Error Recovery in a LL(1) Parser

Compiler Construction & Project
Lecture 7

Error Recovery

- The aim of error detection and recovery is to
  - Detect no errors in a correct program
  - Find as many errors as possible in an incorrect program with a minimum number of spurious errors being detected.
  - After detecting an error, correct parsing of the remainder of the program should commence as soon as possible.

Current What?

- We will assume that the Parser has 2 important instance variables:
  - currentSymbol
    - The ‘kind’ of token currently being processed
    - Equal to the Yytoken ‘symbol’ instance variable
  - currentToken
    - The current instance of Yytoken that is being processed
    - Has instance variables int symbol, String text, int line, int charBegin, int charEnd

Error Recovery in a LL(1) Parser

- Detection of errors is easy with an LL(1) parser.
  - You are expecting a particular symbol, or one of a small group of symbols, and if you do not get such a symbol then an error has occurred.

- If no error recovery is performed then the input token is not advanced and many errors may be flagged at the same location.
  - Flagging spurious errors is confusing and frustrating to the user.

- We will consider two schemes for error recovery in a recursive descent parser.
Scheme 1

- We modify our lexer interface **mustbe**
- This technique is outlined in the paper: "Error Diagnosis and Recovery in One Pass Compilers" by D.A. Turner.
- This approach makes the **mustbe** interface throw input tokens away when we detect an error.
  - We stop throwing input away until we reach a plausible symbol...

**Note**
- This is the basis of every sensible error recovery strategy – **throw erroneous input away**
- Error recovery strategies differ in how much input they discard
- The less input you discard, the better!

We also need to suppress extraneous error messages (we really only want to see the first one).
  - Easy: our error reporting method (we'll call it `syntaxError()`) does not issue an error message when a flag called "recovering" is set to `true`.
  - `syntaxError()` sets this flag...
- Code for `syntaxError()`:

```java
public void syntaxError (String errMsg) {
  if (!recovering) {
    // call to error routine to
    // print/record the error message.
    error(currentToken, errMsg);
    recovering = true;
  }
}
```

**Scheme 1 – mustbe()**

```java
void mustbe (int symbol) {
  if (recovering) {
    while ((currentSymbol != symbol) && (currentSymbol != EOF)) {
      nextSymbol();
    }
    if (currentSymbol == symbol) {
      nextSymbol(); // consume the symbol
      recovering = false; // have recovered
    } else {
      if (currentSymbol == symbol) {
        nextSymbol(); // consume the symbol
      } else {
        syntaxError(errormsg(symbol));
      }
    }
  } else {
    if (currentSymbol == symbol) {
      nextSymbol(); // consume the symbol
    } else {
      syntaxError(errormsg(symbol));
    }
  }
}
```

**Scheme 1 – Performance**

- Pretty poor, actually.
- Leaves (potentially) large amounts of input un-parsed.
- The Java compiler implements this, cf:

```java
class Tst {
  public static void main() {
    int i, j, k, l;
    System.out.println("Hello");
  }
}
[kevin@akurra bug]$ javac Tst.java
Tst.java:4: ';' expected
  int i, j, k, l;
^ 1 error
```

Looks OK doesn’t it?
If we probe a little deeper, we see what the compiler is actually doing

Consider...

```java
class Tst {
    public static void main() {
        int i, j, k, l;
        i = j;
        k = j;
        System.out.println("Hello");
    }
}
```

Discarding too much input!!

```bash
[kevin@akurra bug]$ javac Tst.java
Tst.java:4: ';' expected
    int i j, k, l;
               ^
Tst.java:5: cannot resolve symbol
symbol : variable l
location: class Tst
    i = l;
Tst.java:6: cannot resolve symbol
symbol : variable k
location: class Tst
    k = j;
Tst.java:6: cannot resolve symbol
symbol : variable j
location: class Tst
    k = j;
```

This scheme can be tuned for better performance

This changes, depending on where in the parser `mustbe()` was called from.

This still does not go far enough...

This is the scheme you need to implement...

The basic approach of this scheme is simply that when a subparser locates an error:

- It reports it
- Skips text until a plausible follow symbol is encountered (so that the subparser can continue).

**NOTE:** "Follow" symbols (here) are the set of \( V \) which can logically follow \( V_N \) in the current context.

- This means that this is not exactly the follow symbol set we discussed, but is based on it.

For each subparser, there must be a parameter denoting a set of possible follow symbols. You could use a number of data types to represent this set.
The Method

- Write a method called testAndSkip that takes three parameters:
  - A set of symbols valid at this point
  - A set of additional symbols, not necessarily valid, that should not be skipped over
  - An error message to report
- We can (harmlessly) add calls to this wherever we see fit.
- We always call it on entry and exit from a subparser...

```java
void testAndSkip(SymbolSet validSet, SymbolSet additionalSet, String err) {
    if (!validSet.contains(currentSymbol)) {
        error(currentSymbol, err);
    }
    while (!validSet.contains(currentSymbol) || additionalSet.contains(currentSymbol)) {
        nextSymbol();
    }
}
```

Will Report Error if not in this one
Will stop when it finds one in this set or the first set

Why SymbolSet?

```
public interface SymbolSet {
    public void add(int symbol);
    public boolean contains(int symbol);
    public void addAll(SymbolSet otherSet);
    public void addAll(int[] symbols);
    public SymbolSet union(SymbolSet otherSet);
}
```

- Remember what you have learnt in CS1 on basic types and container classes

Modifying the Parser

- We add a parameter to each parser (fsys – the follow symbol set).
- Each parser calls testAndSkip with the set of valid start symbols, and fsys as parameters, eg:

```java
static SymbolSet fooStart;
static {
    fooStart = new ConcreteSymbolSet();
    fooStart.addAll({tSym1, .. , tSymN});
}

void fooParser(SymbolSet fsys) {
    testAndSkip(fooStart, fsys, "Error at start of a Foo!");
    ...
}
```

- And also at the end, just with fsys
Modifying the Parser

- We can also place calls to this method wherever we observe poor error reporting performance by the compiler
  - Note that this “can go forever”
  - Don’t waste too much time trying to fine-tune the performance of your compiler...

- When we invoke a sub-parser, we add in the symbols valid at this point with the existing fsys
  - So the “factor” subparser gets to have just about all the symbols in fsys – why?

The Java Example

- Our example has a missing comma in a declaration list.
  - This is a relatively common error

- We’d like it to stop skipping input sooner than it does (it stops on the semicolon).
  - Stop on the next identifier?

Almost universal rule:
- Never have identifier in the stopping symbol set!
  - Why?

The Java Example

```java
static SymbolSet commaSet;
static {
    commaSet = new ConcreteSymbolSet();
    commaSet.add(tCOMMA);
}

// A declaration list
while (have(tCOMMA)) {
    mustBe(tIDENTIFIER);
    // We've found that people sometimes omit the commas, so we check...
    testAndSkip(fsys.union(commaSet), fsys, "Error in declaration list missing comma?");
}
```

Some Points to Remember

- Don’t forget to add in the “new” symbols valid at this point in the parse when you call a sub-parser (augmentation of the follow symbol set)

- Sometimes you don’t need to call testAndSkip – Expression/Term/Factor type parsers
  - Once you get to “Expression”, “Factor” is the only way out – just do the skipping there

- Sometimes it might be a good idea to remove some things from the sets...

- TestAndSkip is a tool for error recovery – you can place it where it will help, in addition to the start/end of parsers

- Don’t waste time adding error recovery – just to the basic thing, tune later when you have some time
  - Plus: compiler writers don’t make the same errors as first year students – your tuning may be pointless!