



CNC
Informe Entrega II
Lenguajes de Programación II
Abril - Julio 2010

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

Universidad Simón Bolívar.
Caracas, Venezuela.
9 de julio de 2010

Índice general

1. Introducción	4
1.1. Motivación del proyecto	4
1.2. Breve descripción del problema	5
1.2.1. Palabras reservadas	6
1.2.2. Estructura general de un programa en CNC	6
1.3. Contenido del Informe	7
2. Diseño	8
2.1. Lineamientos	8
2.2. Operadores	8
2.3. Tipos Básicos	9
2.3.1. Int	9
2.3.2. Float	10
2.3.3. Char	11
2.3.4. Bool	12
2.4. Conversión de Tipos Básicos	13
2.4.1. Regla General	14
2.5. Tipos Compuestos	15
2.5.1. Arreglos	15
2.5.2. Registro	16
2.5.3. Unión	18
2.6. Instrucciones	20
2.6.1. Declaraciones	21
2.6.2. Asignación Simple	23
2.6.3. Asignación múltiple	23
2.6.4. Selección	23
2.6.5. Ciclos	25
2.7. Funciones y Procedimientos	26
2.8. Entrada y salida	30

2.8.1. Sintaxis:	30
2.8.2. Entrada:	30
2.8.3. Salida:	30
3. Implementación	31
3.1. Herramientas	31
3.1.1. JFlex y JavaCup	31
3.1.2. ESC/Java	32
3.2. Reseña de elementos implementados	35
3.2.1. Generación de código	44
3.2.2. Cálculo de desplazamiento de variables	49
3.3. Casos de prueba	50
3.3.1. Criterios Generales	50
3.4. Problemas encontrados y soluciones propuestas	58
4. Detalles de compilación y corrida del lenguaje	59
5. Estado actual	60
5.1. Estado final del lenguaje	60
5.2. Errores	61
6. Conclusiones y recomendaciones	63
7. Bibliografía	65

Índice de figuras

1.1. ENIAC	5
2.1. Tipo de almacenamiento	17
2.2. Tipo de almacenamiento	17
2.3. Tipo de almacenamiento de las uniones	19
3.1. Sym	36
3.2. Arbol de herencia de las instrucciones	37
3.3. Herencia de la clase ASTExpresion	39
3.4. Herencia de la clase ASTAcceso	40
3.5. Ejemplo de construcción y chequeo de árboles	42
3.6. Ejemplo de construcción y chequeo de árboles	43
3.7. Arbol de tipo de la variable a	43

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 “inherte” 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

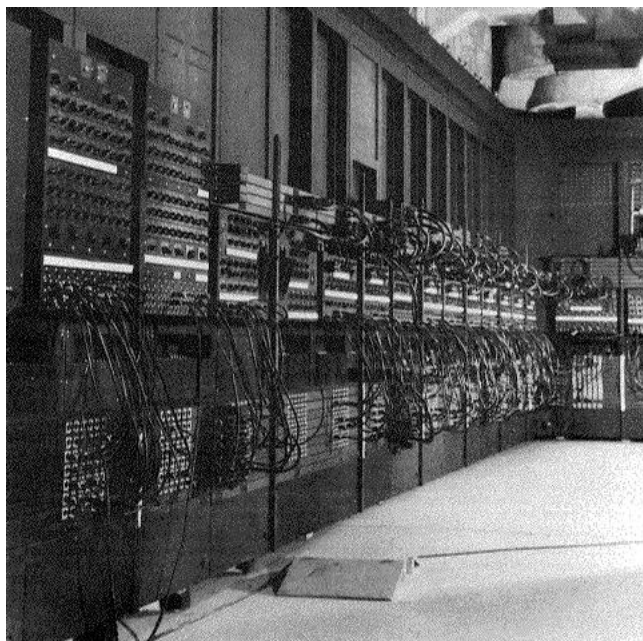


Figura 1.1: ENIAC

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 fases. Primero se crea un analizador lexicográfico o lexer, que va a reconocer todas las palabras reservadas para el lenguaje. Luego se crea un analizador sintético 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 sintético, 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 también están reservados como separadores del lenguaje:

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

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.
- Programa 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 líneas 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 sintaxis 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 tipado. 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

- **Declaración:**

```
int Identificador;
```

- **Detalles:** Con este tipo de datos podrán representarse los números enteros. Éste 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 (sólo se permite con identificadores):
++, --
 - 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

■ Declaración:

```
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(sólo se permite con identificadores):
 - ++, --
- 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

■ Declaración:

```
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 en memoria ya que la memoria está alineada siempre y los registros que utilizamos son todos de tamaño de 64-bits. Ésto evita el uso de mscaras y desplazamientos para obtener un valor.

