Proyecto Compiladores: CTds Compiler

Facundo Molina

Diciembre, 2015

1 Introducción

El proyecto consistió en el diseño e implementación de un compilador para el lenguaje CTds, un lenguaje de programación simple muy similar al lenguaje C. En este documento se presentará la gramática y la semántica del lenguaje CTds, y luego se dará una descripción de cada una de las etapas del desarrollo del compilador CTdsCompiler, explicando las decisiones de diseño más relevantes.

2 Gramática del lenguaje CTds

⟨*program*⟩ → ⟨*class_decl*⟩ $^{+}$

⟨*class_decl*⟩ → **class** ⟨*id*⟩ '`{`' ⟨*field_decl*⟩* ⟨*method_decl*⟩* '`}`'

⟨*field_decl*⟩ → ⟨*type*⟩ { ⟨*id*⟩ | ⟨*id*⟩ '`[`' ⟨*int_literal*⟩ '`]`' }$^{+}$, '`;`'

⟨*method_decl*⟩ → { ⟨*type*⟩ | **void** } ⟨*id*⟩ '`(`' [{ ⟨*type*⟩ ⟨*id*⟩ }$^{+}$,] '`)`' ⟨*body*⟩

⟨*body*⟩ → ⟨*block*⟩
| **extern** '`;`'

⟨*block*⟩ → '`{`' ⟨*field_decl*⟩* ⟨*statement*⟩* '`}`'

⟨*type*⟩ → **int** | **float** | **boolean**

⟨*statement*⟩ → ⟨*location*⟩ ⟨*assign_op*⟩ ⟨*expr*⟩ '`;`'
| ⟨*method_call*⟩ '`;`'
| **if** '`(`' ⟨*expr*⟩ '`)`' ⟨*block*⟩ [**else** ⟨*block*⟩]
| **for** ⟨*id*⟩ '`=`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ ⟨*block*⟩
| **while** ⟨*expr*⟩ ⟨*block*⟩
| **return** [⟨*expr*⟩] '`;`'
| **break** '`;`'
| **continue** '`;`'
| '`;`'
| ⟨*block*⟩

⟨*assign_op*⟩ → '`=`' | '`+=`' | '`-=`'

⟨*method_call*⟩ → ⟨*id*⟩ { .⟨*id*⟩ }* '`(`' [⟨*expr*⟩$^{+}$,] '`)`'

⟨*location*⟩ → ⟨*id*⟩ { .⟨*id*⟩ }*
| ⟨*id*⟩ { .⟨*id*⟩ }* '`[`' ⟨*expr*⟩ '`]`'

⟨*expr*⟩ → ⟨*location*⟩
| ⟨*method_call*⟩
| ⟨*literal*⟩
| ⟨*expr*⟩ ⟨*bin_op*⟩ ⟨*expr*⟩
| '`-`' ⟨*expr*⟩

| ‘!’ ⟨*expr*⟩
| ‘(’ ⟨*expr*⟩ ‘)’

⟨*bin_op*⟩ → ⟨*arith_op*⟩ | ⟨*rel_op*⟩ | ⟨*eq_op*⟩ | ⟨*cond_op*⟩

⟨*arith_op*⟩ → ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’

⟨*rel_op*⟩ → ‘<’ | ‘>’ | ‘<=’ | ‘>=’

⟨*eq_op*⟩ → ‘==’ | ‘!=’

⟨*cond_op*⟩ → ‘&&’ | ‘||’

⟨*literal*⟩ → ⟨*int_literal*⟩ | ⟨*float_literal*⟩ | ⟨*bool_literal*⟩

⟨*id*⟩ → ⟨*alpha*⟩ ⟨*alpha_num*⟩*

⟨*alpha_num*⟩ → ⟨*alpha*⟩ | ⟨*digit*⟩ | ‘_’

⟨*alpha*⟩ → ‘a’ | ‘b’ | ... | ‘z’ | ‘A’ | ‘B’ | ... | ‘Z’

⟨*digit*⟩ → ‘0’ | ‘1’ | ‘2’ | ... | ‘9’

⟨*int_literal*⟩ → ⟨*digit*⟩ ⟨*digit*⟩ *

⟨*bool_literal*⟩ → **true** | **false**

⟨*float_literal*⟩ → ⟨*digit*⟩ ⟨*digit*⟩ * ‘.’ ⟨*digit*⟩ ⟨*digit*⟩ *

3 Semántica del lenguaje CTds

En esta sección se van a detallar las decisiones de más importancia con respecto a la semántica del lenguaje.

3.1 General

Si bien en la gramática un programa CTds consiste de una lista de declaraciones de clases, y esto es considerado de esa forma en las etapas de análisis léxico y sintáctico, a partir de la etapa de análisis semántico se considera que un programa tiene sólo una clase. Esta clase es una lista de declaraciones de atributos seguida de una lista de declaraciones de métodos. Los atributos son variables globales que pueden ser accedidos por cualquier método de la clase. Los métodos representan tanto funciones como procedimientos. Además debe haber exactamente un método **main** sin argumentos, el cuál será el método por donde comenzará la ejecución.

