

Proyecto 1 - (Compiladores CS3025)

Resumen

En este proyecto se implementaron mejoras al compilador del lenguaje IMP-DEC que se desarrolló en el curso de Compiladores CS3025. Las mejoras implementadas fueron las siguientes:

1. **Typechecker y Codegen:** Se implementó un `typechecker` y un `codegen` para la gramática IMP-DEC.
2. **Generación de Código I:** Se modificó el `Scanner` y `Parser` para que soporte la inclusión de comentarios en el código fuente.
3. **Sentencia Do While:** Se implementó la sentencia `do while` en el lenguaje IMP-DEC.

Tabla de Contenidos

- Proyecto 1 - (Compiladores CS3025)
 - Resumen
 - Tabla de Contenidos
 - Gramática del lenguaje IMP-DEC
 - 1. Typechecker y Codegen
 - 1.1. Typechecker
 - Implementación del Typechecker
 - 1.2. Codegen
 - Modificaciones en el Typechecker
 - Implementación del Codegen
 - Ejemplo de uso
 - Consideraciones finales
 - 2. Generación de Código I
 - Cambios en el Scanner
 - Cambios en el Parser
 - Comentarios al final de una declaración de variable o statement
 - Comentarios aislados en cualquier parte del código
 - Cambios en los visitors
 - ImpVisitor
 - ImpValueVisitor e ImpTypeVisitor
 - Ejemplo de funcionamiento
 - Consideraciones finales
 - 3. Sentencia Do While
 - Ejemplo de funcionamiento
 - Consideraciones finales
 - Autores
 - Referencias

Gramática del lenguaje IMP-DEC

La gramática que se utilizó en el proyecto es la siguiente:

```

Program      ::= Body
Body         ::= VarDecList StatementList
VarDecList   ::= (VarDec | Comment)*
VarDec       ::= "var" Type VarList ";" (Comment)?
Type         ::= id
VarList      ::= id ("," id)*
StatementList ::= (Stm ";" (Comment)? | Comment)*
Stm          ::= id "=" Exp |
                "print" "(" Exp ")" |
                "if" Exp "then" Body "else" Body "endif" |
                "while" Exp "do" Body "" "endwhile" |
                "do" Body "enddo" "while" Exp
Exp          ::= BExp
BExp         ::= CEXP (("and" | "or") BExp)?
CEXP         ::= AExp ("==" | "<" | "<=" ) AExp)?
AExp         ::= Term ("+" | "-") Term)*
Term         ::= FExp ("*" | "/" ) FExp)*
FExp         ::= Unary ("*" FExp)?
Unary        ::= "-" Factor | "!" Factor | Factor
Factor       ::= id |
                num |
                true |
                false |
                "(" Exp ")" |
                "ifexp" "(" Exp "," Exp "," Exp ")"
Comment      ::= "//" (~("\n"))*

```

1. Typechecker y Codegen

En esta sección se describe la implementación del Typechecker y Codegen para el lenguaje IMP-DEC.

1.1. Typechecker

El Typechecker es un visitor que recorre el AST parseado y verifica que las variables utilizadas en el programa estén declaradas y que los tipos de las variables sean correctos. En caso de que haya un error, el Typechecker lanza una excepción con un mensaje de error.

Los archivos modificados para implementar el Typechecker son:

- `type_visitor.hh`: Interfaz **TypeVisitor** que funcionará como visitor para el Typechecker.
- `imp_typechecker.hh`: Clase **ImpTypeChecker** que implementa la interfaz **TypeVisitor**. Este a su vez tiene un Environment de tipo **ImpType** que almacenará las variables (key) y sus tipos (value).
- `imp_typechecker.cpp`: Definición de la clase **ImpTypeChecker**. Aquí se implementan los métodos de visitor para cada regla de la gramática.

