Proyecto Compiladores

Renato Aurelio Cernades Ames
Luis Enrique Cortijo Gonazales
Max Brayam Antúnez Alfaro

27 de noviembre de 2023

1. Implementación de Comentarios

1.1. Cambios en el Scanner

Se implementó los comentarios para el lenguaje IMP0. El código se muestra en la siguiente figura 1.

```
case '/':
    //lógica de los comentarios
    c = nextChar();
    if (c == '/') {
        while ((c = nextChar()) != '\n' && c != '\0');
        return nextToken();
    } else {
        rollBack();
        token = new Token( type: Token::DIV);
        break;
    }
```

Figura 1: Obtenido de la función nextToken del *imp_parser.cpp*

Como se puede observar en la figura, cuando encontramos un slash (/), verificamos si existe otro slash, y consumimos caracteres hasta encontrar un salto de linea o fin de archivo. Caso contrario, se toma como un token de división.

2. Generación de código

En esta sección se mostrará la implementación en código y definiciones de codegen el *for loop*, constantes booleanas y operadores *and* y *or*

2.1. Constante TRUE y FALSE

Se implementó la generación de código para las constantes booleanas True y False. La implementación en código se puede ver en la figura 2.

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

Figura 2: Función ubicada en *imp_codegen.cpp*

Las defincione de *codegen* que se usarón fueron las siguientes:

`CodeGen(`*True*`)`
| push 1

`CodeGen(`*False*`)`
| push 0

2.2. Operadores AND y OR

Se implementó la generación de código para los operadores *and* y *or*. La implementación en código se puede ver en la figura 3.

```
case AND:
    op = "and";
    break;
case OR:
    op = "or";
    break;
```

Figura 3: Función ubicada en *imp_codegen.cpp*

Estos operadores forman parte de una operación binaria por lo que la definición de *codegen* es la misma que para *BinaryExp*.

`CodeGen(`*BinaryExp(e1, e2, op)*`)`
| `CodeGen(`*e1*`)`
| `CodeGen(`*e2*`)`
| op

Asumimos que el operador del *BinaryExp* es correcto, pues el parseo y el *typecheck* lo han verificado antes.

2.3. For Loop

Se implementó la generación de código para el *for loop*. La implementación en código se puede ver en la figura 4.

```
//ForStatement
int ImpCodeGen::visit(ForStatement *s) {
    string startLabel = next_label();
    string endLabel = next_label();
    inLoop = true;
    loop_repeat_label = startLabel;
    loop_finish_label = endLabel;

    direcciones.add_level();

    // Se añade la variable temporal al enviroment
    direcciones.add_var( var: s->id,  value: siguiente_direccion++);

    s->e1->accept( v: this);
    codegen( label: nolabel,  instr: "store",  arg: direcciones.lookup( x: s->id));
    codegen( label: startLabel,  instr: "skip");

    // Verifica si e1 <= e2, no salta en el jumpz
    codegen( label: nolabel,  instr: "load",  arg: direcciones.lookup( x: s->id));
    s->e2->accept( v: this);
    codegen( label: nolabel,  instr: "le");
    codegen( label: nolabel,  instr: "jmpz",  jmplabel: endLabel);
    codegen( label: nolabel,  instr: "load",  arg: direcciones.lookup( x: s->id));
    codegen( label: nolabel,  instr: "push",  arg: 1);
    codegen( label: nolabel,  instr: "add");
    codegen( label: nolabel,  instr: "store",  arg: direcciones.lookup( x: s->id));

    s->body->accept( v: this);

    codegen( label: nolabel,  instr: "goto",  jmplabel: startLabel);
    codegen( label: endLabel,  instr: "skip");

    direcciones.remove_level();
    inLoop = false;

    return 0;
}
```

Figura 4: Función ubicada en *imp_codegen.cpp*

Para esta implementación, se tuvo que crear un nuevo nivel de enviroment para guardar la variable del iterador que ha aumentando en cada iteración. El for finaliza cuando la condicion $e1 \leq e2$ es false. Por ejemplo, si $e1 = 5$ y $e2 = 10$, la condición va cumplir para 5, 6, 7, 8, 9 y 10, por lo cual iterará 6 veces hasta pasar el número dado por $e2$. Finalmente al finalizar el for, se elimina el nivel del enviroment creado. La definición formal en codegen lo podemos ver a continuación:

CodeGen(*ForStatement(e1, e2, body)*)
- CodeGen(*e1*)
- store addr(e1)
- L1: skip
- load addr(e1)
- CodeGen(*e2*)
- le
- jmpz L2
- load addr(e1)
- push 1
- add
- store addr(e1)
- CodeGen(*body*)
- goto L1
- L2: skip

3. Implementación del Do-While

3.1. Cambios en el código

Para poder implementar esta nueva instrucción, se creó un nuevo Statement. La implementación se puede ver en la figura 5.

