More On Recursive-Descent Parsing

© Allan C. Milne and Adam Sampson

Abertay University

v12.7.18

Where were we...

… the idea of recursive-descent parsing.

… why LL(1) was a useful property to have – implementing a recursive-descent parser, by writing recogniser methods based on the BNF.

… the relationship between the derivation sequence, the order of method calls in an RD parser, and the abstract syntax tree.

You've seen enough to make a start on the coursework now – semantics still to come...

Agenda.

- Ardkit's parser framework.
- Recognising EBNF Clauses.

(for reference – I'm going to go through this pretty quickly)

- Error Handling.
- Turner's approach.

The *Ardkit* Parser Framework.

- **`IParser`**
  - **`RdParser`**
    - **`RecoveringRdParser`**
- See the *Ardkit* parser reference.

The Parser Contract.

```
public interface IParser {
      List<ICompilerError> Errors { get; }
      bool Parse (TextReader src);
} // end IParser interface.
```

- The Parse() method performs the syntax analysis and returns true if no errors are detected.
- The Errors property provides access to detected errors; will be empty (not null) if no errors detected.

RdParser Overview.

- Requires access to the scanner (we wrote this) to read in tokens from the input.
- A set of standard *RD primitive* methods is provided.
- A *recogniser method* is defined for each non-terminal rule.
- Parsing is initiated by calling the recogniser method for the start symbol.

The *RdParser* Class.

```
public abstract class RdParser : IParser {

  protected IScanner       scanner;        // the scannner.
  protected List<ICompilerError> errs;     // collection of error.

  public RdParser (IScanner scan)  {
    this.scanner = scan;
    this.errs = null;
  } // end Parser constructor method

  … public API methods …

  protected abstract void recStarter ();

  … protected RD primitive methods …

} // end RdParser class.
```

The Public API.

```
public List<ICompilerError> Errors
{ get {  return errs;  } }

public virtual bool Parse (TextReader source) {
  errs       = new List<ICompilerError> ();
  scanner.Init (source, errs);
  scanner.NextToken();           // get the first token.
  try {                          // allow parse to be aborted.
    recStarter ();               // initiate parse.
    mustBe (Token.EndOfFile); // check no other text.
  } catch (CompilerErrorException) {}
  return errs.Count == 0;
} // end Parse method.
```

Creating A Concrete Parser.

- See the *Ardkit* how to use parser page.
- Define a class deriving from *RdParser*.
- The constructor method takes no arguments and calls the base constructor passing an instance of a concrete scanner.
- Override the *recStarter()* recogniser for the rule of the starter symbol of the language.
- Define private RD recogniser methods for each rule in the language specification.

A Concrete Example.

```
public class MyParser : RdParser {

  public MyParser ()
  : base (new MyScanner())
  { }

  protected override void recStarter()
  { … }

  private void recRule1 () { … }
  private void recRule2 () { … }
  … RD recognisers for EBNF rules  …

} // end MyParser class.
```

Using A Concrete Parser.

- Instantiate a parser object.
- Create a *TextReader* for the source program.
- Call the *Parse(…)* method of the parser.
- Process any errors returned.

```
StreamReader source = new StreamReader ("program.txt");

MyParser parser = new MyParser ();

parser.Parse (source);

foreach (CompilerError err in parser.Errors)
  Console.WriteLine (err);

Console.WriteLine ("{0} errors found.",
          parser.Errors.Count);
```

The RD Primitives.

- ***mustBe(…)*** : determines if the correct token type is present as the next input token; if so then consume it, otherwise we have an error.

```
protected virtual void mustBe (String shouldBe) {
   if (have (shouldBe)) {
     scanner.NextToken();
   } else {
     syntaxError (shouldBe);
   }
 } // end mustBe method.
```

- ***have(…)*** : determines if a token type is present as the next input token.

```
protected bool have (String mightBe)
{  return scanner.CurrentToken.Is (mightBe);  }
```

syntaxError Method.

- Call when the current input token is not the token expected.

```
protected virtual void syntaxError (String shouldBe) {
  errs.Add (new SyntaxError (scanner.CurrentToken, shouldBe));
  throw new CompilerErrorException ();
} // end syntaxError method.
```

- Create an appropriate error object and add it to the error collection.
- The *scanner.CurrentToken* object provides info on the position as well as the actual string and the type of the token in error.
- **Throw an exception that will be caught in the *Parse()* method to signal the end of the parsing process.**

Notes On The RD Primitives.

