Sintax direct translation: Boolean Calculator

Abstract

Design of a grammar and his implementation using a top-down/ bottom-top analyzer/translator that represent a boolean calculator using python.

Introduction

The boolean calculator allow this operations represented by the examples of this table:

Input	Output
x := true;	
print x;	The result is 1
y := false and x;	
print not y;	The result is 1
print x and not y;	The result is 1
print not (x and not y);	The result is 0
x := not x;	
z := true or not (x and not y);	

- There are `Id's`
- There are `or` binary and left asociative.
- There are `and` and is binary and left asociative and more priority than the `or` operator.
- There are asign operator `:=`
- There are print sentence `print *values*`
- There are `True` and `False` constants.
- There are `()`

Methodology

Stage : 1

Design the grammar for a bottom-top translator with his translation scheme.

Sintactic and semantic rules

Sintactic rules	Semantic rules
entry -> `print` exprOR `;`	write('The result is {exprOR.s} ;')
entry -> asign `;`	
asign -> `ID` `=` exprOR	table[ID.lexval] = exprOR.s
exprOR -> exprOR `or` exprAND	exprOR.s = exprOR_1.s or exprAND.s
exprOR -> exprAND	exprOR.s = exprAND.s
exprAND -> exprAND `and` boolean	exprAND.s = exprAND_1.s and boolean.s

Sintactic rules	Semantic rules
exprAND -> boolean	exprAND.s = boolean.s
boolean -> not boolean	boolean.s = !boolean.s
boolean -> CBOOLEAN	boolean.s = CBOOLEAN.lexval
boolean -> ID	boolean.s = table[ID.leval]
boolean -> (exprOR)	boolean.s = exprOR.s

Translation scheme

Translation scheme
entry -> print exprOR ; { write(‘The result is {exprOR.s} ;’) }
entry -> asign ;
asign -> ID = exprOR { table[ID.lexval] = exprOR.s }
exprOR -> exprOR or exprAND { exprOR.s = exprOR_1.s or exprAND.s }
exprOR -> exprAND { exprOR.s = exprAND.s }
exprAND -> exprAND and boolean { exprAND.s = exprAND_1.s and boolean.s }
exprAND -> boolean { exprAND.s = boolean.s }
boolean -> not boolean { boolean.s = !boolean.s }
boolean -> CBOOLEAN { boolean.s = CBOOLEAN.lexval }
boolean -> ID { boolean.s = table[ID.leval] }
boolean -> (exprOR) { boolean.s = exprOR.s }

Stage: 2

Adapt the grammar for a top-bottom translator. For do that we have to transform the left recursion to right recursion and refactorize the resulting grammar.

Sintactic rules
entry -> print exprOR ;
def -> asign ;
asign -> ID = exprOR
exprOR -> exprAND exprOR’
exprOR’ -> or exprAND exprOR’
exprOR’ -> e
exprAND -> boolean exprAND’
exprAND’ -> and boolean exprAND’
exprAND’ -> e
boolean -> not boolean
boolean -> CBOOLEAN
boolean -> ID
boolean -> (exprOR)

Stage: 3

Add the translation scheme to stage 2 adapting the semantic rules of stage 1.

Stage: 4

Implement the top-bottom recursive translator resuling from stage 3 using *Python*.