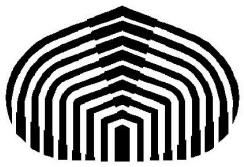

CNC

Informe Entrega I

Lenguajes de Programación II

Abril - Julio 2010

Federico Ponte - 06-40108
María Gabriela Valdés - 05-39020

Universidad Simón Bolívar.
Caracas, Venezuela.
4 de junio de 2010

Índice general

Índice de figuras

Capítulo 1

Introducción

Desde hace ya muchos años que se creó la primera computadora. Ésta fue creada en 1947 con el nombre de ENIAC (Electronic Numerical Integrator And Calculator). Desde ese día en adelante no han parado de aparecer nuevas computadoras cada vez más pequeñas y para diversos usos. Actualmente, las personas no se pueden imaginar un mundo sin éstas y siempre que las usan se hacen la misma pregunta: cómo es que algo ïnherte"tiene la capacidad entender las ordenes que se le dan y lograr cosas tan complejas.

Un modo fácil de responder sería el siguiente: las computadoras lo único que reconocen son 1's y 0's (hay o no hay voltaje), por lo que simplemente se le da una secuencia binaria con lo que uno desea que haga y listo. Pero que sucede si se quiere hacer un programa muy completo que requiera por lo menos de un millón de números binarios. Va a ser muy difícil para un humano y se va a perder mucho tiempo haciéndolo.

Para evitar este tipo de inconvenientes es que surge el estudio lenguajes de programación. La idea es poder tener un conjunto de frases capaces de expresar de un modo mucho más simple lo mismo que los 1's y 0's que entiende una computadora. Éstos pueden tener diferentes paradigmas: funcional, lógico, imperativo, etc. Así como pueden ser interpretados o traducidos.

1.1. Motivación del proyecto

A medida que pasa el tiempo cada día surgen nuevos lenguajes: siempre se está en la búsqueda de facilitarle las tareas al programador o cumplir

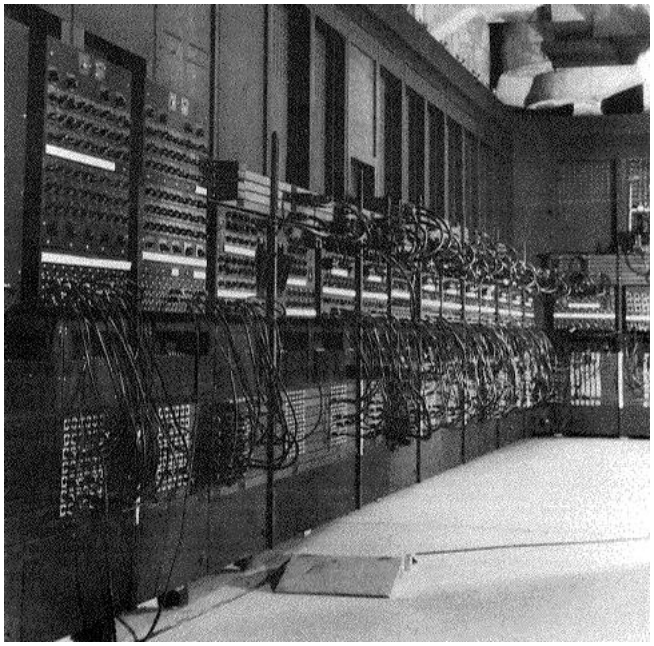

Figura 1.1: ENIAC

ciertas tareas específicas. Lo que nunca se observa es el complejo trabajo que significa diseñar e implementar uno de éstos.

Por ello la idea de este proyecto es justamente diseñar un lenguaje de tamaño medio y crear un compilador para éste.

Además uno como programador siempre tiene disgustos con algunas cosas de cada uno de los lenguajes. Así, al crear un lenguaje propio con las cosas que a uno más le agradan es bastante motivante.

1.2. Breve descripción del problema

Como bien se mencionó antes el problema es ciertamente bastante complicado. El diseño de un lenguaje toma un tiempo considerable y largo, ya que se tiene que considerar una gran cantidad de aspectos. La idea es que éste sea consistente y cumpla con su objetivo.

En cuanto a la implementación, ésta se hace en varias faces. Primero se crea un analizador lexicográfico o lexer, que va a reconocer todas las palabras reservadas para el lenguaje. Luego se crea un analizador sintćtico que se encarga de verificar si un programa está sintácticamente bien formado según los lineamientos de diseño del lenguaje. Después se proceden a hacer chequeos de tipo, también según el diseño del lenguaje. Finalmente el árbol generado del analizador sintáctico, es traducido a código de máquina para luego ser ejecutado.

1.2.1. Palabras reservadas

Las palabras reservadas de nuestro lenguaje y que no pueden ser utilizadas como identificadores son:

```
int float bool char union struct break if elif else switch case
default typedef for while const true false void return main read
print
```

Los siguientes caracteres tambien estan reservados como separadores del lenguajes:

```
( ) { } [ ] ; , .
```

Los siguientes 19 tokens son operadores del lenguaje formados por los caracteres:

```
= == + - ++ -- * / % != < > <= >= || && ! : &
```

Además también están reservados:

- Números literales.
- Caracteres literales.

1.2.2. Estructura general de un programa en CNC

En el diseño del lenguaje se definió que el orden estructural de un programa era el siguiente:

- Una secuencia de declaraciones de variables globales.
- Una secuencia de promesas.
- Una secuencia de declaraciones de funciones y procedimientos.
- Pograma principal *main*.

1.3. Contenido del Informe

El objetivo de este curso es desarrollar un lenguaje de programación y el propósito de este informe es exponer los detalles de diseño de nuestro lenguaje, explicando más específicamente el diseño de tipos básicos y compuestos, instrucciones, funciones y procedimientos. En la sección de implementación se mencionan las herramientas que utilizamos para el desarrollo del lenguaje, unido a la descripción de los elementos más importantes que fueron implementados, así como la especificación de las clases que se construyeron y la estructura general de la gramática. También en esta sección se exponen los problemas que se nos presentaron durante las distintas etapas de desarrollo del lenguaje, así como las soluciones propuestas e implementadas para resolver dichos conflictos. Seguido de este capítulo se explican los detalles de compilación y corrida de la aplicación, para luego terminar con una reseña general del estado actual de nuestro proyecto y de los errores que no pudimos resolver.

Capítulo 2

Diseño

2.1. Lineamientos

El nombre de nuestro lenguaje es "CNC"que quiere decir, "C is not C". En lineas generales, el diseño del mismo está basado en el diseño del lenguaje C. La razón principal es que C es un lenguaje que tiene una sintáxis sencilla pero que al mismo tiempo es muy poderoso y eficiente. Éstas dos últimas son características favorables y ventajosas para un lenguaje y para los programadores que lo utilizan. Además buscamos que nuestro lenguaje sea un poco más amigable que C en cuanto a la notificación y depuración de errores para el programador.

Este lenguaje es estáticamente tipeado. Los chequeos de tipos y declaraciones se hacen en tiempo de compilación. Por la misma razón, el lenguaje no permite reserva de memoria en tiempo de ejecución porque no se hace manejo de la memoria heap; sólo de la memoria estática y de pila para los procedimientos, funciones y llamadas recursivas de los mismos.

2.2. Operadores

A continuación se numeran los operadores en orden de precedencia y se especifica su asociatividad:

- `- (unario), !, --, ++`
- `*, /, %`

- `+, -`
- `<, <=, >, >=`
- `==, !=`
- `&&, ||`
- `=`
- Asociativos a la derecha:

  `=`
- Asociativos a la izquierda:

  `&&, ||, +, -, *, /, %`
- No asociativos:

  `==, !=, <, <=, >, >=, - (unario), !, --, ++`

2.3. Tipos Básicos

2.3.1. Int

- **Sintáxis:**

  ```
  int Identificador;
  ```

- **Detalles:** Con este tipo de datos podrán representarse los números enteros. Los *int* ocupará un tamaño de 8 bytes en memoria.

- **Operadores:**
  - Los operadores de comparacion cuyo resultado es del tipo *bool*:
    - Los operadores de comparacion numérica:
      `<, <=, >, >=`
    - Los operadores de equivalencia numérica:
      `==, !=`

- Los operadores numéricos cuyo resultado es del tipo *int*:
  - El operador de menos unario:
    `-`
  - Los operadores aditivos:
    `+, -`
  - Los operadores de multiplicativos:
    `*, /, %`
  - Los operadores de incremento:
    `++, --`
- El operador de asignación:

  `=`

  Cuyo resultado es el mismo valor que se está asignando, luego de haber hecho las conversiones implícitas necesarias.

