# Proyecto 1 - Compiladores

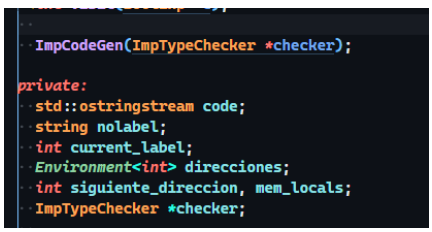
Adrian Sandoval Huamaní - Andres Jaffet Riveros Soto

17 de Junio del 2024

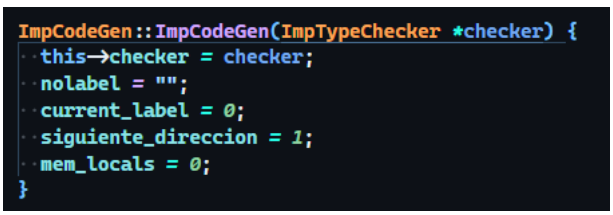
## 1. Validación de tipos

### 1.1. Implementación

Se agrega el constructor dentro del Codegen como párametro al Typechecker. Este Typechecker es guardado como atributo privado dentro del CodeGen. Y el Typechecker nos brinda información sobre el número de variables o espacio de memoria máxima a reservar.




Cambios en `codegen.hh`




Constructor del *Checker*

### 1.2. Explicación

El Typechecker al llamar a su función `typecheck(p)` de *checker* brinda el número máximo de variables que serán usados durante todo el programa. Para obtener este número de variables, se utilizan dos variables **sp** y **max_sp**, donde si **sp** es más grande que **max_sp** actualizamos **max_sp**, esta acción se puede observar en `visit(Body* b)` del typechecker.

```
void ImpCodeGen::codegen(Program* p, string outfname) {
  mem_locals = checker->typecheck(p);
  codegen(nolabel,"alloc",this->mem_locals);
  p->accept(this);
  ofstream outfile;
  outfile.open(outfname);
  outfile << code.str();
  outfile.close();
  cout << "Memoria variables locales: " << mem_locals << endl;

  return;
}
```

Cambios en `codegen.cpp`

# 2. Comentarios

## 2.1. Scanner modificado

En `scanner.cpp` se agregó en el case `/` un caso que al consumir otro `/`, se tiene un bucle que consume cualquier caracter hasta que haya salto de línea o EOF y al final un token `COMMENT` con el contenido del comentario escaneado.

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

Scanner Modificado

## 2.2. Cambios en la gramática

```
. . . . . .
VarDec ::= 'var' Type VarList ';' ['//' comment '\n']
. . . . . .
StatementList ::= Stm (';' ['//' comment '\n'] Stm)*
. . . . . .
```

## 2.3. Parser modificado

En `parseStatementList()`, dentro del bucle while se agrega una condicional para agregar el contenido del comentario sin los caracterres iniciales `//` si consume un `COMMENT` despues de `SEMICOLON`. Caso contrario, continua parseando los Statement sin ello (por defecto el comentario es cadena vacía en este AST).

```
StatementList* Parser::parseStatementList() {
  StatementList* p = new StatementList();
  p->add(parseStatement());
  while(match(Token::SEMICOLON)) {
    if (match(Token::COMMENT)) {
      p->add(parseStatement(),previous->lexema.substr(2));
    } else
      p->add(parseStatement());
  }
  return p;
}
```

`parser.cpp` Modificado

En `parseVarDec()` se agrega una condicional al final de leer toda la decalaración al encontrar un `COMMENT` para extraer el comentario sin los caracterres iniciales `//` después del `SEMICOLON` en caso lo consuma el Scanner.

```
    if (!match(Token::SEMICOLON)) parserError("Expecting semicolon
    if (match(Token::COMMENT)) {
      vd = new VarDec(type,vars,previous->lexema.substr(2));
    } else
      vd = new VarDec(type,vars);
  }
  return vd;
```

`parser.cpp` Modificado

## 2.4. Visitor ImpPrinter

En el visitor del `ImpPrinter` a un `StatementList` se agrega una condicional que imprime el punto y coma junto con el comentario en la misma iteración de la instrucción cuando se extrae el *front* de `s->comments` siendo no vacío. Caso contrario, se imprime solo un punto y coma.

```cpp
int ImpPrinter::visit(StatementList* s) {
  cout << "{" << endl;
  list<Stm*>::iterator it;
  for (it = s->slist.begin(); it != s->slist.end(); ++it) {
    (*it)->accept(this);
    s->comments.pop_front();
    if (s->comments.front() != "") {
      cout << "; //" << s->comments.front();
    } else {
      cout << ";" << endl;
    }
  }
  cout << "}" << endl;
  return 0;
}
```

