INDICE

Introducción

C-TDS es un lenguaje imperativo tipado básico con sintaxis similar al Lenguaje C. Proporciona 3 tipos básicos Int,Float y Boolean y arreglos de cada uno de los tipos antes mencionados. Presenta un conjunto de 3 tipos de asignaciones (=,+=,-=) 1 para operar con valores de cualquiera de los tipos básico y arreglos de ellos y 2 para valores numéricos (Int y Float) y arreglos de ellos. Proporciona la clásica sentencia condicional If-then-else y 2 construcciones para iterar 1 definida (for) y otra indefinida (while). También incorpora las sentencias Break y Continue.
Además, C-TDS permite entrelazar con métodos de la librería básica de C, medio por el cuál permite operaciones de entrada-salida. En las invocaciones a a rutinas externas permite la utilización de elementos del tipo String. Proporciona 2 construcciones básicas para la modularización de los programas: Funciones y Acciones, permitiendo invocaciones recursivas.
C-TDS es un Lenguaje seguro ya que una implementación del mismo, debe chequear en tiempo de ejecución que el acceso a una posición de un arreglo sea válida.
C-TDS proporciona una conjunto básico de operadores, 4 operadores aritméticos (+,-,*,/,%), 4 operadores relacionales (<,>,<=,>=), 2 operadores de equivalencia (==,!=) y 2 operadores lógicos (&&,||).
Otra de las características del lenguaje, es la evaluación lazzy de sus operadores lógicos (&&,||).
En el presente informe se abordará la creación de un Compilador para el Lenguaje C-TDS, el compilador: C-TDS-PCR.

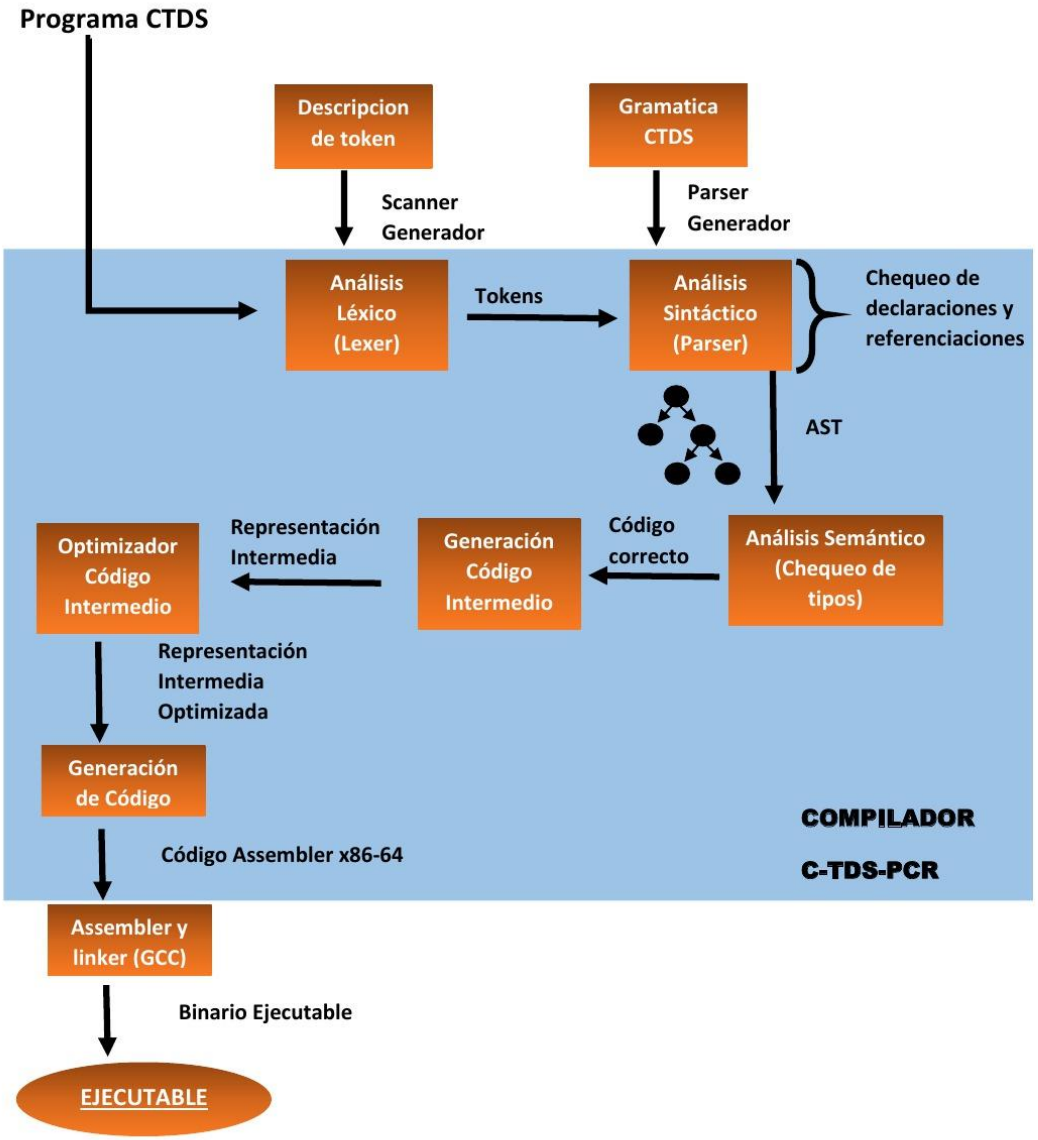
CTDS-PCR un compilador para el Lenguaje C-TDS