3.2 Tipos

Los tipos básicos son **int**, **float** y **boolean**. Se permiten definir arreglos (sólo unidimensionales y de tamaño fijo) para cada uno de los tipos básicos, y son indexados de 0 a $N-1$, donde $N > 0$ es el tamaño.

Los atributos de una clase pueden de ser de tipos básicos o arreglos, al igual que las variables locales de cualquier bloque.

3.3 Alcance y visibilidad de los identificadores

Todos los identificadores deben ser definidos antes de ser utilizados.

En un punto de un programa CTds existen al menos dos *scopes* válidos, el global (identificadores de atributos y métodos de la clase) y el local a un método (identificadores de variables declaradas en el cuerpo). Dentro de los métodos se pueden introducir *scopes* locales mediante bloques (⟨*block*⟩) anidados. Este anidamiento puede causar que un identificador definido en un bloque posterior pueda ocultar alguno definido con el mismo nombre en un bloque previo.

Los nombres de los identificadores son únicos en cada scope. Aunque puede haber atributos o variables con el mismo nombre que un método sin problemas.

3.4 Locaciones de memoria

Sólo hay dos clases de locaciones: variables y arreglos (locales y globales). Cada una tiene un tipo. Los arreglos son alocados en el espacio de datos estático del programa (frame de ejecución).

3.5 Asignaciones

Las asignaciones pueden ser a variables de tipos de básicos o a posiciones de un arreglo. La semántica de las asignaciones define la copia del valor. Una asignación es válida si la locación y la expresión que se intenta asignar tienen el mismo tipo. Las asignaciones de incremento y decremento solo son permitidas para tipos de datos numéricos.

3.6 Invocación y Retorno de Métodos

La invocación de métodos involucra: pasar el valor de los parámetros actuales a los parámetros formales, ejecutar el cuerpo del método invocado y retornar del método invocado (posiblemente con un resultado).

Los argumentos son pasados por valor. Los parámetros formales son considerados como variables locales del método.

Tanto los parámetros como el tipo de retorno de un método deben ser de tipo básico, excepto los métodos que retornan **void**.

Los métodos que retornan **void** solamente pueden ser utilizados como una sentencia y además al final deben tener una sentencia **return** sin una expresión asociada.

Un método que retorna un resultado puede ser invocado como cualquier expresión. Retornan solo si se alcanza una sentencia **return** con una expresión del mismo tipo que el tipo de retorno del método.

3.7 Sentencias de Control

Las sentencias **if** y **while** tienen la semántica estándar.

En la sentencia **for** el $\langle id \rangle$ es la variable índice del ciclo, cuyo valor inicial viene dado por la evualuación de la primera $\langle expr \rangle$. La segunda $\langle expr \rangle$ es el valor final de la variable índice, es decir se cicla mientras el valor de $\langle id \rangle$ sea menor o igual al valor de la segunda $\langle expr \rangle$. Ambas expresiones deben ser de tipo **int**.

Las **expresiones** siguen las reglas usuales de evaluación. Una locación (variables y elementos de un arreglo) es evaluada al valor que contiene en memoria. Los literales enteros, reales y booleanos evaluan a su valor. En cuanto a los operadores todos tienen el significado usual, teniendo en cuenta que para los operadores binarios ambos operandos deben ser del mismo tipo, y los operadores relacionales o de igualdad deben devolver un resultado de tipo **boolean**.

3.8 Llamadas a funciones externas

Se pueden declarar métodos externos reemplazando su cuerpo por la palabra reservada **extern**.

3.9 Reglas extras

Las siguientes son algunas reglas semánticas adicionales:

1. Ningún identificador es declarado dos veces en el mismo bloque.
2. Ningún identificador es utilizado antes de ser declarado.
3. Todo programa tiene una única clase, con un único metodo **main**.
4. Las invocaciones a métodos respetan la aridad con la que los métodos fueron declarados.
5. Las invocaciones a métodos utilizadas como expresión retornan un resultado.
6. La sentencia **return** solo tiene asociada una expresión si el método retorna un valor, si el método es **void** no debe tener ninguna.
7. Un $\langle id \rangle$ usado como una $\langle location \rangle$ debe estar declarado como parámetro, o como variable local o global.
8. La $\langle expr \rangle$ en una sentencia **if** o **while** debe ser de tipo **boolean**.

9. Las expresiones iniciales y finales de un **for** deben ser de tipo **int**.

10. Las sentencias **break** y **continue** solo pueden encontrarse en el cuerpo de un ciclo.

4 Etapas

4.1 Análisis Léxico

El objetivo de esta etapa es construir el analizador léxico del lenguaje, el cuál a partir del código fuente de un programa CTds debe reconocer que todos los símbolos son correctos, y retornar *tokens*. Comunmente estos *tokens* representan una clase de símbolos del lenguaje, de manera que si, por ejemplo, los símbolos '+','-','*','/' y '%' forman parte del lenguaje, podemos agruparlos en la clase de símbolos "Operadores Aritméticos", y expresarlo en el analizador léxico de la siguiente manera:

```
79  /* Arithmetical Operators */
80  "+"         { return symbol(CTdsSymbol.ARITHMETICAL_OP,BinOpType.PLUS); }
81  "-"         { return symbol(CTdsSymbol.ARITHMETICAL_OP,BinOpType.MINUS); }
82  "*"         { return symbol(CTdsSymbol.ARITHMETICAL_OP,BinOpType.MULTIPLY); }
83  "/"         { return symbol(CTdsSymbol.ARITHMETICAL_OP,BinOpType.DIVIDE); }
84  "%"         { return symbol(CTdsSymbol.ARITHMETICAL_OP,BinOpType.MOD); }
```

