Bottom-Up Parsing – LR(1)

 Previously we have studied top-down or LL(1) parsing.

 The idea here was to start with the start symbol and keep expanding it until the whole input was read and matched.

 In bottom-up or LR(1) parsing we do exactly the opposite, we try to match the input to a rule and then keep reducing the input replacing it with the non-terminal of the rule. The last step is to replace the current input with the start-symbol.
In LL(1) parsing we replaced non-terminal symbols with functions that did the expansions and the matching for us.

In LR(1) parsing we use a stack to help us find the correct reductions.

Given the stack an LR(1) parser has four available actions:

- **Shift** – push an input token on the stack
- **Reduce** – pop elements from the stack and replace by a non-terminal
- **Accept** – accept the current program
- **Reject** – reject the current program
Bottom-Up Parsing – LR(1)

Grammar:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;empty&gt;</td>
<td>p + x 1 ;</td>
<td>Shift</td>
</tr>
<tr>
<td>p</td>
<td>+ x 1 ;</td>
<td>Shift</td>
</tr>
<tr>
<td>p +</td>
<td>x 1 ;</td>
<td>Shift</td>
</tr>
<tr>
<td>p + x</td>
<td>1 ;</td>
<td>Reduce</td>
</tr>
<tr>
<td>p + var</td>
<td>1 ;</td>
<td>Reduce</td>
</tr>
<tr>
<td>p + exp</td>
<td>1 ;</td>
<td>Reduce</td>
</tr>
<tr>
<td>p + exp 1</td>
<td>;</td>
<td>Reduce</td>
</tr>
<tr>
<td>p + exp num</td>
<td>;</td>
<td>Reduce</td>
</tr>
<tr>
<td>p + exp exp</td>
<td>;</td>
<td>Reduce</td>
</tr>
<tr>
<td>p exp</td>
<td>;</td>
<td>Shift</td>
</tr>
<tr>
<td>p exp ;</td>
<td>&lt;empty&gt;</td>
<td>Reduce</td>
</tr>
<tr>
<td>stmt</td>
<td>&lt;empty&gt;</td>
<td>Reduce</td>
</tr>
<tr>
<td>prog</td>
<td>&lt;empty&gt;</td>
<td>Accept</td>
</tr>
</tbody>
</table>
Bottom-Up Parsing – LR(1)

```
p + x 1 ;
```

Stack:

- `<empty>`
- `p`
- `p +`
- `p + x`
- `p + var`
- `p + exp`
- `p + exp 1`
- `p + exp num`
- `p + exp exp`
- `p exp`
- `p exp ;`
- `stmt`
- `prog`

Diagram:

```
  prog
   /\
  stmt
   / \
  p   exp
  /   /
/     \
var   num
|     /
|     \
|      
|       
|        
|         
|          
|           
|            
x        1
```
**Bottom-Up Parsing – LR(1)**

Let’s try an illegal sentence

```
p + x s ;
```

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;empty&gt;</td>
<td>p + x s ;</td>
<td>Shift</td>
</tr>
<tr>
<td>p</td>
<td>+ x s ;</td>
<td>Shift</td>
</tr>
<tr>
<td>p +</td>
<td>x s ;</td>
<td>Shift</td>
</tr>
<tr>
<td>p + x</td>
<td>s ;</td>
<td>Reduce</td>
</tr>
<tr>
<td>p + var</td>
<td>s ;</td>
<td>Reduce</td>
</tr>
<tr>
<td>p + exp</td>
<td>s ;</td>
<td>Shift</td>
</tr>
<tr>
<td>p + exp s</td>
<td>;</td>
<td>Reject</td>
</tr>
</tbody>
</table>
Let’s try it with the original grammar!

```
p + x 1 ;
```

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;empty&gt;</td>
<td>p + x 1 ;</td>
<td>Shift</td>
</tr>
<tr>
<td>p</td>
<td>+ x 1 ;</td>
<td>Shift</td>
</tr>
<tr>
<td>p +</td>
<td>x 1 ;</td>
<td>Shift</td>
</tr>
<tr>
<td>p + x</td>
<td>1 ;</td>
<td>Reject</td>
</tr>
</tbody>
</table>

There is a conflict between the lhsvar rule and rhsvar rule here, we do not have enough information to select one rule over the other. This is called a reduce/reduce conflict in bottom-up parsing terminology.

That means, even though our grammar is a perfectly legal context-free grammar, it is not a grammar that can be used by a bottom-up parser, we say that the grammar is not LR(1).

We shouldn’t feel too smug about LL(1) parsers, later on we will encounter perfectly legal CFG’s that are not LL(1).
Bottom-Up Parsing – LR(1)

- LR(1) parsers are implemented in such tools as Yacc (Unix) and Bison (Linux)
- However, the tool we are using, ANTLR, implements LL(1) parsing *

* Actually ANTLR implement LL(k) parsing, more on that later
Parser Generators

- Writing parsers by hand is difficult and time consuming.
- The resulting parsers are difficult to maintain and extend.
- Ideally, we would like a tool that reads a.g file (e.g., exp0.g) and generates a parser from that file.
Parser Generators

Grammar File \[\rightarrow\] Parser Generator \[\rightarrow\] Parser Code (e.g. Java)

That looks very much like a translator!
Parser generators are an example of a domain specific language translator!