C-TDS-PCR implementa todas las características del lenguaje Ctds para computadores con Arquitectura de 64-bits. Además, Ctds-PCR permite las operaciones aritméticas y relacionales entre expresiones de tipo Int y Float mediante la implementación de coerciones implícitas.
Ctds-PCR implementa un conjunto de optimizaciones entre las que se destacan la propagación de constantes, poda de código inalcanzable y optimizaciones en frame que restringen la vida de una variable al tiempo en el que pueden ser referenciadas.
A continuación, se detallará acerca de la arquitectura del compilador, se describirán cada una de las etapas del desarrollo enunciando los problemas más significativos de cada etapa y sus respectivas soluciones. Además se intentará que el documento en sí mismo, sea una bitácora con todas aquellas decisiones que hayan resultado positivas y/o negativas y se intentará dejar en claro que decisiones hubieren sido más convenientes desde nuestra óptica como desarrolladores.

Desarrollo

Consideraciones Básicas

El compilador Ctds-PCR fue desarrollado íntegramente en el Lenguaje de Programación Java, utilizando herramientas como Jflex y Cup para la generación del Lexer y Parser respectivamente. En grandes rasgos el proceso de compilación que realiza Ctds-PCR puede ser visualizado en el siguiente diagrama:



El desarrollo del mismo se distribuyó en 6 etapas:

- Etapa 1: Análisis Léxico y Sintáctico
- Etapa 2: Análisis Semántico
- Etapa 3 :Generación de Bodigo Intermedio
- Etapa 4 :Generación de Código Objeto Enteros y Booleanos
- Etapa 5:Generación de Código Objeto Reales
- Etapa 6 :Optimizaciones

Es importante mencionar, que en varias etapas del proyecto fue necesario revisar o modificar la implementación de las etapas anteriores.

Etapa 1- Análisis Léxico y Sintáctico

En esta primera etapa del proyecto, comenzamos a familiarizarnos con las herramientas Jflex y Cup. La primera fue utilizada para desarrollar el Lexer del compilador C-TDS-PCR y la segunda para la implementación del Parser.
Las principales dificultades surgieron debido al desconocimiento a la hora de utilizar estas herramientas, pero en general fueron lo suficiente amigables como para poder trabajar con ellas en poco tiempo.
A la hora implementar el Parser se presentaron inconvenientes con la gramática, para salvarlos se re factorizó la gramática y se definieron precedencias entre los operadores.
Nos surgieron inconvenientes también con los comentarios multilineas, que fueron solucionados a nivel del análisis léxico mediante la utilización de construcciones de Jflex que permiten la creación de un nuevo estado en el AFD, generado por dicha herramienta. El funcionamiento de esta característica se reduce a que el AFD tenga una transición al nuevo estado cada vez que aparece / seguido de * y en dicho nuevo estado consume todos los caracteres hasta que aparece * seguido de /.
En esta etapa aún no se habían realizado decisiones de diseño de gran importancia sino que se generaron el analizador léxico(lexer) y el analizador sintáctico(parser) de manera de que puedan trabajar mancomunadamente.

Etapa 2- Análisis Semántico

En esta etapa se implementaron chequeos para la serie de restricciones semánticas enunciadas en la descripción del lenguaje. Más adelante enunciaremos todos los chequeos requeridos y como se ha implementado cada uno de ellos. Ahora es importante detallar que para la realización de los diferentes chequeos fue necesaria la creación de 2 estructuras: Tabla de Símbolos y Arbolo de Sintaxis Abstracta.

Tabla de Símbolos

La primera, es una estructura utilizada para llevar la información de los ambientes de las declaraciones y resolver su alcance(sope). La construcción que escogimos es sencilla; básicamente la tabla de símbolos del compilador Ctds-PCR es una pila de listas de declaraciones. Para su implementación fueron generadas dos clases: Simbolizable y Level que proporcionan un set de instrucciones que permiten manejar los alcances de las declaraciones. Entonces, la tabla de símbolos es una pila de levels, donde los levels son listas de Location(VarLocation,ArrayLocation, MethodLocation).

