Assignment 3 – Solution

Advanced Computing Concepts for Engineering

2015

The work that you turn in for this assignment must represent your individual effort. You are welcome to help your fellow students understand the material of the course and the meaning of the assignment questions, however, the answer that you submit must be created by you alone.

Q0. (a) [10] Design a readable context-free grammar for the 'tree expressions' over the alphabet {'(', ')', id, '.'}. Each tree expression is a finite sequence that fits one of the following rules.

- id is a tree expression.
- A sequence of tree expressions surrounded by parentheses is a tree expression.
- A sequence of at least 2 tree expressions surrounded by parentheses, where the last is preceded by a dot, is a tree expression.

Here are some examples of tree expressions

( )
(id)
((id) (id id) id)
(( ) (id id) . id)

The following are not examples of tree expressions

(
id)
( )) (( )
(( ) . (id id) id)
(. (id id))

Solution: Many solutions are possible. Here is one.

Tree	→	'(' Trees ')' | '(' NonEmptyTrees '.' Tree ')' | id
Trees	→	ϵ | Tree Trees
NonEmptyTrees	→	Tree Trees

Note that what I called tree expressions are more commonly known as S-expressions and are used in the syntax of LISP.

(b) [10] Using your grammar from part (a) trace the top-down predictive algorithm on the input

$$(\ (\ )\ (\ \mathsf{id}\ )\ .\ \mathsf{id})\ \$$$

Solution: First add a new start nonterminal where \$ represents the end of input.

$$\text{Tree}' \rightarrow \text{Tree}\ \$$$

	$s_\blacktriangle \alpha$,	t	**Action**
	$\epsilon_\blacktriangle$ Tree′,	(( ) (id) . id) $	Produce Tree′ → Tree $
⊢	$\epsilon_\blacktriangle$ Tree $,	(( ) (id) . id) $	Produce Tree ⟶ '(' NonEmptyTrees '.' Tree ')'
⊢	$\epsilon_\blacktriangle$ '(' NonEmptyTrees '.' Tree ')' $,	(( ) (id) . id) $	Shift
⊢	(▲ NonEmptyTrees '.' Tree ')' $,	( ) (id) . id) $	Produce NonEmptyTrees → Tree Trees
⊢	(▲ Tree Trees '.' Tree ')' $,	( ) (id) . id) $	Produce Tree → '(' Trees ')'
⊢	(▲ '(' Trees ')' Trees '.' Tree ')' $,	( ) (id) . id) $	Shift
⊢	((▲ Trees ')' Trees '.' Tree ')' $,	) (id) . id) $	Produce Trees → ϵ
⊢	((▲ ')' Trees '.' Tree ')' $,	) (id) . id) $	Shift
⊢	( )▲ Trees '.' Tree ')' $,	(id) . id) $	Produce Trees → Tree Trees
⊢	( )▲ Tree Trees '.' Tree ')' $,	(id) . id) $	Produce Tree → '(' Trees ')'
⊢	( )▲ '(' Trees ')' Trees '.' Tree ')' $,	(id) . id) $	Shift
⊢	(( ) (▲ Trees ')' Trees '.' Tree ')' $,	id) . id) $	Produce Trees → Tree Trees
⊢	(( ) (▲ Tree Trees ')' Trees '.' Tree ')' $,	id) . id) $	Produce Tree → id
⊢	(( ) (▲ id Trees ')' Trees '.' Tree ')' $,	id) . id) $	Shift
⊢	(( ) (id▲ Trees ')' Trees '.' Tree ')' $,	) . id) $	Produce Trees → ϵ
⊢	(( ) (id▲ ')' Trees '.' Tree ')' $,	) . id) $	Shift
⊢	(( ) (id)▲ Trees '.' Tree ')' $,	. id) $	Produce Trees → ϵ
⊢	(( ) (id)▲ '.' Tree ')' $,	. id) $	Shift
⊢	(( ) (id) .▲ Tree ')' $,	id) $	Produce Tree → id
⊢	(( ) (id) .▲ id ')' $,	id) $	Shift
⊢	(( ) (id) . id▲ ')' $,	) $	Shift
⊢	(( ) (id) . id)▲ $,	$	Shift
⊢	(( ) (id) . id) $▲ ϵ,	ϵ	

(c) [10] Design an LL(1) CFG for tree expressions. Calculate the selector sets for your grammar.