donde para cada uno de los operadores aritméticos se retorna el *token* **ARITHMETHICAL_OP**. Esta codificación se puede repetir análogamente para el resto de clases de símbolos del lenguaje (operadores relacionales, operadores lógicos, palabras reservadas, etc). Como para etapas posteriores es conveniente tener identificados cada uno de los símbolos, es decir, saber si el operador que se está utilizando es exactamente '+', '-' o cualquiera de los otros, se optó por tener *tokens* individuales por cada uno de los símbolos:

```
86  /* Arithmetical Operators */
87  "+"         { return symbol(CTdsSymbol.PLUS,BinOpType.PLUS); }
88  "-"         { return symbol(CTdsSymbol.MINUS,BinOpType.MINUS); }
89  "*"         { return symbol(CTdsSymbol.TIMES,BinOpType.MULTIPLY); }
90  "/"         { return symbol(CTdsSymbol.DIVIDE,BinOpType.DIVIDE); }
91  "%"         { return symbol(CTdsSymbol.MOD,BinOpType.MOD); }
```

Y del mismo modo, cada símbolo del lenguaje (operadores relacionales, palabras reservadas, etc) tiene asociado un *token* propio.

La herramienta utilizada para generar el analizador léxico fue JFlex, la cuál a partir de una especificación, como la presentada en el código previo, genera el scanner *CTdsScanner* que es capaz de reconocer los símbolos y retornar el *token* correspondiente para cada uno de ellos. Por ejemplo, para una cadena de símbolos que represente una asignación:

$$x = y + 2;$$

CTdsScanner generaría la cadena de *tokens*:

$$ID\ ASSIGN\_OP\ ID\ PLUS\ INT\_LITERAL\ SEMI\_COLON$$

4.1.1 Testing

Se implementó una serie de tests para el analizador léxico, proporcionando en un archivo secuencias de símbolos válidos para el lenguaje y en otro archivo la secuencia de tokens esperados. Y luego del análisis los tokens retornados debían ser exactamente los mismos que los tokens esperados. Se implementaron tests de acuerdo a la clase de tokens a reconocer, es decir, para *literales*, *operadores*, *keywords*, *delimitadores*, *identificadores*, etc.

4.2 Análisis Sintáctico

A partir de los *tokens* que se obtienen al finalizar el análisis léxico, se realiza el análisis sintáctico, el cuál consiste en reconocer que las sentencias están construidas correctamente de acuerdo a la especificación del lenguaje, es decir, que los símbolos aparecen en el orden correcto.

Para esta etapa es necesario un parser, y la herramienta utilizada para generarlo fue CUP. A partir de la especificación de la gramática, en una forma muy similar a la presentada en la sección 2, CUP genera el parser *CTdsParser* del lenguaje. La salida de este parser es el árbol de parsing, donde cada uno de los nodos se corresponde con algún símbolo de la gramática.

```
189 statement ::= location:l EQ expr:e SEMI_COLON                {: RESULT = new AssignStatement(l,
190     |   location:l PLUS_EQ expr:e SEMI_COLON                  {: RESULT = new AssignStatement(l,
191     |   location:l MINUS_EQ expr:e SEMI_COLON                 {: RESULT = new AssignStatement(l,
192     |   method_call:m SEMI_COLON                              {: RESULT = new MethodCallStatemen
193     |   IF L_PAREN expr:cond R_PAREN block:b                  {: RESULT = new IfStatement(cond,b
194     |   IF L_PAREN expr:cond R_PAREN block:ib ELSE block:eb   {: RESULT = new IfStatement(cond,i
195     |   FOR ID:id EQ expr:init COMMA expr:e block:b           {: RESULT = new ForStatement(id,in
196     |   WHILE expr:cond block:b                               {: RESULT = new WhileStatement(con
197     |   RETURN:r expr:e SEMI_COLON                            {: RESULT = new ReturnStatement(e,
198     |   RETURN:r SEMI_COLON                                   {: RESULT = new ReturnStatement(rl
199     |   BREAK:b SEMI_COLON                                   {: RESULT = new BreakStatement(ble
200     |   CONTINUE:c SEMI_COLON                                 {: RESULT = new ContinueStatement(
201     |   SEMI_COLON:s                                          {: RESULT = new SemicolonStatement
202     |   PRINT L_PAREN expr:e R_PAREN SEMI_COLON               {: RESULT = new PrintStatement(e,e
203     |   block:b                                               {: RESULT = b; :}
204     ;
```

