Informe del Compilador C-TDS

Materia: Taller de diseño de software.

Integrantes: Agustin Bauer, Alan Gonzalez, Luciano Putruele.

GLOSARIO:

Introducción:

Diseño e implementación de un compilador para el lenguaje ctds.
En este informe, se va explicar las consideraciones tenidas en cuenta de cada etapa del compilador, además, de las decisiones de diseño elegidas para su construcción.
Por otro lado, se explicara de cada etapa las estructuras de datos diseñadas, para ser utilizadas por cada etapa.
Se suma a este informe, aclaraciones y ejemplificación de algunos algunos aspectos difíciles de entender en el código del compilador.
La organización del siguiente informe va a estar dividida por las etapas del compilador, las cuales cada una contiene una breve introducción a la etapa, las estructuras de datos utilizadas y las decisiones de diseño en la misma. Y por último, las consideraciones, aclaraciones y ejemplos.

Sobre Implementación General:

La implementación del compilador ctds fue desarrollada usando diferentes herramientas, como CUP, Jflex, el lenguajes de programación JAVA, sistema operativo linux de 64bits, editores de texto, etc.

Etapas :

Analizador Sintáctico y Léxico de ctds (Scanner y Parser)

Para desarrollar dicho lenguaje, durante la primera etapa se implementó en el lenguaje Jflex los token del lenguaje teniendo en cuenta las siguientes consideraciones:

1) Las palabras reservadas del lenguaje son: boolean, break, class, continue, else, false, float, for, if, int, return, true, void, while.
2) Los valores enteros son de la forma digito+, donde digito es un valor de [0,9].
3) Los valores reales son de la forma digito+ . digito+, donde digito es un valor de [0,9].
4) Los identificadores son de la forma alpha alphaNumeric, donde alpha es un valor de [a..zA..Z] y alphaNumeric es un valor de [a..zA..Z0..9_].
5) Los tipos de comentario que acepta el lenguaje son el comentario en linea que es de la forma "//comentario " y el comentario multilinea que es de la forma "/* comentario [/n] */".

Luego de haber definido todos los token del lenguaje, se definió la gramática (Análisis Sintáctico del lenguaje) la cual se desarrollo en CUP, estratificando las diferentes reglas para evitar conflictos reduce/reduce & shift/reduce

Analizador Semántico de ctds:

La verificación de las reglas semánticas del lenguaje ctds se realizó utilizando una tabla de símbolos para mantener la información de los identificadores de un programa. Dicha tabla fue implementada como una lista de listas (HashMap). Donde cada elemento del HashMap era del tipo Descriptor (definidos por nosotros).

Para el análisis semántico a cada nodo del árbol de parsing se le asoció un tipo Descriptor, el cual almacenaba la información necesaria para determinar las reglas semánticas en cada punto del programa.

Además, para realizar el analizador semántico se consideró que el lenguaje ctds es de tipos de datos estáticos, es decir que tiene alcance estático y fuertemente tipado.

Sobre la implementación:

Para la etapa de verificación semántica se desarrollaron las siguientes estructuras de datos en Java:

* Tabla de símbolos: implementada como una lista (LinkedList) de ambientes (HashMap), donde cada ambiente está conformado por descriptores (clase abstracta) que pueden ser de tipo simple, función, o arreglo (concretas) que definen sus características.

Consideraciones

- -Consideraciones Jflex

-Definimos el metodo symbol, dentro del cual construimos un Symbol, asi nos abstraemos del numero de linea y de columna.
-En cuanto a los comentarios, se aceptan comentarios simples y multilinea, donde estos ultimos funcionan como en Java, se pueden anidar muchas aperturas pero al cerrar una se cierran todas las anidadas.
-En cuanto a los tipos de los objetos que devuelven los literales (intLit, floatLit), estos son de tipo Integer y Float respectivamente (tipos de Java).
-Las palabras reservadas son: boolean, break, class, continue, else, false, float, for, if, int, return, true, void, while, externinvk.

