.

Simple error recovery corrects errors in a statement, but it does not correct errors beyond this level. For example, incomplete scopes will cause our parser to stop.

These errors can be corrected with scope and secondary error recovery schemes. Scope and secondary error recovery algorithms were considered; however, due to time constraints, we only implemented simple error recovery. This does not provide us with complete error recovery, but the behaviour of the algorithm was found to be suitable for common and simple mistakes; for example, missing a keyword or a semicolon.

Parser has a pointer to an instance of `StepwiseParser`. When an error is found `ParserStateTrial` is constructed from `ParserStateDeferred` and the token stream from `ParserStateScanner`. Then each trial is carried out using the instance of `StepwiseParser` and `ParserStateTrial`. Once we have determine that a repair is possible, we perform the repair on deferred parser state, copy the state to `ParserStateScanner`, and continue parsing the rest of the input. See below for the description of the candidate selection algorithm.

2.2.5.1 Candidate Selection

Simple error recovery algorithm requires us to select a candidate token in order to recover from an error. Burke and Fisher suggest the following repair possibilities.

1. Insert a token into the token stream
2. Delete a token from the token stream
3. Substitute a token in the token stream
4. Merge two tokens in the token stream

In our implementation, the last repair option is not implemented.

To select a repair candidate, we followed [3] closely; however, preference heuristic is not implemented. In case of a tie among the candidates, we simply pick the first candidate found. Our repair candidate selection heuristic is outlined below.

```
For each repair candidate set Insert, Delete,
Substitute

   If there are none keyword candidates
         Prune keyword candidates.

   Find the candidate with the maximum distance, and
   insert it in the set FinalCandidates.

   If there are none keyword candidates in
FinalCandidates
         Prune keyword candidates.

   If there is an insertion repair candidate
         Repair the error using this candidate.

   Else if there is a deletion repair candidate
         Repair the error using this candidate.

   Else if there is a substitution repair candidate
         Repair the error using this candidate.
```

```
   Otherwise, this error cannot be repaired.
```

2.3 SymbolTable

The SymbolTable class will be shared between Parser and Scanner. Because of the nature of a symbol table, we decided to put a restriction on this object so that only one copy may be present in memory at any time (i.e. Singleton design patter). The main use of this SymbolTable will be to hold the user defined Identifier tokens. We also record each occurence of the token and make it available for output. This is the cross reference list.

2.3.1 API

```
SymbolTable* GetInstance(
   ExceptionManager* exceptionManager)
```

This method will instantiate a copy of the symbol table if it does not yet exist in memory. If one copy does exist, it will simply return that object pointer.

```
Token* CreateToken(
   string name,
   string originalName,
   string dataType,
   enum TokenType type,
   int col,
   int line)
```

This method will create a token using the given parameters. Also, if the token is an integer, float or string literal, it will not insert it into the symbol table.

```
void UpdateToken(
   Token* tokenToUpdate,
   string dataType)
```

This method will update the token's data type (not to be confused with token type).

```
Token* FindToken(
   string name)
```

This method will find a token with a matching name given by the input parameter. But it will only search for a match on name, not original name.

```
void PrintAllTokens()
```

This method will log all tokens currently in the symbol table to the exception manager. It is simple method to help with debugging.

```
void PrintCrossReference()
```

This method prints out all the user defined identifier tokens and each occurence in the source file.

2.4 Token

This is a simple class that holds information about the token that was generated. The information that is recorded in a token consists of: name (case insensitive), original name(case sensitive), data type, and line number and column number (where the token was encountered in the source file).

2.4.1 API

```
string GetName()
string GetOriginalName()
int GetLineNumber()
int GetColumnNumber()
```

Basic accessor methods for this class.

```
string GetType()
```

Get the type of the token (e.g. ID, INTEGER, etc).

```
string Serialize()
```

Generates a string representation of the token. The format will be “[TOKENNAME]”.

2.5 ExceptionManager

This class will manage exception and error messages. Each of the components should log either an error or a debug statement directly to the Exception Manager as needed. With a centralized place where we handle output that is useful for the user, we can easily modify the format in one place, rather than multiple places. This also allows us to remove the iostream references from the other classes, leaving only what are needed.

Just like the symbol table, we will only allow one copy of the Exception Manager in memory at any time.

2.5.1 API

```
ExceptionManager* GetInstance()
```

This will get an instance of the Exception Manager.

```
void LogDebugSatement(
   string component,
   string statement)

void LogErrorMessage(
   string component,
   string statement)
```

Methods to log debug and error messages.

```
void PrintDebugStatements()
void PrintErrorMessages()
```

Methods to print debug and error messages.

2.6 ParseTable

This class is responsible for managing and loading parse table generated by the provided tool "ilalr". See "Token, Parse Table and Rules Loading" section for more details on exact low level details.

