Reporte Compiladores

Integrantes:

- Luis Mendez
- Jean Sotomayor

1) Comentarios

A continuación, se presentarán los cambios realizados en la interfaz de *parser* a fin de soportar comentarios en el lenguaje IMP.

Constantes booleanas

```
enum Type { LPAREN=0,...,SLASH};
static const char* token_names[38];
```

```
const char* Token::token_names[38] = {
  "LPAREN" , "RPAREN", "PLUS", "MINUS",
"MULT","DIV","EXP","LT","LTEQ","EQ",
  "NUM", "ID", "PRINT", "SEMICOLON", "COMMA", "ASSIGN", "CONDEXP",
"IF", "THEN", "ELSE", "ENDIF", "WHILE", "DO",
  "ENDWHILE", "ERR", "END", "VAR" , "NOT", "TRUE", "FALSE", "AND", "OR"
  "FOR", "COLON" , "ENDFOR", "BREAK", "CONTINUE","SLASH"};
```

Modificacion del parser

```
case '/':
      c = nextChar();
      if(c =='/'){
        c = nextChar();
        while (c != '\n' && c != '\0') c = nextChar();
        rollBack();
        token = new Token(Token::SLASH,getLexema());

      }
```

Cuando el paser detecta un “/”, validar si es comentario y asignar el token respectivo.

parseStatementList

```
StatementList* Parser::parseStatementList() {
  StatementList* p = new StatementList();
  p->add(parseStatement());
  while(match(Token::SEMICOLON)) {
    while(match(Token::SLASH));
    p->add(parseStatement());
  }
  while(match(Token::SLASH));
  return p;
}
```

En caso de encontrar un ‘comentario’, no hacer nada hasta el cambio de línea.

parseStatement

```
Stm* Parser::parseStatement() {
  Stm* s = NULL;
  Exp* e;
  Body *tb, *fb;
…
 else if (match(Token::SLASH)){
    advance();
  }
…
  return s;
}
```

parseVarDec

```
VarDec* Parser::parseVarDec() {
  VarDec* vd = NULL;
  if (match(Token::VAR)) {
  …
    if (match(Token::SLASH));//NO HACER NADA
    vd = new  VarDec(type,vars);
  }
  return vd;
}
```

Simplemente cuando leo el slash, avanzó hasta leer un salto de línea, donde el comentario termina.

2) Generación de código I

A continuación, se presentan los cambios realizados en la interfaz $ImpCodeGen$ e $ImpTypeChecker$ a fin de generar código para:

Constantes booleanas

$codegen(addr,\ BoolExp(b)) \ = \ push\ b\ ,\ b\ in\ \{0, 1\}$

```
int ImpCodeGen::visit(BoolConstExp* e) {
  codegen(nolabel,"push",e->b ? 1 : 0);
  return 0;
}
```

$tcheck(env,\ BoolExp(b)) \ = \ bool$

```
ImpType ImpTypeChecker::visit(BoolConstExp* e) {
  return booltype;
}
```

Operadores AND y OR

$codegen(addr,\ BinExp(e1, e2, op)) \ =$
$codegen(addr, e1)$
$codegen(addr, e2)$
$op,\ op\ in\ \{..., \ and,\ or\}$

```
int ImpCodeGen::visit(BinaryExp* e) {
  e->left->accept(this);
  e->right->accept(this);
  string op = "";
  switch(e->op) {
  case PLUS: op =  "add"; break;
  case MINUS: op = "sub"; break;
  case MULT:  op = "mul"; break;
  case DIV:  op = "div"; break;
  case LT:  op = "lt"; break;
  case LTEQ: op = "le"; break;
  case EQ:  op = "eq"; break;
  case AND: op = "and"; break;
  case OR: op = "or"; break;
  codegen(nolabel, op);
  return 0;
}
```

$tcheck(BinExp(e1, e2, op)) = int$
if
$tcheck(env, e1) = int \&\&$
$tcheck(env, e2) = int$
$op\ in\ \{plus, minus, mul, div\}$

$tcheck(BinExp(e1, e2, op)) = bool$
if
$tcheck(env, e1) = int \&\&$
$tcheck(env, e2) = int$
$op\ in\ \{lt, leq, eq\}$