- Consideraciones Tabla de simbolos

- Implementada como una lista de Ambientes.
- Cada Ambiente es un HashMap de Descriptores.
-Hay 3 tipos de Descriptores, uno para variables simples, otro para arreglos y otros para funciones.

- Consideraciones CUP

-Tenemos 3 atributos en action code, una tabla de simbolos, un Type que sirve para guardar el tipo con el que se declara una variable o metodo, y una lista de parametros que sirve para guardar los parametros de un metodo que se van a insertar en el ambiente del bloque de dicho metodo (mediante el metodo putParameters ahi declarado).

- Sobre el parser code, tenemos como atributo una lista de bloques, donde se guardan los bloques correspondientes a los metodos y mediante el getASTs, nuestra clase main podra acceder a los mismos. Ademas tenemos un metodo report fatal error para reportar errores.

-Los no terminales son, assign_op, op, arith_op, rel_op, eq_op, cond_op, unary_op, id, block_content , statement, extern,expr, location_decl, method, location, method_call, type, statement_decl, parameters, expr_sequence, externinvk_arg_sequence, program, decl, field_decl, method_decl, externinvk_arg, id_sequence, literal, int_literal, bool_literal, float_literal, string_literal.

-Gramaticas

```
program ::= CLASS id LKEY {: tds.push(new Ambiente());:} decl  RKEY {:tds.pop();:}
              |CLASS id LKEY RKEY;
```

Al abrir la llave, apilamos el ambiente principal y luego de cerrar la llave (luego de todo el programa), desapilamos.

```
  decl ::= field_decl
         | method_decl
         | field_decl method_decl;
```

decl puede estar compuesto por declaraciones, metodos o declaraciones seguidas de metodos.

```
  field_decl ::= type id_sequence      SEMI
               | field_decl type id_sequence SEMI;
```

Una declaracion esta formada por su tipo y una secuencia de identificadores, field_decl incluye una o mas declaraciones.

```
id_sequence ::= location_decl:l {: tds.top().put(l.getId(),l.getDesc()) ;:}
                | id_sequence COMMA location_decl:l {: tds.top().put(l.getId(),l.getDesc()) ;:};
```

Lista de identificadores que pueden ser variables simples o arreglos con su longitud.
Cada identificador que se define es agregado en el ambiente tope de la tabla de simbolos.

```
location_decl ::= id:i {: if (tds.top().get(i)==null)
                              RESULT = new VarLocation(i,new DescriptorSimple
                                                       (i,tipoVar),ileft,iright) ;
                          else
                              parser.report_fatal_error("Variable
redeclarada",ileft,iright);:}
                | id:i LBRACKET int_literal:il RBRACKET {:
                              if (tds.top().get(i)==null)
                                  RESULT = new VarLocation(i,new
DescriptorArreglo(i,tipoVar,il.getValue()),ileft,iright) ;
                              else
                                  parser.report_fatal_error("Variable
redeclarada",ileft,iright);:} ;
```

Location_decl puede ser una variable comun o un arreglo con su logintud.
Si el identificador se encuentra en el nivel actual, reporta un error por variable redeclarada, si no devuelve un VarLocation.

```
method_decl ::= method:m {: tds.top().put(m.getId(),m.getDesc()) ;:}
                | method_decl:md method:m {:
tds.top().put(m.getId(),m.getDesc()) ;:};
```