2.3.2. Float

- **Sintáxis:**

```
float Identificador;
```

- **Detalles:** Con este tipo de datos podrán representarse los números reales. Los *float* ocuparán un tamaño de 8 bytes en memoria.

- **Operadores:**

  - Los operadores de comparacion cuyo resultado es del tipo *bool*:
    - Los operadores de comparacion numérica:
      `<, <=, >, >=`
    - Los operadores de equivalencia numérica:
      `==, !=`
  - Los operadores numéricos cuyo resultado es del tipo *float*:
    - El operador de menos unario:
      `-`
    - Los operadores aditivos:
      `+, -`

    - Los operadores de multiplicativos:
      `*, /`
    - Los operadores de incremento:
      `++, --`
- El operador de asignación:

  `=`

  Cuyo resultado es el mismo valor que se está asignando, luego de haber hecho las conversiones implícitas necesarias.

2.3.3. Char

- **Sintáxis:**

  ```
  char Identificador;
  ```

- **Detalles:** Los caracteres ocuparán 8 bytes en memoria. Éstos serán útiles junto con el tipo de dato *Array*, para poder simular el tipo de dato "String"que nuestro lenguaje no posee explícitamente.

  Es importante justificar que decidimos que los caracteres ocuparían 8 bytes a bajo nivel, para hacer más fácil el manejo de los bits de los registros a la hora de generar código, ya que los registros que utilizamos son todos de tamaño de 64-bits. Ésto nos evita tener que hacer operaciones de *AND*'s y *OR*'s sobre los registros. Además garantizamos que la memoria esta alineada siempre de 8 en 8 bytes.

- **Operadores:**

  - Los operadores de comparacion cuyo resultado es del tipo *bool*:
    - Los operadores de comparacion:
      `<, <=, >, >=`
    - Los operadores de equivalencia:
      `==, !=`
  - El operador de asignación:

    `=`

    Cuyo resultado es el mismo valor que se está asignando, luego de haber hecho las conversiones implícitas necesarias.

2.3.4. Bool

- **Sintáxis:**

```
bool Identificador;
```

- **Detalles:** El tipo de dato *bool* ocuparán 8 bytes en memoria, al igual que los caracteres. Cabe destacar que las expresiones booleanas son las que determinan el control de flujo de las siguientes instrucciones:

```
if, while, for
```

.

Es importante justificar que decidimos que los booleanos ocuparían 8 bytes a bajo nivel, para hacer más fácil el manejo de los bits de los registros a la hora de generar código, ya que los registros que utilizamos son todos de tamaño de 64-bits. Ésto nos evita tener que hacer operaciones de *AND*'s y *OR*'s sobre los registros. Además garantizamos que la memoria esta alineada siempre de 8 en 8 bytes.

- **Operadores:**
  - Los operadores relacionales:

    ```
    ==, !=
    ```

  - El operador de complemento lógico:

    ```
    !
    ```

  - Los operadores lógicos:

    ```
    &&, ||
    ```

  - El operador de asignación:

    ```
    =
    ```

    Cuyo resultado es el mismo valor que se está asignando, luego de haber hecho las conversiones implícitas necesarias.

2.4. Conversión de Tipos Básicos

Cuando se realizan operaciones entre tipos básicos distintos, la conversión se hace implícitamente según la siguiente tabla de cohersión, donde N/A se refiere a que la conversión entre dichos tipos "No Aplica", y por el contrario A significa que "Aplica":

Para los operadores aritméticos:

`+, -, *, /, %`

Tipo	int	float	bool	char
int	int	float	N/A	N/A
float	float	float	N/A	N/A
bool	N/A	N/A	N/A	N/A
char	N/A	N/A	N/A	N/A

Para los operadores de comparación:

`==, !=`

Tipo	int	float	bool	char
int	A	A	N/A	A
float	A	A	N/A	N/A
bool	N/A	N/A	A	N/A
char	A	N/A	N/A	A

`<, <=, >, >=`

Tipo	int	float	bool	char
int	A	A	N/A	A
float	A	A	N/A	N/A
bool	N/A	N/A	N/A	N/A
char	A	N/A	N/A	A

Para los operadores booleanos:

`&&, ||`

Tipo	int	float	bool	char
int	N/A	N/A	N/A	N/A
float	N/A	N/A	N/A	N/A
bool	N/A	N/A	A	N/A
char	N/A	N/A	N/A	N/A

Para el operador de asignación, y los parámetros de llamadas a subrutinas:

Tipo	int	float	bool	char
int	int	int	N/A	N/A
float	float	float	N/A	N/A
bool	N/A	N/A	bool	N/A
char	char	N/A	N/A	char

2.5. Tipos Compuestos

2.5.1. Arreglos

- **Sintáxis:**

```
Tipo [Expresion]...[Expresion] Identificador;
```

- **Detalles:** Cada [Expresión] representa una dimensión del arreglo y la expresión indica su tamaño. El tipo de la Expresión, debe ser obligatoriamente *Int*. Además es importante mencionar que *tipo* puede ser cualquoer tipo básico antes especificado, así como un registro o unión que se explicarán a continuación.

  En cuanto al almacenamiento en memoria, este tipo de dato se va alamcenar por filas, es decir, se recorre el arreglo de izquierda a derecha y de arriba a abajo, y de este modo se almacenan, al igual que en el lenguaje C.

- **Operadores:**
  - El operador de asignación:

    `=`

    Será válido siempre y cuando las dimensiones y el tipo base de los arreglos involucrados en la operación sean iguales.
  - Los operadores de comparación:

    `== y !=`

    Serán válidos siempre y cuando las dimensiones y el tipo base de los arreglos involucrados en la operación sean iguales.
- **Inicialización:**

```
Tipo [Expresion1]...[Expresionn] Identificador =
{ { {x, y, ... };  {p, q, ... }; ... }; ... ; { {z, w, ...}; ... } };
```

  Donde la lista exterior contiene tantos elementos, a su vez listas, como indique la Expresión1, así, cada una de estas listas será de tamaño igual a la Expresión2, y así sucesivamente.
- **Acceso:** Para acceder a cierta pocisión del arreglo, la notación que debe seguirse es la siguiente:

```
Identificador[Expresion1]...[Expresionn];
```

  Donde cada Expresión_i debe ser de tipo *Int* con valores válidos entre 0 y n - 1, donde n es el tamaño de la dimensión a la que le corresponden dichos corchetes.

2.5.2. Registro

- **Sintáxis:**

```
struct {
  Tipo1 Identificador1;
  Tipo2 Identificador2;
  ...
```

```
  Tipon Identificadorn;
} Identificador;
```

- **Detalles:** Los registros son estructuras cuyo cuerpo lo conforman tipos básicos o compuestos. Cada "atributo" de la estructura viene especificado con su tipo y nombre correspondiente. Es importante recalcar que todos los identificadores dentro de la estructura deben ser diferentes entre si.

  El almacenamiento en memoria será como un bloque sin compactación, es decir, si se tiene:

```
struct {
  bool a;
  int b;
} x;
```

  Esta estructura se almacenaría así:

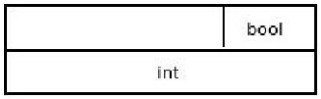

Figura 2.1: Tipo de almacenamiento

  y no así:

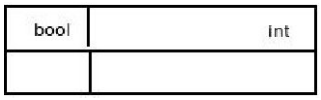

Figura 2.2: Tipo de almacenamiento

  Se escogió este tipo de almacenamiento porque a pesar de que se pierde memoria, se gana eficiencia en el programa.

- **Operadores:**

- El operador de asignación:

  `=`

  Será válido siempre y cuando la cantidad de campos, el orden de los mismos, los identificadores y el tipo asociado a cada uno de ellos, sean iguales en ambos de los registros involucrados en la operación.

- **Inicialización:**

```
struct {
  Tipo1 Identificador1;
  Tipo2 Identificador2;
  ...
  Tipon Identificadorn;
} Identificador =
{Identificador1 = Valor1; ... ; Identificadorn = Valorn} ;
```

- **Acceso:** Para acceder a cierto campo del registro, la notación que debe seguirse es la siguiente:

```
Identificador.Campo_i;
```

  Donde Campo_i, debe ser uno de los Identificadores_i especificados en la declaración.

2.5.3. Unión

- **Sintáxis:**

```
union {
  Tipo1 Identificador1;
  Tipo2 Identificador2;
  ...
  Tipon Identificadorn;
} Identificador;
```

- **Detalles:** La sintáxis de las uniones es igual a la de los registros. Lo que diferencia a las uniones de los registros es que éstas sólo pueden tener un campo activo de entre todos los que se especificaron en su declaración, éste se actualiza y cambia a través de la operación de asignación.