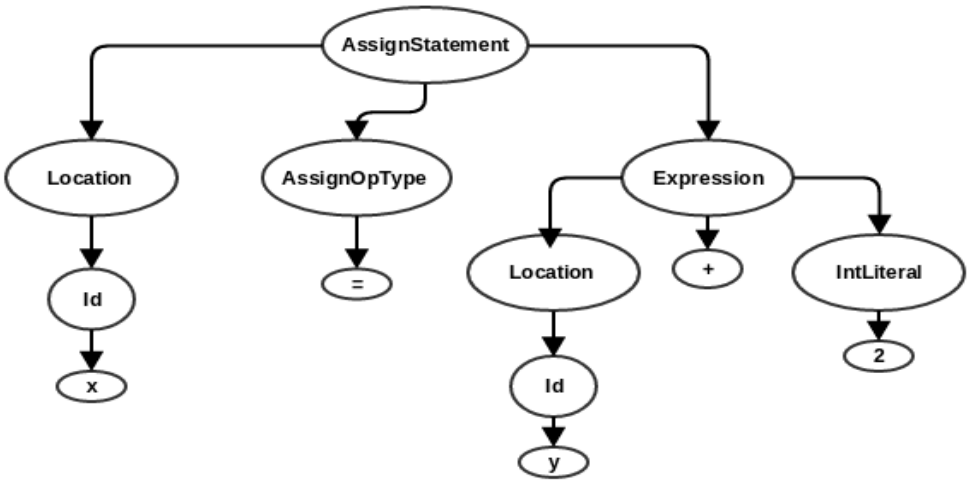
En la figura se observa la parte de la definición de la gramática correspondiende a “statements”. Si bien las tres primeras sentencias, correspondientes a asignaciones, podrían haber sido expresadas en sólo una con un operador genérico, ya que lo único que cambia entre ellas es el operador de asignación, se realizó de esta manera para obtener mayor precisión a la hora de constuir el árbol de parsing. A modo de ejemplo, para la cadena de tokens del ejemplo de la sección 4.1:

ID ASSIGN_OP ID PLUS INT_LITERAL SEMI_COLON

que se corresponde con la primer regla de “statements”:

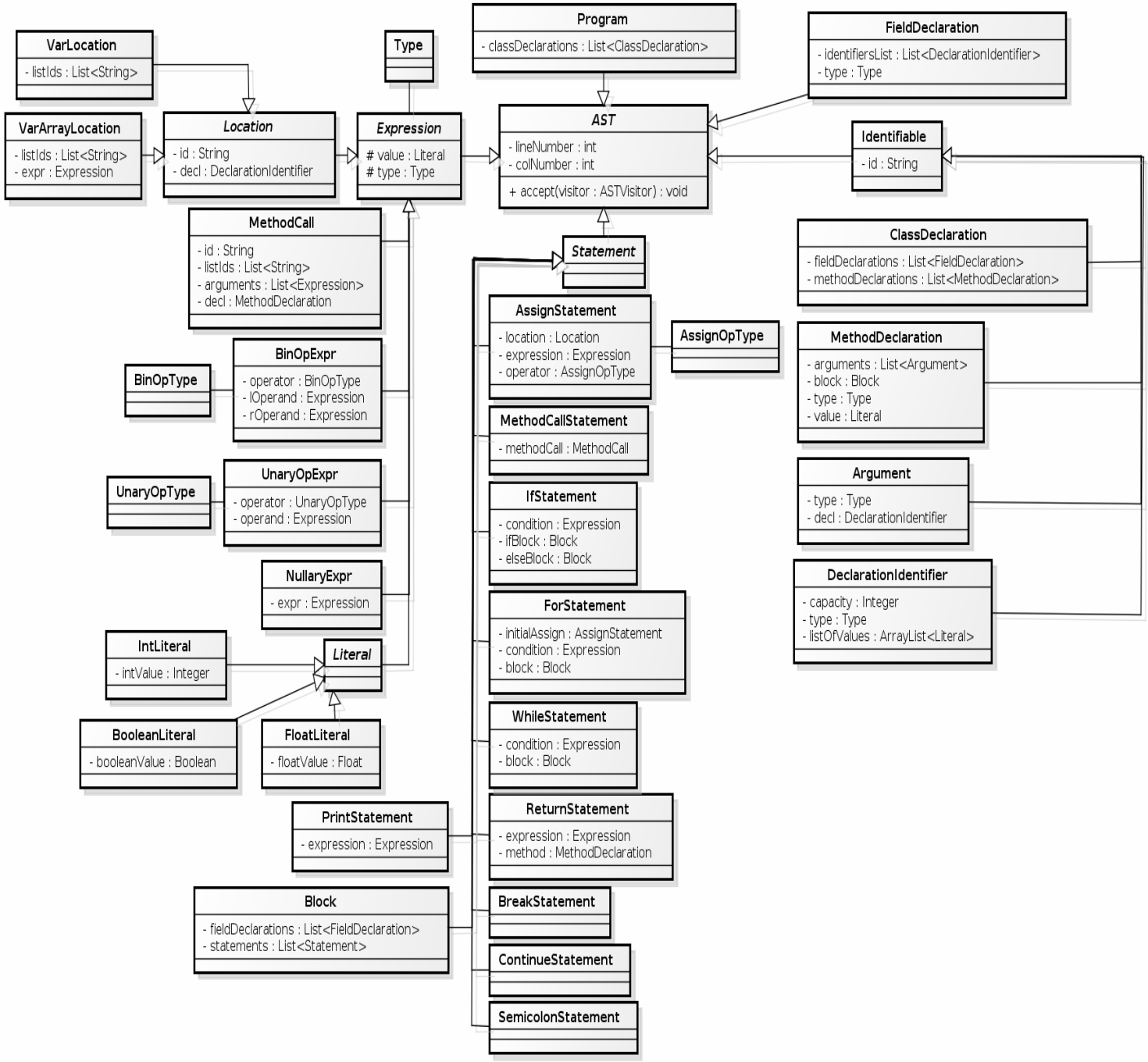
```
location:l EQ expr:e SEMI_COLON
```

en el árbol de parsing tendríamos:



De haber tenido la tres primeras reglas generalizadas en sólo una, se debería haber hecho algún procesamiento extra para determinar de qué operador se trataba. En esta etapa la mayoría de las decisiones tuvieron que ver con esa misma idea, tratar de tener la mayor información posible a la hora de constuir el árbol.

Para la construcción del AST se diseñó la siguiente jerarquía de clases:



Básicamente cuando se genera el AST, la raíz es un elemento *Program*, a partir del cuál se puede comenzar a acceder a objetos *ClassDeclaration*, luego a *FieldDeclaration* y *MethodDeclaration*, a partir de estos últimos a elementos *Argument* y al objeto *Block*, y luego a elementos *Statement*, y así sucesivamente a cada nodo.

4.2.1 Testing

Los tests implementados para comprobar el correcto funcionamiento del parser, fueron realizados a partir de archivos con cadenas de texto, algunas pertenecientes al lenguaje CTds y otras no pertenecientes al lenguaje (con errores sintácticos), y verificando que para las pertenecientes al lenguaje el parser las reconoce correctamente, y para aquellas no pertenecientes se obtienen errores.

4.3 Análisis Semántico

Esta etapa consiste en chequear las reglas semánticas del lenguaje, entre las que se encuentran la compatibilidad de tipos, la correcta declaración y uso de los identificadores, y todas las detalladas en la sección 3. Para verificar estas reglas se implementaron *visitors* del AST generado durante el parsing:

4.3.1 Mains

El visitor *CheckMainVisitor* es el más simple de todos, ya que su objetivo es determinar la cantidad de métodos **main** para comprobar si hay exactamente un método **main** (es lo esperado) o más de uno. Para implementarlo, sólo es necesario mirar cada nodo del árbol que se corresponda con la declaración de un método (*MethodDeclaration*) y determinar si su identificador es **main** o no.

4.3.2 Declaraciones

El visitor *CheckDeclarationVisitor* permite determinar si los identificadores son duplicados o no, si han sido correctamente declarados antes de ser utilizados, etc. La idea es utilizar una tabla de símbolos a modo de pila, donde a medida que se va profundizando un nuevo nivel en el AST (pasamos de un nodo que representa una *ClassDeclaration* a otro que representa una *FieldDeclaration*, o de uno que representa una *MethodDeclaration* a otro que representa un *Statement* de asignación, etc) se va apilando el segundo nodo en la pila. El nivel actual, por ejemplo el de *ClassDeclaration*, es visto como una lista, donde se van agregando los nodos que no representan un nuevo nivel con respecto al actual, es decir nodos *ClassDeclaration*.

De esta manera, para detectar duplicados simplemente es necesario recorrer la lista del nivel actual y comprobar que no existe ningún identificador igual que el identificador del elemento actual. Y para verificar si un identificador ha sido declarado correctamente antes de ser utilizado hay que buscarlo en los niveles anteriores de la pila.

Testing. Para el testing del *CheckDeclarationVisitor* se implementaron tests con programas CTds correctos, para los cuáles el *visitor* no debe encontrar errores, y tests con programas CTds con errores semánticos sobre declaraciones, es decir, con identificadores duplicados, variables no declaradas, métodos sin sentencia **return**, etc.

4.3.3 Tipos

El visitor *CheckTypeVisitor* determina si los tipos han sido definidos correctamente de acuerdo a la semántica:

- si las asignaciones a locaciones respetan el tipo que fue utilizado en la declaración de las locaciones
- si las operaciones son realizadas entre expresiones de tipos equivalentes
- si los operadores en las expresiones son compatibles con los tipos
- si los métodos retornan expresiones del mismo tipo que el de su declaración
- si los parámetros actuales tienen el mismo tipo que los parámetros formales
- si las expresiones de las sentencias condicionales tienen tipo **boolean**

De la misma manera que para las declaraciones, se usa una tabla de símbolos a modo de pila. Para la visita de cada nodo del AST la forma de proceder es la siguiente: se realiza el chequeo de tipos si es necesario, por ejemplo en la condición de un *IfStatement* (el tipo debe ser **boolean**), y si no hay errores se continúa visitando el interior del nodo (*Block* en el caso del *IfStatement*). Si se encuentra algún error, directamente se agrega a la lista de errores de tipo y no se visita el interior del nodo actual. Para los casos en los que no es necesario un chequeo de tipos, por ejemplo una *FieldDeclaration*, se agrega el nodo a la pila (en el caso de declaraciones, ya que sus identificadores pueden ser necesarios), y se visita el nodo siguiente.