Method_decl puede ser un metodo o varios metodos. Se inserta en el ambiente del tope de la tabla de simbolos cada metodo.

```
method        ::= type:t id:i LPAREN RPAREN block:b {:
                      parser.asts.add(b);
                          if (tds.top().get(i)==null)
                              RESULT = new VarLocation(i,b,new
DescriptorFuncion(i,t,null),ileft,iright) ;
                          else
                              parser.report_fatal_error("Metodo
redeclarado",ileft,iright); :}
                    |VOID id:i LPAREN RPAREN block:b {: parser.asts.add(b);
                          if (tds.top().get(i)==null)
                              RESULT = new VarLocation(i,b,new
DescriptorFuncion(i,Type.VOID,null),ileft,iright) ;
                          else
                              parser.report_fatal_error("Metodo
redeclarado",ileft,iright); :}
                | VOID id:i LPAREN parameters:p  {:params = p;:} RPAREN
block:b {:  parser.asts.add(b);
                          if (tds.top().get(i)==null)
                              RESULT = new VarLocation(i,b,new
DescriptorFuncion(i,Type.VOID,p),ileft,iright) ;
```

```
                                                    else
                                                        parser.report_fatal_error("Metodo
redeclarado",ileft,iright);
                                                params = null; :}
                                            | type:t id:i LPAREN parameters:p {:params = p;:} RPAREN
block:b {:  parser.asts.add(b);
                                                if (tds.top().get(i)==null)
                                                    RESULT = new VarLocation(i,b,new
DescriptorFuncion(i,t,p),ileft,iright) ;
                                                    else
                                                        parser.report_fatal_error("Metodo
redeclarado",ileft,iright);
                                                params = null; :}  ;
```

La declaracion de un metodo puede estar compuesta por el tipo del valor de retorno seguido de un identificador (nombre del metodo), seguido de () y luego su bloque, o puede estar compuesto por el tipo del valor de retorno seguido de un identificador seguido de una lista de parametros entre (), seguido de su bloque. En todos los casos se inserta en la lista asts el bloque del metodo y se verifica que el identificador no este contenido dentro del mismo ambiente en la tabla de simbolos, si es asi se devuelve un nuevo VarLocation, y si no se devuelve un error. En el caso en que el metodo posea parametros, estos son insertados en la lista params definida globalmente.
Tratar a VOID como un caso especial de tipo fue una desicion de implementación que tomamos durante el desarrollo. Podria no haberse tomado como un caso especial.

```
  parameters ::= type:t id:i {: LinkedList<DescriptorSimple> l = new
LinkedList<DescriptorSimple>() ;
                                                    l.add(new DescriptorSimple(i,t));
                                                    RESULT = l; :}
                                | parameters:p COMMA type:t id:i {:  p.add(new
DescriptorSimple(i,t));
                                                                    RESULT = p ; :};
```

Parameters puede ser un parametro formado por su tipo y un identificador o muchos parametros. En el caso base se crea una lista de descriptores simples y se inserta el parametro, mientras que en el caso recursivo se insertan los demas parametros.

```
  block ::=  LKEY {:tds.push(new Ambiente()) ; putParameters();:} block_content:b  RKEY {:/*if
(!visitor.visit(b).isUndefined())*/ RESULT = b;
        tds.pop();:};
```

Un bloque esta formado por una llave que abre, su contenido y una llave que cierra. Cuando se abre el bloque se crea y se inserta en el tope de la tabla de simbolos un nuevo ambiente con los parametros del metodo, si es que tiene. Cuando se cierra el bloque se devuelve el contenido y se desapila.

```
  block_content ::= statement_decl:sd {: RESULT = new Block(sd);:}
                                    | field_decl  {: RESULT = new Block(null);:}
                                    | field_decl statement_decl:sd  {: RESULT = new Block(sd);:}
                                    |{: RESULT = new Block(null);:};
```

block_content puede estar conformado por un statement_decl, o por declaraciones y luego statement_decl, en estos casos se devuelve un nuevo bloque con sus statement_decl. O puede estar

conformado solo por declaraciones o puede estar vacio.

```
type ::= INT {:tipoVar = Type.INT;  RESULT = Type.INT ; :}
                        | FLOAT {:tipoVar = Type.FLOAT; RESULT = Type.FLOAT ; :}
                        | BOOLEAN {:tipoVar = Type.BOOL; RESULT = Type.BOOL ; :};
```