`imp_printer.cpp` modificado

También, el visitor del `ImpPrinter` a un `VarDec` se agrega una condicional que imprime el punto y coma junto con el comentario al final cuando `vd->comment` sea no vacío. Caso contrario, se imprime solo un punto y coma

```cpp
int ImpPrinter::visit(VarDec* vd) {
  bool first = true;
  cout << "var " << vd->type << " ";
  list<string>::iterator it;
  for (it = vd->vars.begin(); it != vd->vars.end(); ++it){
    if (!first) cout << ", ";
    first = false;
    cout << *it;
  }
  if (vd->comment != "")
    cout << "; //" << vd->comment;
  else
    cout << ";" << endl;
  return 0;
}
```

`imp_printer.cpp` modificado

## 2.5. Ejemplo

Para este código `ejemplo11.imp`:

```
var int x, y; // comentario
x = 12; y = x + 3; // otro comentario
print(ifexp(x >y,100,200)+2);
// comentario mas
print(y + 1)
```

El resultado en el printer y TypeChecker:

```
Program :
var int x, y; // comentario
{
x = 12;
y = x + 3; // otro comentario
print(ifexp(x > y,100,200) + 2); // comentario mas
print(y + 1);
}

Run program:
202
16

Compiling to: .\ejemplo11.imp.sm
Max stack size: 2
Memoria variables locales: 2
```

Resultado `ejemplo11.imp`

# 3. Do-While

## 3.1. Cambios en la gramática

```
Stm ::= id '=' Exp | 'print' '(' Exp ')' |
        'if' Exp 'then' Body ['else' Body] 'endif' |
        'while' Exp 'do' Body 'endwhile' |
        'do' Body 'while' Exp
```

## 3.2. Implementación

Creamos un AST DoWhileStatement junto a sus atributos. Además, agregar los visitors de ImpCodeGen, ImpInterpreter, ImpPrinter y ImpTypeChecker para este AST y modificar el método del Parser `parseStatement()` agregando una condicional donde acepte el `Token::DO` para que consuma toda la instrucción DoWhile.

```
class DoWhileStatement : public Stm {
public:
  Exp* cond;
  Body *body;
  DoWhileStatement(Exp* c, Body* b);
  void accept(ImpVisitor* v);
  void accept(TypeVisitor* v);
  ~DoWhileStatement();
};
```

AST del *DoWhile*

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

  codegen(l1,"skip");
  s->body->accept(this);
  s->cond->accept(this);
  codegen(nolabel,"jmpz",l2);
  codegen(nolabel,"goto",l1);
  codegen(l2,"skip");

  return 0;
}
```

Codegen del *DoWhile*

```
int ImpInterpreter::visit(DoWhileStatement* s) {
  do {
    s->body->accept(this);
  } while (s->cond->accept(this));
  return 0;
}
```

Interpreter del *DoWhile*

```
else if (match(Token::DO)) {
  tb = parseBody();
  if (!match(Token::WHILE)) parserError("Esperaba 'while'");
  e = parseBExp();
  s = new DoWhileStatement(e,tb);
}
```

Nuevo parseo *DoWhile* como nuevo `Stm`

```
int ImpPrinter::visit(DoWhileStatement* s) {
  cout << "do" << endl;
  s->body->accept(this);
  cout << "while (";
  s->cond->accept(this);
  cout << ")";
  return 0;
}
```

Printer *DoWhile*

```
void ImpTypeChecker::visit(DoWhileStatement* s) {
    s->body->accept(this);
    s->cond->accept(this);
    return;
}
```

TypeChecker *Do While*

### 3.3. Ejemplo

Para este código `ejemplo22.imp`:

```
var int x, y; x = 30; y = 1; do x = x / 2; print(x) while x >  y; print(x)
```

El resultado en el printer y TypeChecker:

```
adria@ASUS ~\..\..\ProjCompilers  main > ./compile.exe ".\ejemplo22.imp"
Program :
var int x, y;
{
x = 30;
y = 1;
do
{
x = x / 2;
print(x);
}
while (x > y);
print(x);
}

Run program:
15
7
3
1
1

Compiling to: .\ejemplo22.imp.sm
Max stack size: 2
Memoria variables locales: 2
```

Resultado `ejemplo22.imp`

## 4. Anexos

- Repositorio del proyecto https://github.com/Sandovl0593/ProjCompilers