Durante el parseo del programa, en primera instancia se hace un push del nivel global y se le adhieren todas las declaraciones de variables globales, luego se le añaden los métodos uno a uno. Cuándo se parsean las declaraciones de los métodos se hace un push del nivel con alcance local al método, se adhieren a dicho nivel las declaraciones de variables locales al método y luego cada vez que se abre un bloque se hace un push y se adhieren las declaraciones de variables locales al bloque. Cada vez que se cierra un bloque se hace un pop, y cada vez que se finaliza de parsear un método otro pop, y así hasta finalizar el parseo del programa donde se hace un pop del nivel global.
El comportamiento mencionado, permite chequear que cuándo se referencia una variable la misma este definida y además permite escoger correctamente la variable referenciada. Es decir si se define una variable A global y una variable A local al método m, dentro de m se deberá referenciar al A local pero dentro del método m2 que no redefine A, una referencia a A debería resolverse como una referencia a la A global. Además permite chequear que no se redefinan declaraciones con el mismo nombre.

Árbol de Sintaxis Abstracta:

El Árbol de Sintaxis Abstracta, más conocido por sus siglas en ingles AST(Abstract Sintax Tree),es una estructura que nos acompañara durante el resto del desarrollo del compilador y sobre esta se realizarán gran parte de los chequeos semánticos. El mismo, se construye durante el proceso de Análisis Léxico y Sintáctico.
En este punto, nuestra decisión fue recabar la mayor cantidad de información posible y ordenarla de manera tal de facilitar todos los procesos siguientes que trabajaran directamente con el árbol o indirectamente, es decir sobre alguna representación generada a partir de este.
Es por ello, que se definió la siguiente jerarquía de clases:

Ver jerarquía de clases, en la carpeta Documentación

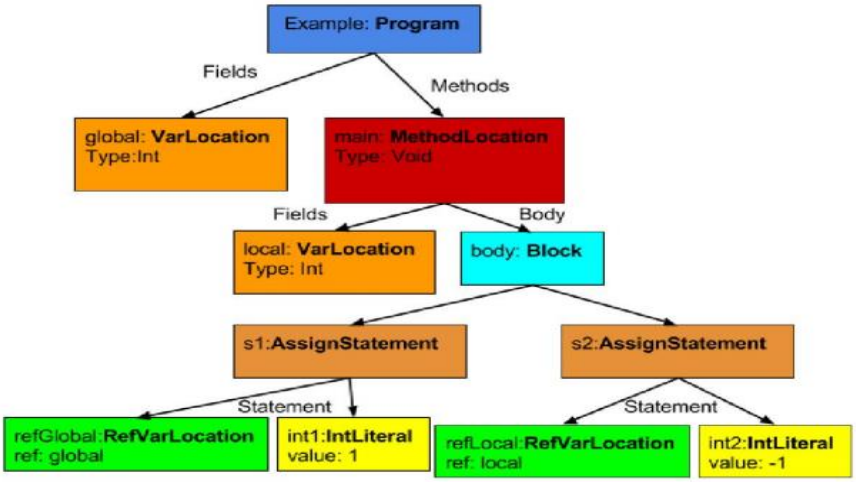
A modo de ser más explicativos, el siguiente programa Ctds:

```
class Example{

Int global;

        void main(){
        Int local;
        global=1;
        local=-1;
        }
}
```

Sería representado con el siguiente AST:



En general, creemos que el diseño de la jerarquía de clases y el armado del AST resulto ser una buena decisión ya que disponemos de toda la información del programa a compilar en una representación donde fue relativamente fácil aplicar los distintos tratamientos requeridos, siguiendo la estructura del patrón Visitor.
El diseño del AST nos permitió resolver varios chequeos semánticos, la generación de código intermedio y la realización de optimizaciones sin mayores inconvenientes. En general los problemas ocurridos fueron resueltos adhiriendo más información al AST(en forma de atributos de las diferentes clases) o re factorizando la jerarquía de clases como por ejemplo en el siguiente caso:

En un principio, no se habían distinguido entre la definición de una variable y el uso o referenciación a ella, lo cuál traía aparejado inconvenientes, sobretodo cuándo se quería referenciar arreglos. Por lo que se re factorizó la jerarquía de clases definiendo la clase abstracta RefLocation y sus derivados (RefVarLocation y RefArrayLocation.)

Ahora, ya explicados los justificativos para la creación de la Tabla de Símbolos y el AST, ahora explicaremos como han sido resueltos cada uno de los siguientes chequeos semánticos:

1. Ningún identificador es declarado dos veces en un mismo bloque.

2. Ningún identificador es usado antes de ser declarado.

3. Todo programa (clase) contiene la definición de un método llamado main. Este método no tiene parámetros. Notar que la ejecución comienza con el método main.

4. El Int Literal en la declaración de un arreglo debe ser mayor a cero (es la longitud del arreglo).

5. El número y tipos de los argumentos en una invocación a un método debe ser iguales al número y tipos declarados en la definición del método (los parámetros formales y los reales deben ser iguales).

6. Si la invocación a un método es usada como una expresión, el método debe retornar un resultado.

7. Los literales String solo pueden ser usados como argumentos en métodos externinvk.

8. Una sentencia return solo tiene asociada una expresión si el método retorna un valor, si el método no retorna un valor (es un método void) entonces la sentencia return no puede tener asociada

ninguna expresión.