Testing. Para el testing del *CheckTypeVisitor* se implementaron tests con programas CTds correctos y tests con programas CTds con errores de tipos, y verificando que en este último caso los errores introducidos son detectados.

4.4 Intérprete

En esta etapa se diseñó e implementó un intérprete de CTds, *InterpreterVisitor*, el cuál a partir del AST generado en el parsing, y luego de comprobar que el programa no tiene errores semánticos (es decir, ejecutando los visitors *CheckMainVisitor*, *CheckDeclarationVisitor* y *CheckTypeVisitor*), comienza desde el método **main** a ejecutar cada una de las sentencias respetando la semántica de cada una:

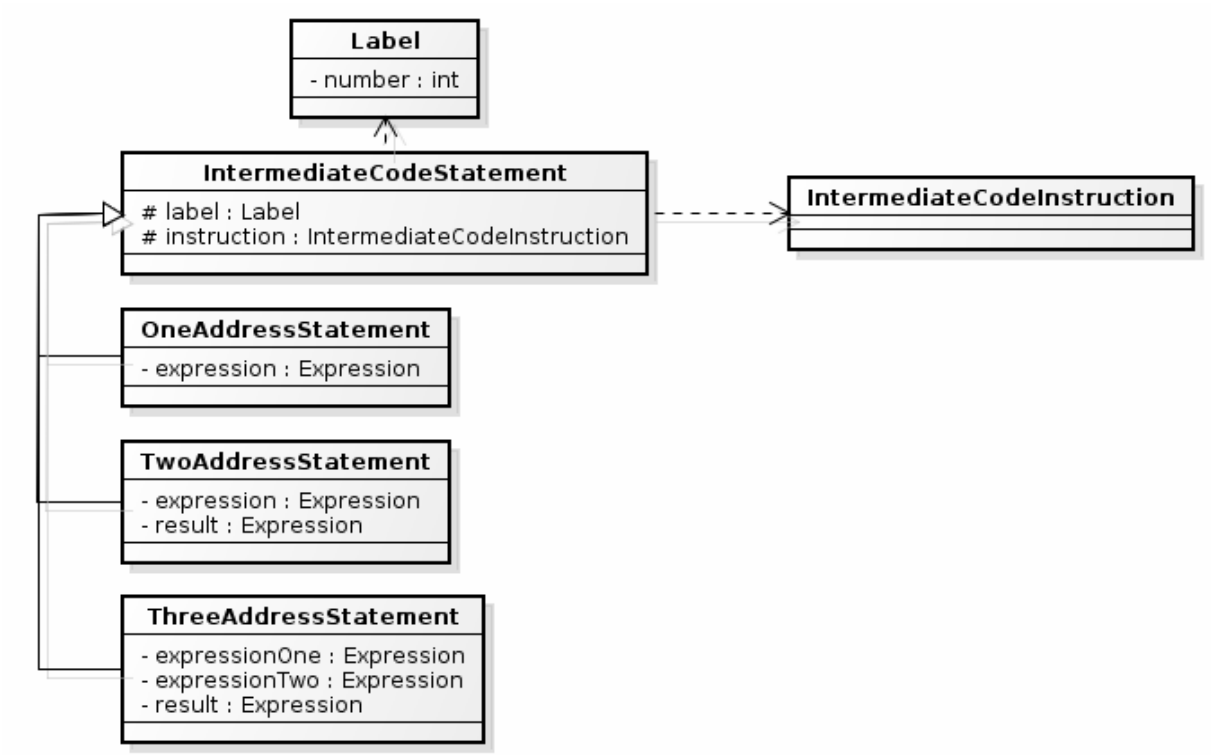
- para las asignaciones, se evalua la expresión y se asigna el valor en la locación.
- para las expresiones unarias, se evalua la expresión y luego se evalua el resultado con el operador
- para las expresiones binarias, se evaluan ambas expresiones y luego se aplica el operador sobre el valor de cada una de ellas
- para las llamadas a métodos se evaluan los argumentos, se asignan los valores a los parámetros formales y se comienza a ejecutar las sentencias del cuerpo del método invocado
- para las sentencias **if** se evalua la expresión booleana, si el resultado es **true** se ejecutan las sentencias del bloque del **if**, si es **false** se ejecutan las sentencias del bloque del **else**.
- para las sentencias **while** mientras la evaluación de la expresión sea **true** se ejecutan las sentencias del cuerpo del ciclo.

Se omiten algunas sentencias ya que su ejecución es similar a alguna de las descriptas.

4.5 Generación de Código Intermedio

En la generación de código intermedio se implementó el *IntermediateCodeGeneratorVisitor* con el objetivo de generar una representación intermedia de bajo nivel del código CTds, a partir de la cuál será posible generar el código objeto del programa.

Para esta etapa se diseñó una jerarquía de clases para representar las sentencias del código intermedio:



La idea es, a partir del AST del programa, generar una lista de *IntermediateCodeStatement*, donde cada elemento de la lista puede ser:

- *IntermediateCodeStatement*: por ejemplo para la instrucción *RET*, que en ocasiones no requiere expresiones.
- *OneAddressStatement*: para las instrucciones *JUMPF* (Salto por falso) y *JUMP* (Salto incondicional) que solo requieren un label al cuál saltar. O la instrucción *PUSH* que necesita la expresión que se desea agregar a la pila.
- *TwoAddressStatement*: para la instrucción *ASSIGN* que necesita dos expresiones, una conteniendo la locación y la otra conteniendo la expresión que se va a asignar. O también la instrucción *CALL*, que requiere la expresión de la llamada al método y la locación en donde almacenar el resultado.