En cuanto a los operadores de comparación estos simplemente van a usar el valor ascii. Es decir, si se compara á'¿'b', ésto devuelve falso ya que el código ascii de á es 97 y el de 'b' 98.

■ Operadores:

- Los operadores de comparacion cuyo resultado es del tipo *bool*:
 - Los operadores de comparacion:
 - <, <=, >, >=

- Los operadores de equivalencia:
`==, !=`
- Los operadores numéricos cuyo resultado es del tipo *char*:
 - Los operadores de incremento(sólo se permite con identificadores):
`++, --`
- 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

■ Declaración:

```
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
```

```
.
```

Al igual que con los caracteres, los booleanos van a ocupar 8 bytes a bajo nivel por el mismo motivo de hacer más fácil el manejo de los bits en memoria a la hora de generar código ya que la memoria está alineada siempre.

■ 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 coherción, 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

<, <=, >, >=

Para los operadores booleanos:

&&, ||

En este caso se indica el tipo resultante de asignar fila con columna. Para el operador de asignación, y los parámetros de llamadas a subrutinas:

La conversin de flotante a entero truncando el valor entero del flotante. En éste caso se pierde información.

2.4.1. Regla General

- **int:** Son compatibles con los ellos mismos, flotantes y caracteres.
- **float:** Son compatibles con los ellos mismos y enteros.
- **bool:** Sólo son compatibles con ellos mismos.
- **char:** Son compatibles con ellos mismos y los enteros.

Todo ésto siguiendo las reglas antes descritas.

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

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

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

- **Declaración:**

Tipo [Constante entera]...[Constante entera] Identificador;

- **Detalles:** Cada [Constante entera] representa una dimensión del arreglo y la expresión indica su tamaño. Además es importante mencionar que *tipo* puede ser cualquier 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 almacenar 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 asignables entre sí. Es decir que si se tiene un arreglo de flotantes y éste se asigna a un arreglo de enteros, lo que se va a hacer es ir moviendo casilla por casilla del primer arreglo al segundo haciendo los cast de flotante a entero.

- **Inicialización:**

Tipo [Constante entera1]...[Constante enteran] Identificador =
{ { {x, y, ... }; {p, q, ... }; ... }; ... ; { {z, w, ...}; ... } };

Donde la lista exterior contiene tantos elementos, a su vez listas, como indique la Constante entera1, así, cada una de estas listas será de tamaño igual a la Constante entera2, y así sucesivamente. Además x,y,p,q,z y w son expresiones.

- **Acceso:** Para acceder a cierta posición 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 sí.

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:

y no así:

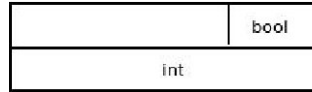


Figura 2.1: Tipo de almacenamiento



Figura 2.2: Tipo de almacenamiento

Claro ésta que en este ejemplo no va de acuerdo al lenguaje ya que los booleanos van a ocupar 8 bytes. Simplemente la idea era reflejar lo que es compactación. 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:

Con el almacenamiento del campo activo en memoria lo que se busca es evitar errores de asignación: asignar dos uniones con el campo activo diferente o también asignar a un campo activo que en verdad no lo está. 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.

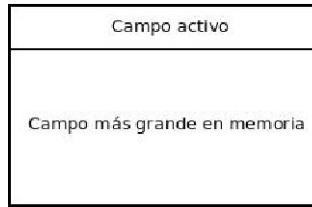


Figura 2.3: Tipo de almacenamiento de las uniones

■ Operadores:

- El operador de asignación:

=

Será válido siempre y cuando la cantidad de campos, el orden de los mismos (en el caso que se asigne una union con otra), los identificadores y el tipo asociado a cada uno de ellos, sean asignables en ambas de las uniones involucrados en la operación. Si se usa un literal de unin, el campo que se especifique en la asignación, será el campo activo de la unión.

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

- El operador de hasactive:

`hasactive`

La idea de este operador es permitir al programador en tiempo de ejecución saber si la unión tiene el campo dado activo. Éste devuelve un booleano. Su uso es de la siguiente forma:

```
Union hasactive Campo
```

Donde la el primer operando es el identificador de la unión y el segundo un identificador con el campo que se quiere saber si se encuentra activo. Un ejemplo real de uso seria el siguiente:

```
union{
    int a;
```

```

        bool b;
    } unionC;

    ...

    if(unionC hasactive b){
        ...
    }
    else{
        ...
    }

```

- **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 y además debe ser el campo activo, sino es un error.