ANTLR is a parser generator, it translates a .g file into parser code written in Java.
Using ANTLR

- Let’s assume we have a project Exp1Bytecode
- Assume that I have a folder
  
  /Users/lutz/CSC402 Projects/EXP1BYTECODE
  
  that holds all the source files
Generating a Parser for exp0

Assume we have our grammar for exp0 stored in a file called exp0.g

```plaintext
grammar exp0;

prog    : stmt+ ;
stmt    : 'p' exp ';' |
        's' lhsvar exp ';' |
exp     : '+' exp exp |
        '-' exp exp |
        '(' exp ')' |
        rhsvar |
        num |
lhsvar  : 'x' | 'y' | 'z' |
rhsvar  : 'x' | 'y' | 'z' |
num     : '0' ... '9' |

WS : ( ' ' | \t | \r | \n ) { $channel=HIDDEN; ;
```
Generating a Parser for exp0
Generating a Parser for exp0
Generating a Parser for exp0

```java
import org.antlr.runtime.*;

public class <grammar name> {
    public static void main(String[] args) throws Exception {
        // check if we have an input file, if not print out error and abort
        if (args.length == 0) {
            System.out.println("Usage: java <grammar name> <input file>");
            System.exit(0);
        } else {
            System.out.println("Processing: " + args[0]);
        }

        // set up and initialize our lexer and parser objects
        // open up the input file
        ANTLRFileStream input = new ANTLRFileStream(args[0]);
        // create the lexer with the input stream
        <grammar name>Lexer lexer = new <grammar name>Lexer(input);
        // create a token stream for the parser
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // create a parser object with the token stream
        <grammar name>Parser parser = new <grammar name>Parser(tokens);

        // call the toplevel recursive descent parsing function -- start symbol
        parser.start_rule();
    }
}
```

The Parser Driver Template
Generating a Parser for exp0
Generating a Parser for exp0

```
$ ls
exp0.g  exp0.java  exp0Lexer.java  exp0Parser.java  exp0.tokens
$

$ javac *.java
$

$ ls
exp0.class  exp0.java  exp0Lexer.java  exp0Parser.java
exp0.g  exp0Lexer.class  exp0Parser.class  exp0.tokens
$
```
Generating a Parser for exp0

```
$ cat p1.exp
s x 1;
s y 2;
p + x y;

$ java exp0 p1.exp
Processing: p1.exp
$ 
```
Actions

- Making the generated parser do something useful.
- In the hand-coded parser you can add code anywhere in order to make the parser do something useful...like counting ‘p’ statements.
- In parsers generated by parser generators we use something called ‘actions’ we insert into the grammar.
Actions

Actions can be inserted anywhere in the grammar using a pair of braces

Example:

```java
prog : ({System.println("parsing stmt");} stmt)+
```
Idea

- Build a reader that reads an exp0 program and prints out the kind of statements and expressions as they are being recognized.
- Call the program ‘exp0finder’
- We want to use ANTLR and the exp0.g grammar.
- First step: extend the grammar with actions.
grammar exp0finder;

// prints out a message as the rules for
// expanding a non-terminal are selected.

@members {
void symbol(String s) {
    System.out.println(s);
}
}

prog : ({symbol("stmt")}; stmt )+ ;

stmt : {symbol("p-stmt");} 'p' exp ';';
   | {symbol("s-stmt");} 's' lhsvar exp ';';
   ;

exp : {symbol("+ exp");} '+' exp exp
    | {symbol("- exp");} '-' exp exp
    | {symbol("() exp");} '(' exp ')' 
    | {symbol("rhsvar");} rhsvar
    | {symbol("num");} num
    ;

lhsvar : {symbol("x");} 'x'
        | {symbol("y");} 'y'
        | {symbol("z");} 'z'
        ;

rhsvar : 'x' | 'y' | 'z'
        ;

num : '0' | '1' | '2' | '3' | '4' | '5'
    | '6' | '7' | '8' | '9'
    ;

// tell the parser to ignore white space
WS : (' ' | \t | \v | \n) { $channel=HIDDEN; };
import org.antlr.runtime.*;

public class exp0finder {
    public static void main(String[] args) throws Exception {

        // check if we have an input file, if not print out error and abort
        if (args.length == 0) {
            System.out.println("Usage: java exp0finder <input file> ");
            System.exit(0);
        } else {
            System.out.println("Processing: " + args[0]);
        }

        // set up and initialize our lexer and parser objects
        // open up the input file
        ANTLRFileStream input = new ANTLRFileStream(args[0]);

        // create the lexer with the input stream
        exp0finderLexer lexer = new exp0finderLexer(input);

        // create a token stream for the parser
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        // create a parser object with the token stream
        exp0finderParser parser = new exp0finderParser(tokens);

        // call the toplevel recursive descent parsing function -- start symbol
        parser.prog();
    }
}
Running the Reader

```
s x 1;
s y 2;
p (+ x y);
```

```
bash-3.2$ java exp0finder p1.exp0
Processing: p1.exp0
stmt
s-stmt
x
num
stmt
s-stmt
s-stmt
y
num
stmt
p-stmt
() exp
+ exp
rhsvar
rhsvar
bash-3.2$
```
Assignment

- Assignment #2 – see website