No hay consideraciones.

```
statement_decl ::= statement:s {: LinkedList<Statement> l = new LinkedList<Statement>() ;
                                                                l.add(s);
                                                                RESULT = l; :}
                                                | statement_decl:sd statement:s {:  sd.add(s);

                RESULT = sd ; :};
```

statements_decl puede estar formado por un statement (caso base) o por varios statements. En el caso base se crea una lista de statements, se inserta el statement y se devuelve la lista, y en el caso inductivo se agregan los statement y se devuelve la lista.

```
statement ::= location:loc assign_op:aop expr:e SEMI {: RESULT = new
AssignStmt(loc,aop,e,locleft,locright) ; :}
                        | method_call:m SEMI {: RESULT = new MethodCallStmt(m) ; :}
                        | extern:e SEMI {: RESULT = new ExternStmt(e) ; :}
                        | IF LPAREN expr:e RPAREN block:b1 ELSE block:b2 {: RESULT = new
IfStmt(e,b1,b2) ; :}
                        | IF LPAREN expr:e RPAREN block:b {: RESULT = new IfStmt(e,b) ; :}
                        | FOR id:i EQ expr:e COMMA expr:c block:b {:Descriptor d = tds.search(i);

        if(d != null)

                if(d.getClase().equals("descriptorSimple")){

                        if(d.getTipo() == Type.INT)

                                RESULT = new ForStmt(i,e,c,b); }

                else

                        parser.report_fatal_error("Los tipos no concuerdan",ileft,iright);

        else

                parser.report_fatal_error("Variable no declarada",ileft,iright); :}
                        | WHILE expr:e block:b {: RESULT = new WhileStmt(b,e) ; :}
                        | RETURN expr:e SEMI {: RESULT = new ReturnStmt(e)          ; :}
                        | RETURN SEMI       {: RESULT = new ReturnStmt(); :}
                        | BREAK SEMI    {: RESULT = new BreakStmt() ; :}
                        | CONTINUE SEMI {: RESULT = new ContinueStmt() ; :}
                        | SEMI  {: RESULT = new SkipStmt() ; :}
                        | block:b {: RESULT = b ; :};
```

En el statement del for, se verifica que el id ya este definido como un descriptor simple y ademas de

tipo entero.

```
assign_op ::= EQ {: RESULT = AssignOpType.ASSIGN;:}
                | PLUSEQ  {: RESULT = AssignOpType.INCREMENT;:}
                | MINUSEQ  {: RESULT = AssignOpType.DECREMENT;:};
```

assign_op retorna el simbolo = en el caso de una asignacion simple, un += en el caso de una asignacion para incrementar o un -= para decrementar.

```
method_call ::= id:i LPAREN RPAREN {: Descriptor d = tds.search(i);
                                                    if (d!=null)
                                                        if
(d.getClase().equals("descriptorFuncion"))
                                                            RESULT =
new MethodCall(i,d.getTipo(),null,ileft,iright) ;
                                                        else

parser.report_fatal_error("Funcion no declarada",ileft,iright);
                                                    else

parser.report_fatal_error("Los tipos no concuerdan",ileft,iright); :}
              | id:i LPAREN expr_sequence:l RPAREN {: Descriptor d =
tds.search(i);
                                                    if (d!=null)
                                                        if
(d.getClase().equals("descriptorFuncion"))
                                                            RESULT =
new MethodCall(i,d.getTipo(),l,ileft,iright) ;
                                                        else

parser.report_fatal_error("Funcion no declarada",ileft,iright);
                                                    else

parser.report_fatal_error("Los tipos no concuerdan",ileft,iright); :};
```

