AUTÓMATAS, TEORÍA DE LENGUAJES Y COMPILADORES

1º CUATRIMESTRE 2017

TRABAJO PRÁCTICO ESPECIAL

GRUPO 2 - LA MAQUINARIA

CARLA BARRUFFALDI, 55421

LUCIANO BIANCHI, 56398

TOMÁS CERDÁ, 56281

MARCELO LYNCH, 56287

Índice

Índice 1

Introducción - Idea subyacente y objetivo del lenguaje. 2

Consideraciones realizadas 3
- Tipado dinámico 3
- Tipado débil 3
- Arreglos (listas) 3
- Funciones nativas 3
- Runtime environment 4
- Testing 4

Sintaxis 4
- Declaración y/o definición de variables 4
- Tipos de datos 4
  - Texto 4
  - Número 4
  - Lista 4
- Ciclo While 5
- Ciclo Until 5
- Sentencias de decisión 5
- Operadores relacionales 5
- Operadores aritméticos 5
  - Lista 6
    - Suma 6
    - Resta 6
  - Texto 6
    - Suma 6
    - Resta 6
    - Producto 6
    - División 6
- Operadores lógicos 7

Descripción del desarrollo del TP. 7

Descripción de la gramática. 10

Dificultades encontradas en el desarrollo del TP. 11

Futuras extensiones, con una breve descripción de la complejidad de cada una 12
- Extensiones del lenguaje. 12
  - Declaración y definición de funciones 12
  - Funciones que devuelvan valores 12
  - Nuevos elementos de control de flujo. 12
- Extensiones de la implementación. 12
  - Mayor expresividad para comunicar errores. 12
  - Mejor manejo de memoria. Garbage Collection. 13

Introducción - Idea subyacente y objetivo del lenguaje.

El presente informe detalla el desarrollo del lenguaje de programación que se decidió bautizar **prose**.

La filosofía de **prose** se resume en lo que podría ser su *slogan*: “el lenguaje de programación que puede usar tu abuela”. La idea fue implementar un lenguaje sencillo, como de scripting, en español y con el lenguaje más natural posible para definir la algoritmia del programa, al punto que a simple vista podría parecer pseudocódigo (en este sentido se asemeja a Python).

El Hola, Mundo de **prose** es un programa de una línea:

```
mostrar ‘¡Hola, mundo!’
```

La programación en **prose** es flexible y se acomoda al lenguaje natural, priorizando la legibilidad y entendimiento de un programa cualquiera con solo leerlo. Así, existe más de una forma de programar lo mismo. Por ejemplo, los siguientes son dos programas que resultan en la misma compilación:

```
que x valga 4
que y valga x + 2

suponga que "x es menor a y"
entonces mostrar x
listo.
```

Y:

```
que x valga 4
que y sea (x + 2)

si pasa que "x es menor que y" entonces:
      mostrar x
listo
```

En este mismo sentido, el lenguaje es de tipado dinámico y no es fuertemente tipado (el tipado no es 100% débil, pues hay algunas operaciones entre tipos distintos que sí fallan, pero se implementan la mayoría de las operaciones para intentar tener la menor cantidad de errores en ejecución).
El lenguaje está pensado como para scripting, es decir, no provee funcionalidad para que el usuario defina e invoque sus propias funciones. Sin embargo, se agregan funciones nativas para proveer mayor funcionalidad que solamente los operadores aritméticos.

Se detallan a continuación estas cuestiones.

Consideraciones realizadas

Se enumeran a continuación las decisiones especiales e inclusiones no previstas en el enunciado que se realizaron desarrollando el lenguaje. Los detalles de implementación asociados a los ítems que siguen se describen en el próximo apartado.

Tipado dinámico

En virtud de la misión del lenguaje se decidió desarrollar **`prose`** con tipado dinámico. Los tipos existentes son:

- texto (cadena de caracteres)
- número (internamente, enteros o de punto flotante)
- lista (colección ordenada de datos, no homogénea)

pero una variable puede guardar cualquier tipo de dato y ser definida sin declaración.