  Para las uniones también se cumple que su cuerpo lo conforman tipos básicos o compuestos. Cada "atributo" de la estructura viene especificado con su tipo y nombre correspondiente. Es importante recalcar que todos los identificadores dentro de la estructura deben ser diferentes entre si.

  La representación en memoria de este tipo complejo sigue el siguiente esquema:

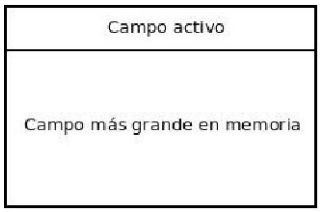

  Figura 2.3: Tipo de almacenamiento de las uniones

  Con el almacenamiento del campo activo en memoria lo que se busca es evitar errores de asignación. Por otro lado, como las uniones sólo poseen un campo activo a la vez, basta con reservar espacio en memoria igual al campo de mayor tamaño.

- **Operadores:**

  - El operador de asignación:

    `=`

    Será válido siempre y cuando la cantidad de campos, el orden de los mismos, los identificadores y el tipo asociado a cada uno de ellos, sean iguales en ambas de las uniones involucrados en la operación. El campo que se especifique en la asignación, será el campo activo de la unión.

```
Identificador = {Campo_i = Valor_i} ;
```

- **Inicialización:**

```
union {
  Tipo1 Identificador1;
  Tipo2 Identificador2;
  ...
  Tipon Identificadorn;
} Identificador = {Campo_i = Valor_i} ;
```

- **Acceso:** Para acceder a cierto campo de la unión, la notación que debe seguirse es la siguiente:

```
Identificador.Campo_i;
```

  Donde Campo_i, debe ser uno de los Identificadores_i especificados en la declaración.

2.6. Instrucciones

El conjunto de instrucciones que se tiene es el mismo que el de la mayoría de los lenguajes imperativos modernos. Se usó la misma sintáxis que C, C++ y Java con el objetivo de facilitar el aprendizaje del lenguaje. Además éstas proporcionan suficiente poder de expresión por lo que no se agregó alguna instrucción fuera de lo común.

2.6.1. Declaraciones

- **Sintáxis:**

```
tipo Identificador;
tipo Identificador, ..., Identificador;
tipo Identificador = Expresion;
tipo Identificador, ..., Identificador = Expresion;
```

- **Detalles:** La declaración es igual que en la mayoría de lenguajes. Se puede hacer una declaración con o sin inicialización. En el tercer y cuarto caso se asigna el mismo valor de la expresión a cada uno de los valores declarados.

  Un detalle importante de las declaraciones, es que éstas se pueden hacer en los bloques de los *if*, *for*, *switch* y *while*. En estos casos las variables sólo van a existir y van a tener alcance en la ejecución de dicho bloque.

  Cabe destacar, que no permitimos declaraciones anidadas de variables con igual identificador dentro de los bloques de las instrucciones *if*, *for*, *switch* y *while*, pero si dentro de las subrutinas.

  Por ejemplo:

```
int a;

if (a > 3) {
  a = 3;
  int a;
  a = 2;
}
```

  Esto daría error porque la variable *a* ya estaba declarada fuera del *if*.

```
int a;

void proc(int a) {
  a = 3;
}
```

  Esto no daría error porque existe una variable, en este caso pasada como parámetro, que tiene alcance dentro del procedimiento y solapa a la variable global.

```
int a;

void proc(int a) {
 if(a > 3) {
   a = 3;
   int a;
   a = 2;
 }

}
```

Este ejemplo da error, porque la variable *a* está pasada como parámetro a la subrutina, y luego se hace una declaración de una variable con igual identificador. Es importante destacar que el error no es porque existe una variable global *a*. Si por ejemplo el parámetro del procedimiento se denotara por el identificador *b*, en vez de *a*, el programa no daria error.

La declaración de constantes sigue la misma sintáxis antes mencionada excepto que se debe agregar la palabra reservada *const* antes de especificar el tipo. Así se puede hacer la distinción de declaración entre variables y constantes. Además al declarar un identificador como constante, es obligatorio darle un valor inicial. Esta decisión de diseño da mayor legibilidad al programador.

2.6.2. Asignación Simple

- **Sintáxis:**

```
Identificador = Expresion;
```

- **Detalles:** El operador de asignación es el "=". A pesar que muchas personas opinan que ":="debe ser el operador, se elegió éste porque es el más usado entre los lenguajes que conocemos. La idea es que un programador con experiencia en otros lenguajes, tenga la capacidad de aprender el nuestro rápidamente.

2.6.3. Asignación múltiple

- **Sintáxis:**

```
Identificador = Identificador = ... = Expresion;
```

- **Detalles:** La idea de ésta instrucción es darle la facilidad al programador de asignarle el mismo valor a un conjunto de variables en una sola instrucción. Esto aumenta la legibilidad del código y lo compacta. Es importante recordar que para esta instrucciń el operador "=" es asociativo hacia la derecha.

2.6.4. Selección

- **If:**
  - **Sintáxis:**
    - ```
      if (Expresion Booleana) {
        Instrucciones ...
      }
      ```
    - ```
      if (Expresion Booleana) {
        Instrucciones ...
      }
      else {
        Instrucciones ...
      }
      ```
    - ```
      if (Expresion Booleana) {
        Instrucciones ...
      }
      elif (Expresion Booleana) {
        Instrucciones ...
      }
      elif (Expresion Booleana) {
        Instrucciones ...
      }
      ...
      else {
        Instrucciones ...
      }
      ```
      ```

- **Detalles:**
  Lo que se busca con éstas tres formas de escribir una selección es darle la facilidades al programador.

  Se eligió la palabra *elif* porque da una mayor legibilidad y comodidad frente a la posibilidad de colocar *else if* como en muchos otros lenguajes.

  Se puede notar que es obligatorio tener las expresiones booleanas siempre entre paréntesis. Igual sucede con el uso de llaves para encerrar las instrucciones. Esto es para mantener el código más entendible y organizado para el programador.

- **Switch:**

  - **Sintáxis:**

```
switch (Expresion) {
  case Constante:
       Instrucciones ...
  case Constante:
       Instrucciones ...
  ...
  default:
       Instrucciones ...
}
```

  - **Detalles:** Nuestro *switch* es muy parecido al *switch* de Java y C. Lo único que es diferente es que los valores junto a los cases deben constantes, el objetivo de esto es hacer más eficiente el código que se genere y evitar errores de programación. Además no se tiene que colocar *break* en los casos. Es bien sabido que esta última opción es poco usada por ser una tarea tediosa para el programador.

    Además el tipo de la Expresión dentro del *switch* así como de cada constante asociada a un *case*, debe ser un tipo Básico, de los mencionados y explicados anteriormente.

Por otro lado, es importante mencionar que para que la instrucción sea válida, el tipo de la expresión entre paréntesis del *switch*, debe ser igual al tipo de las constantes en cada uno de los *case* de la instrucción.

2.6.5. Ciclos

- **For:**
  - **Sintáxis:**

```
for (Asignacion o declaracion
     con inicializaci\'on;
     Expresion Booleana;
     Asignacion o expresion con
     operadores ++ o --) {
     Instrucciones ...
}
```

  - **Detalles:** En la primera parte del *for* sólo se puede tener una asignación o declaración simple o múltiple con inicialización. La idea es evitar que el programador genere código complicado de entender y leer. La segunda parte de la sección limitada entre paréntesis es una expresión booleana que va a indicar la condición de parada del ciclo. Finalmente se coloca una asignación (puede ser simple o múltiple), o una expresión con los operadores de incremento "++" o "-". Con esto se busca que el código sea simple pero expresivo al mismo tiempo.

- **While:**
  - **Sintáxis:**

```
while (Expresion Booleana) {
      Instrucciones ...
}
```

  - **Detalles:** La notación y significado de esta instrucción es igual a la de los *while* de la mayoria de los lenguajes imperativos.

2.7. Funciones y Procedimientos

- **Declaración:**
  - **Sintáxis:**
    - **Procedimiento:**

```
void nombre(tipo &parametro1,
            tipo parametro2, ... ,
            tipo &parametron) {
  Instrucciones...
}
```

    - **Función:**

```
tipo nombre(tipo &parametro1,
            tipo parametro2, ... ,
            tipo &parametron) {
  Instrucciones...
}
```

  - **Detalles:** Para la declaración de procedimientos y funciones se eligió la sintáxis de C, C++ y Java porque es fácil de entender y usar. Además mantiene una sintáxis consistente con las instrucciones *if*, *for*, *switch* y *while*, donde el conjunto de instrucciones se encuentra limitado entre llaves.