- *ThreeAdressStatement*: para las instrucciones *ADDI* (suma de enteros) y *ADDF* (suma de flotantes), que utilizan las dos expresiones a sumar y además la locación en donde se va a almacenar el resultado.

Todas las instrucciones mencionadas corresponden a objetos del tipo *IntermediateCodeInstruction*.

También se utilizaron dos pilas, una para ir apilando los labels de inicio de los nodos y otra para los labels de fin. Esto es útil para algunos nodos, por ejemplo los que representan **for** o **while**, ya que para algunas sentencias, como **break** o **continue**, es necesario saber el label de inicio o fin.

4.6 Generación de Código Objeto

Para generar el código assembler a partir del código intermedio se implementó *AssemblerCodeGenerator*. El cuál lo que hace es ir recorriendo cada elemento de la lista de *IntermediateCodeStatement* creada en la etapa anterior y generando las instrucciones assembler necesarias para representar cada *IntermediateCodeInstruction*. A continuación se muestran algunos ejemplos del código assembler generado para distintas sentencias:

4.6.1 Asignaciones

- CTds:

  $x = 2;$

- Código intermedio:

  $ASSIGN \;\; 2 \;\; x$

- Assembler:

  $movl \;\; \$2, \;\; \%ebx$
  $movl \;\; \%ebx, \;\; -4(\%ebp)$

  Donde $-4(\%ebp)$ es la dirección de memoria reservada para x.

4.6.2 Operación aritmética con enteros

- CTds:

  $y = x + 2;$

- Código intermedio:

  $ADDI \;\; x \;\; 2 \;\; t0$
  $ASSIGN \;\; t0 \;\; y$

  Donde $t0$ es una variable temporal.

- Assembler:

  $movl \;\; -4(\%ebp), \;\; \%ebx$
  $addl \;\; \$2, \;\; \%ebx$
  $movl \;\; \%ebx, \;\; -12(\%ebp)$
  $movl \;\; -12(\%ebp), \;\; \%ebx$
  $movl \;\; \%ebx, \;\; -8(\%ebp)$

  Donde $-4(\%ebp)$, $-8(\%ebp)$ y $-12(\%ebp)$ son las direcciones de memoria de x, y y $t0$ respectivamente. Notar que para el resto de las operaciones aritméticas sólo cambia la instrucción *addl* por la correspondiente.

4.6.3 Operación aritmética con flotantes

- CTds:

  $y = x + 2.0;$

- Código intermedio:

  $ADDF \;\; x \;\; 2.0 \;\; t0$
  $ASSIGN \;\; t0 \;\; y$

  Donde $t0$ es una variable temporal.

- Assembler:

  $flds \;\; -4(\%ebp)$
  $fadds \;\; .label2$
  $fstps -12(\%ebp)$
  $movl \;\; -12(\%ebp), \;\; \%ebx$
  $movl \;\; \%ebx, \;\; -8(\%ebp)$

  Donde $-4(\%ebp)$, $-8(\%ebp)$ y $-12(\%ebp)$ son las direcciones de memoria de x, y y $t0$ respectivamente, y $.label2$ es el label para la constante 2.0. Notar que para el resto de las operaciones aritméticas sólo cambia la instrucción $fadds$ por la correspondiente.

4.6.4 Declaraciones de métodos

- CTds:

  int sum(**int** a, **int** b){
  return $a + b$;
  }

- Código intermedio:

  $INITML \;\; sum$
  $RESERVE \;\; amount$
  $ADDI \;\; a \;\; b \;\; t0$
  $RET \;\; t0$

  $INITML$ es la instrucción que representa el inicio de la declaración de un método, cuyo nombre en este caso es sum y $RESERVE$ es la instrucción utilizada para reservar en la pila tanto espacio como el valor de $amount$.

- Assembler:

  $.globl \;\; sum$
  $.type \;\; sum, \;\; @function$
  sum :
  $pushl \;\; \%ebp$
  $movl \;\; \%esp, \;\; \%ebp$
  $movl \;\; 8(\%ebp), \;\; \%ebx$
  $addl \;\; 12(\%ebp), \;\; \%ebx$
  $movl \;\; \%ebx, \;\; -4(\%ebp)$
  $movl \;\; -4(\%ebp), \;\; \%eax$
  $leave$
  ret

  Donde $8(\%ebp)$ y $12(\%ebp)$ son las direcciones de memoria de los parámetros a y b respectivamente, y $-4(\%ebp)$ es la dirección de memoria $t0$. Los resultados de los métodos siempre son almacenados en el registro eax.

4.6.5 Invocaciones a métodos

- CTds:

  $x = sum(x, y);$

- Código intermedio:

  $PUSH\ \ y$
  $PUSH\ \ x$
  $CALL\ \ sum\ \ t1$
  $ASSIGN\ \ t1\ \ x$

  Donde $t1$ es una variable temporal.