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

Figura 5: Obtenido del archivo *imp.hh*

Se modificó el parser para reconocer el nuevo Statement *DoWhileStatement* como se puede ver en la figura 6.

```
  } else if (match(Token::DO)) {
    tb = parseBody();
    if (!match(Token::WHILE))
      parserError("Esperaba 'while'");
    e = parseExp();
    s = new DoWhileStatement(e,tb);
  }
  else {
    cout << "No se encontro Statement" << endl;
    exit(0);
  }
  return s;
}
```

Figura 6: Obtenido de la función *Parser::parseStatement*

Esta parte del código identifica específicamente las sentencias do-while. Al encontrar el Token *DO*, el analizador procede a parsear el cuerpo del bucle y luego espera el Token *WHILE* para parsear la condición del bucle. Si el Token está presente, se crea una nueva sentencia *DO-WHILE* con el cuerpo y la condición parseados. Si no se encuentra una sentencia reconocible, el programa imprime un mensaje de error y se detiene.

3.2. Cambios en la gramática

La gramática original se modificó para incluir la sentencia `do-while` de la siguiente manera:

Gramática Original:

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

Gramática Modificada:

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

3.3. Tcheck y Codegen

Para la verificación de tipos y generación de código de la sentencia `do-while`, se implementaron las siguientes funciones:

Tcheck:

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

Figura 7: Obtenido del archivo *imp_typechecker.cpp*

Este fragmento de código pertenece al componente `TypeChecker` del compilador, el cual es responsable de validar los tipos dentro de una sentencia `do-while`. Inicia invocando el método `accept` en el cuerpo de la sentencia (`body`), permitiendo así la verificación de tipos de manera recursiva para todas las sentencias anidadas. Posteriormente, procede a validar la condición del bucle (`cond`), también a través del método `accept`. En caso de que la condición no coincida con el tipo esperado (`booltype`), se imprime un mensaje de error señalando que la condición debe ser de tipo booleano, seguido de la terminación del programa. Esto garantiza que la semántica de la sentencia `do-while` sea consistente en términos de tipos de datos.

Codegen:

```
int ImpCodeGen::visit(DoWhileStatement *s) {
    string startLabel = next_label();
    string compareLabel = next_label();
    string endLabel = next_label();
    inLoop = true;
    loop_repeat_label = compareLabel;
    loop_finish_label = endLabel;

    codegen(startLabel, "skip");
    s->body->accept(this);

    codegen(compareLabel, "skip");
    s->cond->accept(this);
    codegen(nolabel, "jmpz", endLabel);
    codegen(nolabel, "goto", startLabel);
    codegen(endLabel, "skip");

    inLoop = false;
    return 0;
}
```

Figura 8: Obtenido del archivo *imp_codegen.cpp*

El fragmento de código corresponde al módulo `CodeGen` del compilador y maneja la generación de código para la sentencia `do-while`. Se comienza generando etiquetas únicas para marcar el inicio (`startLabel`), el punto de comparación (`compareLabel`), y el final (`endLabel`) del bucle. Variables de control de flujo como `inLoop` y `loop_repeat_label` se configuran para mantener el estado actual del procesamiento del bucle.

Tras emitir la etiqueta de inicio, se invoca el método `accept` en el cuerpo del bucle. Posteriormente, se emite la etiqueta de comparación y se procede con la evaluación de la condición del bucle. Utiliza una instrucción de salto condicional (`jmpz`) para decidir si el bucle debe continuar o terminar, basándose en el resultado de la condición. Si la condición evalúa a verdadero, se emite un salto incondicional (`goto`) para reiniciar el bucle. Al final, se emite la etiqueta que señala la conclusión del bucle y se establece la variable de control `inLoop` a falso, antes de retornar cero, indicando que la generación de código para la sentencia `do-while` ha terminado.

4. Implementación de sentencias Break y Continue

4.1. Cambios en el código

Para la implementación de las sentencias Break y Continue, se crearon dos nuevos Statements, además se agregaron los Tokens *BREAK* y *CONTINUE* para que puedan ser reconocidos en el *Parser* y finalmente se modificó la función *Parser::parseStatement* para que puedan ser reconocidos correctamente.