    Además se le agrega la opción al programador de indicar si quiere pasaje por copia/resultado o solamente copia. Para ello se dispone del símbolo "&", al utilizarlo, se indica que el pasaje de parámetros con el que se desea trabajar es de tipo copia/resultado. Luego, el pasaje de parámetros por defecto, sin utilizar el símbolo "&", es tipo copia.
    Los procedimientos van a manejar alcance de variables tanto local como globlal. Ejemplo:

```
int b;

void proc(int b) {
 b = 3;
 return;
}
```

En este caso a la variable *b* del procedimiento es a la que se le asigna el valor 3 y no a la variable global, declarada fuera del procedimiento.

Tanto los procedimientos como las funciones pueden tener la instrucción *return* dentro de su cuerpo. Esta instrucción permite a las subrutinas terminar su ejecución. En el caso de las funciones, además, devolver un valor del mismo tipo especificado en la declaración.

Para las funciones es obligatorio devolver un valor. El siguiente ejemplo debe dar error:

```
int a(bool b) {
 int c;
 if ( b ) {
  return c;
 }
 else {
  c = 3;
 }
}
```

Para los procedimientos no es obligatorio cumplir esta condición. El *return* serviría para salir de la subrutina en cierto momento, sino la salida del mismo se produce al final de la ejecución de todas las instrucciones dentro del cuerpo. En caso de utilizarse la instrucción *return* dentro de un procedimiento, ésta no esta asociada a ningún valor o identificador particular, ya que los procedimientos no retornan ningún valor al terminar.

Algo importante que mencionar es que el *main* es manejado como una subrutina por lo que es válido que entre sus instrucciones se encuentre un *return* con un valor o identificador asociado. Ésto permite al programador retornar, por ejemplo, resultados de la ejecución del programa principal.

Aunque la sintáxis del *main* especifica que el programa principal

puede tener un valor de retorno, la idea es ser flexibles en este caso particular y dejar al programador la decisión de usar o no la instrucción *return*, y en el caso de hacerlo, utilizarla simplemente para terminar el programa, o para devolver un valor.

Nuevamente para el caso particular del *main*, aunque éste es manejado como una función, es importante destacar que dicha "función" no puede ser llamada por otra subrutina declarada en el programa, ni tampoco por ella misma.

- **Invocación:**
  - **Sintáxis:**

```
funcion/procedimiento(valor1, valor2, ... , valorn);

Identificador = funcion(valor1, ... , valorn);
```

  - **Detalles:** El orden de evaluación va a ser de izquierda a derecha.

    Cabe resaltar que en la invocación tanto a funciones como a procedimientos, no hace falta colocar el símbolo de "&" delante de ninguno de los parámetros reales, porque dicha especificación se hizo en la declaración de la subrutina en cuestion.

- **Promesas de procedimientos y funciones:**
  - **Sintáxis:**
    - **Procedimiento:**

```
void nombre(tipo &parametro1,
            tipo parametro2, ... ,
            tipo &parametron);
```

    - **Función:**

```
tipo nombre(tipo &parametro1,
            tipo parametro2, ... ,
            tipo &parametron);
```

  - **Detalles:** Para atacar el problema de llamadas mutuamente recursivas el lenguaje va a disponer de la misma estrategia de C

. El programador va a poder colocar las firma de una función o procedimiento y así cuando este sea usado se pueda chequear su correcta llamada y uso.

Las promesas tienen que ser declaradas después de la declaración de variables globales, pero antes de la declaración de los procedimientos/funciones con cuerpo. Además, cada promesa tiene que ser cumplida luego de su declaración, esto es, debe existir la definición del cuerpo de la subrutina, en algún punto futuro del programa, es decir, si se tiene la siguiente promesa:

```
int funcion1(int a, bool &b);
```

En alguna parte del código del programa tiene que ser declarado el cuerpo de la función:

```
int funcion1(int a, bool &b){
  ...
}
```

Capítulo 3

Implementación

El lenguaje que estamos utilizando para desarrollar el lenguaje es Java 1.4 [3]

3.1. Herramientas

3.1.1. JFlex y JavaCup

- **JFlex 1.4.3:** JFlex es un analizador y generador lexicográfico hecho en Java, que genera código en Java. A la herramienta se le proporciona un archivo que posee toda la especificación lexicográfica del lenguaje. En este archivo se especifican cuáles son las palabras reservadas del lenguaje y los caracteres (incluyendo caracteres ascii, números y caracteres especiales) válidos y reconocibles en el diseño de un programa hecho en este lenguaje.

  Toda la documentación que utilizamos provino del manual de usuario disponible que se encuentra en internet [1].

- **JavaCup 0.1:** JavaCup es un analizador y generador sintáctico que también está hecho y genera código en Java. A la herramienta se le proporciona un archivo con la gramática del lenguaje, es decir, toda la especificación sintáctica válida en un programa hecho en nuestro lenguaje. En este archivo se especifican cuales son los símbolos terminales y no terminales de las producciones que conforman la gramática,

seguido de éstas últimas.

La información utilizada para el uso de esta herramienta también provino del manual de usuario que se encuentra en internet [2].

- **NASM:** Es un ensamblador de 80x86 y x86-64 diseñado para la portabilidad y modularidad. Éste soporta un gran rango de formatos de objeto, incluyendo Linux y BSD* a.out, ELF, COFF, Mach-O, Microsoft 16-bit OBJ, Win32 y Win64. También puede producir archivos binarios. Su sintaxis está diseñada para ser simple y fácil de entender, igual a la de Intel, pero menos compleja. Soporta todas las arquitecturas x86 conocidas, y tiene un gran soporte de macros.

  Su documentación puede ser encontrada en internet [4]

3.1.2. ESC/Java

Esta es una herramienta que se encarga de hacer chequeos estáticos más avanzados que los que pueden detectarse a tiempo de compilación. Para ello debe especificársele un poco más de información antes de cada procedimiento o especificación de clase en donde desee usarse.

Para esta entrega se busco darle más utilidad a esta herramienta. Se registraron los siguientes resultados para cada clase:

Resultados de la primera pasada:

Tipo	Arreglo	Registro	Union	Basico
0 cautions	0 cautions	24 cautions	24 cautions	0 cautions
0 warnings	0 warnings	11 warnings	11 warnings	0 warnings

ASTExpresion	ASTAritmetica	ASTIdentificador	ASTAsignacionExpr
1 caution	1 caution	7 cautions	18 cautions
0 warnings	7 warnings	2 warnings	1 warning

Resultados de la segunda pasada:

ASTBool	ASTConst
1 caution	2 cautions
2 warnings	0 warnings

ASTInstruccion	ASTAsignacion	ASTSwitch	ASTWhile
0 cautions	18 cautions	17 cautions	1 warning
0 warnings	0 warnings	4 warnings	0 warnings

ASTIf	ASTBloque
17 cautions	23 cautions
3 warnings	0 warnings

Tipo	Arreglo	Registro	Union	Basico
0 cautions	4 cautions	1 caution	1 caution	4 cautions
0 warnings	1 warning	5 errors	5 errors	13 warnings

ASTExpresion	ASTAritmetica	ASTIdentificador	ASTAsignacionExpr
0 cautions	1 caution	1 caution	1 caution
0 warnings	206 errors	30 errors	63 errors

ASTBool	ASTInvocarExpresion	ASTConst
1 caution	18 cautions	1 cautions
310 errors	0 warnings	41 errors

ASTInstruccion	ASTAsignacion	ASTInvocar	ASTSwitch	ASTWhile
0 cautions	1 caution	0 cautions	1 cautions	1 caution
0 warnings	99 errors	15 warnings	68 errors	18 errors

ASTIf	ASTBloque	ASTLiteralUR	ASTPrintConstante
1 caution	1 caution	1 caution	1 caution
38 errors	6 errors	10 warnings	12 errors

Sym	SymVar	SymProc	SymTable
0 cautions	0 cautions	0 cautions	0 cautions
0 warnings	0 warnings	0 warnings	3 warnings

ASTAcceso	ASTAccesoArreglo	ASTAccesoUR
0 cautions	0 cautions	0 cautions
1 warning	0 warnings	3 warnings

AssemblerInfo ASTLiteralArreglo	ASTCast	Resultado
1 caution 0 cautions	1 caution	0 cautions
32 errors 6 warnings	56 errors	0 warnings

ASTPrintIdentificador	ASTRead	Promesa
1 caution	1 caution	0 cautions
9 errors	63 errors	0 warnings

3.2. Reseña de elementos implementados

Como bien se explicó en la introducción la fase de compilación tiene varias etapas; y en el desarollo de nuestro lenguaje buscamos cubrir en su totalidad cada una de ellas.