- Assembler:

  $pushl\ \ -8(\%ebp)$
  $pushl\ \ -4(\%ebp)$
  $call\ \ sum$
  $movl\ \ \%eax,\ \ -12(\%ebp)$
  $movl\ \ -12(\%ebp),\ \ \%ebx$
  $movl\ \ \%ebx,\ \ -4(\%ebp)$

  Donde $-4(\%ebp)$ $-8(\%ebp)$ y $-12(\%ebp)$ son las direcciones de memoria de x, y y $t0$ respectivamente. Mediante *call sum* se comienzan a ejecutar las instrucciones del cuerpo del método *sum* descriptas anteriormente.

Compilación. Ya que el código generado por el *AssemblerCodeGenerator* es el mismo que el generado por el lenguaje C, el *GNU Assembler*, puede ser compilado utilizando *gcc*. De esta manera también se puede linkear fácilmente código escrito en otros lenguajes.

4.7 Análisis y Optimización

El objetivo de esta etapa es analizar el código para aplicar distintas posibles optimizaciones tratando de reducir los tiempos de ejecución. Algunas de las posibles optimizaciones son:

- Resolver operaciones entre expresiones constantes. Una vez realizados los chequeos semánticos necesarios, aquellas operaciones que involucren solo constantes se pueden calcular y sólo dejar el valor total de la expresión. De esta manera se pueden reducir la cantidad de cálculos en etapas posteriores.

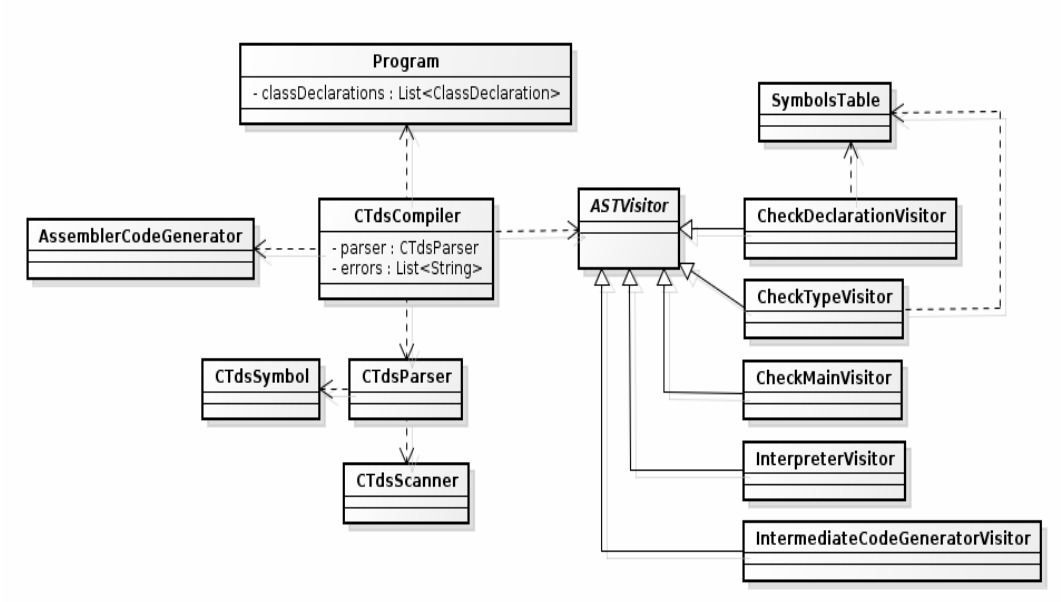
- Reducir la cantidad de instrucciones *movl* generadas. En muchas ocaciones, al utilizar variables temporales como en el ejemplo **Operación aritmética con enteros** de la sección anterior, se generan instrucciones *movl* innecesarias.

- No generar código de bajo nivel para sentencias que no se ejecutan a través de ninguna traza. Por ejemplo, en una sentencia **if** con una condición que siempre evalua a **true**, las sentencias del cuerpo del **else** nunca se ejecutarán, por lo tanto no es necesario generar código para esas sentencias.

Si bien esta etapa no se llevó a cabo, es necesario notar que hay que ser precisos en el análisis y las optimizaciones que se deseen efectuar, ya que realizar un excesivo análisis del código puede llevar a elevar los tiempos de compilación, y también no ser precisos en la optimización, más aún cuando se trata de código de bajo nivel, puede llevar a generar programas que dejen de respetar la semántica del lenguaje.

5 Resultado final

Después de llevar a cabo cada una de las etapas del desarrollo del compilador *CTdsCompiler*, se concluyó con la siguiente estructura:



donde el orden de ejecución del compilador es básicamente el mismo de las etapas, es decir:

1. Análisis Léxico: *CTdsScanner*
2. Análisis Sintáctico: *CTdsParser*
3. Análisis Semántico: *CheckDeclarationVisitor*, *CheckTypeVisitor* y *CheckMainVisitor*
4. Generación de Código Intermedio: *IntermediateCodeGeneratorVisitor*
5. Generación de Código Objeto: *AssemblerCodeGenerator*

y sólo se va continuando a la etapa siguiente si no se obtuvo ningún error en la etapa anterior.