9. La expresión en una sentencia return debe ser igual al tipo de retorno declarado para el método.

10. Un <id> usado como una <location> debe estar declarado como un parámetro o como una variable local o global.

11. En toda locación de la forma <id>[<expr>]
(a) <id> debe ser una variable arreglo (array), y
(b) el tipo de <expr> debe ser int.

12. La <expr> en una sentencia if o while debe ser boolean.

13. Los operandos de <arith_op>’s y <rel_op>'s deben ser de tipo int o float.

14. Los operandos de <eq_op>’s deben tener el mismo tipo (int, float o boolean).

15. Los operandos de <cond_op>’s y el operando de la negación (!) deben ser de tipo boolean.

16. La <location> y la <expr> en una asignación, <location> = <expr>, deben tener el mismo tipo.

17. La <location> y la <expr> en una asignación incremental o decremental, <location> += <expr> o <location> -= <expr>, deben ser de tipo int o float.

18. Las expresiones <expr> iniciales y finales de un for deben ser de tipo int.

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

Explicación de las soluciones implementadas:

1, 2,10: Son chequeadas en el analizador sintáctico mediante la implementación de la tabla de símbolos, tal como se mencionó anteriormente.

3: Este chequeo también es realizado en el Analizador Sintáctico, esta decisión se tomó desde el comienzo y se mantuvo hasta el final de la implementación del compilador Ctds-PCR. Quizás resulte en un diseño más modular realizar este chequeo más adelante.

4: Este chequeo es realizado en el analizador sintáctico, durante el parseo de la definición de un arreglo.

7:Es resuelto por la definición de la gramática. En el único lugar que se esperan cadenas de caracteres es en el cuerpo de un externinvk.

5,6,8,9,12-18 :Para dichos chequeos se implementó la clase TypeCheckVisitor que implementa una un visitador para chequear los tipos. Vale mencionar que se tomó una importante decisión en cuanto a las operaciones permitidas: se permite que los operadores binarios que operan con Int o Float hayan sido extendidos y permiten operar con operandos de uno u otro tipo indistintamente. Vale hacer una salvedad, en el caso de la división, si opera con dos expresiones de tipo int, tiene la semántica de la división entera, mientras que si al menos uno de sus operadores es de tipo float tiene la semántica de la división en flotantes. El operador mod (%) solo permite operar con valores de tipo int.
Los operandos de tipo relacionales y de equivalencia operan con valores enteros y flotantes

indistintamente.
La decisión de permitir operar con elementos numéricos pero de tipos distintos tiene que ver con la posibilidad de ofrecer una característica que genere más flexibilidad a la hora de programar en C-TDS. Quizás hubiere sido interesante implementar warnings en estos casos, esto fue planificado pero no se llevó a cabo.

11: La parte a) es chequeada durante el análisis sintáctico a través de la tabla de símbolos. La parte b) es chequeada por la clase TypeCheckVisitor.

19: Se implementó la clase BreakContinueCheckVisitor que es otra instancia del patrón visitor que recorre el AST y chequea lo expresado en el ítem 19.

Etapa 3 - Generación de Código Intermedio

Esta etapa del compilador se retorna una representación intermedia del código, luego, a partir de esta representación intermedia se generará código assembler para la plataforma x86-64. La motivación que resulto en la elección de la generación de código para esta arquitectura fue la intención de indagar en la programación a bajo nivel de la arquitectura mencionada.
En concordancia con lo antes mencionado, y con el interés de resolver gran parte de la generación de código assembler en esta etapa; se diseñó un código de 3 direcciones(con a lo sumo 3 operandos) con 49 instrucciones. Como características principales de este código podemos mencionar:

*Se generó una instrucción por operador aritmético, lógico, relacional y de igualdad. Además se diferencian las instrucciones que operan con valores de tipo Int con las que operan con valores de tipo Float en operadores aritméticos y relacionales.

*También se implementaron instrucciones para generar las coerciones para permitir operaciones aritméticas y relacionales entre operadores de tipo Float e Int.

*Como operando las instrucciones reciben Expresiones. Recordemos que en la jerarquía de clases del AST las declaraciones de variables y procedimientos (Locations) son expresiones.

A continuación enumeraremos las 49 instrucciones que se utilizaron para representar la semántica de ejecución de cada construcción del lenguaje:

ProgramDecl: Instrucción para definir un programa. *(Sintaxis: ProgramDecl Label)*

BlockBegin: Instrucción para indicar el comienzo de la declaración de un bloque, el operando representa la cantidad de bytes a reservar al inicio del bloque. *(Sintaxis: BlockBegin IntLiteral)*

BlockEnd: Instrucción para indicar el fin de la declaración de un bloque, el operando representa la cantidad de bytes a devolver al finalizar la ejecución del bloque. *(Sintaxis: BlockEnd IntLiteral)*

MethodDecl: Instrucción para definir la declaración de un método y encargarse de generar el ambiente necesario para su ejecución(prologo). *(Sintaxis: MethodDecl MethodLocation)*