Es importante destacar que cada una de las clases implementadas tiene un método *printTree* que muestra por consola el arbol construido hasta el momento.

- **Lexer:** Está perfectamente implementado. Se tiene todo el código en el archivo Scanner.jflex. Éste programa se encargará de pasar el valor de los tokens al Parser.
- **Gramática:** Se encuentra en el archivo Parser.cup. Contiene todas las producciones y símbolos necesarios para generar el lenguaje.
- **Tabla de Símbolos:** Ésta se implementó como una tabla hash que va de nombres a símbolos. Dentro del Parser.cup, se manejan exactamente 2 tablas de símbolos, *actual* y *anterior*. Se decidieron utilizar 2 tablas para poder manejar las declaraciones anidadas dentro de procedimientos e instrucciones. Con estas 2 tablas es suficiente porque cada vez que se entra a un nivel nuevo de anidamiento, lo que se hace es crear una tabla nueva cuyo padre es la tabla *anterior* (esto es lo que nos permitirá reestablecer el estado actual de las tablas cuando nos salgamos del nivel de anidamiento) y convertir la tabla *anterior* en la tabla *actual*.

  El método insertar va a chequear que el elemento que se desea agregar, no haya sido declarado en ningún momento anterior (cuenta los anidamientos). Para hacer este chequeo se utiliza un valor que es pasado como parámetro con el nombre de *profundidad*, que permite hacer la verificación de la existencia o no de una variable declarada con el mismo identificador en un nivel de anidamiento mayor al actual.

  El método *existeProfundidad*, difiere con el método *exist* en que éste revisa hasta cierta profundidad también dada, pero su valor de retorno es booleano.

La otra operación que puede hacerse sobre tablas es la verificación de la existencia de un elemento. En el caso que exista devuelve el tipo del símbolo, de lo contrario el valor de retorno es *null*.

Finalmente también se puede obtener un símbolo específico de la tabla, con el método *getSym*, lo que permite verificar si el identificador que se busca es un procedimiento o una variable.

- **Sym:** Esta clase contiene 2 atributos que representan el nombre y tipo, en caso de instancias de SymVar, de la variable declarada, y en el caso de instancias de SymProc, de subrutinas declaradas. Su jerarquía es la siguiente:

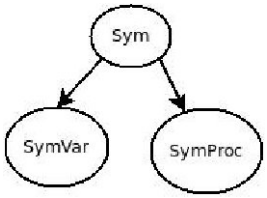

Figura 3.1: Sym

  - **SymVar:** Representa un símbolo de una variable de algun tipo, tanto básico, como compuesto. El atributo *isConst* indica si el identificador fue declarado como variable o constante. En cambio el atributo *offset* determina cual es el desplazamiento de la variable con respecto al marco de pila(Frame Pointer) o al bloque de memoria estático, dependiendo ésto de si la variable es global o no.
  - **SymProc:** Representa un símbolo de una subrutina, tanto procedimiento como función. Sus atributos son una lista llamada *in* que guarda los parametros de entrada de la subrutina, *ref* que es una lista que contiene booleanos indicando si los argumentos son por referencia o no, *tamlocal* que es el tamao en bytes que se necesita para las variable locales y un bloque de instrucciones con el cuerpo de la subrutina. El atributo *bloque* es el que contiene la tabla de símbolos del procedimiento con las declaraciones locales al mismo.

- **AST:** Los árboles sintácticos se implementaron por medio de varias clases con la siguiente jerarquía:

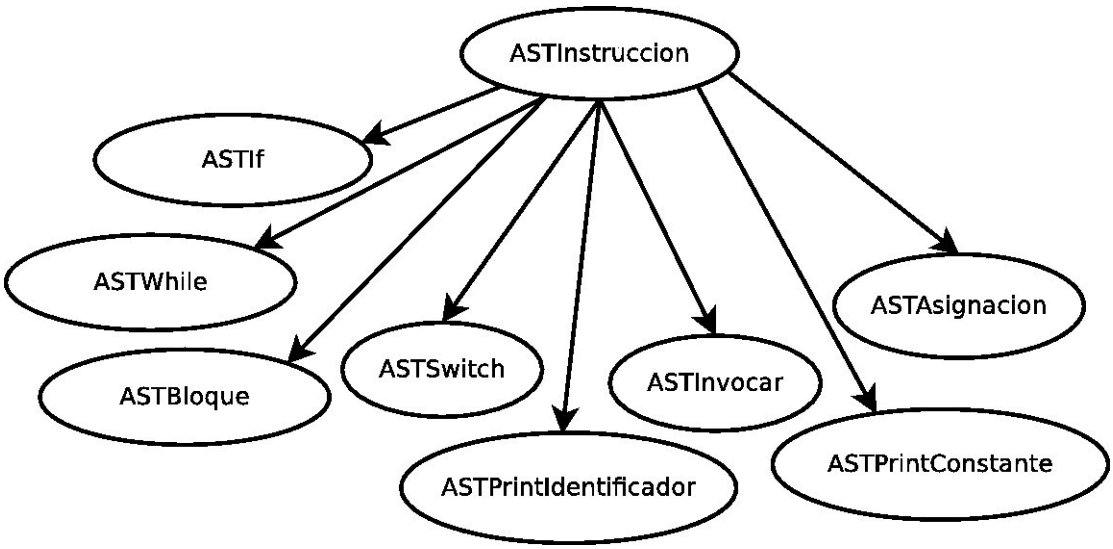

Figura 3.2: Arbol de herencia de las instrucciones

La clase padre ASTInstrucción, tiene tres atributos. Dos booleanos que indican si retorna o hace break. El último atributo es un String con el nombre de la instrucción.

El método *update* se encarga de verificar si la instrucción retorna o hace break dentro de su cuerpo, y de actualizar los atributos de la clase que manejan estos casos.

La mayora de las clases poseen un atributo Tipo llamado state que permite sabe si hay errores, de que tipo son las expresiones, si las instrucciones estan bien. Como su nombre lo dice lleva el estado de los árboles.

- **ASTAsignacion:** Consta de tres campos:
  - LinkedList ids: lista de ASTIdentificadores.
  - LinkedList casts: lista de ASTCast.
  - ASTExpresion expr: expresión que se va a asignar.

- boolean isDeclaration: indica si la asignacion es de una declaració.

- **ASTInvocar:** Consta de cuatro campos:
  - String nombre: nombre de la subrutina que se quiere llamar.
  - LinkedList expresionEntrada: lista de ASTExpresion, con los argumentos de entrada.
  - Tipo state: tipo del valor que devuelve la subrutina.

  El método *check* de esta clase, sólo es llamado cuando ya se sabe que la subrutina que se quiere invocar ha sido declarada anteriormente.
- **ASTIf:** Consta de tres campos:
  - LinkedList cond: cada uno de las expresiones booleanas que se tienen en los if y elif.
  - LinkedList bloques: los bloques de código de los if y elif.
  - ASTBloque els: El bloque de código del else.
- **ASTWhile:** Sus campos son:
  - ASTExpresion cond: condición que debe cumplir para estar en el ciclo.
  - ASTBloque bloque: bloque de código a ejecutar.
- **ASTBloque:**
  - SymTable table: Tabla de símbolos correspondiente del bloque.
  - LinkedList insts: Lista de las instrucciones.
- **ASTIdentificador:**
  - SymTable table: Tabla donde se encontró el identificador.
  - ASTAcceso acceso: El acceso que se le haga al identificador ([] de arreglo o . de campo de un registro o unión).
- **ASTSwitch:** Sus atributos son:
  - ASTExpresion exp: La expresión del switch.
  - LinkedList cases: Lista de ASTConst, con cada una de las constantes de los cases.
  - LinkedList bloques: Lista de ASTBloque, con cada uno de los bloques de los cases.
  - ASTBloque def: ASTBloque del caso default.

Para ver que están bien lo que se revisa es que cada uno de los bloques este bien. Además se ve que el tipo de la expresión corresponda con cada una de las constantes.

- **ASTExpresión:** Posee los siguientes atributos:
  - String value: Nombre del operador principal de la expresión.
  - Tipo state: tipo de la expresión.
  - ASTExpresion left: Expresión izquierda.
  - ASTExpresion right: Expresión derecha. En el caso que sea unitaria ésta es nula.
  - boolean canCheck: Si es posible chequear la expresión o no, esto es porque una expresión puede ser una llamada a una subrutina que todavía no ha sido declarada.