Solution: (Of course there are other possibilities.)

Tree′	→	Tree $	{'(', id}
Tree	→	id	{id}
Tree	→	'(' AfterLeft	{'('}
AfterLeft	→	')'	{')'}
AfterLeft	→	Tree MoreTree	{'(', id}
MoreTree	→	'.' Tree ')'	{'.'}
MoreTree	→	')'	{')'}
MoreTree	→	Tree MoreTree	{'(', id}

(d) [10] Design a recursive descent parser based on your grammar from part (c). Your parser should either return a tree representing the input or call 'error'. The tree is formed from 4 kinds of nodes

- NULL nodes are leaf nodes. I.e., they have no children. They correspond to the tree expression ()

- ID nodes are leaf nodes. They represent the expression id.
- CONS nodes have exactly 2 children. They represent expressions of the form (x . y) where x and y are (respectively) represented by the left and right children of the CONS node. That is (x . y) is represented by CONS(X, Y) where X and Y are trees representing x and y.

Furthermore a list without a dot such as

$$(x_0\ x_1\ \cdots\ x_n)$$

(with $n \geq 0$) will be represented by a tree

$$\text{CONS}(X_0, \text{CONS}(X_1, \cdots \text{CONS}(X_n, \text{NULL}) \cdots)) .$$

And list with a dot such as

$$(x_0\ x_1\ \cdots\ x_n\ .\ x_{n+1})$$

(with $n \geq 0$) will be represented by a tree

$$\text{CONS}(X_0, \text{CONS}(X_1, \cdots \text{CONS}(X_n, X_{n+1}) \cdots)) .$$

Solution:

```
// Tree'   →   Tree $    {'(', id}
proc Tree'() is
    val result = Tree() ; expect($) ; return result

// Tree    →   id              {id}
// Tree    →   '(' AfterLeft   {'('}
proc Tree() is
    if t(0) = id then ( consume ; return ID )
    else if t(0) = '(' then ( consume ; return AfterLeft() )
    else error

// AfterLeft   →   ')'              {')'}
// AfterLeft   →   Tree MoreTree    {'(', id}
proc AfterLeft() is
    if t(0) = ')' then ( consume ; return NULL )
    else if t(0) ∈ {'(', id} then (
        val left = Tree()
        val right = MoreTree()
        return CONS(left, right) )
    else error
```

```
  // MoreTree   →   '.' Tree ')'        {'.'}
  // MoreTree   →   ')'                 {')'}
  // MoreTree   →   Tree MoreTree       {'(', id}
proc MoreTree() is
    if t(0) = '.' then (
        consume() ;
        val tree = Tree() ;
        expect(')') ;
        return tree )
    else if t(0) ∈ {')'} then (
        consume() ;
       return NULL )
    else if t(0) ∈ {'(', id} then (
        val left = Tree()
        val right = MoreTree()
        return CONS(left, right) )
    else error
```

(e) Optional: Code your parser in Java and test it. If he gets a chance, your instructor will provide some JUnit test cases.

Q1 [10]. In class we proved the unsolvability of the halting problem in two ways. We proved that a Turing Machine can not solve the halting problem for Turing Machines and that 'idealized Java' can not solve the halting problem for 'idealized Java'.

Show that the halting problem for 'idealized Java' can not be solved by a Turing Machine You may assume either that any Turing Machine can be translated into an equivalent idealized Java program, and conversely, any idealized Java program can be translated into an equivalent Turing Machine.

Solution:

Suppose that there is a TM hj that takes as input an idealized Java program and an input for the program, always terminates, and always gives an answer of true if the Java program halts on the given input and false if it does not. Then hj could be translated into a idealized Java program that does the same. But no such idealized Java program exists, as shown in class.

Alternative solution:

Again suppose that there is a TM hj that takes as input an idealized Java program and an input for the program, always terminates, and always gives an answer of true if the Java program halts on the given input and false if it does not. Then we could compose hj with a Turing machine that translates Turing machines to Java. This would make a Turing machine that takes a TM and a string as input and reports whether the TM halts on that input. This contradicts Turing's result that no such TM exists.