$tcheck(BinExp(e1, e2, op)) = bool$
if
$tcheck(env, e1) = bool \&\&$
$tcheck(env, e2) = bool$
$op\ in\ \{and, or\}$

```
ImpType ImpTypeChecker::visit(BinaryExp* e) {
  ImpType t1 = e->left->accept(this);
  ImpType t2 = e->right->accept(this);
  ImpType result, argtype;
  argtype = inttype;
  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;
  case AND: case OR:
    result = booltype; argtype = booltype;
    break;
  }
  if (!t1.match(argtype) || !t2.match(argtype)) {
    cout << "Tipos en BinExp deben de ser: " << argtype << endl;
    exit(0);
  }

  return result;
}
```

For statement

$codegen(addr,\ ForStm(i,\ e1,\ e2,\ bd))\ =$
$codegen(addr,\ e1)$
$store\ addr(i)$
$LENTRY:\ skip$
$load\ addr(id)$
$codegen(addr,\ e2)$
le
$jmpz\ LEND$
$codegen(addr,\ bd)$
$load\ addr(i)$
$push\ 1$
add
$store\ addr(i)$
$goto\ LENTRY$
$LEND:\ skip$

```
int ImpCodeGen::visit(ForStatement* s) {
  string l1 = next_label();
  string l2 = next_label();

  direcciones.add_var(s->id, siguiente_direccion++);

  s->e1->accept(this);
  codegen(nolabel,"store",direcciones.lookup(s->id));

  codegen(l1,"skip");
  codegen(nolabel,"load",direcciones.lookup(s->id));
  s->e2->accept(this);
  codegen(nolabel, "le");
  codegen(nolabel,"jmpz",l2);
  s->body->accept(this);
  codegen(nolabel,"load",direcciones.lookup(s->id));
  codegen(nolabel,"push", 1);
  codegen(nolabel,"add");
  codegen(nolabel,"store",direcciones.lookup(s->id));

  codegen(nolabel,"goto",l1);
  codegen(l2,"skip");

  return 0;
}
```

$tcheck(env,\ ForStm(i,\ e1,\ e2,\ bd))$
$if\ i$
$tcheck(env,\ e1)\ =\ int\ \&\&$
$tcheck(env,\ e2)\ =\ int\ \&\&$
$tcheck(env, bd)$

```
void ImpTypeChecker::visit(ForStatement* s) {
  ImpType t1 = s->e1->accept(this);
  ImpType t2 = s->e2->accept(this);
  if (!t1.match(inttype) || !t2.match(inttype)) {
    cout << "Tipos de rangos en for deben de ser: " << inttype << endl;
    exit(0);
  }
  env.add_level();
  env.add_var(s->id,inttype);
  s->body->accept(this);
  env.remove_level();
 return;
}
```

3) Sentencia do-while

A continuación, se presentarán los cambios realizados en la interfaz a fin de soportar comentarios en el lenguaje IMP.

Clases añadidas

```
class DoWhileStatement : public Stm {

public:

    Exp* condition;

    Body* body;

    DoWhileStatement(Body* b, Exp* c);

    int accept(ImpVisitor* v);

    void accept(TypeVisitor* v);

    ~DoWhileStatement();

};
```

```
DoWhileStatement::DoWhileStatement(Body* b, Exp* c) : body(b),
condition(c) {}

int DoWhileStatement::accept(ImpVisitor* v) {
    return v->visit(this);
}
void DoWhileStatement::accept(TypeVisitor* v) {
    return v->visit(this);
}
DoWhileStatement::~DoWhileStatement() {
    delete body;
    delete condition;
}
```

Parser

```
Stm* Parser::parseStatement() {
  Stm* s = NULL;
  Exp* e;
  Body *tb, *fb;
….
 else if (match(Token::DO)) {
        tb = parseBody();
        if (!match(Token::WHILE)) {
            parserError("Esperaba 'while' después de 'do'");
        }
        e = parseExp();
        s = new DoWhileStatement(tb, e);
…
  return s;
}
```