2.6.1 API

```
ParseTableEntry* GetEntry(
   int state,
   int token)
```

This message will get an entry from the Parse Table given state and token.

2.7 RuleList

This class is responsible for managing and loading the rule list generated by the provided tool "ilalr". See "Token, Parse Table and Rules Loading" section for more details on exact low level details.

2.7.1 API

```
RuleEntry* GetRule(int id)
```

Retrieve a rule from the Rule List given a rule id.

3.0 Testing

Testing a program of this size requires a good testing strategy. We made sure that after each component was ready for testing, we performed unit tests on them first. After we were satisfied with the unit test we then combined components one at a time and continued our testing.

Nearing the end of the assignment required us to test the program as a whole.

Picking our test cases was an important part of testing. We created test cases based on real world experiences (e.g. a missing semicolon, a miss-spelled keyword, or mismatched braces). We also made a lot of boundary test cases to stress our compiler as much as possible.

3.1 Testing the Grammar

To test our grammar, we wrote our cfg and ran it through various programs, namely bison, flex and ilalr. We found out that using ilalr did not give us the most intuitive error messages, so we used bison. At the same time, we made sure that the grammar being accepted in bison was still being accepted into ilalr.

Testing the grammar further required our parser to be completed, so most of the grammar tests were done with the parser tests.

3.1.1 Dynamic Testing

To make debugging the context-free grammar easy, we have written a converter that takes the parse table generated by the tool "ilalr", and generated binary data.

```
Output from "ilalr" --> Converter -->
TokenDefinitions.cc/.h
                                        parsetable.bin
                                        rulelist.bin
```

ParseTable.h and RuleList.h define the format of parsetable.bin and rulelist.bin respectably.

Parse Table Binary Format

```
struct _ParseTable {
    int                 numStates;
    int                 numVocabularies;
    ParseTableEntry     table[];
}

struct ParseTableEntry {
    enum ParseTableAction       action;     // SHIFT,
REDUCE, SHIFT_REDUCE, ERROR, ACCEPT
    union {
        int                     state;
        int                     ruleId;
    } actionParameter;
}
```

Rule List Binary Format

```
struct _RuleList {
    int         numRules;
    RuleEntry   rules[];
}

struct RuleEntry {
    int                     id;
    int                     numRHS;
    int                     nonterminal;
}
```

parsetable.bin and rulelist.bin are loaded in to memory using mmap() system call. This eliminates dynamic memory allocation.

Another approach is to write a script to convert all the data into C++ source code. However, this approach requires frequent recompilation when debugging the CFG. With our approach, it is possible to avoid compilation of the application when token values are not modified, but grammar is modified.

3.2 Testing the Scanner

In order to test the scanner, we chose test cases that would stress the tokenization function of the scanner. We also have a flag that will disable the parser and only show the tokenized output of the input file.

3.2.1 Test 1

This is a simple test of the scanner. We just pass in a very basic file and make sure that the scanner tokenizes the input correctly. This is the simple test:

```
package Identifier is
body
begin
    null;
end;
```

The output of this test is:

```
[PACKAGE] [ID Identifier] [IS] [BODY] [K_BEGIN] [NIL]
[SEMICOLON] [END] [SEMICOLON]
```

We can see from this output that the input file is being tokenized correctly. Note that the output that the scanner has produced is simple a human-readable representation of the token. Tokens and strings are not actually passed from the scanner to the parser.

3.2.2 Test 2

Because we can disable the parsing stage of our program, this test file will just have a listing of all the keywords in our grammar. This test will ensure that every reserved word is being tokenized correctly and not being tokenized as an identifier.

```
abs and array begin body case constant declare else
elsif end exception exit for function if in is loop
mod not null of or others out package pragma private
procedure raise range record return reverse separate
subtype then type use when while access all
```

The output is as expected; all of the tokens are keywords and not identifiers:

```
[ABS][AND][ARRAY][K_BEGIN][BODY][CASE][CONSTANT]
[DECLARE][ELSE][ELSIF][END][EXCEPTION][EXIT][FOR]
[FUNCTION][IF][IN][IS][LOOP][MOD][NOT][NIL][OF][OR]
[OTHERS][OUT][PACKAGE][PRAGMA][PRIVATE][PROCEDURE]
[RAISE][RANGE][RECORD][RETURN][REVERSE][SEPARATE]
[SUBTYPE][THEN][TYPE][USE][WHEN][WHILE][ACCESS][ALL]
```

Note that the output has been formatted so that it can be easily read.

3.2.3 Test 3

After a test with all the keywords, we found appropriate that we test all the symbols in the grammar too. Here is the input file:

```
- + / /= < <= > >= = * ** ( ) | , . .. => : ; <> :=
```

And the output is:

```
[MINUS][PLUS][DIV][NEQ][LT][LEQ][GT][GEQ][EQUAL]
[MULT][EXP][LPAREN][RPAREN][PIPE][COMMA][DOT]
[RANGE_OP][ARROW][COLON][SEMICOLON][UNLIMITED]
[ASSIGNMENT_OP]
```

As it can be seen from the output, each operator is being tokenized properly.