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

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

Figura 9: Obtenido del archivo *imp.hh*

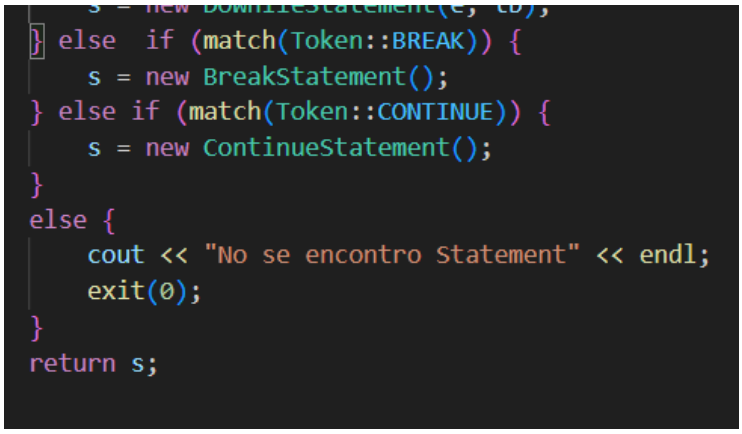

Figura 10: Obtenido del archivo *imp_parser.cpp*

4.2. Cambios en la gramática

La gramática se ha extendido para incluir las nuevas sentencias `break` y `continue`. A continuación, se muestra cómo se modificó la gramática original:

Gramática Original:

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

Gramática Modificada:

```
Stm ::= id "=" Exp |
        "print" "(" Exp ")" |
        "if" Exp "then" Body ["else" Body] "endif" |
        "while" Exp "do" Body "endwhile" |
        "for" id ":" Exp "," Exp "do" Body "endfor" |
        "do" Body "while" Exp |
        "break" |
        "continue"
```

Las nuevas sentencias permiten la interrupción del flujo normal de ejecución dentro de las estructuras de control de bucles, donde `break` termina el bucle inmediatamente, y `continue` pasa al siguiente ciclo de iteración.

4.3. Tcheck y Codegen

4.3.0.1 Break

En el componente `TypeChecker`, la sentencia `Break` se implementa mediante una función de visita que no lleva a cabo ninguna operación, lo que indica que no hay comprobaciones de tipo asociadas con esta sentencia. La generación de código para `Break`, sin embargo, es más compleja: la función `visit` correspondiente en el módulo `CodeGen` verifica si `Break` se encuentra dentro de un bucle.

Si no es así, imprime un mensaje de error y termina la ejecución del programa. Si se confirma que `Break` está dentro de un bucle, se genera un salto incondicional (*goto*) hacia la etiqueta que señala el final del bucle, terminando su ejecución de manera efectiva.

```
void ImpTypeChecker::visit(BreakStatement *) { return;};
```

Figura 11: Método de visita de la sentencia *Break* en *imp_typechecker.cpp*

```
int ImpCodeGen::visit(BreakStatement *s) {
    if (loop_finish_label.empty()) {
        cout << "Break necesita estar dentro de un bucle" << endl;
        exit(0);
    }
    codegen(nolabel, "goto", loop_finish_label);
    return 0;
}
```

Figura 12: Generación de código para la sentencia *Break* en *imp_codegen.cpp*

4.3.0.2 Continue

La sentencia `Continue` sigue un enfoque similar en el `TypeChecker` al de `Break`, con una función de visita que no efectúa ninguna comprobación de tipos. En la generación de código, la función `visit` para `ContinueStatement` también verifica que la sentencia se utilice dentro de la estructura de un bucle. De no ser así, el compilador emite un mensaje de error y se detiene. En caso afirmativo, se genera un salto incondicional (*goto*) hacia la etiqueta que indica el inicio o la condición del bucle, permitiendo que la ejecución continúe con la siguiente iteración del bucle.

```
void ImpTypeChecker::visit(ContinueStatement *) { return;};
```

Figura 13: Método de visita de la sentencia *Continue* en *imp_typechecker.cpp*

```
int ImpCodeGen::visit(ContinueStatement *s) {
    if (loop_repeat_label.empty()) {
        cout << "Continue necesita estar dentro de un bucle" << endl;
        exit(0);
    }
    codegen(nolabel, "goto", loop_repeat_label);
    return 0;
}
```

Figura 14: Generación de código para la sentencia *Continue* en *imp_codegen.cpp*

5. Anexos

Repositorio de Github: https://github.com/Maxantunez404/Proyecto_Compiladores