Steps to Work on Assignment 5

Jianguo Lu

February 2, 2021

(Remember to read this slides together with the lecture slides. They complement each other)

Outline

Run the example code posted on the course web site

- Create a clean directory
  - This is very important: if you use a directory in previous assignments, you may use some programs left from previous assignments and you are not aware of that
- Download the following Java programs (rightclick then 'save as')
  - RecursiveDescent.java
  - Symbol.java
  - Calc3Scanner.java
  - Calc2Symbol.java
- Note that Symbol, Calc3Scanner, Calc2Symbol are classes from earlier examples. They are part of a scanner.
- Our focus is to write the parser, i.e., RecursiveDescent.java
- Compile everything and run

```
>javac *.java
>java RecursiveDescent
```

Understand the program: Read input in RecursiveDescent.java

```
static int pointer = -1;
static ArrayList tokens = new ArrayList();

 public static void main(String[] args) throws Exception {
  Calc3Scanner scanner = new Calc3Scanner(new FileInputStream(new
       File("calc2.input")));
  Symbol token;
  while ((token=scanner.yylex()).sym!= Calc2Symbol.EOF) {
    tokens.add(token);
  }
  tokens.add(token);    // add EOF as the last token in the array
  ...  ...
}
```

- It reads the input into *tokens* array. Note that *tokens* is declared as an instance variable of the class.
- Each token is of type Symbol. We reuse the Symbol class from A3 and A4.
- We reuse Calc3Scanner that is generated before.
- Note that yylex() is the default method to get the next token in scanner that is generated by JLlex.
- Calc2Symbol is used to record the type of the symbol (e.g., whether it is an ID or EOF).

Understand the program: Parse the input


```
public static void main(String[] args) throws Exception {
  ...
  boolean legal= program() && nextToken().sym==Calc2Symbol.EOF;
  System.out.println(legal);
}
```

- Call the recursive descent parser

Understand the program: Parse the input

```
public static void main(String[] args) throws Exception {
  ...
  boolean legal= program() && nextToken().sym==Calc2Symbol.EOF;
  System.out.println(legal);
}
```

- Call the recursive descent parser
- The start symbol is 'program'.

Understand the program: Parse the input


```
public static void main(String[] args) throws Exception {
  ...
  boolean legal= program() && nextToken().sym==Calc2Symbol.EOF;
  System.out.println(legal);
}
```

- Call the recursive descent parser
- The start symbol is 'program'.
- The input is valid if it is a 'program' and we reached the EOF.

Understand the 'program' method

```
/**  program—>statement program
      program—>statement
*/
static boolean program() {
  check whether  "program—>statement program" can be applied
  if not, check whether  "program—>statement" can be applied
}
```

- Recall the grammar rule for 'program', which defines a sequence of statements
- e.g., the derivation for three statements *statement statement statement* is

$$\begin{aligned} program &\Rightarrow statement\ program \\ &\Rightarrow statement\ statement\ program \\ &\Rightarrow statement\ statement\ statement \end{aligned} \tag{1}$$

- We will try these two rules in sequence
- If the first rule (recursion rule) fails, we fall back to try the second one (the base case)
- How to implement this in real Java?

Understand the 'program' method


```
/**  program-->statement program
      program-->statement
*/
static boolean program() throws Exception


  if (statement() && program())  return true;
```

- Try the first rule

Understand the 'program' method

```
/**  program-->statement program
      program-->statement
*/
static boolean program() throws Exception


  if (statement() && program())  return true;

  if (statement())  return true;
```

- Try the first rule
- Try the second rule

```
/**  program-->statement program
      program-->statement
*/
static boolean program() throws Exception

  int savePointer = pointer;
  if (statement() && program())  return true;
  pointer = savePointer;
  if (statement())  return true;
```

- Try the first rule
- Try the second rule
- When the first rule fails, move the pointer back to the starting position

Understand the 'program' method


```
/**  program-->statement program
      program-->statement
*/
static boolean program() throws Exception

  int savePointer = pointer;
  if (statement() && program())  return true;
  pointer = savePointer;
  if (statement())  return true;
  pointer = savePointer;
  return false;
```

- Try the first rule
- Try the second rule
- When the first rule fails, move the pointer back to the starting position
- When both rules fails, fall back to the starting position