method_call puede ser una llamada a un metodo con o sin parametros, en ambos casos se verifica que el metodo se encuentre en la tabla de simbolos y que sea un descriptor de tipo funcion para retornar un nuevo MethodCall, en caso contrario retorna un error.

```
extern ::=  EXTERNINVK LPAREN string_literal:s COMMA type:t RPAREN {: RESULT
= new Extern(s,t,null,sleft,sright)  ;:}
              | EXTERNINVK LPAREN string_literal:s COMMA VOID RPAREN
{: RESULT = new Extern(s,Type.VOID,null,sleft,sright)  ;:}
              | EXTERNINVK LPAREN string_literal:s COMMA type:t COMMA
externinvk_arg_sequence:l RPAREN {: RESULT = new Extern(s,t,l,sleft,sright)  ;:}
              | EXTERNINVK LPAREN string_literal:s COMMA VOID COMMA
externinvk_arg_sequence:l RPAREN {: RESULT = new Extern(s,Type.VOID,l,sleft,sright)  ;:};
```

En todos los casos se retorna un nuevo Extern con sus caracteristicas.

```
location ::= id:i {:  Descriptor d = tds.search(i);
                                    if (d!=null)
```

```
                                                    if (d.getClase().equals("descriptorSimple"))
                                                        RESULT = new
VarLocation(i,d,ileft,iright);
                                                    else
                                                        parser.report_fatal_error("Los tipos no
concuerdan",ileft,iright);
                                                else
                                                    parser.report_fatal_error("Variable no
declarada",ileft,iright); :}
                                | id:i LBRACKET expr RBRACKET {:  Descriptor d = tds.search(i);
                                                if (d!=null)
                                                    if (d.getClase().equals("descriptorArreglo"))
                                                        RESULT = new
VarLocation(i,d,ileft,iright);
                                                    else
                                                        parser.report_fatal_error("Los tipos no
concuerdan",ileft,iright);
                                                else
                                                    parser.report_fatal_error("Variable no
declarada",ileft,iright);:}  ;
```

location puede ser una variable o un arreglo, en ambos casos se verifica que no se encuentren en el ambiente actual de la tabla de simbolos, si es asi se devuelve el descriptor (simple o de arreglo) si no se devuelve un error.

```
        expr_sequence ::= expr:e {: LinkedList<Expression> l = new LinkedList<Expression>() ;
                                                        l.add(e);
                                                        RESULT = l; :}
                                    | expr_sequence:es COMMA expr:e {: es.add(e);

        RESULT = es  ;:} ;
```