Implementación del Typechecker

Primero, se modificó el enum **ImpType**, que antes era un atributo de la clase **ImpValue**, creando una clase `ImpType` que contendrá los tipos de las variables.

```

class ImpType {
public:
    enum TType { NOTYPE=0, VOID, INT, BOOL, FUN };
    static const char* type_names[5];
    TType ttype;
    vector<TType> types;
    bool match(const ImpType&);
    bool set_basic_type(string s);
    bool set_basic_type(TType tt);
    bool set_fun_type(list<string> slist, string s);
private:
    TType string_to_type(string s);
};

```

Luego, se creó la interfaz **TypeVisitor** que tendrá los métodos visit para cada regla de la gramática. También se creó la clase **ImpTypeChecker** que implementa la interfaz **TypeVisitor**. Finalmente, se implementaron los métodos de visitor para cada regla de la gramática.

Los cambios se pueden ver en los archivos mencionados anteriormente.

1.2. Codegen

El Codegen es un visitor que recorre el AST y genera el código máquina en un nuevo archivo.

Los archivos modificados para implementar el Codegen son:

- `imp_codegen.hh`: Clase **ImpCodeGen** que implementa la interfaz **ImpVisitor**. Este a su vez tiene un Environment de tipo **int** que almacenará las variables en direcciones (key) y sus valores (value).
- `imp_codegen.cpp`: Definición de la clase **ImpCodeGen**. Aquí se implementan los métodos de visitor para cada regla de la gramática. Además, se almacenará cada instrucción en una variable de tipo **ostream** para luego escribirlos en un archivo.

Modificaciones en el Typechecker

Para implementar el Codegen, se modificó el Typechecker con el fin de calcular la memoria necesaria para las variables globales y locales previamente. Como el typechecker recorre todo el AST, durante ese proceso también calcula la memoria necesaria. Esto se hizo con el fin de mapear la memoria necesaria antes de la ejecución del codegen y así poder realizar un alloc de la memoria necesaria para las variables globales.

Para ello, se agregaron nuevas variables como contadores en **ImpTypeChecker**, así como dos métodos `sp_incr` y `sp_decr` para incrementar y decrementar el stack pointer.

```

class ImpTypeChecker : public TypeVisitor {
public:
    ImpTypeChecker();
    int sp, max_sp, next_direc, mem_locals;
private:
    Environment<ImpType> env;
    ImpType booltype;
    ImpType inttype;
    void sp_incr(int n);
    void sp_decr(int n);
public:
    // visitors ...
}

```

Se realizó esta modificación con el fin de que el Codegen pueda contener al Typechecker como atributo y así poder acceder a los contadores de memoria.

Implementación del Codegen

Primero se creó la clase **ImpCodeGen** que implementa la interfaz **ImpVisitor**. Esta clase tendrá un atributo de tipo **ImpTypeChecker** para poder acceder a los contadores de memoria. Además, se creó un atributo de tipo **ostream** para almacenar las instrucciones y un atributo de tipo **ofstream** para escribir las instrucciones en un archivo. Estos cambios se pueden visualizar en el archivo `imp_codegen.hh`.

```

class ImpCodeGen : public ImpVisitor {
private:
    std::ostringstream code; // Código máquina generado
    string nolabel;
    int current_label;
    Environment<int> direcciones;
    ImpTypeChecker typechecker; // Typechecker
    int siguiente_direccion, mem_locals;
    void codegen(string label, string instr);
    void codegen(string label, string instr, int arg);
    void codegen(string label, string instr, string jmplabel);
    string next_label();

public:
    // visitors ...

}

```

Luego, se implementaron los métodos de visitor para cada regla de la gramática en el archivo `imp_codegen.cpp`. En estos métodos se generan las instrucciones en código máquina y se almacenan en el atributo **code** de tipo **ostringstream** con los métodos `codegen` que se crearon en la clase **ImpCodeGen**. Además, se ejecuta el **Typechecker** al comienzo del programa para calcular la memoria necesaria para las variables globales y locales.

```

void ImpCodeGen::codegen(Program *p, string outfname) {
    ...
    // Typechecker execution
    typechecker.typecheck(p); // Se ejecuta ni bien se ejecuta el codegen

    p->accept(this); // Generación de código

    ofstream outfile;
    outfile.open(outfname);
    outfile << code.str(); // Escribir las instrucciones en un archivo .sm
    outfile.close();

    cout << "Max memory for local variables: " << mem_locals << endl;
    cout << "Max stack size: " << typechecker.max_sp << endl; // Máximo tamaño de la pila
    return;
}

void ImpCodeGen::visit(Program *p) {
    codegen(nolabel, "alloc", typechecker.mem_locals); // Alloc de la memoria necesaria para las variables locales
    p->body->accept(this);
    codegen(nolabel, "halt"); // Fin del programa
    return;
}