  Esta clase tiene el método *update* que se encarga de actualizar el campo *state* de la expresión y el método *check* que indica si dicha expresión es váilida o no.

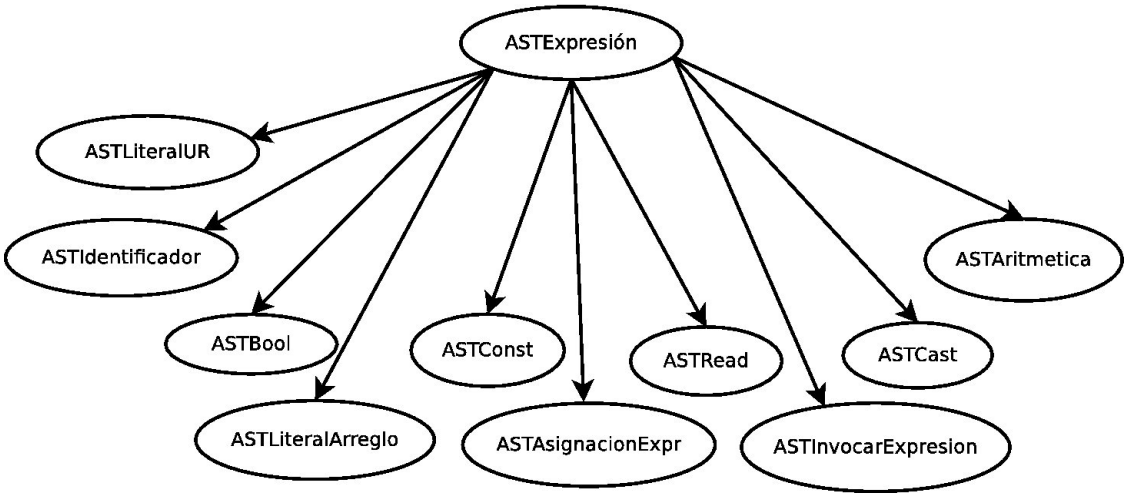

Figura 3.3: Herencia de la clase ASTExpresion

- **ASTAsignacionExpr:**
  - ASTIdentificador id: Identificador al que se asigna la expresión en cuestión.
  - ASTCast cast: Indica si hay que hacerle una conversión al tipo de la expresión.

- **ASTInvocarExpresión:** Contiene los mismo tres campos que ASTInvocar.
- **ASTAritmética:** Tiene dos árboles, de los cuales uno puede ser nulo en el caso que la operación sea unaria.
- **ASTBool:** Es igual a la clase ASTArimética, sólo que difiere en los chequeos que realiza: las expresiones que se pueden comparar son diferentes a las que se pueden sumar, restar, dividir, etc.
- **ASTConst:** Se usa para almacenar las constantes, por cuatro campos: entero, flotante, caracter y booleano. Cuando se crea se le asigna un valor a uno de estos.
- **ASTAcceso:** Esta clase representa los posibles accesos que se le pueden hacer a un identificador ([] o .).

  El único atributo de la clase denotado por el identificador *hijo*, se utiliza para manejar n cantidad de accesos seguidos a un mismo identificador.
  Sus subclases son las siguientes:

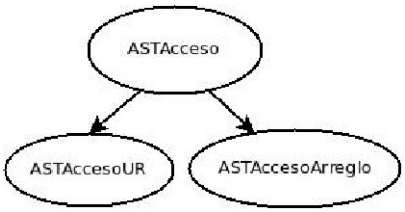

Figura 3.4: Herencia de la clase ASTAcceso

- **ASTAccesoUR:** Representa el acceso a un campo de un registro o una unión. Por ello sólo tiene un String que contiene dicho campo éste.
- **ASTAccesoArreglo:** Como su nombre lo dice representa el acceso a un arreglo. Su único atributo es una ASTExpresion que representa el índice del arreglo que se quiere acceder.
- **ASTPrintConstante:** Sus atributos son:
  - ASTConst constante: La constante que se quiere imprimir.
- **ASTPrintIdentificador:** Sus atributos son:
  - ASTIdentificador iden: El identificador que se quiere imprimir.

- **ASTRead:** Ésta clase no tiene atributos, y se crean instancias de la misma para que a la hora de generar código de dicha instrucción, se sepa que ésta es una instrucción de lectura.

- **ASTCast:** Es una clase para "adornar" el los AST de constantes e identificadores. Se crean instancias de ésta clase para saber cuando debe hacerse la traducción necesaria para la transformación de un tipo de dato básico a otro básico también.
- **ASTLiteralArreglo:** Sus atributos son:
  - LinkedList arreglos: lista de listas que contiene el literal que se quiere asignar.
  - boolean flag: indica si ha ocurrido un error revisando el literal.
- **ASTLiteralUR:** Sus atributos son:
  - LinkedList asignaciones: son las asignaciones que se le quieren hacer a una estructura o unión.

Funcionamiento general:

La idea es ir verificando todo lo que se puede a medida que se va parseando. Por eso se tienen en los árboles los métodos update y check. El primero lo que va a hacer es ver como quedaron los árboles abajo de él y a partir de ello en la variable state se indica si se encontró un error (en ese caso state es null) o sino en que estado quedó.

Ejemplo 1:

De la expresión

```
true == (a+b > 1*1.25)
```

resulta el siguiente árbol:
Todo esto se va haciendo a medida que se ejecutan las reglas en la gramática.

En cuanto a las delcaraciones, éstas van a ser almcenadas en las tablas de símbolos y además son traducidas como asignaciones para construir su árbol correspondiente.

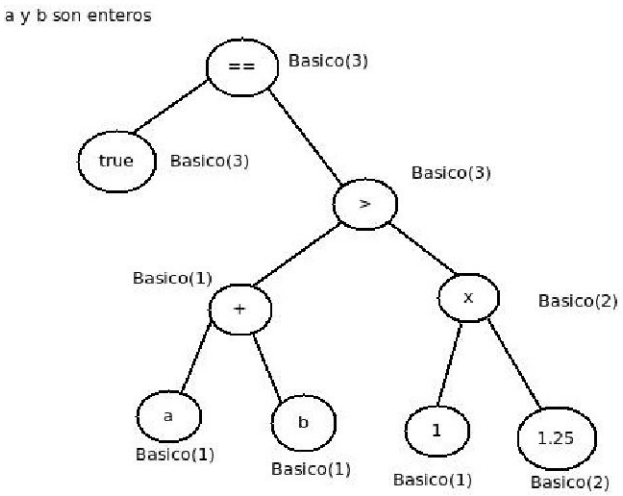

Figura 3.5: Ejemplo de construcción y chequeo de árboles

Otro detalle es que para lograr tener las declaraciones anidadas, lo que se hizo es tener dos tablas de símbolos: la actual y la anterior. Entonces siempre que se entre a un bloque, la tabla actual se vuelve una nueva, y la anterior la actual. Al salir vuelven al estado original.

Ejemplo 2:

Se tiene:

```
struct {
  float campo1;
}[5] a;

a[1].campo1 == 5
```

El árbol que se construye de la expresión dada es el siguiente:

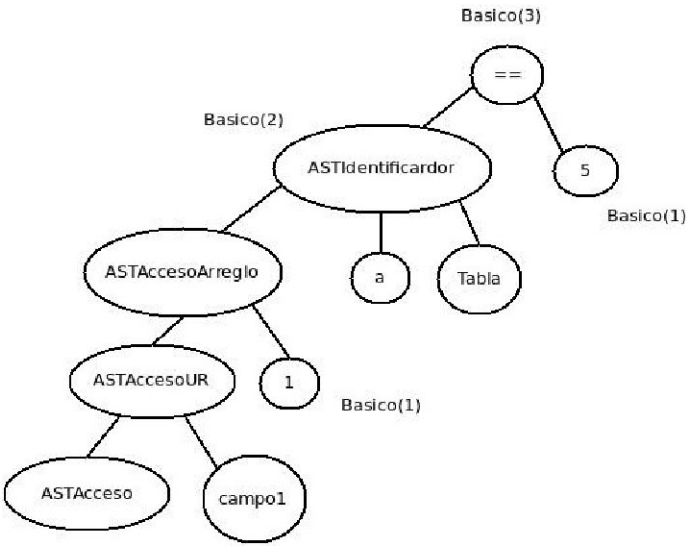

Figura 3.6: Ejemplo de construcción y chequeo de árboles

En los accesos se calcula el tipo resultante, en este caso un flotante. Luego se revisa si es posible equivaler eso con un entero, cosa que es posible.