MethodDeclEnd: Instrucción de finalización de un método cuya semántica es la realización de operaciones que deban de ejecutarse al final del método (*epilogo*). *(Sintaxis: MethodDeclEnd MethodLocation)*

LocationDecl: Instrucción para la declaración de una variable. *(Sintaxis:LocationDecl Location)*

Assign: Instrucción que representa una asignación. *(Sintaxis: Assign expr RefLocation)*

AddF, AddI, SubF, SubI: Instrucciones con la semántica de las operaciones aritméticas de suma(Add) y resta (Sub), distinguiendo reales(..F) de enteros(..I). *(Sintaxis: AddI|...|SubI Expr, Expr, RefLocation)*

MinusI, MinusF: Instrucciones que representa el opuesto de un valor, MinusI caso de enteros y MinusF caso de reales. *(Sintaxis: MinusF|...|MinusI Expr, RefLocation)* Vale mencionar, que el cálculo de opuestos de literales(introducidos directamente por el usuario) es realizado durante el parseo.

MultI, MultF,DivI,DivF, Mod: Instrucciones de multiplicación y división, la letra F distingue el caso de reales, mientras la letra I el caso de enteros. En el caso de DivI se realiza la división entera entre los valores y la instrucción mod solo se utiliza para enteros. *(Sintaxis: MultI|...|Mod Expr, Expr, RefLocation)*

And, Or: Instrucciones para las operaciones lógicas binarias. *(Sintaxis: And|Or Expr, Expr, Location)*

Not: Instrucción para la operación lógica unaria de la negación. *(Sintaxis: Not Expr, RefLocation)*

EqualI, DifI, EqualF, DifF, GEI, GEF, LEI, LEF, GrtI, GrtF, LesI, LesF: Instrucciones para las operaciones relacionales, igual(Equal), distinto(Dif), mayor o igual(GE), menor o igual(LE), mayor(Grt), menor(Les). A cada instrucción se le agrega al final una letra para distinguir el caso de reales(F) y el caso de enteros(I). *(Sintaxis: Equal|Dif| GE|LE|Grt|Les Expr, Location)*

Jmp, JTrue, JFalse: Instrucciones de salto, donde se distingue el salto obligatorio(Jmp), el salto condicional por valor verdadero(JTrue) y el salto condicional por valor falso(JFalse). *(Sintaxis: JTrue|JFalse Expresión Label) y (Sintaxis: Jmp Label).*

Call,CallWithReturn: Instrucciones para realizar llamados a métodos, donde se distinguen los casos *void*(Call) de los casos con retorno (CallWhitReturn). *(Sintaxis: Call Label) y (Sintaxis: CallWithReturn label, RefLocation). El argumento de tipo RefLocation indica el lugar de destino a donde será almacenado el valor de la función, posterior al retorno.*

ParamPush, ParamPop: Instrucciones para realizar en las invocaciones a métodos. Se distingue el push de los mismos (ParamPush) del pop de los parámetros (ParamPop). *(Sintaxis: ParamPush Expression Location) y (Sintaxis: ParamPop IntLiteral). Donde Location en ParamPush representa el destino donde será pusheado el valor. IntLiteral en ParamPop significa la cantidad de bytes a recuperar luego de la invocación a un método.*

CallExtern,CallExternWithReturn: Instrucciones para realizar llamados a métodos externos, se distingue el caso *void*(CallExtern) de los casos con valores de retornos (CallExternWithReturn). *(Sintaxis: CallExtern Label) y (Sintaxis: CallExternWithReturn label, RefLocation).El argumento de tipo RefLocation indica el lugar de destino a donde será almacenado el valor de la función, posterior al retorno.*

Ret: Instrucción para generar el retorno de una función. *(Sintaxis: Ret expression).*

ToFloat: Instruccion de conversion, la misma convierte un valor de entero(int) a real(float). (*Sintaxis: ToFloat Expresion Location)*

ReadArray, WriteArray: Instrucciones definidas sobre arreglos, las mismas permiten leer de un arreglo(ReadArray) y escribir en un arreglo(WriteArray). *(Sintaxis: ReadArray ArrayLocation dir Location) y (Sintaxis: WriteArray Expr dir ArrayLocation).*

PutLabel: Instrucción para la generación de un label. *(Sintaxis: PutLabel Label)*

PutStringLiteral: Intruccion para definir un literal de tipo String o Float*(Sintaxis: PutStringLiteral StringLiteral)*