The pattern for writing a recursive descent parser

- Each non-terminal corresponds to a method
- Each rule corresponds to a if statement
- The if statement involves calls of methods (Nonterminals)
- What if there are terminals? e.g.,
- How to implement the Expr rule? (note that there is a terminal '+')

```
/** expr--> term+expr
    expr--> term
*/
```

Exp example


```
/** expr--> term+expr
    expr--> term
*/
static boolean expr() throws Exception

  if (term() && nextToken().sym==Calc2Symbol.PLUS && expr()) return true;
```

- Try the first rule. The meaning is that we want to see a term, a terminal '+', and an expression in that sequence. Note the different from the 'program' example, i.e., here we need to treat a terminal '+'. using nextToken().

Exp example

```
/** expr--> term+expr
    expr--> term
*/
static boolean expr() throws Exception

  if (term() && nextToken().sym==Calc2Symbol.PLUS && expr()) return true;

  if (term()) return true;
```

- Try the first rule. The meaning is that we want to see a term, a terminal '+', and an expression in that sequence. Note the different from the 'program' example, i.e., here we need to treat a terminal '+'. using nextToken().
- Try the second rule

Exp example

```
/** expr--> term+expr
    expr--> term
*/
static boolean expr() throws Exception
  int savePointer = pointer;
  if (term() && nextToken().sym==Calc2Symbol.PLUS && expr()) return true;
  pointer = savePointer;
  if (term()) return true;
```

- Try the first rule. The meaning is that we want to see a term, a terminal '+', and an expression in that sequence. Note the different from the 'program' example, i.e., here we need to treat a terminal '+'. using nextToken().
- Try the second rule
- When the first rule fails, move the pointer back to the starting position

Exp example

```
/** expr--> term+expr
    expr--> term
*/
static boolean expr() throws Exception
  int savePointer = pointer;
  if (term() && nextToken().sym==Calc2Symbol.PLUS && expr()) return true;
  pointer = savePointer;
  if (term()) return true;
  pointer = savePointer;
  return false;
```

- Try the first rule. The meaning is that we want to see a term, a terminal '+', and an expression in that sequence. Note the different from the 'program' example, i.e., here we need to treat a terminal '+'. using nextToken().
- Try the second rule
- When the first rule fails, move the pointer back to the starting position
- When both rules fails, fall back to the starting position

The most common error

What if we write the grammar as follows

```
/** expr--> expr+term
    expr--> term
*/
```

The most common error

What if we write the grammar as follows

```
/** expr--> expr+term
    expr--> term
*/
```

The corresponding program would be

```
static boolean expr() throws Exception
  ...
  if (expr() && nextToken().sym==Calc2Symbol.PLUS && term()) return true;
  ...
  if (term()) return true;
  ...
```

The most common error

What if we write the grammar as follows

```
/** expr--> expr+term
    expr--> term
*/
```

The corresponding program would be

```
static boolean expr() throws Exception
  ...
  if (expr() && nextToken().sym==Calc2Symbol.PLUS && term()) return true;
  ...
  if (term()) return true;
  ...
```

What is the problem of this code?

Infinit loop


```
static boolean expr() {
  ...
  if (expr() ...) ...
  ...
}
```

- Your program will freeze
- There is an infinite loop when you have a recursive call without moving forward the token pointer
- Why the previous example does not have infinite loop: it tests 'term()' first. While doing so, it consumes some tokens, i.e., moving the pointer forward.
- How to avoid infinite loop: change the grammar so that it is not left recursive.

Avoid infinite loops

- Test your program before submitting
- If your submission page hangs there for a long time, probably your code has infinite loop
- If that happens, email us to kill your process.

Outline

Renaming and Copying your previous Scanner

- Rename RecursiveDescent.java into A5.java
- Copy your A4Scanner.java into your A5 directory, and rename it into A5Scanner.java
- Copy your A4Sym.java into your A5 directory, and rename it as A5Sym.java
- Symbol.java will remain to be the same
- Change your main method to read from a5.tiny.
- Copy a test file in the directory. Make sure it is named 'a5.tiny'
- Try to compile and run:

```
javac *.java
```

- If compilation goes well, try to run the parser

```
java A5.java
```

- There could be error messages ...