```

Los demás cambios se visualizan en el archivo `imp_codegen.cpp`.

Ejemplo de uso

Para utilizar el Typechecker y Codegen se debe ejecutar el siguiente comando:

```

make compiler
./compile.exe "ejemplo11.imp"

```

Input:

```
var int i, j;
i = 1;
do
    j = i * 2;
    print(j);
    i = i + 1;
enddo while i < 5;
```

Output en consola:

```
Program :
var int i, j;
i = 1;
do {
j = i * 2;
print(j);
i = i + 1;
} enddo while (i < 5);
```

Run program:

```
2
4
6
8
```

```
Compiling to: ejemplo11.imp.sm
Max memory for local variables: 2
Max stack size: 2
```

Archivo "ejemplo11.imp.sm":

```
alloc 2
push 1
store 1
L0: skip
load 1
push 2
mul
store 2
load 2
print
load 1
push 1
add
store 1
load 1
push 5
lt
jmpz L1
goto L0
L1: skip
halt
```

Ejecutando el svm:

```
make svm
./svm.exe "ejemplo22.imp.sm"
```

Output:

```
Reading program from file ejemplo11.imp.sm
Program:
alloc 2
push 1
store 1
L0: skip
load 1
push 2
mult
store 2
load 2
print
load 1
push 1
add
store 1
load 1
push 5
lt
jmpz L1
goto L0
L1: skip
halt
-----
Running ....
2
4
6
8
Finished
stack [ 0 5 8 ]
```

Consideraciones finales

Como se está trabajando solamente con dos tipos de variable (bool, int), el Environment del Codegen es de tipo **int** para facilitar la generación de código. Además, por simplicidad la dirección 0 no se utiliza, por lo que se comienza a contar desde la dirección 1. Del mismo modo, algunas operaciones que no soporta el **svm** no se implementaron en el codegen, como la exponenciación ****** o negación **!**. Finalmente, la generación de código para las funciones aún no está implementado.

⚠ Reporte: ¿Cómo se calculó la memoria necesaria para las variables globales? \ Para calcular la memoria necesaria tanto para las variables globales como para las variables locales se utilizó el Typechecker como atributo de la clase del Codegen. En el Typechecker se recorre todo el AST, por lo que se creó una variable que almacene la memoria necesaria para las variables globales y locales. Luego, este valor es pasado al Codegen para que pueda realizar un alloc de la memoria necesaria para las variables globales.

2. Generación de Código I

En esta sección se describe la implementación de la inclusión de comentarios en el código fuente.

Como se ve en la [gramática](#), se añadió la regla `Comment ::= "//" (~("\n"))` que permite la inclusión de comentarios en el código fuente. Esta regla permite que se incluyan comentarios de una sola línea al final de una **declaración de variable (VarDec)**, o **statement (stm)**, así como comentarios aislados en **cualquier parte del código**.

Para implementar esta funcionalidad se modificó ligeramente la gramática:

- Se modificó el `VarDecList` para que pueda aceptar un `VarDec` o un `Comment`

```
// Antes          Después
VarDecList ::= (VarDec)* ➔ VarDecList ::= (VarDec | Comment)*
```

- `VarDec` tienen al final un `Comment` opcional.

```
// Antes          Después
VarDec ::= "var" Type VarList ";" ➔ VarDec ::= "var" Type VarList ";" (Comment)?
```

- Se modificó el `StatementList` para que cada `Stm` esté seguido de un `;` y luego de un `Comment` opcional o sea un `Comment` solo.

```
// Antes          Después
StatementList ::= Stm (";" Stm)* ➔ StatementList ::= (Stm ";" (Comment)? | Comment)*
```

- Se añadió la regla `Comment ::= "//" (~("\n"))` * que permite la inclusión de comentarios en el código fuente.

