Documentatie LFDC

Opris Vlad Marius 935, Socaciu Mihai 938

**Github link:** <https://github.com/Socca98/LFTC-2020>

Lab6 – Parser LR(0)

**Requirements**

***Statement: Implement a parser algorithm (cont)***

PART 2: Deliverables

1. Algorithm corresponding to parsing tables (if needed) and parsing strategy
2. Class ParserOutput - DS and operations corresponding to choice 2.a/2.b/2.c ([lab 5](https://moodle.cs.ubbcluj.ro/mod/assign/view.php?id=2841)) (required operations: transform parsing tree into representation; print DS to screen and to file)

**Remark**:  
- if the table contains conflicts, you will be helped to solve them. It is important to print message containing row (symbol in LL(1), respectively state in LR(0)) and column (symbol) where the conflict appears. For LL(1) values (αα,i) might also help

Parsing strategy: **LR(0)**

Parser.py

LR(0) parsing algorithm bottom-up to construct the syntax tree.

**Properties**:

grammar – the Grammar

workingStack – stack

inputStack – stack

output – list of production rule integer (stack)

Methods:

**closure** (productions)

* **Description**: Constructs one clojure of the canonical collections
* **In**: productions – List of productions for the closure
* **Out**: closure ~ list of tuples

**Ex**: [('S1', ['.', 'S']), ('S', ['.', 'aA']), ('S', ['.', 'bB'])]

**go\_to** (state, symbol)

* **Description**: Transition from a state to another using a terminal or non-terminal.

Used in generating parsing table and the canonical collection.

* **In**: state – String; symbol – String
* **Out**: list of productions on which we call closure() to obtain all in that state

**canonical\_collection** ()

* **Description**: Constructs all closures, kept in a list of lists.

ex: [

[('S1', ['.', 'S']), ('S', ['.', 'aA']), ('S', ['.', 'bB'])],

[('S1', ['S', '.'])],

...

]

* **In:** -
* **Out**: collection of clojures

**generate\_table** ()

* **Description**: Generates the parsing table used to check the input tokens.

Obtains the canonical collection then iterates using go\_to().

A dictionary for each state ”i”.

* **In:** -
* **Out**: parsing table. List of dictionaries containing action and maybe non/terminals

**ex:** [

{'action': 'shift', 'S': 1, 'A': 2, 'a': 3, 'b': 4},

{'action': 'acc'},

{'action': 'shift', 'A': 6, 'a': 3, 'b': 4},

{'action': 'reduce 2'},

{'action': 'reduce 1'}

]

**parse** (input\_string)

* **Description**: Constructs parse tree using the generated parsing table.

inputStack - list of strings. PIF code for each token read from txt file.

['33', '18', '19', '16', '25', '0', '6', '1', '15', '31', '0', '15', '17']

table – same as in generate\_table()

workingStack - used to keep symbols and actions while iterating inputStack

['0']

* **In**: input\_string: taken from PIF (

**ex**: ‘aabb’

‘program, {, statement’

* **Out**: output - [0,2,1,1,2]

Production rule numbers needed to obtain the input\_string starting from grammar.S

Representation of the parsing tree = **derivations string**

ParserOutput.py

Properties:

parser – The LR(0) parser

Methods:

**derivations\_string** (output\_parser)

* **Description**: Constructs one clojure of the canonical collections
* **In**: output\_parser – list of integers representing reduce states
* **Out**: result – list of tuples; list of production rules converted from their number

**print\_to\_console** (output\_parser)

* **Description**: Print derivations string to console.
* **In**: output\_parser – list of integers representing reduce states

**write\_to\_file** (output\_parser)

* **Description**: Write derivations string to file.
* **In**: output\_parser – list of integers representing reduce states

Lr(0) conflicts

In the **generate\_table()** function, for each state/clojure we count the number of shift/reduce/accept operations. If the length of productions in that state does not correspond to the count, it means the state consists of conflicts.

Ex: first\_rule\_cnt = 3, second\_rule\_cnt = 1

But state I7 contains 4 productions and len(state) != first\_rule\_cnt, so one is a reduce action.

In case counting is not equal to the number of productions we signal the **state** and the **symbol** of the conflict:

else: else:  
 conflict\_msg = 'Conflict! State I' + str(index) + ': ' + str(state) + '\nSymbol: ' + beta[0]  
 raise (Exception(conflict\_msg))

**The UML diagram**