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

- **Sintaxis:**

```
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 caso que no se inicialicen, no se tienen valores por defecto. Se debe revisar que las variables estén inicializadas a la hora de usarlas. 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 daría error.

La declaración de constantes sigue la misma sintaxis 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 eligió 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. Un detalle importante es que el identificador puede tener accesos.

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 instrucción el operador “=” es asociativo hacia la derecha. Y los identificadores pueden tener accesos.

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 casos deben ser 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:

- **Sintaxis:**

```

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 mayoría de los lenguajes imperativos. Es decir, mientras se cumpla una condición booleana se va a ejecutar el código del bloque.

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 sintaxis de C, C++ y Java porque es fácil de entender y usar. Además mantiene una sintaxis 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 global. 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, bien sea a un programa que lo llamo o por la salida standar, los 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.

■ 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 cuestión.

■ Promesas de procedimientos y funciones:

- **Sintaxis:**

- **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 la 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){
    ...
}
```

2.8. Entrada y salida

Ésta es sólo permitada para los tipos básicos.

2.8.1. Sintaxis:

2.8.2. Entrada:

En una asignación o inicialización se hace lo siguiente:

```
TipoBasico Identificador1, ... = read TipoBasico;
Identificador1, ... = read TipoBasico;
```

Se debe cumplir que los tipos sean asignables entre sí.

2.8.3. Salida:

Se pueden imprimir constantes o el valor de un identificador, si éste es de tipo básico.