Para calcular el tipo resultante de un acceso lo que se hace es ver el árbol de Tipo de a:

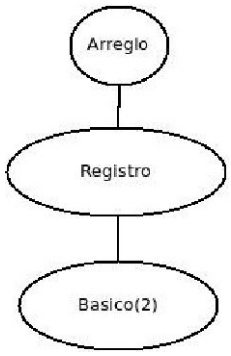

Figura 3.7: Arbol de tipo de la variable a

Se recorre al mismo tiempo que el árbol de acceso, con lo que se puede

determinar si el acceso es válido y cual es su resultado.

3.2.1. Generación de código

Se decidió que el código a generar es para arquitecturas de 64-bits, para ello se usan las instrucciones y sintáxis adecuada siguiendo la documentación de NASM.

Los registros del lenguaje son los siguientes: *rax, rbx, rcx, rdx, rdi, rsi, r8, r9, r10, r11, r12, r13, r14 y r15*. El stack pointer y el frame pointer son *rsp y rbp* respectivamente.

Esta información se tiene en una clase estática llamada *AssemblerInfo*. Además ésta posée procedimientos para salvar y restaurar registros, generar el código inicial del archivo de salida, entre otras.

Por otro lado cada uno de los AST tiene un método llamado *generateCode*; dependiendo de la clase, el número y tipo de parámetros varía. En el caso de las instrucciones, dicho método posee sólo dos parámetros: el primero es el descriptor del archivo de salida donde se va a escribir el código ensamblador y el segundo es el número de registro disponible para usar dentro de la generación de código de dicha instrucción.

El método de generación de código de las expresiones tiene éstos mismos dos parámetros descritos anteriormente, a los que se le suman otros dos, que son usados por las expresiones booleanas. Ambos representan "etiquetas", de *si* y *no*, que indican a que parte del código se debe saltar, si la expresión es verdadera, o falsa.

3.2.2. Cálculo de desplazamiento de variables

Para las variables globales simplemente se calcula el espacio que van a usar ya que éstas pueden ser usadas en cualquier parte del código. Y éste espacio es reservado estáticamente. En el archivo de salida, a este espacio se le asocia el identificador "static".

Por otro lado lo que se hace con las locales es calcular cual es la mayor cantidad de memoria que se va a necesitar en el programa siguiendo una cadena de bloques. Para ello se lleva en una variable "desplazamiento", el

espacio de cada variable declarada; junto con ésta variable, se usa una pila de enteros de la siguiente forma: cuando se entre un bloque nuevo se salva el valor del desplazamiento que se llevaba y al salir se recupera. De este modo se consume menos memoria y además se evita el problema de tener que estar empilando variables, que diferencia de las globales, se almacenan en la pila.

Ejemplo :

```
int a,b,c;
float d;

int main {
  int x,y,z;

  if (...) {
     int j,k,l;
     while (...) {
       int t,y,u;
     }
  }
  else {
     int o,p;
  }
}
```

Para éste caso se van a reservar 32 bytes para las variables globales, que corresponen a 4 bytes por cada una de las siguientes variables: a, b, c y d; y 72 bytes para el *main* y sus respectivas variables locales, que en el peor de los casos, si el flujo del programa se va por la rama del *if*, se requerirá espacio para las siguientes variables: x, y, z, j, k, l, t, y, u.

3.3. Casos de prueba

3.3.1. Criterios Generales

- Los tipos de prueba que se van a hacer son de caja negra, es decir, del comportamiento observable y externo del sistema.

- Se quiere simular las situaciones más comunes de un programador usando el lenguaje.
- Análisis de los casos bordes.

A continuación se muestran todos los casos de pruebas con sus respectivas intenciones y salidas esperadas.

- **Asignaciones:**

  Intención:

  Cumple adecuadamente tanto con las asignaciones globales, como con las locales de un prodecimiento, es decir que calcula de forma correcta los desplazamientos y coloca en memoria los valores de las expresiones evaluadas. Además se va a comprobar que las asignaciones múltiples, simples y en las expresiones funcionan adecuadamente.

Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
asignaciones1.lang	Asignación de variables globales	Impresión de un 3 y 6	Correcto
asignaciones2.lang	Asignación de variables locales	Impresión de un 4 y 16	Correcto
asignaciones3.lang	Asignaciónes entre variables globales y locales	Impresión de un 3 y 15.2	Correcto
asignaciones4.lang	Asignación mútiple	Impresión de un tres 49	Correcto
asignaciones5.lang	Asignación mútiple con diferentes tipos	Impresión de 45, 45, 45, 49.0, '-'	Correcto
asignaciones6.lang	Asignación como expresión	Impresión de 45	Correcto

- **Entrada/Salida:**

  Intención:

  La librerias proporcionadas por el lenguaje para que el programador tenga la capacidad de hacer entrada y salida funcionan tal cual como se prevée.

Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
es1.lang	Se leen adecuadamente los tipos básicos	Impresión de cada uno de los tipos básicos que se ingresó	Correcto
es2.lang	Se imprimen los diferentes tipos básicos	Impresión de 1, 2.3, true, false y 'z'	Correcto
es3.lang	Impresión de tipos compuestos	Dos errores indicando que sólo se puede imprimir y leer tipos básicos	Correcto

- **Expresiones:**

  Intención:

  Las expresiones con todos los diversos operadores y tipos funcionan adecuadamente tal cual como lo prevée la defición del lenguaje.

  - **Del tipo Bool:**

    Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
booleanos1.lang	Las expresiones simples funcionan	Impresión de true	Correcto
booleanos2.lang	Probar la fusión de todos los operadores booleanos	Impresión de flase	Correcto

- **Del tipo Char:**

  Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
caracteres1.lang	Expresión simple donde se asigna un caracter	Impresión de á'	Correcto

- **Del tipo Int:**

  Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
enteros1.lang	Expresión simple	Impresión de 3	Correcto
enteros2.lang	Expresión semi-compleja donde se requieren salvar registros	Impresión de 20	Correcto
enteros3.lang	División y módulo	Impresión de 2 y 1	Correcto
enteros4.lang	Expresión compleja	Impresión de -22	Correcto

- **Del tipo Float:**

  Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
flotantes1.lang	Expresión simple	Impresión de 3.4	Correcto
flotantes2.lang	Expresión semi-compleja donde se requieren salvar registros	Impresión de 20.8	Correcto
flotantes3.lang	División	Impresión de 2.0	Correcto
flotantes4.lang	Módulo no es permitido con los flotantes	Error indicando que la operación es inválida	Correcto

- **Generales:**

  Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
generales1.lang	Expresión con flotante y enteros	Impresión de 6.2	Correcto
generales2.lang	Expresión con flotante y enteros	Impresión de 6	Correcto
generales3.lang	Expresión booleana con flotantes, enteros y caracteres	Impresión de true	Correcto
generales4.lang	Expresión con char y enteros	Impresión de 'C'	Correcto

- **For:**

  Intención:

  Comprobar que la ejecución del código generado es la adecuada.

  Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
for1.lang	For sencillo con una asignación simple	Impresión de 0, 1, 2, 3, 4, 5, 6, 7, 8, 9	Correcto
for2.lang	For con un switch dentro	Impresión de 2, 5, 7 y 9	Correcto
for3.lang	Dos For's anidados	Impresión del 3 al 49 seis veces	Correcto

- **Funciones/Procedimientos:**

  Intención:

  Demostrar que la implemantación de las promesas funciona adecuadamente. Es decir, tanto la firma concuerte con la función/procedimiento como tal. También ver que se obliga a retornar en las funciones, que lo que se retorne es del tipo adecuado y que todo este bien declarado.

Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
fp1.lang	Los procedimientos y funciones se leen bien	Sin errores	Correcto
fp2.lang	Procedimiento ya declarado	Error	Correcto
fp3.lang	La funciones estan obligadas a retornar	Error	Correcto
fp4.lang	Función que retorna	Sin errores	Correcto
fp6.lang	Se debe retornar el tipo de la función	Error	Correcto
fp7.lang	Las promesas de declaracion de funciónes y procedimientos funciónan	Sin errores	Correcto
fp8.lang	No se cumple la promesa	Error	Correcto
fp9.lang	Llamada a función incorrecta	Error	Correcto
fp10lang	Solo se declara la promesa de una función o procedimiento pero nunca su cuerpo	Error	Correcto
fp11lang	Varios errores de promesas no declaradas y no cumplidas	Error	Correcto, aunque sólo imprime un error y no se recupera bien de éste
fp12lang	Error de tipos al usar una función en una expresión	Error	Correcto
fp13.lang	Llamada a procedimiento incorrecta	Error	Correcto

- **If:**

  Intención:

  Comprobar que la ejecución del código generado es la adecuada.

  Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
if1.lang	If sencillo para comprobar comparación de flotantes	Depende de la entrada. Con 3.2, imprime 3.2	Correcto
if2.lang	If simple	Imprime 3	Correcto
if3.lang	If simple con un else	Imprime 2	Correcto
if4.lang	If simple con varios elif	Imprime 0	Correcto
if5.lang	If anidados	Imprime 1 y 3	Correcto

- **Literales:**

  Intención:

  El perfecto funcionamiento de los literales para los tipos compuestos (arreglos, registros y uniones).

Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
literales1.lang	Literal de un arreglo simple	Sin errores	Correcto
literales2.lang	Literal de un struct	Sin errores	Correcto
literales3.lang	Literal de una union	Sin errores	Correcto
literales4.lang	Literal de una estructura con errores: los campos a asignar no concuerdan con la declaración de tipo.	Error	Correcto
literales5.lang	Sólo se puede asignar un campo de las uniones	Error	Correcto
literales6.lang	El campo al que se le quiere asignar no existe	Error	Correcto
literales7.lang	Declaración de un literal de una matriz	Sin errores	Correcto
literales8.lang	Error declarando una matriz, ya que un arreglo difiere de tamaño con el resto	Error	Correcto
literales9.lang	Error de tipos	Error	Correcto
literales10.lang	Error de tipos	Error	Correcto, pero da errores de más que son falsos

- **Switch:**

  Intención:

  Comprobar que la ejecución del código generado es la adecuada.

Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
switch1.lang	Switch simple	Impresión de 3	Correcto
switch2.lang	Switch simple con booleanos	Imprime 10	Correcto
switch3.lang	Switch con flotantes	Imprime 90	Correcto
switch4.lang	Switch con If's y For's	Imprime true	Correcto
switch5.lang	Switch anidados	Imprime 23	Correcto

- **While:**

  Intención:

  Comprobar que la ejecución del código generado es la adecuada.

  Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
while1.lang	While sencillo	Impresión de 1, 2, 3, 4, 5, 6, 7, 8, 9 y 10	Correcto
while2.lang	While sencillo con expresión booleana más compleja	Depende de la entrada. Con 10 imprime del 11 al 99	Correcto
while3.lang	While con If	Impresión de veintinueve 50's, un 100, veintinueve 80's y diecinueve 1000's	Correcto
while4.lang	While anidados	Depende del número ingresado	Correcto

3.4. Problemas encontrados y soluciones propuestas

El primer problema importante al que tuvimos que enfrentarnos fue que el uso de Java 1.4.2 complica el uso de tipos. Para se tuvo que hacer uso de muchos casts.

También el poco conocimiento de Jflex y JavaCup complicó en cierto modo el trabajo, ya que se perdió una gran cantidad de tiempo aprendiendo a usar estas herramientas. En especial JavaCup que dio muchos errores de tipo a las hora de ir armando los árboles. Para ello se le tuvo que indicar a éste, en la declaración de los no terminales de la gramática, qué devolvía cada producción y él se encarga de hacer los casts.

Capítulo 4

Detalles de compilación y corrida del lenguaje

El uso es bastante simple una vez descargado el archivo, abra el terminal y vaya a la ubicación donde lo descargó. Una vez hecho esto, va a necesitar descomprimir el archivo que esta en el formato .tar.gz. Para ello ejecute el siguente comando:

```
$> tar zxf nombre_archivo
```

Luego se tiene que ejecutar un script de bash que se encargara de compilar todos los archivos y generar el archivo de salida y además sejecutarlo.

```
$> sh makefile nombre_archivo_prueba nombre_archivo_salida
```

Es importante destacar que debe darse toda la ruta del archivo de prueba que desee ejecutarse. Además, el archivo de salida se creara con el nombre especificado por la cónsola, y se encontrará dentro de la carpeta NAMS.

Capítulo 5

Estado actual

5.1. Estado final del lenguaje

En lineas generales nuestro lenguaje está completamente operativo.

La gramática ya está completa y diseñada para los requisitos del trimestre pasado y esta primera entrega Se crean los árboles correspodientes para los tipos tanto básicos como compuestos y para las expresiones e instrucciones.

El lenguaje hace chequeos de variables no declaradas en su totalidad, siguiendo lo anteriormente mencionado en el caso de las instrucciones *if*, *for*, *while* y *switch*, donde las variables declaradas dentro de los bloques de cada una de estas instrucciones, sólo tendrán alcance dentro de estos mismos.

En cuanto al chequeo de tipos, se realizan chequeos de posibles coerciones válidas entre tipos iguales y distintos para el caso de tipos básicos, y las posibles asignaciones para tipos compuestos, incluyendo asignaciones de literales para cada uno de estos tipos compuestos.

En cuanto a la generación de código, se logró cubrir completamente la generación de código de expresiones de todos los tipos básicos, así como de las siguientes instrucciones: asignación, ambos condicionales *If*, y *Switch*, ambas iteraciones *For* y *While*. Las variables y expresiones utilizadas en todas estas intrucciones antes mencionadas, son de tipos básicos únicamente.

También cubrimos la declaración de funciones y procedimientos, total

o parcialmente. Esto es, permitimos al programador hacer promesas de declaración de subrutinas sin el cuerpo de las mismas, para permitir su uso y llamada dentro de otras subrutinas declaradas por debajo de ellas. Como consecuencia de ésta ventaja, se hace el chequeo correspondiente de la definición del cuerpo de promesas de subrutinas hechas.

Se agregó la funcionalidad de entrada y salida. Actualmente la E/S es a través de la cónsola. Todavía no tenemos E/S a través de archivos.

En la carpeta principal con los archivos del lenguaje, se encuentran también varios casos de prueba que se utilizaron para probar los chequeos antes mencionados. En lineas generales se hicieron casos de prueba donde se demuestra que los errores se estan detectando y mostrando por pantalla. También incluimos casos de prueba libres de errores.

5.2. Errores

Actualmente no está implementada la generación de código de variables de tipos compuestos, es decir, arreglos, registro e uniones.

Además en la generación de código de las iteraciones *For* y *While*, no se hace el chequeo de posibles *break*'s dentro de dichas instrucciones.

Otro problema que tenemos en relación a la generación de código en general, es que por alguna razón que aun desconocemos, el manejo de la pila del lenguaje ensamblador que escogimos, *NASM*, sólo puede hacerse a través de las instrucciones *Push* y *Pop*, y no explícitamente con el registro *Stack Pointer*.

Actualmente, ésto nos está generando complicaciones para la generación de código de tipos compuestos, ya que la declaración y almacenamiento de los mismos en la pila, requieren de mucho más espacio en la misma que lo que reserva un simple *Push* o *Pop*.

Capítulo 6

Conclusiones y recomendaciones

Al ir diseñando e implementando cada uno de los detalles del lenguaje uno se va percatando lo complicado que es esta tarea. El hecho de tener que tomar en cuenta una gigantesca cantidad de aspectos para tratar de cumplir con la mayor cantidad posible de ellos, conlleva a veces ha tomar decisiones erróneas. Uno se puede desviar del objetivo principal del lenguaje. Por ello se recomienda no crear algo con una gran cantidad de opciones de tipos, instrucciones, etc., ya que esto por el contrario, puede traer como consecuencia un mal diseño del lenguaje, debido al gran tamaño de la información que debe manejarse.

Otro detalle que es importante mencionar es que una vez pasado a la fase de implementación del lenguaje, es bueno tener un diseño claro, para no estar constamente cambiando el código. Ésto puede reducir significativamente la cantidad de errores. Es preferible perder varias horas para tener un diseño lo mejor pensado posible, a haber pasado a la fase de implementación y después tener que modificar algo del diseño en cada momento.

Una posible recomendación para la siguiente entrega es formular de una manera distinta y más sencilla, la construcción de los árboles de tipos compuestos. Actualmente, la manera en que tenemos formuladas las clases de *ASTAcceso* y las clases que extienden de ella, *ASTAccesoArreglo* y *ASTAccesoUR*, nos hicieron muy tediosa la generación de código de dichos *l-values* al punto que no nos dio suficiente tiempo para entregar la generación de código de ninguno de los accesos a tipos compuestos.

Capítulo 7

Bibliografía

- [1] http://www.jflex.de/manual.html.
- [2] http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html.
- [3] http://java.sun.com/j2se/1.4.2/docs/api/.
- [4] http://www.nasm.us/doc/.