Cambios en el Scanner

Para implementar la inclusión de comentarios en el código fuente, se modificó el Scanner para que asigne un token a los comentarios. Para ello, cuando el Scanner detecta un `"//"` en el input procede a saltarse todos los caracteres hasta encontrar un salto de línea `"\n"`.

Primero, se agregó el token `COMMENT` en los archivos `imp_parser.hh` y `imp_parser.cpp`.

```
// imp_parser.hh
class Token {
    enum Type {
        ...
        COMMENT,
        ...
    };
}

// imp_parser.cpp
const char *Token::token_names[32] = {
    ...
    "COMMENT",
    ...
};
```

Luego, se modificó el método `next_token()` en el archivo `imp_scanner.cpp` para que detecte los comentarios.

```

Token *Scanner::nextToken() {
    char c;
    ...
    else if (strchr("()+-*/;=<,>", c)) {
        switch(c) {
            ...
            case '/':
                c = nextChar();
                if (c == '/') {
                    while (c != '\n') c = nextChar(); // Hasta encontrar un salto de línea
                    rollBack();
                    token = new Token(Token::COMMENT, getLexema()); // COMMENT
                } else {
                    rollBack();
                    token = new Token(Token::DIV);
                }
                break;
            ...
        }
    }
}

```

Cambios en el Parser

Para que el Parser soporte la inclusión de comentarios de una sola línea en el código fuente, se consideraron dos casos:

1. Comentarios al final de una declaración de variable (`VarDec`) o statement (`Stm`).
2. Comentarios aislados en cualquier parte del código.

Comentarios al final de una declaración de variable o statement

Después de cada `VarDec` o `Stm` se puede incluir un comentario, como en la [gramática](#). Para ello, se creó una nueva clase `Comment` la cual almacenará el comentario al final de un `VarDec` o `Stm`.

```

/* imp.hh */
// Clase Comment para los comentarios en línea
class Comment {
public:
    string comment;
    Comment();
    Comment(string comment);
    void accept(ImpVisitor* v); // Solo se utiliza para el printer
    ~Comment();
};

```

Además, se agregó como atributo de las clases derivadas de `Stm` y de `VarDec` un puntero a un objeto de tipo `Comment` para almacenar comentarios en línea.


```

/* imp.hh */
// Modificación del AssignStatement
class AssignStatement : public Stm {
public:
    ...
    Comment* cmt;
    AssignStatement(Var* var, Exp* exp, Comment* cmt);
    ...
};

// La misma modificación se realizó en cada Stm y VarDeclaration

/* imp.cpp */
// Se modificaron los constructores

```

Luego, se agregó un nuevo método para parsear el comentario

```

/* imp_parser.hh */
class Parser {
    ...
    Comment* parseComment();
    ...
};

/* imp_parser.cpp */
Comment *Parser::parseComment() {
    Comment *c = NULL;
    if (match(Token::COMMENT)) {
        c = new Comment(previous->lexema);
    }
    return c;
}

```

Y finalmente, se modificó el parseo de las reglas de `Stm` y `VarDec` para que puedan incluir comentarios de acuerdo a la gramática.

```

/* imp_parser.cpp */

// Modificación al parser de VarDec
VarDec *Parser::parseVarDec() {
    VarDec *vd = NULL;

    if (match(Token::VAR)) {
        Comment* comment = NULL;
        ...
        comment = parseComment();

        vd = new VarDec(type, vars, comment);
    }
    return vd;
}

// Modificación al parser de StatementList
StatementList *Parser::parseStatementList() {
    StatementList *p = new StatementList();
    Stm* stm;
    stm = parseStatement();

    while (stm != NULL) {
        p->add(stm);
        stm = parseStatement();
    }
    return p;
}

// Modificación al parser de Stm
Stm *Parser::parseStatement() {
    Stm *s = NULL;
    Exp *e;
    Body *tb, *fb;
    Comment *comment;

    // AssignStatement Parser
    if (match(Token::ID)) {
        ...
        comment = parseComment();
        s = new AssignStatement(lex, e, comment);

        // PrintStatement Parser
    } else if (match(Token::PRINT)) {
        ...
        comment = parseComment();
        s = new PrintStatement(e, comment);
    }

    // Se realizó lo mismo para los demás Stm

    return s;
}