Tipado débil

Al mismo tiempo, el principio de que el lenguaje sea lo más amigable posible lleva a un intento de que las operaciones del usuario fallen lo menos posible. Así, el lenguaje es en esencia débilmente tipado: se permite por ejemplo "sumar" textos y números (resulta en un texto con la concatenación), restar textos (si `texto1` y `texto2` son textos, `texto1 - texto2` resulta en un texto con todas las ocurrencias de `texto2` en `texto1` eliminadas).

Sin embargo, no todas las operaciones pueden resolverse sin errores (como en algunos lenguajes débilmente tipados, que en lugar de error devuelven un valor especial *indefinido*, por ejemplo). Operaciones a las que no se les pudo asociar un sentido semántico razonable (u obvio) no se implementaron, y resultan en errores en tiempo de ejecución (y terminación del programa). Un ejemplo es si se quiere restar una lista de un texto.

Arreglos (listas)

`prose` provee el tipo de datos lista, análogo a un arreglo tradicional, que permite guardar una colección de datos ordenados. La lista no es homogénea, es decir, una misma lista puede alojar valores de distintos tipos.

Funciones nativas

Al no poder definirse funciones nuevas sino solamente programas individuales, se decidió proveer desde el lenguaje una serie de funciones nativas para que el programador tenga cierta funcionalidad más allá de la operatoria. Las funciones actualmente incorporadas tienen estas invocaciones, donde expresión es una expresión aritmética, un literal o una variable:

- `mostrar` `expresión` : imprime el resultado de `expresión` en pantalla
- `leer (numero | texto) a` `variable` : lee de entrada estándar hasta '\n' y guarda lo ingresado en `variable`. La variable no necesita estar declarada

previamente (pero debe ser una variable y no puede ser, por ejemplo, la posición de una lista).

- `anexar expresión a variable` : Equivalente a hacer `que variable valga variable + (expresión)`, aunque variable debe ser de tipo texto o lista
- `incrementar variable` **:** Equivalente a hacer que variable valga variable + 1. Debe ser de tipo número.
- `decrementar variable` **: Análogo a incrementar.**
- `pasar variable a mayúscula[s]` **:** Pasa los caracteres de variable a mayúsculas. Variable debe ser de tipo texto.
- `pasar variable a minúscula[s]`**:** Análogo a pasar a mayúscula.

Runtime environment

Para lograr lo especificado en los puntos anteriores se programó una especie de *runtime environment* en C con el que se compila el código arrojado por YACC: en definitiva, todo el manejo de variables y la operatoria se realiza mediante llamadas a estas funciones, es decir, se resuelven en tiempo de ejecución. Por este motivo no hay *type safety* en compilación.

Testing

Para testear el lenguaje, se desarrolló un pequeño script de Ruby en el cual se especifica el nombre del archivo de **`prose`** a ejecutar, y el valor que se espera leer por salida estándar, ya que por el momento éste es el único punto de acceso que se tiene a un programa escrito en **`prose`**. El objetivo de ésto fue hacer que los tests dependan lo menos posible del funcionamiento del lenguaje, ya que es justamente lo que se quiere probar.

Sintaxis

Declaración y/o definición de variables

```
que nombre_variable (sea | valga) expresión
```

Tipos de datos

Texto

Entre comillas simples: `que hola valga 'Hola mundo'`

Número

Puede o no ser entero: `que pi sea 3.14`

Lista

Se declaran con `'[ ]'`, y sus elementos separados por `','`. Se acceden como en C.

```
que lista valga [1, 2, 3]
```

```
mostrar lista[1] => '2'
```

Ciclo While

```
mientras que "expresión booleana" (hacer: | entonces:)
     bloque de sentencias
y repetir.
```

Ciclo Until

```
hasta que pase "expresión booleana" (hacer: | entonces:)
     bloque de sentencias
y repetir.
```

Sentencias de decisión

```
(si pasa que | suponga que) "expresión booleana" (hacer:|entonces:)
     bloque de sentencias
[ en cambio si "expresión booleana"
     bloque de sentencias
] *
[ (si no: | de otro modo: | en otro caso:)
     bloque de sentencias
]
listo.
```

