Partes del Examen 2 – 2023.2

2) El lenguaje LP1 describe un subconjunto de las fórmulas de la lógica proposicional. LP1 está compuesto de variables proposicionales, los conectivos lógicos binarios *and* y *or* (denotando conjunción y disyunción, respectivamente ), el conectivo lógico *not (*denotando negación), formulas entres pares de paréntesis y las constantes booleanas *true* y *false*.

a) (2 pts) Escribir una gramática NO Ambigua de LP1 que permita la implementación de un analizador sintáctico (Parser) usando el método de descenso recursivo; esto ultimo implica que la gramatica no puede tener recursion a la izquierda. Considerar que (1) *not* tiene mayor precedencia que *and* y *or*, (2) *or* y *and* tienen igual orden de precedencia y (3) *and* y *or* asocian a la izquierda. Por ejemplo:

- 
  - "*p and q or r and s*" es equivalente a "*((p and q) or r) and s*".
  - "*not p and q*" es equivalente a "*(not p) and q"*.

Para esto, usar los siguientes no-terminales:
LP1 ::= ... Unary ::= ... Factor ::= ...

b) (3 pts) Escribir un analizador sintáctico para LP1 en C++ que, además, genere el árbol sintáctico abstracto de LP1. Asumir que:

- Tenemos la clase *Token* con tipos de token ID, LPAREN, RPAREN, AND, OR y NOT, la función *match(Token::TokenType tt)* y el puntero *previous*.
- Tenemos definidas la superclase Formula, las subclases BinaryFormula, *UnaryFormula*, *ParenthFormula*, *IdFormula* y *BoolConstFormula*, asi como los constructores necesarios para construir el arbol.
- Existen los operadores AND, OR y NOT definidos como elementos del tipo enumerador *Formula::Op* y la funcion Parser::TokenToOp(Token) que retorna el operador correspondiente de tipo *Formula::Op*.

c) (1 pt) Deseamos ahora extender el lenguaje LP1 – llamémosle LP2 - para que incluya al conectivo lógico *implies* (=>) asumiendo que *implies* asocia a la derecha y que tiene **menor** order de precedencia que *and* y *or*. Esto quiere decir lo siguiente:

- 
  - "p implies q implies r" es equivalente a "p implies (q implies r)"
  - "p or q implies r and s" es equivalente a "(p or q) implies (r and s)".

**Escribir la gramática de LP2** como una extension de LP1 - re-usar los no-terminales usados en (a). Hint: Solo es necesario agregar el no-terminal LP2 pero, ¿donde?

d) (1 pt) Implementar *Formula parseLP2()*, es decir, el analizador sintactico para LP2 que ademas genera el arbol sintactico abstracto (AST) correspondiente.

3a) Dado el lenguaje IMP definido por la siguiente gramatica:

Program ::= Body
Body ::= VarDecList StatementList
VarDecList ::= (VarDec)*
VarDec ::= “var” Type VarList “;”
Type ::= "int" | "bool"
VarList :: id (“,” id)*
StatementList ::= Stm (“;” Stm)*
Stm ::= id "=" Exp | "print" "( Exp ")" | "if" Exp "then" Body ["else" Exp] "endif" | "while" Exp "do" Body "endwhile"
Exp ::= CExp CExp ::= AExp (('<'|'<='|'==') AExp)?
AExp ::= Term ((´+´ | ´-´) Term)* Term ::= Factor ((´*´|´/´) Factor)*
Factor ::= num | ´(´ Exp ´)´ | id

Queremos agregar lo siguiente:

- Las constantes booleanas *true* y *false*.
- Un nuevo no-terminal *BExp* para formar expresiones con los operadores booleanos *and* y *or*. Estos nuevos operadores deberan tener menor precedencia que el resto de operadores. Asi, "*x > 2 and 6 <= 10*" debera interpretarse como *(x > 2) and (6 <= 10)* - ¡lo obvio! Ademas, estos operadores deberan asociar a la derecha, por ejemplo, "*x and y or w and z*" debera ser interpretado como "*x and (y or (w and z))*".
- Una nueva sentencia para especificar for loops con un iterador entero. Por ejemplo, queremos escribir:
  for i : 0, x+2 in accum = accum * i endfor
  donde 0 y x+2 son expresiones, y la variable i toma valores de *0* a *x+2*.

Modificar la gramatica de arriba para incluir *true*, *false*, *BExp* y *for-endfor*. Solo incluir los cambios, no es necesario re-escribir toda la gramatica.

3b) Typechecking
Tomado en cuenta la gramatica y las consideraciones de la pregunta (3a), especificar las reglas de typecheking para *true y false*, *BExp* y *for-endfor*. Esto puede hacerse definiendo el metodo *visit* en ImpTypeChecker o definiendo las reglas para tcheck(). Para esta y la siguiente pregunta, asumir que existen las nuevas clases:

```
class BoolConstExp : public Exp { public: bool value; .... };  // para true y false
class ForStatement: public Exp { string var; Exp *e1, *e2; Body* body }
// BExp is un BinaryExp!!
```

y que el typechecking de BinaryExp es:

```
ImpType ImpTypeChecker::visit(BinaryExp* e) {
  ImpType t1 = e->left->accept(this);
  ImpType t2 = e->right->accept(this);
  if (!t1.match(inttype) || !t2.match(inttype)) {
    cout << "Tipos en BinExp deben de ser int" << endl;
```

```
    exit(0);
  }
  ImpType result;
  switch(e->op) {
  case PLUS:   case MINUS:
  case MULT:   case DIV:  case EXP:
    result = inttype;
    break;
  case LT: case LTEQ: case EQ:
    result = booltype;
    break;
  }
  return result;
}
```

Notar que la variable en el for-loop crea un nuevo scope / binding.

3c) Tomando en cuenta lo dicho en 3a) y 3b), implementar el codigo del interprete para *BoolConstExp* y *ForExp*, y modificar el codigo del interprete para *BinaryExp*. Es decir, implementar

```
int ImpInterpreter::visit(BoolConstExp* e)
int ImpInterpreter::visit(FoExp* e)
```

y modificar

```
int ImpInterpreter::visit(BinaryExp* e) {
  int v1 = e->left->accept(this);
  int v2 = e->right->accept(this);
  int result = 0;
  switch(e->op) {
  case PLUS: result = v1+v2; break; case MINUS: result = v1-v2; break;
  case MULT: result = v1 * v2; break; case DIV: result = v1 / v2; break;
  case EXP:
    result = 1;
    while (v2 > 0) { result *= v1; v2--; }
    break;
  case LT: result = (v1 < v2) ? 1 : 0; break; case LTEQ: result = (v1 <= v2) ? 1: 0; break;
  case EQ: result = (v1 == v2) ? 1 : 0; break;
  }
  return result;
}
```

No olvidar que la variable en ForExp genera un nuevo scope.