```

Comentarios aislados en cualquier parte del código

Los comentarios aislados pueden ir en cualquier parte del código. Para ello, se crearon dos nuevas clases que solo contendrán un comentario:

- `CommentStatement` que hereda de `Stm`
- `CommentVarDec` que hereda de `VarDec`

Nota: Se definió a `VarDec` como una interfaz que hereda a `VarDeclaration` (que contiene la declaración de variables) y `CommentVarDec` (que contiene el comentario). Con el fin de que `VarDecList` pueda aceptar ambos.

```
/* imp.hh */
// Comentarios aislados en los statements
class CommentStatement : public Stm {
public:
    string comment;
    CommentStatement(string comment);
    void accept(ImpVisitor* v);
    void accept(ImpValueVisitor* v);
    void accept(TypeVisitor* v);
    ~CommentStatement();
};

// Interfaz VarDec
class VarDec {
public:
    bool isComment;
    VarDec(bool ic);
    virtual void accept(ImpVisitor* v) = 0;
    virtual void accept(ImpValueVisitor* v) = 0;
    virtual void accept(TypeVisitor* v) = 0;
    virtual ~VarDec() = 0;
};

// Comentarios aislados en la declaración de variables
class CommentVarDec : public VarDec {
public:
    string comment;
    CommentVarDec(string comment);
    void accept(ImpVisitor* v);
    void accept(ImpValueVisitor* v);
    void accept(TypeVisitor* v);
    ~CommentVarDec();
};
```

Luego, se modificó el parser para que pueda parsear los comentarios aislados en cualquier parte del código.

```

/* imp_parser.cpp */
// Modificación al parser de VarDec
VarDec *Parser::parseVarDec() {
    VarDec *vd = NULL;

    if (match(Token::COMMENT)){
        vd = new CommentVarDec(previous->lexema);
    } else if (match(Token::VAR)) {
        ...
        vd = new VarDeclaration(type, vars, comment);
    }
    return vd;
}

// Modificación al parser de Stm
Stm *Parser::parseStatement() {
    Stm *s = NULL;
    ...
    else if (match(Token::COMMENT)){
        s = new CommentStatement(previous->lexema);
    }
}

```

Cambios en los visitors

ImpVisitor

Se modificó el ImpVisitor y ImpPrinter para que imprima los comentarios en el código fuente.

```

/* imp_visitor.hh */
class ImpVisitor {
public:
    ...
    virtual void visit(VarDeclaration* e) = 0;
    virtual void visit(CommentVarDec* e) = 0;
    virtual void visit(CommentStatement* e) = 0;
    virtual void visit(Comment* e) = 0;
    ...
}

/* imp_printer.hh */
class ImpPrinter : public ImpVisitor {
public:
    ...
    void visit(VarDeclaration*);
        void visit(CommentVarDec*);
    void visit(CommentStatement*);
    void visit(Comment* e);
    ...
}

```

Y se establecieron algunas reglas de printeo para los comentarios en el ImpPrinter.

```

/* imp_printer.cpp */
// Modificación del VarDeclaration
void ImpPrinter::visit(VarDeclaration *vd) {
    ...
    if (vd->cmt == NULL){
        cout << ";" << endl;
        return;
    }

    cout << ";\t";
    vd->cmt->accept(this);
}

// Se realizó lo mismo para los demás Stm que tienen comentarios en línea como atributo

void ImpPrinter::visit(CommentVarDec *c) {
    cout << c->comment << endl;
}

void ImpPrinter::visit(CommentStatement *s) {
    cout << s->comment << endl;
}

void ImpPrinter::visit(Comment* c) {
    cout << c->comment << endl;
}