Change the main method in A5.java

```
public static void main(String[] args) throws Exception {
  BufferedWriter bw=new BufferedWriter(new FileWriter("a5.output"))
      ;
  A5Scanner scanner=new A5Scanner(new FileInputStream(new File("A5.
      tiny")));
  Symbol token;
  while ((token=scanner.yylex()).sym!= A5Sym.EOF) {
    tokens.add(token);
  }
  tokens.add(token);
  boolean legal= program() && nextToken().sym==A5Sym.EOF;
  bw.write((legal)?"legal":"illegal");
  bw.close();
}
```

- It is mostly the same as in our starter code except
  - input/output file names
  - classes for the scanner
- note that yylex() is the default method to get the next token in scanner that is generated by JLlex. Make sure that it is the the method name in the scanner
- Make sure you have EOF in A5Sym. If your earlier A4Sym.java does not have that symbol, or you have a different symbol name, you can add one or change it manually.

Make it run

```
static boolean statement() throws Exception {
  int savePointer = pointer;
  if (assignment() && nextToken().sym == Calc2Symbol.SEMI) {
    return true;
  }
  pointer = savePointer;
  return false;
}
```

- There could be many possible error messages in your previous step, depending on your A4Scanner and A4Sym.

Make it run

```
static boolean statement() throws Exception {
  int savePointer = pointer;
  if (assignment() && nextToken().sym == Calc2Symbol.SEMI) {
    return true;
  }
  pointer = savePointer;
  return false;
}
```

- There could be many possible error messages in your previous step, depending on your A4Scanner and A4Sym.
- e.g., you need to replace Calc2Symbol systematically with A5Sym

Make it run

```
static boolean statement() throws Exception {
  int savePointer = pointer;
  if (assignment() && nextToken().sym == Calc2Symbol.SEMI) {
    return true;
  }
  pointer = savePointer;
  return false;
}
```

- There could be many possible error messages in your previous step, depending on your A4Scanner and A4Sym.
- e.g., you need to replace Calc2Symbol systematically with A5Sym
- You need to align the symbol name (here SEMI) with the symbol name in your A4.

Make it run

```
static boolean statement() throws Exception {
  int savePointer = pointer;
  if (assignment() && nextToken().sym == Calc2Symbol.SEMI) {
    return true;
  }
  pointer = savePointer;
  return false;
}
```

- There could be many possible error messages in your previous step, depending on your A4Scanner and A4Sym.
- e.g., you need to replace Calc2Symbol systematically with A5Sym
- You need to align the symbol name (here SEMI) with the symbol name in your A4.
- After these renaming operations, the program should be able to run.

Make it run

```
static boolean statement() throws Exception {
  int savePointer = pointer;
  if (assignment() && nextToken().sym == Calc2Symbol.SEMI) {
    return true;
  }
  pointer = savePointer;
  return false;
}
```

- There could be many possible error messages in your previous step, depending on your A4Scanner and A4Sym.
- e.g., you need to replace Calc2Symbol systematically with A5Sym
- You need to align the symbol name (here SEMI) with the symbol name in your A4.
- After these renaming operations, the program should be able to run.
- You can try to parse simple 'program's like

```
x=x+1
```

Make it run

```
static boolean statement() throws Exception {
  int savePointer = pointer;
  if (assignment() && nextToken().sym == Calc2Symbol.SEMI) {
    return true;
  }
  pointer = savePointer;
  return false;
}
```

- There could be many possible error messages in your previous step, depending on your A4Scanner and A4Sym.
- e.g., you need to replace Calc2Symbol systematically with A5Sym
- You need to align the symbol name (here SEMI) with the symbol name in your A4.
- After these renaming operations, the program should be able to run.
- You can try to parse simple 'program's like

```
x=x+1
```

- Up to now, it is exactly the same as RecursiveDescent.java, except renaming and try to use your scanner in A4.

Outline

Add one rule at a time

- Implement all the rules in the Tiny language specification

Add one rule at a time

- Implement all the rules in the Tiny language specification
- But always add one rule at a time, and test the rule with a test case once it is added

Add one rule at a time

- Implement all the rules in the Tiny language specification
- But always add one rule at a time, and test the rule with a test case once it is added
  - e.g., program is a sequence of functions. And define a java method for function.

Add one rule at a time

- Implement all the rules in the Tiny language specification
- But always add one rule at a time, and test the rule with a test case once it is added
  - e.g., program is a sequence of functions. And define a java method for function.
  - Watch out for infinite loops (left recursion).