- They are defined as `protected` since they must be accessible to be used in the RD recogniser methods defined in a concrete parser subclass.

- *mustBe(...)* and *syntaxError(...)* are defined as `virtual` because they may be overridden in a subclass to provide a more sophisticated error recovery method.
  - *We will see this in* RecoveringRdParser *later...*

Recognising EBNF Clauses.

- BNF and EBNF.
- Some Notation.
- Recognising EBNF Clauses.
- Putting It All together.

Recognising BNF.

- The RD process we have seen up to now is used with standard BNF rules.
  - The recursive nature of these rules is reflected in the recursive calls of recogniser methods. (remember AST vs. call tree last time?)

- Hence the name recursive-descent;
  - *recursive* recogniser method calls
  - *descending* from the start symbol.

Recognising EBNF.

- EBNF clauses are **iterative** rather than **recursive** constructs.
  - They sit within a rule with other BNF.
  - They are recognised within the same overall approach used previously for BNF.
- An EBNF clause is recognised
  - by an iterative program statement
  - within the recogniser method of the rule in which it occurs.

Some Notation.

- The following is used in defining the EBNF recogniser code:
- α, β productions of terminal, non-terminal and/or further EBNF clauses.
- HAVE(α) The series of *have()* calls connected with || to check for the director set of α.
- REC(α) Recogniser code for α, might include *mustBe*(), recogniser method calls and further EBNF recogniser code.

Recognising (…)?

```
( α | … | β )?   →   if (HAVE(α)) REC(α)
                     … … …
                     else if (HAVE(β)) REC(β)
```

To be pedantic add
`else {}`

```
… ( , Identifier)? …
```

```
if (have(",")) {
  mustBe(",");
  mustBe(Token.IdentifierToken);
}
else {}  // do nothing
```

```
… ( : <Form> | <Block>)? …
```

```
if (have(":")) {
  mustBe (“:”);
  recForm();
} else if (have("let") ||
           have("for") || … ) {
  recBlock();
}
```

Recognising (...)

```
( α | … | β )  →  if (HAVE(α)) REC(α)
                  … … …
                  else...
```

```
else if (have(β)) REC(β)
else syntaxError("…");
```

```
( + | - )
```

```
if (have("+"))
  mustBe("+");
else if (have("-"))
  mustBe("-");
else
  syntaxError("+ or –");
```

```
(+Integer | -Integer | <Expr>)
```

```
if (have("+")) {
  mustBe("+");
  mustBe(Token.IntegerToken);
}
else if (have("-")) {
  mustBe("-");
  mustBe(Token.IntegerToken);
}
else recExpr();
```

Recognising (...)*

```
( α | … | β )*
 → while (HAVE(α)||…||HAVE(β)) {
       if (HAVE(α)) REC(α)
          …  …  …
       else REC(β)
   }
```

```
(to <Expr> | ,<Expr>)*
```

```
while (Have("to") || have(",")) {
  if (have("to"))
  { mustBe("to"); recExpr(); }
  else
  { mustBe(","); recExpr(); }
}
```

```
( , Identifier)*
```

```
while (have(",")) {
  mustBe(",");
  mustBe(Token.IdentifierToken);
}
```

Recognising (…)+

```
( α | … | β )+
 → do {
     if (HAVE(α)) REC(α)
     … … …
     else REC(β)
  } while (HAVE(α)||…||HAVE(β));
```

```
else if (have(β)) REC(β)
else syntaxError("…");
```

```
(to <Expr> | ,<Expr>)+
```

```
do {
  if (have("to"))
  { mustBe("to"); recExpr(); }
  else
  { mustBe(","); recExpr(); }
} while (have("to") || have(","));
```

```
( , Identifier )+
```

```
do {
  mustBe(",");
  mustBe(Token.IdentifierToken);
} while (have(","));
```

Putting It All Together.

```
<Block> ::= <Statement> |
            begin (<Declaration>)* (<Statement>)* end ;
```

```
private void recBlock() {
  if (have("let") || have("for") ||
      have("get") || have("put") )
    recStatement();
  else if (have("begin")) {
    mustBe("begin");
    while (have("declare"))
      recDeclaration();
    while ( ! have("end"))
      recStatement();
    mustBe("end");
  }
  else syntaxError("<Block>");
}
```

Note we test for not being the token that follows; sometimes more efficient and better for errors.