```

ImpValueVisitor e ImpTypeVisitor

Como los comentarios no afectan la semántica del programa, no se realizaron cambios sustanciales en estos visitors. No obstante, para que no se genere un error al visitar un comentario, se añadió un método `accept` en las nuevas clases que solo se ignorarán.

```

/* imp_interpreter.hh */
void ImpInterpreter::visit(CommentStatement *s) {
    return;
}

void ImpInterpreter::visit(CommentVarDec *c) {
    return;
}

/* imp_typechecker.hh */
// Se realizó lo mismo para el TypeChecker

```

Ejemplo de funcionamiento

La inclusión de comentarios en el código fuente se puede ver en el siguiente ejemplo:

Input:

```

var int x, y; // Enteros
// Booleana
var bool b;
x = 3; // Asignacion
b = x < 5; // Comparacion
if b then
    y = x*2; // Multiplicacion
else // Multiplicacion 2
    y = x*3;
endif; // Fin del if
// Print
print(y);

```

Output:

```

Program :
var int x, y; // Enteros
// Booleana
var bool b;
x = 3; // Asignacion
b = x < 5; // Comparacion
if (b) then {
y = x * 2; // Multiplicacion
}
else {
// Multiplicacion 2
y = x * 3;
}
endif; // Fin de if
// Print
print(y);

Type checking:
Type checking OK

Run program:
6

```

Consideraciones finales

Es complicado manejar los comentarios en el código fuente, ya que estos pueden estar en cualquier parte del código. Por ello, se decidió implementar dos tipos de comentarios: los comentarios en línea y los comentarios aislados. Los comentarios en línea se pueden incluir al final de una declaración de variable o statement, mientras que los comentarios aislados pueden ir en cualquier parte del código. Aun así, hay casos en los que los comentarios no se imprimen correctamente, como en el caso de los comentarios aislados que están al final de un `if` o `else`. Finalmente, se puede mejorar la implementación de los comentarios para que se puedan incluir comentarios de bloque.

⚠ Reporte: ¿Qué cambios se hicieron al scanner y/o parser para lograr la inclusión de comentarios?

Como se vió en esta sección [2. Generación de Código I](#), se modificó el Scanner para que asigne un token a los comentarios y el Parser para que pueda parsear los comentarios de acuerdo a la gramática para así poder imprimirlos.

3. Sentencia Do While

En esta sección se describe la implementación de la sentencia `do while` en el lenguaje IMP-DEC. Para ello, se añadió la regla `DoWhileStatement` en la [gramática](#).

```
Stm ::= "do" Body "enddo" "while" Exp
```

Para implementar esta sentencia, primero se agregó la palabra reservada "enddo" en el scanner, que determinará el final de la sentencia **do** para proceder con el **while**

```
// imp_parser.hh
class Token {
    enum Type {
        ...
        ENDDO,
        ...
    };
}

// imp_parser.cpp
const char *Token::token_names[33] = {
    ...
    "COMMENT",
    ...
};

Scanner::Scanner(string s) : input(s), first(0), current(0) {
    ...
    reserved["enddo"] = Token::ENDDO;
    ...
}
```

Luego, se creó la clase `DoWhileStatement` que hereda de `Stm` y que contendrá el cuerpo del `do` y la condición del `while`.

```
/* imp.hh */
class DoWhileStatement : public Stm {
public:
    Body *body;
    Exp* cond;
    Comment* cmt;
    DoWhileStatement(Exp* c, Body *b, Comment* cmt);
    void accept(ImpVisitor* v);
    void accept(ImpValueVisitor* v);
    void accept(TypeVisitor* v);
    ~DoWhileStatement();
};

/* imp.cpp */
// Se crearon los constructores y métodos accept
```