Para la generación de las instrucciones se decidió volver a utilizar el patrón Visitor, visitando el AST y generando el conjunto de instrucciones del código anteriormente descripto que representen la semántica de ejecución del nodo. A posteriori, cada instrucción del código de 3 direcciones será traducida a lenguaje ensamblador.
Durante la etapa de generación del TAC(Tree Address Code) se crean un conjunto de variables auxiliares que permiten almacenar valores de los cálculos intermedios y que son necesarias para la traducción al lenguaje ensamblador.
Es importante mencionar que todos los valores son almacenados en 4 bytes, por tanto todas las operaciones son realizadas con dicho tamaño.
Es válido también aclarar que el conjunto de instrucciones mencionado es el conjunto final al que se arribó y que no todas fueron incorporadas desde un principio, sino que algunas fueron necesarias para implementar las características ciertas características del lenguaje.
En general, se trataron de respetar las convenciones del lenguaje C, desde la generación de código intermedio.
También es necesario mencionar que durante esta etapa quedan completamente definidos las direcciones relativas de cada una de las variables locales que serán almacenadas en el Stack, se definen respecto a la distancia del base pointer(rbp). La asignación de offset comienza a partir del análisis sintáctico. Finalizada dicha etapa todas las variables definidas por el programador tienen una dirección relativa.
Como inconvenientes, podemos mencionar que desde el principio se generó un TAC para la generación de assembler x86-64, lo cual resulta ser una elección pésima de diseño si en el futuro se tuviere la intención de generar assembler para otras arquitecturas, como por ejemplo en 32 bits. A modo de enfatizar lo mencionado, en el código de 3 direcciones se definen los destinos en donde serán almacenados los parámetros en la invocación de un método y los lugares donde se esperaran los resultados (en caso de que el método sea una función).

Etapa 4 - Generación de Código Objeto Enteros y Booleanos

En esta etapa se generaron las traducciones a lenguaje ensamblador del subconjunto del TAC que operan con operandos enteros y booleanos. Como los problemas más grandes habían sido resueltos en la generación del código de tres direcciones no resulto demasiado complejo la traducción a lenguaje ensamblador. Los principales inconvenientes surgieron por el desconocimiento del lenguaje ensamblador x86-64.

Etapa 5- Generación de Código Objeto Reales

En esta etapa se generaron las traducciones a lenguaje ensamblador del subconjunto del TAC que operan con operandos de tipo flotante.

Se necesitó realizar un tratamiento especial a la hora de crear literales de tipo Float, debido a que en assembler es necesario definirlos como un label.
Siguiendo con nuestra premisa de respetar las convenciones del lenguaje C, cuando se realizan invocaciones a métodos con parámetros de tipo float, los primeros 8 se apilan en los registros(xmm0,...,xmm7). Si hubiere más de 8 parámetros flotantes, son pasados a través de la pila. El resultado de la invocación a un método que retorna un valor de tipo Float, es almacenado en xmm0.
Nuevamente los principales inconvenientes se reportaron debido al desconocimiento de la

programación en assembler x86-64. Además de ello, tuvimos mayores problemas debido a que el lenguaje C maneja la pila de invocaciones con alineamiento en 16 bytes y al apilar siempre de a 4 bytes desalineábamos la pila. Es por ello que fue necesario reservar espacios en la pila múltiplos de 16.

Etapa 6 - Optimizaciones

Se realizaron 3 optimizaciones: Optimización del Frame, Propagación de Constantes y Poda de código muerto o inalcanzable.

Optimización del Frame:

En principio, el compilador reserva espacios para todas las variables locales a un método (las definidas por el usuario y las generadas por el TAC) y ese espacio permanece reservado durante toda la ejecución del método que las contiene. En esta optimización se reasignaron offset para restringir el tiempo de vida de las variables solo en al espacio temporal en el que pueden ser referenciadas.
Para ello, fue necesario que cada bloque tenga asociado el conjunto de todas las variables que tienen vida en él. Esta información es recabada durante la generación del código intermedio. A posteriori se definió una nueva instancia del patrón visitor que se encarga de reasignar los offset de las variables. Este visitador se encarga también de setearle un valor de offset al bloque, dicho valor representa los bytes que deberá reservar para el conjunto de todas las variables locales a él.
Además fue necesario agregar al TAC las instrucciones BlockBegin y BlockEnd para delimitar el comienzo y final del bloque.
Veamos el funcionamiento de la optimización con el siguiente programa C-TDS:

```
class testFrameOptimization{
        void main(){
        int a,b,c,d;
        {
                int e,f,g,h;
                {
                        int i,j,k,l;
                }
        }
        }
}
```

Código sin optimización de Frame

```
.file "testFrameOptimization"
.text
.globl main
main:
enter $48,$0
movl $0, -36(%rbp)
movl $0, -40(%rbp)
movl $0, -44(%rbp)
movl $0, -48(%rbp)
movl $0, -20(%rbp)
movl $0, -24(%rbp)
movl $0, -28(%rbp)
movl $0, -32(%rbp)
movl $0, -4(%rbp)
movl $0, -8(%rbp)
movl $0, -12(%rbp)
movl $0, -16(%rbp)
mov %rbp, %rsp
leave
ret
```

Código con optimización de Frame

```
.file "testFrameOptimization"
.text
.globl main
main:
enter $0,$0
subq $16 ,%rsp
movl $0, -4(%rbp)
movl $0, -8(%rbp)
movl $0, -12(%rbp)
movl $0, -16(%rbp)
subq $16 ,%rsp
movl $0, -20(%rbp)
movl $0, -24(%rbp)
movl $0, -28(%rbp)
movl $0, -32(%rbp)
subq $16 ,%rsp
movl $0, -36(%rbp)
movl $0, -40(%rbp)
movl $0, -44(%rbp)
movl $0, -48(%rbp)
addq $16 ,%rsp
addq $16 ,%rsp
addq $16 ,%rsp
mov %rbp, %rsp
leave
ret
```

En el ejemplo vemos como, en el código sin optimizaciones el tiempo de vida de las variables locales a un método es la duración de toda la ejecución del método, mientras que en el código con optimizaciones solo viven el tiempo en que pueden ser referenciadas.