Another Example.

```
<Expression> ::= <Term> ( (+|-) <Term>)* .
```

```
private void recExpression() {
  recTerm();
  while (have("+") || have("-")) {
    if (have("+"))
         mustBe("+");
    else mustBe("-");
    recTerm();
  }
}
```

Don't check for error since the while ensures we have a + or –.

.

Error Handling.

- Error Handling.
- Error Detection.
- Error Reporting.
- Error Recovery.

Error Handling.

- *Detection* : find syntax errors, i.e. a token that isn't possible within the BNF derivation.
- *Reporting* : report on the location and cause of the error; this may be directly displayed or saved to some collection for later output.
- *Recovery* : take corrective action to allow the parsing to terminate or continue.

Error Detection.

There are 3 points at which errors are detected.

- In the scanner where there is an invalid character or token.
- In the *mustBe()* method where an expected token is not present.
- In a recogniser method where an explicit test is made and the current token does not match that expected for any of the alternative productions.

Error Reporting.

- When an error is detected it may be displayed immediately.
  - The *currentToken* object has the location and string information required.
  - the current source line is not available, unless read in another pass of the source stream.
- The error can be recorded in a collection of *Error* objects and output or processed after the parsing is completed.

Error Recovery.

This can take the form of

- *Termination* : terminate the parse immediately on an error being detected.
- *Recovery* : attempt to continue the parse by resynchronising the input stream with the parse.
- *Correction* : attempt to correct the error before continuing the parse.

Turner's Approach.

- Introduction.
- Implementing Recovery Mode.
- Implementing Recovery.
- Some Observations.

Introduction.

- A mechanism for simple error recovery designed specifically for RD parsing.
- Attributed to D.A.Turner (see his 1974 paper).
- Places no restrictions on the LL(1) specification.
- Is language-independent.
- The parser is considered to be in either normal mode or *recovering* mode.

A Recovering Parser.

```
public abstract class RecoveringRdParser : RdParser {

  private bool recovering;

  public RecoveringRdParser (Iscanner scan)
  : base (scan)
  { recovering = false; }

  public bool IsRecovering
  { get { return recovering; } }

  protected override void mustBe (String shouldBe)
  { … … …}

  protected override void syntaxError (String shouldBe) {
    if (recovering) return;
    errs.Add (new SyntaxError (scanner.CurrentToken, shouldBe));
    recovering = true;
  }
} // end RecoveringRdParser class.
```

Implementing Recovery.

```
protected override void mustBe (String shouldBe) {
  if (recovering) {
    while (!have(shouldBe) && !scanner.EndOfFile) {
      scanner.NextToken();
    }
    if (scanner.EndOfFile) return;
    scanner.NextToken();
    recovering = false;
  }
  else {
    if (have(shouldBe)) {
      scanner.NextToken();
    } else {
      syntaxError (shouldBe)
    }
  }
} // end mustBe method.
```

Implementation Comments.

- Inherits from the "one-shot" *RdParser* class met previously.
  - See *Ardkit parser reference.*
- Overrides *mustBe(…)* and *syntaxError(…)* to implement the recovery process.
- Exposes additional *IsRecovering* property that will be used in semantic analysis.
- Creating a parser for a specific language is exactly the same as before except that it will inherit from *RecoveringRdParser* rather than *RdParser*.

Some Observations.

- Recovery not initiated if error detection takes place explicitly in recogniser methods;
  - i.e. by calling *syntaxError(…)* .
- All errors are detected, recovery is not perfect; consider handling the following error types.
  - missing token **`a := 1`**
  - incorrect token **`lt a := 2`**
  - additional token **`let let a := 3`**
- Recovery mechanism can be tuned for a specific BNF.

Summary.

So Now You Can …

… write recogniser methods for all BNF and EBNF rules.

… describe the different aspects of error handling.

… include error detection and recovery in your parser.

… appreciate the subtlety of error recovery.

… write a parser for any LL(1) specification!