Generador de código para el do while

ImpCodeGen

ImpCodeGenint visit(DoWhileStatement*);

```
int ImpCodeGen::visit(DoWhileStatement* s) {
  string l1 = next_label();
  string l2 = next_label();
  codegen(l1, "skip");
  s->body->accept(this);
  s->condition->accept(this);
  codegen(nolabel, "jmpz", l2);
  codegen(nolabel, "goto", l1);
  codegen(l2, "skip");
  return 0;
}
```

Lógica del do-while

ImpInterpreter

```
int visit(DoWhileStatement*);
```

```
int ImpInterpreter::visit(DoWhileStatement* s) {
    do {
        bool condition_result = s->condition->accept(this);
        if (condition_result) {
            s->body->accept(this);
            if (breaks) { breaks = false; break; }
            if (continues) { continues = false; }
        } else {break;}
    } while (true);
    return 0;
}
```

Validaciones adicionales

ImpTypeChecker

```
void visit(DoWhileStatement*);
```

```
void ImpTypeChecker::visit(DoWhileStatement* s) {
  if (!s->condition->accept(this).match(booltype)) {
    cout << "Condicional en DoWhileStm debe ser de tipo: " << booltype
<< endl;
    exit(0);
  }
  s->body->accept(this);
  return;
}
```

4) Sentencias break y continue

Para el procesamiento de estas nuevas sentencias se agregaron nuevas clases AST:

```
class BreakStatement : public Stm {
  public:
  BreakStatement();
  int accept(ImpVisitor* v);
  void accept(TypeVisitor* v);
  ~BreakStatement();
};

class ContinueStatement : public Stm {
  public:
  ContinueStatement();
  int accept(ImpVisitor* v);
  void accept(TypeVisitor* v);
  ~ContinueStatement();
};
```

En el scanner se añadieron nuevos tokens y en el parser se añadió un nuevo caso en $parseStatement$

```
Stm* Parser::parseStatement() {
  Stm* s = NULL;
  Exp* e;
  Body *tb, *fb;
  if (match(Token::ID)) {
      ...
    } else if (match(Token::BREAK)) {
    s = new BreakStatement();
  } else if (match(Token::CONTINUE)) {
    s = new ContinueStatement();
  }
  else {
    ...
  }
  return s;
}
```

Asimismo, se añadió a todos visitors los métodos visit para estas nuevas clases AST.

Los cambios más importantes se encuentran en el intérprete, como sigue:

```
class ImpInterpreter : public ImpVisitor {
private:
  Environment<int> env;
  bool breaks;
  bool continues;

public:
  int visit(BreakStatement*);
  int visit(ContinueStatement*);
      ...
};
```

```
int ImpInterpreter::visit(Program* p) {
  breaks = false;
  continues = false;
  p->body->accept(this);
  return 0;
}
```

```
int ImpInterpreter::visit(BreakStatement* s) {
  breaks = true;
  return 0;
}

int ImpInterpreter::visit(ContinueStatement*) {
  continues = true;
  return 0;
}
```

```
int ImpInterpreter::visit(StatementList* s) {
  list<Stm*>::iterator it;
  for (it = s->slist.begin(); it != s->slist.end(); ++it) {
    (*it)->accept(this);
    if (breaks or continues) break;  // salir
  }
  return 0;
}
```

Cuando el intérprete visita un $BreakStatement$ o $ContinueStatement$ setea a true la variable breaks o continues respectivamente.
Estas variables se encargan de cortar el flujo de visita a los statements en un $StatementList$ que puede estar en el body de una sentencia While o For.

```
int ImpInterpreter::visit(WhileStatement* s) {
  while (s->cond->accept(this)) {
    s->body->accept(this);
    if (breaks) {  breaks = false; break; }
    if (continues) {  continues = false; continue; }
  }
  return 0;
}

int ImpInterpreter::visit(ForStatement* s) {
  int n1 = s->e1->accept(this);
  int n2 = s->e2->accept(this);
  env.add_level();
  env.add_var(s->id);
  for (int i = n1; i <= n2; i++) {
    env.update(s->id,i);
    s->body->accept(this);
    if (breaks) {  breaks = false; break; }
    if (continues) {  continues = false; continue; }
  }

  env.remove_level();
 return 0;
}
```