Add one rule at a time

- Implement all the rules in the Tiny language specification
- But always add one rule at a time, and test the rule with a test case once it is added
  - e.g., program is a sequence of functions. And define a java method for function.
  - Watch out for infinite loops (left recursion).
  - Make sure the pointer is put back every time the rule fails (the IF statement is false)

Add one rule at a time

- Implement all the rules in the Tiny language specification
- But always add one rule at a time, and test the rule with a test case once it is added
  - e.g., program is a sequence of functions. And define a java method for function.
  - Watch out for infinite loops (left recursion).
  - Make sure the pointer is put back every time the rule fails (the IF statement is false)
- This process is time consuming, but would be less frustrating.

Outline

RecursiveDescent parser is slow

- Is it really slow?

RecursiveDescent parser is slow

- Is it really slow?
  - Try to run on some long programs. e.g., there is a long test case in our test case list.

RecursiveDescent parser is slow

- Is it really slow?
    - Try to run on some long programs. e.g., there is a long test case in our test case list.
    - You can observe that the parser pauses for a few seconds.

RecursiveDescent parser is slow

- Is it really slow?
  - Try to run on some long programs. e.g., there is a long test case in our test case list.
  - You can observe that the parser pauses for a few seconds.
- Why it is slow?

RecursiveDescent parser is slow

- Is it really slow?
  - Try to run on some long programs. e.g., there is a long test case in our test case list.
  - You can observe that the parser pauses for a few seconds.
- Why it is slow?
  - It tries rules one after another. It does not know beforehand which rule should be used.

RecursiveDescent parser is slow

- Is it really slow?
  - Try to run on some long programs. e.g., there is a long test case in our test case list.
  - You can observe that the parser pauses for a few seconds.
- Why it is slow?
  - It tries rules one after another. It does not know beforehand which rule should be used.
    - In LR parsing, we always know which is the write rule to use

RecursiveDescent parser is slow

- Is it really slow?
  - Try to run on some long programs. e.g., there is a long test case in our test case list.
  - You can observe that the parser pauses for a few seconds.
- Why it is slow?
  - It tries rules one after another. It does not know beforehand which rule should be used.
    - In LR parsing, we always know which is the write rule to use
    - There is no backtracking in LR and LL parsing

RecursiveDescent parser is slow

- Is it really slow?
  - Try to run on some long programs. e.g., there is a long test case in our test case list.
  - You can observe that the parser pauses for a few seconds.
- Why it is slow?
  - It tries rules one after another. It does not know beforehand which rule should be used.
    - In LR parsing, we always know which is the write rule to use
    - There is no backtracking in LR and LL parsing
    - That is the whole point of developing advanced parsing techniques

RecursiveDescent parser is slow

- Is it really slow?
  - Try to run on some long programs. e.g., there is a long test case in our test case list.
  - You can observe that the parser pauses for a few seconds.
- Why it is slow?
  - It tries rules one after another. It does not know beforehand which rule should be used.
    - In LR parsing, we always know which is the write rule to use
    - There is no backtracking in LR and LL parsing
    - That is the whole point of developing advanced parsing techniques
    - Speed matters very much.

RecursiveDescent parser is slow

- Is it really slow?
  - Try to run on some long programs. e.g., there is a long test case in our test case list.
  - You can observe that the parser pauses for a few seconds.
- Why it is slow?
  - It tries rules one after another. It does not know beforehand which rule should be used.
    - In LR parsing, we always know which is the write rule to use
    - There is no backtracking in LR and LL parsing
    - That is the whole point of developing advanced parsing techniques
    - Speed matters very much.
- It scans the input multiple passes (recall that we need to put the pointer to the previous location again and again).

RecursiveDescent parser is slow

- Is it really slow?
  - Try to run on some long programs. e.g., there is a long test case in our test case list.
  - You can observe that the parser pauses for a few seconds.
- Why it is slow?
  - It tries rules one after another. It does not know beforehand which rule should be used.
    - In LR parsing, we always know which is the write rule to use
    - There is no backtracking in LR and LL parsing
    - That is the whole point of developing advanced parsing techniques
    - Speed matters very much.
- It scans the input multiple passes (recall that we need to put the pointer to the previous location again and again).
  - LR or LL parsing scan input once