Propagación de Constantes:

Se generó una nueva instancia del patrón visitor que se encarga de visitar el AST calculando y propagando todas las expresiones que puedan calcularse en tiempo de ejecución. Cuándo un nodo contiene un sub-árbol con una expresión que fue resuelta como una constante, se reemplaza ese sub-árbol por una hoja de tipo Literal que representa el valor calculado. La propagación de constantes es resuelta por la clase: ConstPropVisitor.
Veamos el funcionamiento de la optimización con el siguiente programa C-TDS:

```
class testConstPropOptimization{
void main(){
        int a;
        float b;
        a=((2+2)*4)/2;  //debe dar 8
        b=((2.0+2.0)*4.0)/2.0;  //debe dar 8.0
        }
}
```

Código sin Propagación de Constantes

```
.file "testConstPropOptimization"
.text
.globl main
main:
enter $32,$0
movl $0, -4(%rbp)
movss .FloatLiteral_0.0, %xmm3
movss %xmm3 ,-8(%rbp)
movl $2, %eax
addl $2, %eax
movl %eax, -12(%rbp)
movl -12(%rbp), %eax
imull $4, %eax
movl %eax, -16(%rbp)
movl -16(%rbp), %eax
movl $0,%edx
movl $2, %ecx
idivl %ecx
movl %eax, -20(%rbp)
movl -20(%rbp), %ecx
movl %ecx, -4(%rbp)
movss .FloatLiteral_2.0, %xmm0
addss .FloatLiteral_2.0, %xmm0
movss %xmm0, -24(%rbp)
movss -24(%rbp), %xmm0
mulss .FloatLiteral_4.0, %xmm0
movss %xmm0, -28(%rbp)
movss -28(%rbp), %xmm0
divss .FloatLiteral_2.0, %xmm0
movss %xmm0, -32(%rbp)
movl -32(%rbp), %ecx
movl %ecx, -8(%rbp)
mov %rbp, %rsp
leave
ret
.FloatLiteral_0.0:
 .float 0.0
.FloatLiteral_2.0:
 .float 2.0
.FloatLiteral_4.0:
 .float 4.0
```

Código con Propagación de Constantes

```
.file "testConstPropOptimization"
.text
.globl main
main:
enter $16,$0
movl $0, -4(%rbp) //Inicializacion de variable.
movss .FloatLiteral_0.0, %xmm3 //Inicializacion de variable.
movss %xmm3 ,-8(%rbp)
movss .FloatLiteral_8.0, %xmm3
movss %xmm3 ,-8(%rbp)
mov %rbp, %rsp
leave
ret
.FloatLiteral_0.0:
 .float 0.0
.FloatLiteral_8.0:
 .float 8.0
```

En el ejemplo podemos ver como en el código generado con propagación de constantes, los valores a asignar son definidos en tiempo de compilación y por consiguiente son literales en el código traducido. En el código sin optimizaciones, se realizan las cuentas viéndose reflejado en el número de líneas.

Poda de Código Muerto:

En principio, no iba a ser generada esta implementación pero nos pareció un despropósito recabar la información proporcionada por la propagación de constantes y no utilizarla para podar código muerto. Es por ello que se generó otra instancia del patrón Visitor que se encarga de visitar todos los nodos Statements del AST y en cuánto divisa un return, break o continue poda todo el código que existe a continuación. Además si encuentra un if, while o for cuya ejecución pueda decidirse en tiempo de compilación elimina aquellas partes que resultan inalcanzables. Por ejemplo si tenemos

while(false){...} poda completamente el sub-árbol con raíz WhileStatement.
Veamos el funcionamiento de la optimización con el siguiente programa C-TDS:

```
class testPruneUnreachableOptimization{
void main(){
        int a;
        for i=0, 9{
                a=a+1;
                continue;
                a=17;
                a=16;
                break;
        }
}
}
```

