CS 3240

Project Phase II

Jordan Killpack

Ka Young Kim

Byung Min

Jay Zuerndorfer

Scanner for LL(1) Grammar

LL1GrammarScanner class takes in grammar specification and token specification as text files. The scanner needs to be instantiated with the names of the two text files. The scanner is implemented based on the following assumptions:

1. The grammar specification is given in BNF form; therefore, a rule must be expressed as <symbol> ::= _expression_.
2. The grammar specification does not contain any left-recursion and has been left-factored.
3. A nonterminal is enclosed with a set of "<>".
4. The first nonterminal in the grammar specification is the start symbol. For example, in the case Mini-RE, <MiniRE-program> needs to be the very first word in the grammar specification.
5. Per PhaseII-Clarifications.doc, the special notation for epsilon is "<epsilon>". Although it is in the form of nonterminal, the scanner handles it differently than nonterminals.

The scanner first reads the token specification and generates a collection of tokens. Then it reads the grammar specification and associate each symbol (nonterminal) with the expression that defines it. In other words, each nonterminal on the left hand side of "::=" is mapped to the expression on the right hand side. If it successfully scans the grammar specification, it outputs a rule map keyed by the symbols (instances of Nonterminal) themselves, a list of tokens, and the start symbol. For convenience, it also outputs nonterminal map keyed by the names of the nonterminals (String).

LL(1) Parser Generator

Similarly to LL1GrammarScanner, LL1ParserGenerator class takes in grammar specification and token specification as text files. The parser generator instantiates an LL1GrammarScanner with the grammar and token specifications. Then, it uses the rule map generated by the scanner to construct a parsing table. If a rule exists for a given pair of a token and a nonterminal, the parser generator associates the token and the nonterminal with the rule. To generate a parsing table, the parser generator constructs first and follow sets for each nonterminal prior to constructing the parsing table. The algorithm for constructing first set and follow set runs until it determines that the terminating condition has met. Note that the algorithms are likely to run infinitely if the input grammar is left recursive.

Mini-RE Parser

MiniReLL1Parser class takes in a Mini-RE script and Mini-RE grammar specification. It first generates the parser generator; then, it uses the parsing table outputted by the parser generator to create an AST for the input script. Before constructing the AST, the parser checks the validity of the input script. If the input script seems to be invalid, it throws an exception. Otherwise, it constructs the AST for the script. For the construction of an AST, the parser maintains two different stacks: parsing stack for the grammar and the input stack for the input script. The input script is parsed and each word in the script is pushed onto the stack in reverse manner so that the top of the stack is “begin” and the bottom of the stack is “end.” Then, the parser pushes the start symbol onto the parsing stack and recursively builds tree until the input stack is empty. If parsing is successful, both stacks should be empty.

Mini-RE Interpreter

The Mini-RE interpreter simply traverses the parse tree created by the parser described above. There are only a few types of statements in Mini-RE-- assignment statements, replace statements, and print statements-- so it was easy to simply use the structure of the grammar to figure out what type of statement a given <statement> node in the parse tree represented. We stored variables in memory by keeping track of identifier / value pairs in two hashmaps: one for integer types and another for string list types. Overall, the interpreter section of our code is quite straightforward. We initially tested this functionality using a simple recursive descent parser, but as soon as the parser generator was complete, we swapped out the recursive descent parser.

Phase 1 Code

In anticipation of Phase 2, we fixed our Phase 1 code. It now passes all the test cases provided on T-Square.

Test Cases

Test Case for LL(1) Parser Generator

Three grammar specifications from the textbook (Louden) have been tested: test_grammar1_ll1.txt, test_grammar1_ll2.txt, test_grammar3_ll1.txt. Their token specifications are named: test_token1_ll1.txt, test_token2_ll1.txt, test_token3_ll1.txt respectively. The driver for LL1ParserGenerator test is LL1ParserGeneratorTest.java.

Following is the test result of test_grammar1_ll1.txt.

Grammar Specification:

```
<exp> ::= <term> <exp'>
<exp'> ::= <addop> <term> <exp'> | epsilon
<addop> ::= + | -
<term> ::= <factor> <term'>
<term'> ::= <mulop> <factor> <term'> | epsilon
<mulop> ::= *
<factor> ::= ( <exp> ) | number
```

Token Specification:

```
+ - * ( ) number
```

Output:

```
First Sets
First(<exp>) = {  (  number  }
First(<exp'>) = {  +  -  epsilon  }
First(<addop>) = {  +  -  }
First(<term>) = {  (  number  }
First(<term'>) = {  *  epsilon  }
First(<mulop>) = {  *  }
First(<factor>) = {  (  number  }


Follow Sets
First(<exp>) = {  (  number  }
First(<exp'>) = {  +  -  epsilon  }
First(<addop>) = {  +  -  }
First(<term>) = {  (  number  }
First(<term'>) = {  *  epsilon  }
First(<mulop>) = {  *  }
First(<factor>) = {  (  number  }


Parsing Table
format: (<nonterminal>,terminal)
```

```
      <nonterminal> ::= rule

(<exp'>,+)
      <exp'> ::= <addop> <term> <exp'>
(<addop>,+)
      <addop> ::= +
(<exp'>,-)
      <exp'> ::= <addop> <term> <exp'>
(<addop>,-)
      <addop> ::= -
(<term'>,*)
      <term'> ::= <mulop> <factor> <term'>
(<mulop>,*)
      <mulop> ::= *
(<exp>,()
      <exp> ::= <term> <exp'>
(<term>,()
      <term> ::= <factor> <term'>
(<factor>,()
      <factor> ::= ( <exp> )
(<exp>,number)
      <exp> ::= <term> <exp'>
(<term>,number)
      <term> ::= <factor> <term'>
(<factor>,number)
      <factor> ::= number
```

Test Case for LL(1) Interpreter

Six test scripts (minire_test_script1.txt - minire_test_script6.txt) were tested for LL(1) Interpreter. InterpreterLL1Test.java is the driver for the interpreter.