July 11, 2014

Compilers

Surname, Name	
Student identifier	

1. Specify in *Lex* a program that takes as input a text file and prints the text lines (sequences of characters terminated by a newline) which are composed of exactly three **w** letters separated between them by other characters.

2. Given the following grammar G in BNF notation, we ask to transform G into an equivalent non left-recursive grammar G* and then, based on the complete parsing table, determine whether G* is LL(1).

$A \rightarrow B\ \mathbf{a} \mid \boldsymbol{\varepsilon}$
$B \rightarrow C\ \mathbf{a}$
$C \rightarrow A\ \mathbf{b} \mid \boldsymbol{\varepsilon}$

3. Given the BNF at point 2 above, we ask to (*i*) outline the corresponding LR(1) parsing table, (*ii*) trace the LR(1) parsing of phrase **baa**, and (*iii*) based on the trace, outline the syntax tree of the given phrase.

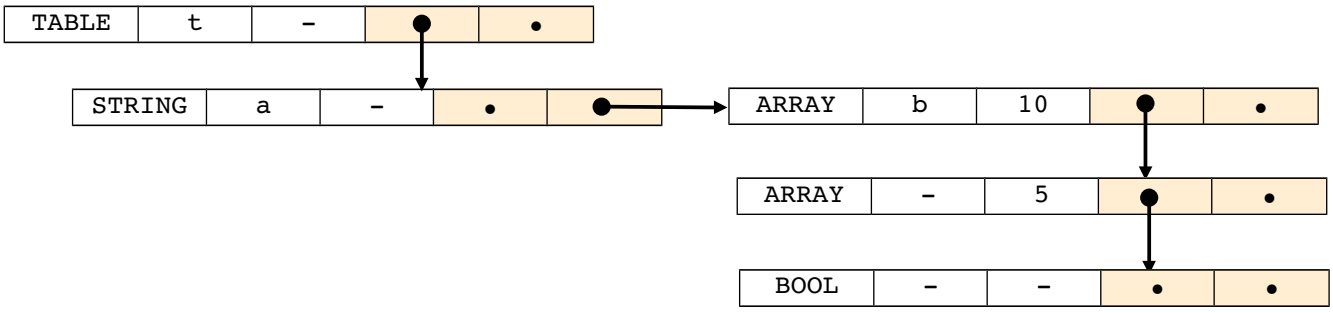
4. A language for type definitions is specified by the following BNF:

def → **id :** *type*
type → **int** | **string** | **bool** | *table-type* | *array-type*
table-type → **table (** *attr-list* **)**
attr-list → *def* **,** *attr-list* | *def*
array-type → **array [num]** *of type*

We ask to codify in *Yacc* the generator of the corresponding type trees, assuming that each node is qualified by a `domain` ∈ {`INT`, `STRING`, `BOOL`, `TABLE`, `ARRAY`}, a `name` (for variables and attributes), a `dimension` (size of the array), and pointers `child` (first child) and `brother` (right brother). For instance, the following declaration:

```
t: table(a: string, b: array [10] of array [5] of bool)
```

shall generate the following type tree:



Note: If the type of the variable is simple, it shall be represented by a single node.

5. Based on all reasonable semantic constraints of a strongly typed language, specify the attribute grammar relevant to the following BNF (in particular, in **foreach** loop, *expr* shall be an array with element type equal to the type of variable **id**):

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-stat* | *if-stat* | *foreach-stat*
def-stat → *id-list* **:** *type*
id-list → **id ,** *id-list* | **id**
type → **int** | **bool** | *array-type*
array-type → **array [intconst] of** *type*
assign-stat → **id :=** *expr*
expr → *expr* **+** *expr* | *expr* **and** *expr* | **-** *expr* | **not** *expr* | **(***expr* **)** | **id** | **intconst** | **boolconst**
if-stat → **if** *expr* **then** *stat-list* **else** *stat-list* **endif**
foreach-stat → **foreach id in** *expr* **do** *stat-list* **endfor**

assuming each node of the type tree being qualified by fields `domain` ∈ {`INT`, `BOOL`, `ARRAY`}, `size` (array dimension), and `child` (pointer to array element type), and the availability of the following auxiliary functions:

- `insert(name,type)`: inserts variable `name` and its `type` into the symbol table;
- `lookup(name)`: returns type of variable `name` (if cataloged) or **nil**;
- `typeEqual(t1,t2)`: checks the equality of types `t1` and `t2`;
- `simpleNode(domain)`: creates a type node for `domain` ∈ {`INT`, `BOOL`};
- `arrayNode(size,type)`: creates an array type node with dimension `size` and child type `type`;
- `error(message)`: prints relevant error `message` and terminates the analysis.

6. Given the language for the manipulation of integers, defined by the following BNF,

program → *stat-list*
stat-list → *stat* *stat-list* | *stat*
stat → **id :=** *expr*
expr → *expr* **+** *expr* | *expr* ***** *expr* | **id** | **num**

assuming a concrete syntax tree where nodes are qualified by fields `symbol` (the grammar symbol), `lexeme` (lexical string), `p1` (pointer to first child), `p2` (pointer to second child), and `p3` (pointer to third child), we ask to specify a procedure of P-code generation based on the following requirements:

a. Operands are evaluated from left to right;

b. Unlike the addition, the multiplication is evaluated in short circuit, based on the following rule: if the first operand is 0 then the result is 0 (otherwise, fully evaluation of multiplication is required);

c. The language of the P-machine includes the following set of instructions:

LDA <id>	(loading of address of variable **<id>** on stack)
LOD <id>	(loading of value of variable **<id>** on stack)
LDC <const>	(loading of integer constant **<const>** on stack)
ADD	(addition)
MUL	(multiplication)
EQU	(equality, pushing either `TRUE` or `FALSE` on the stack)
JMF <label>	(jumps if `FALSE`)
JMP <label>	(unconditional jump)
LAB <label>	(implicit address)
STO	(store)

Note: In the P-machine, integers cannot be treated as booleans.