expr_sequence puede ser una expr o varias expr. Se define una lista donde se van insertando las expresiones.

```
        expr ::= location:loc {: RESULT =loc ; :}
                | method_call:mc {: RESULT = mc ; :}
                | extern:e {: RESULT = e  ;:}
                | literal:l {: RESULT =l ; :}
                | expr:e PLUS expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.PLUS,e2,eleft,eright) ; :}
                | expr:e MINUS expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.MINUS,e2,eleft,eright) ; :}
                | expr:e TIMES expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.TIMES,e2,eleft,eright) ; :}
                | expr:e MOD expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.MOD,e2,eleft,eright) ; :}
                | expr:e DIVIDE expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.DIVIDE,e2,eleft,eright) ; :}
                | expr:e LT expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.LT,e2,eleft,eright) ; :}
```

```
                | expr:e GT expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.GT,e2,eleft,eright) ; :}
                | expr:e LTEQ expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.LTEQ,e2,eleft,eright) ; :}
                | expr:e GTEQ expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.GTEQ,e2,eleft,eright) ; :}
                | expr:e EQEQ expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.EQEQ,e2,eleft,eright) ; :}
                | expr:e NOTEQ expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.NOTEQ,e2,eleft,eright) ; :}
                | expr:e AND expr:e2 {: RESULT = new
BinOpExpr(e,BinOpType.AND,e2,eleft,eright) ; :}
                | expr:e OR expr:e2 {: RESULT = new BinOpExpr(e,BinOpType.OR,e2,eleft,eright)
; :}
                | MINUS expr:e {: RESULT = new
UnaryOpExpr(UnaryOpType.MINUS,e,eleft,eright) ; :}
                | NOT expr:e {: RESULT = new
UnaryOpExpr(UnaryOpType.NOT,e,eleft,eright) ; :}
                | LPAREN expr:e RPAREN  {: RESULT = e; :};

        externinvk_arg_sequence ::= externinvk_arg:e {: LinkedList<Object> l = new
LinkedList<Object>() ;

        l.add(e);

        RESULT = l; :}
                                                        | externinvk_arg_sequence:es COMMA
externinvk_arg:e {: es.add(e);

                                                        RESULT = es  ;:} ;

        externinvk_arg ::= expr:e {: RESULT = e  ;:}
                                        | string_literal:s {: RESULT = s  ;:};

        unary_op ::= NOT {: RESULT = UnaryOpType.NOT  ;:} %prec UMINUS
                                | MINUS {: RESULT = UnaryOpType.MINUS  ;:} %prec UMINUS ;
        bin_op ::= arith_op:ao {: RESULT = ao  ;:}
                                | rel_op:ro {: RESULT = ro ;:}
                                | eq_op:eo {: RESULT = eo  ;:}
                                 | cond_op:co {: RESULT = co  ;:};
        arith_op ::= PLUS {: RESULT = BinOpType.PLUS  ;:}
                                | MINUS{: RESULT = BinOpType.MINUS  ;:}
                                 | TIMES {: RESULT = BinOpType.TIMES  ;:}
                                | MOD {: RESULT = BinOpType.MOD  ;:}
                                | DIVIDE {: RESULT = BinOpType.DIVIDE  ;:};
        rel_op ::= LT {: RESULT = BinOpType.LT  ;:}
                        | GT {: RESULT = BinOpType.GT  ;:}
                        | LTEQ {: RESULT = BinOpType.LTEQ  ;:}
                        | GTEQ {: RESULT = BinOpType.GTEQ  ;:};
        eq_op ::= EQEQ {: RESULT = BinOpType.EQEQ  ;:}
                        | NOTEQ {: RESULT = BinOpType.NOTEQ  ;:} ;
        cond_op ::= AND {: RESULT = BinOpType.AND  ;:}
```

```
                                    | OR {: RESULT = BinOpType.OR  ;:};
        literal ::= int_literal:il {: RESULT = il ; :} | float_literal:fl {: RESULT = fl ; :} |
bool_literal:bl {: RESULT = bl ; :};
        id ::= IDENTIFIER:i {: RESULT = i ;:};
        int_literal ::= INT_LITERAL:n {: RESULT = new IntLiteral(n,nleft,nright) ;:};
        bool_literal ::= TRUE:t {: RESULT = new BoolLiteral(t,tleft,tright) ;:}
                                            | FALSE:f {: RESULT = new BoolLiteral(f,fleft,fright) ;:} ;
        float_literal ::= FLOAT_LITERAL:f {: RESULT = new FloatLiteral(f,fleft,fright);:} ;
        string_literal ::= STRING_LITERAL:s {: RESULT = s ;:};
```

- Consideraciones TypeCheckVisitor
  - Tenemos una lista donde se guardan los posibles errores de tipos que se encuentren.
  -Si un visit falla devuelve UNDEFINED en caso contrario devuelve el tipo del parametro (VOID si es una sentencia).
  -Las hojas del arbol sintactico son los literales y los identificadores (VarLocation).

- Consideraciones TACGenerator
  - A la informacion para generar el codigo intermedio la guardamos en objetos de tipo TACCommand con su tipo de operación y las expresiones que usa.
  - Para los distintos tipos de operación usamos un enumerado llamado TACOpType.
  - En el TACGenerator hay una lista de TACCommand que representa el codigo intermedio, los atributos beginIter y endIter sirven para marcar el comienzo y fin de un bucle, commId y labelId son los numeros commando y etiqueta, y tambien hay una pila usada para almacenar los beginIter y endIter.
  - La clase Pair es usada para gaurdar el beginIter y endIter como par en la pila.