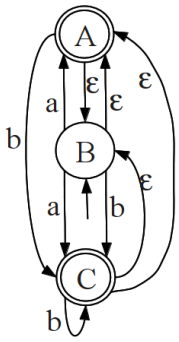
September 10, 2012

Compilers

Surname, Name	
Email	

1. After generating the DFA equivalent to the following NFA, specify the BNF expressing the regular language relevant to the DFA.



2. Codify the recursive-descent parser of the language defined by the following EBNF, also checking that each phrase ends with EOF.

program → {*stat*}
stat → (*def-stat* | *assign-stat* | *if-stat* | *while-stat*) **;**
def-stat → *type* **id** {**,** **id**}
type → **int** | **string** | **bool**
assign-stat → **id** **:=** *expr*
expr → **id** *op expr* | **(** *expr* **)** | **id** | **const**
op → **+** | **-** | ***** | **/** | **and** | **or**
if-stat → **if** *expr* **then** {*stat*}$^{+}$ [**else** {*stat*}$^{+}$] **endif**
while-stat → **while** *expr* **do** {*stat*}$^{+}$ **endwhile**

3. With reference to the following BNF, after constructing the portion of the LR(1) parsing automaton up to the states reached by a single transition from the initial state, check whether this portion of automaton includes conflicts:

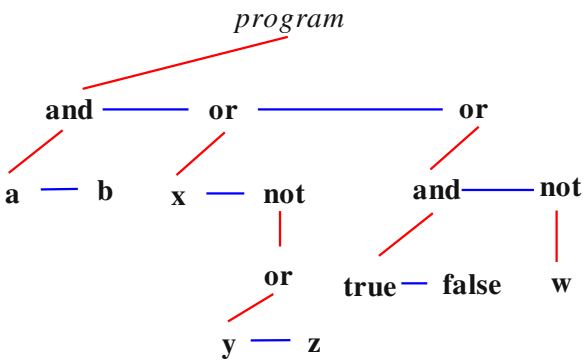
A → **a** *B* | *C*
B → **b** *B* | *C*
C → *C* **b** | **a**

4. Codify in Yacc the generator of the binary abstract trees relevant to the language defined by the following BNF:

program → *expr-list*
expr-list → *expr* **;** *expr-list* | *expr* **;**
expr → *expr* **or** *term* | *term*
term → *term* **and** *factor* | *factor*
factor → **not** *factor* | **(** *expr* **)** | **id** | **boolconst**

based on the following example:

```
a and b;
x or not (y or z);
true and false or not w;
```



5. With reference to the following BNF:

program → *stat-list*
stat-list → *stat* **;** *stat-list* | *stat* **;**
stat → *def-stat* | *assign-stat* | *if-stat* | *for-stat*
def-stat → **id :** *type*
type → **int** | **bool**
assign-stat → **id :=** *expr*
expr → *expr* + *expr* | *expr* * *expr* | *expr* **or** *expr* | *expr* **and** *expr* | **not** *expr* | **id** | **intconst** | **boolconst**
if-stat → **if** *expr* **then** *stat-list* **else** *stat-list* **endif**
for-stat → **for id** = *expr* **to** *expr* **do** *stat-list* **endfor**

Specify the attribute grammar based on the following constraints:

- Variable names are unique;
- Referenced variables shall exist;
- Arithmetic and logical operators are applied to integers and booleans, respectively;
- Conditions are of type boolean;
- Within the **for** statement, the counting variable is of type integer;
- No mixed expressions are allowed;
- The lexical value of identifiers is stored in the `lexval` field of the tree node;
- A symbol table is used to catalog variables by means of the following functions:
  `void insert(name, type)`: inserts variable `name` with `type`;
  `Type lookup(name)`: returns the type of variable `name` (`INT`, `BOOL`) if cataloged, otherwise `NULL`;
- In case of semantic error, function `error(string message)` is called, which prints the relevant error `message` before terminating the analysis.

6. With reference to the BNF defined in point 5, assuming that the concrete syntax tree of a phrase is binary (pointers: `child`, `brother`), we ask to codify a procedure for P-code generation based on the following requirements:

- The symbol table defined in point 5 is available;
- Within the concrete syntax tree, each lexical value is stored as a string in field `lexval`;
- The language of the P-machine includes the following set of instructions:
  - `NEI` *id*: allocates integer variable *id* in data memory;
  - `NEB` *id*: allocates boolean variable *id* in data memory;
  - `LDA` *id*: load address of variable *id*;
  - `LOD` *id*: load value of variable *id*;
  - `LDC` *string*: load constant *string*;
  - `LAB` *label*: create *label*;
  - `GOF` label: conditional jump (to false);
  - `JMP` *label*: unconditional jump;
  - `PLUS`, `TIMES`: *addition*; *multiplication*,
  - `AND`, `OR`: `NOT`: *conjunction*, *disjunction*; *negation*,
  - `LTE`: *less than or equal* (≤),
  - `STO`: store;
- The auxiliary function `emit(string [, string ])` is used to print an instruction of the P-machine.