```
print Constante;
print Identificador;
```

El identificador puede tener accesos.

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, parecida 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 desarrollo 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 árbol 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 convertir la tabla *anterior* en la tabla *actual* y ésta última va a ser una nueva tabla cuyo padre es la tabla *anterior* (es decir que antes era *actual*). Cuando se sale del anidamiento se hace el proceso completamente inverso: la *actual* pasa a ser la *anterior* y ésta última pasa a ser su tabla padre.

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 dada y su valor de retorno es booleano. En cambio el otro busca en todas las tablas hasta encontrar el primero,

donde retorna una tupla con el tipo y la tabla en que se encontró.

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:

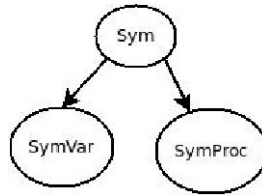


Figura 3.1: Sym

- **SymVar:** Representa un símbolo de una variable de algún 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 esto 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 tamaño en bytes que se necesita para las variables 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:

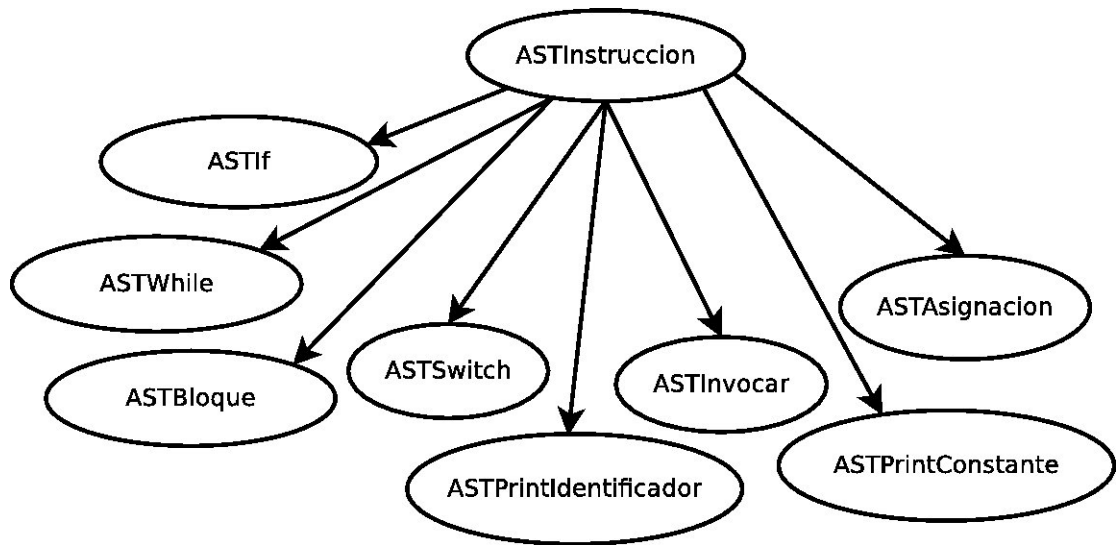


Figura 3.2: Arbol de herencia de las instrucciones

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 mayoría de las clases poseen un atributo Tipo llamado `state` que permite saber 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 asignación es de una declaración.
- **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.
- **ASTReturn:** Sus atributos son:
 - int offset: Desplazamiento con respecto al FP, donde se tiene que colocar el valor de retorno.

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.

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 válida o no.

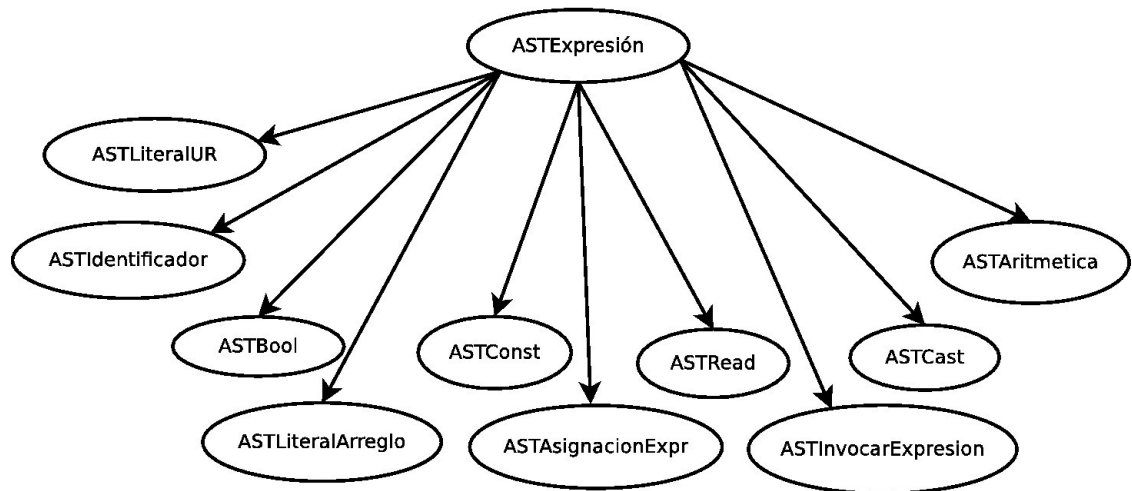


Figura 3.3: Herencia de la clase ASTExpresion

- **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.

Esta clase sólo es usada para adornar los árboles de expresiones, por lo que en el caso de una asignación o a la hora de empilar parametros los casts se tienen que generar. El problema por el cual no se adornó todo es porque existe la asignación múltiple y la de tipos compuestos, donde los casteos son mucho más complicados.

- **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 ASTAritmética, 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:

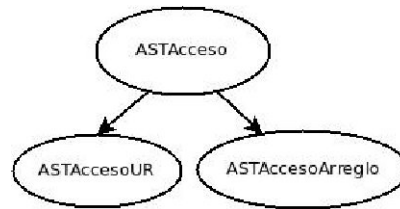


Figura 3.4: Herencia de la clase ASTAcceso

- **ASTAccesoUR:** Representa el acceso a un campo de un registro o una unión. Por ello sólo tiene un String que contiene el campo que se quiere accesar.
- **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” los AST de algunas expresiones bsicas. 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:

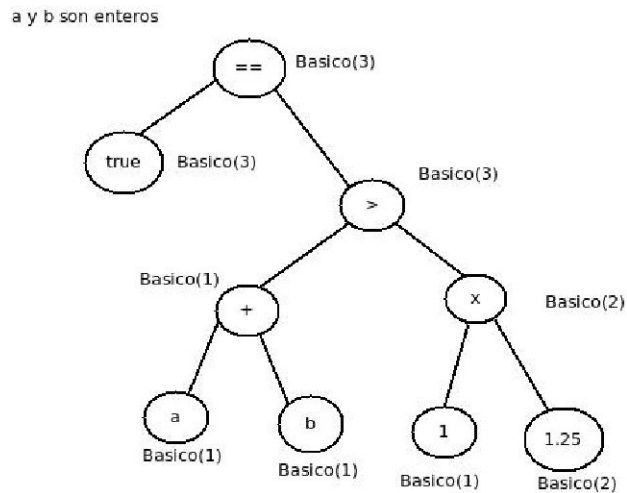


Figura 3.5: Ejemplo de construcción y chequeo de árboles

Todo esto se va haciendo a medida que se ejecutan las reglas en la gramática.

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

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:

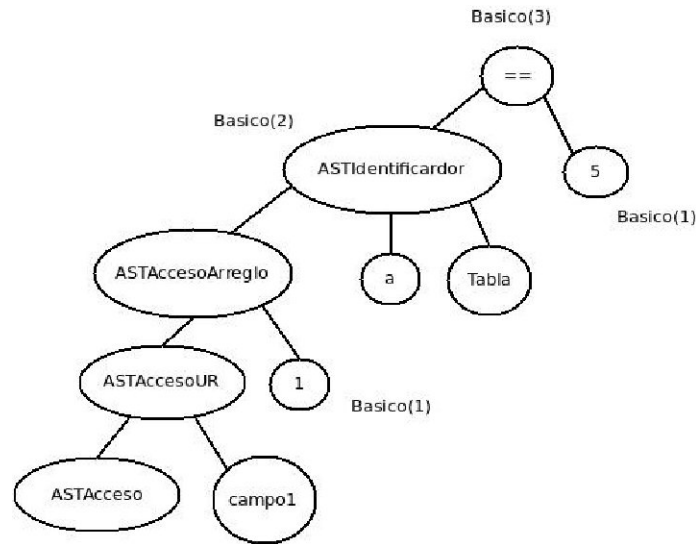


Figura 3.6: Ejemplo de construcción y chequeo de árboles

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 :

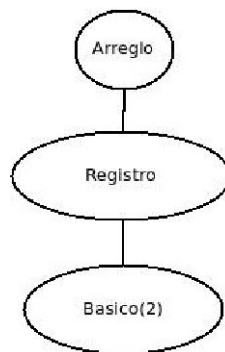


Figura 3.7: Arbol de tipo de la variable a

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 posee 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 cuatro parámetros: el primero es el descriptor del archivo de salida donde se va a escribir el código ensamblador, el segundo es el número de registro disponible para usar dentro de la generación de código de dicha instrucción, el tercero el nombre de la etiqueta cuando se quiere hacer un break y el último la etiqueta a donde hacer el salto de retorno de una función o procedimiento.

Para obtener nuevas etiquetas se hace uso de *newLabel* en la clase *AssemblerInfo*.

En el caso del ASTif el código se genera de la siguiente forma:

```
Iterator itc = cond.iterator();
Iterator itb = bloques.iterator();

String si = AssemblerInfo.newLabel();
String no = AssemblerInfo.newLabel();
String end = AssemblerInfo.newLabel();

((ASTExpresion) itc.next()).generateCode(fd,nextReg, si, no);
fd.write(si + ":\n");
((ASTBloque) itb.next()).generateCode(fd, nextReg);
```

```

fd.write("jmp " + end + "\n");

while(itc.hasNext()) {
    si = AssemblerInfo.newLabel();
    fd.write(no + ":\n");
    no = AssemblerInfo.newLabel();
    ((ASTExpresion) itc.next()).generateCode(fd, nextReg, si, no);
    fd.write(si + ":\n");
    ((ASTBloque) itb.next()).generateCode(fd, nextReg);
    fd.write("jmp " + end + "\n");
}

fd.write(no + ":\n");
if(els != null)
    els.generateCode(fd, nextReg);

fd.write(end + ":\n");

```

Éste funciona del siguiente modo:

- Se genera el código del primer if (este siempre va a estar). Para ellos se llama al primer condicional del iterador. Se genera el código del primer condicional. Se ve que se le pasan las dos etiquetas de salto ya que es una expresión booleana. Posterior se coloca la etiqueta si y el código a ejecutar del if, con un salto al final. Ésto son las siguientes líneas:

```

((ASTExpresion) itc.next()).generateCode(fd,nextReg, si, no);
fd.write(si + ":\n");
((ASTBloque) itb.next()).generateCode(fd, nextReg);
fd.write("jmp " + end + "\n");

```

- Luego se tienen que generar el código de las expresiones y bloques de los *elif*. Ésto es seguir recorriendo el iterador de los condicionales y bloques. A medida que se hace ésto se va generando el código. Se sigue el mismo esquema que se uso para el primer condicional, sólo que hay que agregar la etiqueta no para cuando una expresión booleana falle, salte a la siguiente. Corresponde a la siguiente parte:

```

while(itc.hasNext()) {
    si = AssemblerInfo.newLabel();
    fd.write(no + ":\n");
    no = AssemblerInfo.newLabel();
    ((ASTExpresion) itc.next()).generateCode(fd, nextReg, si, no);
    fd.write(si + ":\n");
    ((ASTBloque) itb.next()).generateCode(fd, nextReg);
    fd.write("jmp " + end + "\n");
}

```

- Finalmente ya sólo queda generar el código del else en el caso que se encuentre. Para ello se revisa que no sea nulo, y en el caso que se cumpla esto se genera su código. Y se coloca la etiqueta end, que es para cuando se halla ejecutado un bloque salte al final de todo el if. Sería lo siguiente:

```

fd.write(no + ":\n");
if(els != null)
    els.generateCode(fd, nextReg);

fd.write(end + ":\n");

```

En el caso del switch, while y for es muy parecido a ésta generación de código. En el caso del switch no hay ningún tipo de optimización. El while tiene una pequeña optimización pensando en el pipelining que tiene el procesador. Consiste en lo siguiente: primero ésta el código del bloque, luego viene una etiqueta y finalmente la expresión booleana. Así se disminuye el número de saltos necesarios.

La asignación si es mucho mas complicada ya que en el lenguaje se permite el uso de accesos y el poder asignar tipos complejos. Por lo que el código es super extenso y toma en cuenta muchos casos: asignar un identificador, un literal, una expresion booleana, los cast que hay que hacer, retorno de una función, entre otras cosas.

En el caso del break y return es simplemente generar una etiqueta que salte a los parametros que le llegan. El return si es mas complejo ya que

requiere en el caso que se encuentre en una función el colocar en el espacio de la pila para el retorno el valor de ésta. Por lo que el código es el mismo de la asignación con un salto.

En el caso de la invocación de un procedimiento o función se siguen las siguientes convenciones:

- **Llamador:** A partir del parámetro que le indica que registro puede usar, salva los registros necesarios. Por ejemplo si el número del próximo registro es el 5, entonces salva los primeros 5 registros. Luego se encarga de empilar los parámetros. Llama a la función. Desempila los parámetros. En el caso que tenga un valor de retorno, si este no es booleano simplemente lo deja en la pila. Y finalmente restaura los registros.
- **Llamado:** Primero guarda la dirección de retorno en la pila, luego el FP y lo actualiza. Después aumenta la pila con respecto a las variables locales. En esta implementación no se salvan registros ya que no se hace uso de registros para variables locales. Al salir se quita de la pila las variables locales, se restaura el antiguo FP y se salta a la dirección de retorno.

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.

En este caso el problema en que se incurrió fueron los flotantes. Tanto para las expresiones aritméticas como para las booleanas se tuvo que generar un código especial, donde se hace uso del FPU. También como las expresiones booleanas siempre hacen un salto en vez de retornar el valor. Esto agrego muchos casos.

La generación de código para una expresión aritmética binaria es la siguiente:

```
String reg = AssemblerInfo.getNombresRegAtPos(nextReg);  
String nreg = AssemblerInfo.getNombresRegAtPos(nextReg+1);
```



```

        left.generateCode(fd, nextReg, si, no);
AssemblerInfo.saveReg(fd, nextReg + 1);
right.generateCode(fd, nextReg + 1, si, no);
fd.write(OPERACION + reg + ", " + nreg + "\n");
    AssemblerInfo.restoreReg(fd, nextReg + 1);

```

Lo que se hace es simple: se genera el código del lado izquierdo usando el registro que se le asignó. Luego se llama el método `saveReg` de la clase `AssemblerInfo`, que revisa el número del registro que se está usando y de ser este mayor al número total de registros genera código para salvarlo. Luego se genera la expresión del lado derecho usando el siguiente registro, por eso el `nextReg + 1`. Finalmente lo que queda es hacer la operación entre los registros. Cabe actor que el método `getNombresRegAtPos` da el `String` del registro a partir de su número. Y bueno al final de ser necesario se restaura el registro, es exactamente el opuesto al método `saveReg`.

En el caso de los flotantes se tienen que cargar los valores a la pila de registros del FPU y hacer la operación adecuada y devolver el valor.

Se hace lo siguiente:

```

String reg = AssemblerInfo.getNombresRegAtPos(nextReg);
String nreg = AssemblerInfo.getNombresRegAtPos(nextReg+1);

left.generateCode(fd, nextReg, si, no);
AssemblerInfo.saveReg(fd, nextReg + 1);
right.generateCode(fd, nextReg + 1, si, no);

fd.write("push "+reg+"\n");
fd.write("fld qword ["+AssemblerInfo.getSp()+"]\n");

fd.write("push "+nreg+"\n");
fd.write("OPERACION qword ["+AssemblerInfo.getSp()+"]\n");

fd.write("fstp qword ["+AssemblerInfo.getSp()+" + 8]\n");

fd.write("pop " + nreg + "\n");

```

```
fd.write("pop " + reg + "\n");

AssemblerInfo.restoreReg(fd, nextReg + 1);
```

Es parecido al caso de los enteros, pero en este caso se tienen que cargar los valores del FPU y sólo se puede hacer desde memoria. Al igual se genera el código del lado izquierdo. Se salva el registro si es necesario, se genera el lado derecho. Ahora hace push del registro con el primer resultado. Se carga ese valor al FPU mediante la instrucción fld. Luego se coloca en memoria el segundo operando y se hace la operación. Finalmente se coloca el resultado en memoria, en este caso en la posición de memoria donde se encuentra el primer registro empujado, con el objetivo que cuando se haga pop de él ya tenga el resultado en él.

En el caso de las expresiones booleanas, *generateCode* se van a ver tres casos cuando se compara enteros o caracteres, flotantes y booleanos. Cada uno tiene su forma de generar el código. En cuanto a las aritméticas simplemente están el caso de los flotantes y enteros.

Por último como detalle importante es que cuando se usa una función como expresión esta va a dejar empujado su valor de retorno, exceptuando en el caso que sea un booleano.

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 empujando 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 corresponden a 8 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 principalmenteson 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.

■ **Llamadas:**

Intención:

Ver que la generación de funciones y procedimientos funcione adecuadamente.

Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
11.lang	Se quiere ver que las llamadas a funciones simples están funcionando	Impresión de a,3, a,7 y 9	Correcto
12.lang	Comprobar que efectivamente los parámetros son pasados por valor y que pueden ser modificados dentro del procedimiento	Impresión de 3,5, a y 3	Correcto
13.lang	Comprobar el pasaje por valor/resultado	Impresión de 3,3, 2 y 2	Correcto
14.lang	Ver que se hacen los casts en la llamada	Impresión de 3.0	Correcto
15.lang	Ver que los casts estén con los parametros valor/resultado	Impresión de 3.0, 4.2 y 4	Correcto

Caso de Prueba	Objetivo	Salida Esperada	Resultado
l6.lang	Pasaje de parametros de un arreglo sirve. Al igual que el uso de literales.	Impresión de 1	Correcto
l7.lang	El pasaje de parametros de literales es el adecuado	Impresión de 3	Correcto
l8.lang	Comprobar el pasaje por valor/resultado de estructuras complejas	Impresión de 5	Correcto
l9.lang	Comprobar que la recursin y el return con tipos bsicos sirve	Impresión del factorial número dado	Correcto
l10.lang	Comprobar que la recursin y el return con tipos bsicos sirve	Impresión del fibonacci número dado	Correcto
l11.lang	el return de tipos complejos sirve	Impresión 4, 11, 2.03 y h.	Incorrecto: Falla en la generación de código

■ Complejos:

Intención:

Ver que los accesos y asignaciones a los complejos funcionan de forma correcta

Casos:

Caso de Prueba	Objetivo	Salida Esperada	Resultado
c1.lang	Se desea comprobar que la instrucción hasctive funcione	Impresión de true	Correcto
c2.lang	Se quiere comprobar que las asignaciones en los arreglos	Impresión de 2,3, 4 y 1	Correcto
c3.lang	Ver que se indican los errores de indexación	Impresión del error	Correcto
c4.lang	Las asignaciones en los arreglos funcionan y adems que se indique si se sale del índice	Impresión de 3 y 8 seguido de un error	Correcto
c5.lang	Comprobar que las asignaciones en estrucutras, arreglos y uniones funcionan adecuadamente. Adems del uso de literales	Impresión de 16, true y 0.02	Correcto
c6.lang	Ver que se indican en los errores a la hora de usar un campo de la unión que no está activo.	Impresión de error	Correcto

■ **Oficiales:**

Intención:

Estos son los casos de prueba obligarios para todos los grupos traducidos a este lenguaje específico.

Casos:

Caso de Prueba	Resultado
fibIterativo.lang	Correcto
fibLogaritmico.lang	Correcto
generarPermutaciones.lang	Correcto
multiplicacionMatrices.lang	Correcto
raizCuadrada.lang	El programa se ejecuta sin problemas, pero los valores aproximados de la raíz cuadrada no son buenos
saludFinanciera.lang	El programa se ejecutó sin problemas es posible que tenga errores la eje- cución en vista que siempre dio que la empresa es saludable
variantes.lang	Incorrecto: no compiló

Para el caso de prueba de llamadas l5.lang se va ir mostrando paso a paso la correctitud de éste.

En el procedimiento principal que empieza con la etiqueta main primero se reserva memoria. Se reservan 8 bytes demás porque los desplazamientos empiezan en 8. El motivo es que no se puede tocar la sección de memoria con desplazamiento 0, la cual sería [rbp], el motivo es que posiblemente se guarde algo en esa región de memoria que es usado por las instrucciones enter y leave.