Luego, se modificó el parser para que pueda parsear la sentencia `do while` de acuerdo a la gramática.

```

/* imp_parser.cpp */
Stm *Parser::parseStatement() {
    Stm *s = NULL;
    Exp *e;
    Body *tb, *fb;
    Comment *comment;

    ...
    // DoWhileStatement Parser
    else if (match(Token::DO)) {
        tb = parseBody();
        if (!match(Token::ENDDO)) {
            parserError("Expecting 'endwhile'");
        }
        if (!match(Token::WHILE)) {
            parserError("Expecting 'while'");
        }
        e = parseBExp();
        if (!match(Token::SEMICOLON))
            parserError("Expecting semicolon at end of statement declaration");
        comment = parseComment();
        s = new DoWhileStatement(e, tb, comment);
    }
    ...
    return s;
}

```

Finalmente, se modificaron los visitors para que puedan visitar la nueva sentencia `DoWhileStatement`.


```

/* imp_visitor.hh */
class ImpVisitor {
public:
    ...
    virtual void visit(DoWhileStatement* e) = 0;
    ...
}

// Se realizó lo mismo para ImpValueVisitor e ImpTypeVisitor y sus respectivas implementaciones

/* imp_printer.hh */
void ImpPrinter::visit(DoWhileStatement *s) {
    cout << "do {" << endl;
    s->body->accept(this);
    cout << "} enddo while (";
    s->cond->accept(this);
    cout << ");";

    if (s->cmt == NULL) {
        cout << endl;
        return;
    }

    cout << "\t";
    s->cmt->accept(this);
}

/* imp_interpreter.hh */
void ImpInterpreter::visit(DoWhileStatement *s) {
    ImpValue v;
    do {
        s->body->accept(this);
        v = s->cond->accept(this);
    } while (v.bool_value);
    return;
}

/* imp_typechecker.hh */
void ImpTypeChecker::visit(DoWhileStatement* s) {
    s->body->accept(this);
    ImpType tcond = s->cond->accept(this);
    if (tcond != TBOOL) {
        cout << "Type error en DoWhile: esperaba bool en condicional" << endl;
        exit(0);
    }
    return;
}

```

Ejemplo de funcionamiento

La sentencia do-while se puede ver en el siguiente ejemplo:

Input:

```

var int accum, x;
x = 4;
accum = 0;
do
    accum = accum + x;
    x = x - 1;
enddo while 0 < x; // Fin de bucle
print(x);
print(accum);

```

Output:

```

Program :
var int accum, x;
x = 4;
accum = 0;
do {
    accum = accum + x;
    x = x - 1;
} enddo while (0 < x); // Fin de bucle
print(x);
print(accum);

Type checking:
Type checking OK

Run program:
0
10

```

Consideraciones finales

Se optó por utilizar el token `enddo` para determinar el final de la sentencia `do` y proceder con el `while`. Aunque se podría haber utilizado un token `while` para determinar el final del `do`, se decidió utilizar `enddo` para que sea más claro el final de la sentencia y diferencialo del `WhileStatement`.

⚠ Reporte: Indicar el cambio a la gramática y los puntos donde se hicieron cambios al código. Además, proveer las definiciones de `tcheck` y `codegen` usadas

Como se vió en esta sección [3. Sentencia Do While](#), se añadió la regla `DoWhileStatement` en la gramática y se modificó el parser para que pueda parsear la sentencia `do while` de acuerdo a la gramática. Además, se modificaron los visitors para que puedan visitar la nueva sentencia `DoWhileStatement`.

Autores

Benjamin Soto



Fabrizio Vilchez



Jeffrey Monja



Benjamin Soto

<https://github.com/SotoBenjamin>
(<https://github.com/SotoBenjamin>)

Fabrizzio Vilchez

<https://github.com/Fabrizzio20k>
(<https://github.com/Fabrizzio20k>)

Jeffrey Monja

<https://github.com/jeffreymonjacaastro>
(<https://github.com/jeffreymonjacaastro>)

Referencias

- W. Appel. (2002) Modern compiler implementation in Java. 2.a edición. CambridgeUniversity Press.
- Kenneth C. Louden. (2004) Compiler Construction: Principles and Practice.Thomson.