3.2.4 Test 4

This test will show that the scanner can detect errors and continue tokenization of the input.

```
package jfa!f9
4.e
10_39_f.324
33.33.33.33
"hello
world"
```

And the output:

```
Error: Invalid Character in an identifier. On line: 1
package jfa!f9
          ^
Error: Invalid Character. On line: 2
4.e
  ^
Error: Invalid Character. On line: 3
10_39_f.324
      ^
Error: Extra dot in a float literal. On line: 4
33.33.33.33
     ^
Error: Invalid Character. On line: 5
"hello
      ^
Error: You have an ending quote but not a beginning
one. On line: 6
world"
^
```

It can be seen that the scanner detects errors and can recover from then and continue scanning.

3.2.5 Test 5

This test was created to stress the parser as much as possible. Because the stress test is very long in length, it was omitted from the document and can be located at test/StressTest2.adb. Also, since the tokenized output from our scanner was too long to be put into our document, it can also be located in test/StressTest2.output.

3.3 Testing the Parser

Testing the parser required test files that were focused on various different ways we can write our test programs. Consequently, testing the parser also tests our grammar at the same time.

When an Ada/CS program is parsed by our program, the output will either be “Accepted” meaning that the test file is a valid Ada/CS program, or will return errors that our error recovery mechanism found.

For our following test cases, we will only run through valid test programs as we have a special section dedicated to error cases and recovery.

3.3.1 Test 1

Using the same methodology, we first tested our parser with a very simple test case. This test case is exactly the same as the simple test used in testing the scanner:

```
package Identifier is
body
begin
    null;
end;
```

Our parser accepted this program, which was the expected result.

3.3.2 Test 2

This test will test the various ways we can build arithmetic expressions and boolean expressions. A test like this will also test our grammar's correctness.

```
package identifier is
body
begin
   a:=1+1; a:=1 - 1; a:=1 * 1; a:=1/1; a:=b/c;
a:=d*e;
   a:=-1- -1; a:=-2*-2; a:=-2/-5; a:= -2**3;
   a:=a**(b**2*2+power/3); a:=a+(b-4)/c*-2+f**99/4;
   a:=5+6*2/19 ** 25;
   a:=-b/-c; a:=-b**c; a:=-b*-c; a:=b**(b**(b**b));

   a:= (3+4)<(8**9);
   a:= a>=3 and b<4; a:= a>3 or b<4; a:= a>3 or not
b<4 and not b>5;
   a:= a>(3<= (8**9) or 3=f**3) and not a < (b or not
c) >= d /= c;
   a:= a>3 and then b<3; a:= a>3 or else b<3; a:= b>3
or else not b<3;
   a:= a=b or not (a or else (c and (b and then c)));
end;
```

As expected, our parser correctly accepts this valid Ada/CS program.

3.3.3 Test 3

This is a test to see if our parser can correctly parse programs that are a few levels deep with nested statements. The test file can be located at test/Nested.adb.

Running this program through our parser generated no errors, which was the expected response.

3.3.4 Test 4

This test is a stress test for our parser. The input file is the same as the stress test used in the scanner testing and can be located in the same path (test/StressTest2.adb).

The stress test passed.

3.4 Testing Error Recovery

The following test cases are meant to test our error recovery mechanism.

3.4.1 Test 1

This test case is a very simple error test case. It is simply missing a semi-colon after the null statement.

```
package Test1 is
body
begin
    null
end;
```

When running this program through our parser, the output is:

```
Repair Action on line 4 : INSERT Token SEMICOLON
    null
        ^
Rejected. Number of Error(s): 1
```

As it can be seen, the parser correctly determines that the source file did have an error, located that error, and made a suggestion on what the possible fix can be. The suggestion was to insert a semi-colon after the word null.

3.4.2 Test 2

This test case will be very similar to test case 1, but with an extra semi-colon.

```
package Test1 is
body
begin
    null;;
end;
```

And the output is:

```
Repair Action on line 4 : INSERT Token ID
    null;;
         ^
Rejected. Number of Error(s): 1
```

Again, the parser correctly identified the error and made a suggestion to insert an identifier token where the carrot is.

3.4.3 Test 3

This test uses the stress test file, as described above. Using this large test case, we created multiple grammatical errors through out the file to test how well our parser can recover from errors and continue parsing.

3.5 Test Suite

As with any program, doing a few test cases are never enough. We have thoroughly tested our compiler with many Ada/CS programs that we have written ourselves. Because we have so many, we felt that it would be better to list them here and give a brief description on what it tests in our compiler. All test files can be located in the test/ directory of our compiler.