```
enter 16, 0
```

Luego se le asigna el valor tres a la variable a. Para ellos se coloca el 3 en el registro rax y la dirección de memoria de a, que es 8 de desplazamiento con respecto al rbp y se le asigna el valor.

```
mov rax, 3
mov rbx, rbp
sub rbx, 8
mov [rbx], rax
```

Posterior a ésto viene el empilamiento de parametros para ello se hace lo siguiente. Primero como se va a empilar la variable a, se saca su valor de la dirección de memoria. Eso son las tres primeras líneas. Luego se transforma ese valor a un flotante, ya que ese es el tipo del parámetro de entrada, haciendo uso de la instrucción `fild`. Luego se empila ese valor, y como el parametro es por valor resultado se empila la dirección de memoria y se llama al procedimiento.

```
mov rax, rbp
sub rax, 8
mov rbx, [rax]
push rbx
fild qword [rsp]
fstp qword [rsp]
pop rbx
push rbx
push rax
call procproc
```

Después de la llamada se desempila la dirección del registro en `rax`. Se saca el parametro en `rbx` y se procede a transformarlo a un entero. Para ello se hace uso de la instrucción `fistp`. Finalmente se almacena ese valor en la dirección de memoria.

```
pop rax
pop rbx
push rbx
fld qword [rsp]
fistp qword [rsp]
pop rbx
mov [rax], rbx
```

Queda por imprimir los valores para ello se salva el registro `rdi`, el cual es usado para pasar los parametros a imprimir a las funciones correspondientes. Primero se coloca el valor de `a` y luego el número 10 que corresponde al salto de línea en la tabla `ascii`.

```
push rdi
mov rax, rbp
```



```

sub rax, 8
mov rdi, [rax]
call print_int
pop rdi
push rdi
mov rdi, 10
call print_char
pop rdi

```

En cuanto al código del procedimiento que se llama, primero se reserva el espacio de memoria. En éste caso no hay variables locales, pero a pesar de eso se reservan 8 por lo dicho anteriormente de que los desplazamientos siempre van a comenzar en 8 por el problema que se tuvo. Luego se procede a imprimir su valor por lo cual su dirección se encuentra 24 bytes por encima del `rbp`. Se llama a la función de imprimir.

```

enter 8, 0
push rdi
mov rax, rbp
sub rax, -24
mov rdi, [rax]
call print_float
pop rdi
push rdi
mov rdi, 10
call print_char
pop rdi

```

Finalmente se le asigna el valor 4.2 a la variable `a` y se manda a imprimir. El valor 4.2 es transformado a un flotante de doble precisión el cual tiene como valor hexadecimal `0x4010cccccccccd`. Al terminar se ejecuta la instrucción `leave` que es la opuesta a `enter` y la instrucción `ret` que desempila la dirección de retorno y vuelve a donde antes estaba ejecutando.

```

mov rax, 0x4010cccccccccd
mov rbx, rbp
sub rbx, -24
mov [rbx], rax
push rdi
mov rax, rbp

```

```
sub rax, -24
mov rdi, [rax]
call print_float
pop rdi
push rdi
mov rdi, 10
call print_char
pop rdi
label0:
leave
ret
```

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 siguiente comando:

```
$> tar xzf 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 se ejecutarlo.

```
$> 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 consola, y se encontrará dentro de la carpeta NAMS.

Capítulo 5

Estado actual

5.1. Estado final del lenguaje

En líneas 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 correspondientes 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 instrucciones antes mencionadas, son de tipos básicos únicamente.

También cubrimos la declaración de funciones y procedimientos. 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 consola. 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 líneas generales se hicieron casos de prueba donde se demuestra que los errores se están 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 constantemente 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/>.