Codigo sin poda de statements inalcanzables

```
.file "testPruneUnreachableOptimization"
.text
.globl main
main:
enter $16,$0
movl $0, -8(%rbp)
movl $0, -4(%rbp)
.For_Loop_5:
movl -4(%rbp), %eax
movl $9, %edx
cmpl %edx, %eax
jl .L0
movl $0, %eax
jmp .Continue0
.L0:
movl $1, %eax
.Continue0:
movl %eax, -12(%rbp)
cmpl $0, -12(%rbp)
je .End_For_5
movl -8(%rbp), %eax
addl $1, %eax
movl %eax, -16(%rbp)
movl -16(%rbp), %ecx
movl %ecx, -8(%rbp)
movl -4(%rbp), %eax
addl $1, %eax
movl %eax, -4(%rbp)
jmp .For_Loop_5
movl $17, -8(%rbp)
movl $16, -8(%rbp)
jmp .End_For_5
movl -4(%rbp), %eax
addl $1, %eax
movl %eax, -4(%rbp)
jmp .For_Loop_5
.End_For_5:
mov %rbp, %rsp
leave
ret
```

Codigo con poda de statements inalcanzables

```
.file "testPruneUnreachableOptimization"
.text
.globl main
main:
enter $16,$0
movl $0, -8(%rbp)
movl $0, -4(%rbp)
.For_Loop_5:
movl -4(%rbp), %eax
movl $9, %edx
cmpl %edx, %eax
jl .L0
movl $0, %eax
jmp .Continue0
.L0:
movl $1, %eax
.Continue0:
movl %eax, -12(%rbp)
cmpl $0, -12(%rbp)
je .End_For_5
movl -8(%rbp), %eax
addl $1, %eax
movl %eax, -16(%rbp)
movl -16(%rbp), %ecx
movl %ecx, -8(%rbp)
movl -4(%rbp), %eax
addl $1, %eax
movl %eax, -4(%rbp)
jmp .For_Loop_5
movl -4(%rbp), %eax
addl $1, %eax
movl %eax, -4(%rbp)
jmp .For_Loop_5
.End_For_5:
mov %rbp, %rsp
leave
ret
```

En el ejemplo podemos ver como son podados aquellas líneas de código que se encuentran después de la sentencia continue cuándo se utiliza la optimización de poda de statements inalcanzables.

```
class testPruneUnreachableOptimization{
void main(){
        int a;
        if(-1<0){
                a=1;
        }
        else{
                a=0;
        }
        while(-1>0){
                a=a+1;
        }
        externinvk("printf",void, "%d\n", a);
        }
}
```

```
.file "testPruneUnreachableOptimization"
.text
.globl main
main:
enter $16,$0
movl $0, -4(%rbp)
movl $-1, %eax
movl $0, %edx
cmpl %edx, %eax
jl .L0
movl $0, %eax
jmp .Continue0
.L0:
movl $1, %eax
.Continue0:
movl %eax, -8(%rbp)
cmpl $1, -8(%rbp)
je .if_5
jmp .endif_5
.if_5:
movl $1, -4(%rbp)
jmp .endElse_6
.endif_5:
movl $0, -4(%rbp)
.endElse_6:
.While_condition_18:
movl $-1, %eax
movl $0, %edx
cmpl %edx, %eax
jg .L1
movl $0, %eax
jmp .Continue1
.L1:
movl $1, %eax
.Continue1:
movl %eax, -12(%rbp)
cmpl $0, -12(%rbp)
je .End_While_18
movl -4(%rbp), %eax
addl $1, %eax
movl %eax, -16(%rbp)
movl -16(%rbp), %ecx
movl %ecx, -4(%rbp)
jmp .While_condition_18
.End_While_18:
movl $.StringLiteral0 , %edi
movl -4(%rbp) , %esi
movl $0, %eax
call printf
mov %rbp, %rsp
leave
ret
.StringLiteral0:
.string "%d\n"
```

```
.file "testPruneUnreachableOptimization"
.text
.globl main
main:
enter $0,$0
subq $16 ,%rsp
movl $0, -4(%rbp)
subq $0 ,%rsp
movl $1, -4(%rbp)
addq $0 ,%rsp
movl $.StringLiteral0 , %edi
movl -4(%rbp) , %esi
movl $0, %eax
call printf
addq $16 ,%rsp
mov %rbp, %rsp
leave
ret
.StringLiteral0:
 .string "%d\n"
```

Conclusión

Si bien al final creemos haber arribado a un compilador con las características propuesta el enunciado del proyecto, creemos que algunas cuestiones del diseño pueden ser erróneas o por lo menos discutibles.
En principio, creemos que hubiese resultado en un mejor diseño diferenciar bien cada una de las etapas. Por ejemplo, la asignación de direcciones relativas o de desplazamiento respecto al ebp(offset) comienza en el análisis sintáctico y se finalizan en el TAC. Es decir todas las variables creadas por el usuario reciben su "dirección" en el análisis sintáctico y las auxiliares o generadas por el TAC durante la generación del TAC.
Por otro lado, los métodos conocen la dirección donde van a estar sus parámetros durante el análisis sintáctico, cuestión que podría no ser positiva en caso de generar código para máquinas con diferente arquitectura. También volvemos a recalcar el estrecho vínculo entre el TAC y el assembler x86-64 cuestión que dificultaría la generación de código para otras arquitecturas.
A modo de cierre, podemos mencionar que fue de gran aprendizaje la experiencia de desarrollar y "diseñar" un compilador.