`8ball.adb` - a magic 8 ball program
`MatrixMult.adb` - a matrix multiplying program
`Nested.adb` - a test for many nested statements
`ScannerErrorTest.adb` - a test that has lexical errors
`SortAlgorithms.adb` - some sorting algorithms
`StressTest.adb` - a stress test with some errors
`StressTest2.adb` - a stress test
`TestCase1.adb` - test case with missing semi colon
`TestCase2.adb` - test case with extra semi colon
`TestCase3.adb` - test case with more than 1 error
`TestCase4.adb` - test case with extra semi colon
`TestCase5.adb` - test case with multiple errors
`TestCase6.adb` - test case with multiple errors
`aggregate.adb` - test our grammar for aggregate
`arraytest.adb` - test our grammar for arrays
`assigntest.adb` - test grammar for assign statement
`attributetest.adb` - test grammar for attributes
`blocktest.adb` - test grammar for block statements
`caseSensitive.adb` - test case insensitivity
`casetest.adb` - test grammar for case statements
`conditional.adb` - test grammar for if conditions
`constanttest.adb` - test grammar for constants
`distance.adb` - a simple program
`elsetest.adb` - test grammar for else
`empty.adb` - an empty file test
`enumtest.adb` - test grammar for enums
`factorial.adb` - a simple factorial program
`forloop.adb` - test grammar for for loops
`iftest.adb` - test grammar for if statements
`operators.adb` - test grammar for operators
`optest.adb` - test scanner for operators
`rangedint.adb` - test grammar for integers
`recordvardenottest.adb` - test grammar for records and variants
`subtypetest.adb` - test grammar for type and subtypes
`wierd.adb` - test scanner for string literal parsing

4.0 Program output

4.1 Cross reference

As required, our program prints out a cross reference listing of all the user defined symbols of an input program.

Here is a simple Ada/CS program:

```
 1 package EightBall is
 2 body
 3     seed : Integer;
 4
 5     procedure seed(s : in Integer) is
 6     begin
 7         seed := s;
 8     end;
 9
10     function rand return Integer is
11     begin
12         seed := seed * 1103515245 + 12345;
13         return seed;
14     end;
15
16
17     str : String;
18     ans : Integer := 0;
19 begin
20     Read(str);
21
22     seed(str'Len);
23     ans := rand;
24
25     if ans mod 2 = 0 then
26         Write("YES!"; ", YEEEES!");
27     else
28         Write("No"; " Really No");
29     end if;
30 end;
```

The cross reference output is:

```
[ID ans]: Occurences
                : line: 18 col: 5
                : line: 23 col: 5
                : line: 25 col: 8
[ID EightBall]: Occurences
                : line: 1 col: 9
[ID Integer]: Occurences
                : line: 3 col: 12
                : line: 5 col: 27
                : line: 10 col: 26
                : line: 18 col: 11
[ID Len]: Occurences
```

```
                    : line: 22 col: 14
    [ID rand]: Occurences
                    : line: 10 col: 14
                    : line: 23 col: 12
    [ID Read]: Occurences
                    : line: 20 col: 5
    [ID s]: Occurences
                    : line: 5 col: 20
                    : line: 7 col: 17
    [ID seed]: Occurences
                    : line: 3 col: 5
                    : line: 5 col: 15
                    : line: 7 col: 9
                    : line: 12 col: 9
                    : line: 12 col: 17
                    : line: 13 col: 16
                    : line: 22 col: 5
    [ID str]: Occurences
                    : line: 17 col: 5
                    : line: 20 col: 10
                    : line: 22 col: 10
    [ID String]: Occurences
                    : line: 17 col: 11
    [ID Write]: Occurences
                    : line: 26 col: 9
                    : line: 28 col: 9
```

As it can be seen from the cross reference output, the compiler records all user defined symbols and records where it occurred and each occurrence throughout the source file.

5.0 References

1. Aho, A. V., Sethi, R., and Ullman, J. D., Compilers, Principles, Techniques, and Tools. Addison-Wesley, 1986.

2. Appel, A. W., Modern Compiler Implementation in Java. Cambridge University Press, Cambridge, 2002.

3. Burke, M. G., and Fisher, G. A., A practical method for LR and LL syntactic error diagnosis and recovery. In ACM Transactions on Programming Languages and Systems. (Vol. 9, No. 2, April 1987). ACM, New York, 1982, pp. 164-197.

4. Burke, M. G., and Fisher, G. A., A practical method for syntactic error diagnosis and recovery. In Proceedings of the SIGPLAN 82 Symposium on Compiler Construction (Jun 23-25, 1982, Boston). ACM, New York, 1982, pp. 67-78.

5. Fischer, C., LeBlanc, R., and Cytron, R. K., Crafting a Compiler Second Edition Draft. Addison Wesley Longman.