RecursiveDescent parser is slow

- Is it really slow?
  - Try to run on some long programs. e.g., there is a long test case in our test case list.
  - You can observe that the parser pauses for a few seconds.
- Why it is slow?
  - It tries rules one after another. It does not know beforehand which rule should be used.
    - In LR parsing, we always know which is the write rule to use
    - There is no backtracking in LR and LL parsing
    - That is the whole point of developing advanced parsing techniques
    - Speed matters very much.
- It scans the input multiple passes (recall that we need to put the pointer to the previous location again and again).
  - LR or LL parsing scan input once
  - Hence the complexity of modern parsers is $O(n)$ where n is the length of the input

RecursiveDescent parser is slow

- Is it really slow?
  - Try to run on some long programs. e.g., there is a long test case in our test case list.
  - You can observe that the parser pauses for a few seconds.
- Why it is slow?
  - It tries rules one after another. It does not know beforehand which rule should be used.
    - In LR parsing, we always know which is the write rule to use
    - There is no backtracking in LR and LL parsing
    - That is the whole point of developing advanced parsing techniques
    - Speed matters very much.
- It scans the input multiple passes (recall that we need to put the pointer to the previous location again and again).
  - LR or LL parsing scan input once
  - Hence the complexity of modern parsers is $O(n)$ where n is the length of the input
- How to make it faster?

How to make it faster?


```
Exp -->Term + Exp
Exp -->Term
```

- Consider the rules (and the corresponding Java program) for parsing Exp

How to make it faster?


```
Exp —>Term + Exp
Exp —>Term
```

- Consider the rules (and the corresponding Java program) for parsing Exp
- We try the first rule (scan the term)

How to make it faster?


```
Exp -->Term + Exp
Exp -->Term
```

- Consider the rules (and the corresponding Java program) for parsing Exp
- We try the first rule (scan the term)
- If the first rule does not work out, we try the second rule.

How to make it faster?

```
Exp —>Term + Exp
Exp —>Term
```

- Consider the rules (and the corresponding Java program) for parsing Exp
- We try the first rule (scan the term)
- If the first rule does not work out, we try the second rule.
- The input is scanned from the very beginning

How to make it faster?


```
Exp -->Term + Exp
Exp -->Term
```

- Consider the rules (and the corresponding Java program) for parsing Exp
- We try the first rule (scan the term)
- If the first rule does not work out, we try the second rule.
- The input is scanned from the very beginning
- Time is wasted by trying the first rule and scanning the input

How to make it faster?

```
Exp —–>Term + Exp
Exp —–>Term
```

- Consider the rules (and the corresponding Java program) for parsing Exp
- We try the first rule (scan the term)
- If the first rule does not work out, we try the second rule.
- The input is scanned from the very beginning
- Time is wasted by trying the first rule and scanning the input
- The problem is 'Term' is repeated in two rules

How to make it faster?


```
Exp —>Term + Exp
Exp —>Term
```

- Consider the rules (and the corresponding Java program) for parsing Exp
- We try the first rule (scan the term)
- If the first rule does not work out, we try the second rule.
- The input is scanned from the very beginning
- Time is wasted by trying the first rule and scanning the input
- The problem is 'Term' is repeated in two rules
- We can factor out the common terms, then backtracking is not needed

```
Exp     —>Term  Terms
Terms —>  + Exp   |Epsilon
```

How to make it faster?

```
Exp  —>Term + Exp
Exp  —>Term
```

- Consider the rules (and the corresponding Java program) for parsing Exp
- We try the first rule (scan the term)
- If the first rule does not work out, we try the second rule.
- The input is scanned from the very beginning
- Time is wasted by trying the first rule and scanning the input
- The problem is 'Term' is repeated in two rules
- We can factor out the common terms, then backtracking is not needed

```
Exp      —>Term  Terms
Terms —>   + Exp    |Epsilon
```

- It is called left factoring

How to make it faster?


```
Exp -->Term + Exp
Exp -->Term
```

- Consider the rules (and the corresponding Java program) for parsing Exp
- We try the first rule (scan the term)
- If the first rule does not work out, we try the second rule.
- The input is scanned from the very beginning
- Time is wasted by trying the first rule and scanning the input
- The problem is 'Term' is repeated in two rules
- We can factor out the common terms, then backtracking is not needed

```
Exp     -->Term  Terms
Terms -->  + Exp   |Epsilon
```

- It is called left factoring
- There are other techniques, together they are called predicative parsing.