Operadores relacionales

Similares a los operadores tradicionales, sólo que en lugar de usar símbolos, se expresan coloquialmente.

```
v1 < v2 : v1 es menor (a|que) v2
v1 <= v2 : v1 es menor (a|que) o (es | vale) v2
Análogamente con mayores reemplazando menor por mayor.

v1 == v2 : v1 (es | vale) v2
v1 != v2 : v1 no (es | vale) v2
```

Operadores aritméticos

`Operadores soportados: *, +, -, /, - (unario)`
En este caso son iguales a los lenguajes tradicionales, salvo en las operaciones que no tienen comportamientos estandarizados, como al hacer `número / texto`.

A su vez, se soportan las siguientes operaciones inmutables con respecto a listas y textos no tan tradicionales:

Lista

Suma

```
que resultado sea lista1 + lista2
```

Guarda en `resultado` una nueva lista con los elementos de la `lista1` seguidos de los elementos de la `lista2`.

```
que resultado sea lista + elemento
```

Guarda en `resultado` una nueva lista con los elementos de `lista` seguido de `elemento`, donde `elemento` corresponde a un tipo de datos distinto al tipo lista.

Resta

```
que resultado sea lista1 - lista2
```

Guarda en `resultado` una nueva lista con los elementos de la `lista1` que no están contenidos en la `lista2`.

```
que resultado sea lista - elemento
```

Guarda en `resultado` una nueva lista con los elementos de `lista` sin la presencia de `elemento`, donde `elemento` corresponde a un tipo de datos distinto al tipo lista.

Texto

Suma

```
que resultado sea texto + elemento
```

Guarda en `resultado` la concatenación entre `texto` y la representación textual de `elemento`. `elemento` no puede ser del tipo lista.

Resta

```
que resultado sea texto1 - texto2
```

Guarda en `resultado` un nuevo texto que contiene el texto `texto1` sin la presencia de `texto2`.g

Producto

```
que resultado sea texto * número
```

Guarda en `resultado` la concatenación de `texto` con si mismo `número` veces.

División

```
que resultado sea texto / número
```

Guarda en `resultado` una nueva lista con subtextos de longitud `número`, excepto el último elemento que es de longitud *resto(longitud(texto) / número)* si el resto es mayor a 0.

Operadores lógicos

Nuevamente, son similares a los de los lenguajes tradicionales, pero se expresan coloquialmente.

```
e1 AND e2: e1 y ademas e2
e1 OR e2: e1 o bien e2
NOT (e1): no (e1)
```

Cabe aclarar que todas las palabras resaltadas previamente son *palabras reservadas* del lenguaje. Los símbolos '**:**' y '**.**' son opcionales en todos los casos. Las sentencias se terminan con '\n', y los espacios en blanco que no son esenciales son ignorados.

Descripción del desarrollo del TP.

Lo primero que se realizó fueron las producciones de la gramática, (que se exhiben en el siguiente apartado). El lenguaje se pensó originalmente como tipado, pero a la hora de implementarlo se ideó la forma de lograr el tipado dinámico, que resultó mejor alineado a los objetivos de fondo ya descritos.

La tokenización desde LEX se pensó para lograr el estilo de prosa que caracteriza a **`prose:`** es por ese motivo que más de una expresión *matchea* con el mismo token.

La compilación que se implementó es de prose al lenguaje C. La mayoría de las sentencias escritas en prose terminan en C como llamadas a funciones del runtime environment (escritas también en C), que realizan las acciones necesarias (por ejemplo, una asignación termina utilizando la función `assign` del runtime environment).

Para lograr el código C se genera un árbol sintáctico abstracto (AST, por sus siglas en inglés), que representa la estructura y valores concretos del programa que escribió el programador. Luego de generar el árbol, el código se genera recorriéndolo desde la raíz en un recorrido DFS. Los nodos del árbol tienen distintos tipos según lo que estén representando. El archivo `ast.h` contiene la definición de las estructuras correspondientes y la documentación asociada.

Como ejemplo, consideremos el programa:

```
que x valga 5
mientras que pase "x es menor a 7"
      incrementar x
y repetir.
```

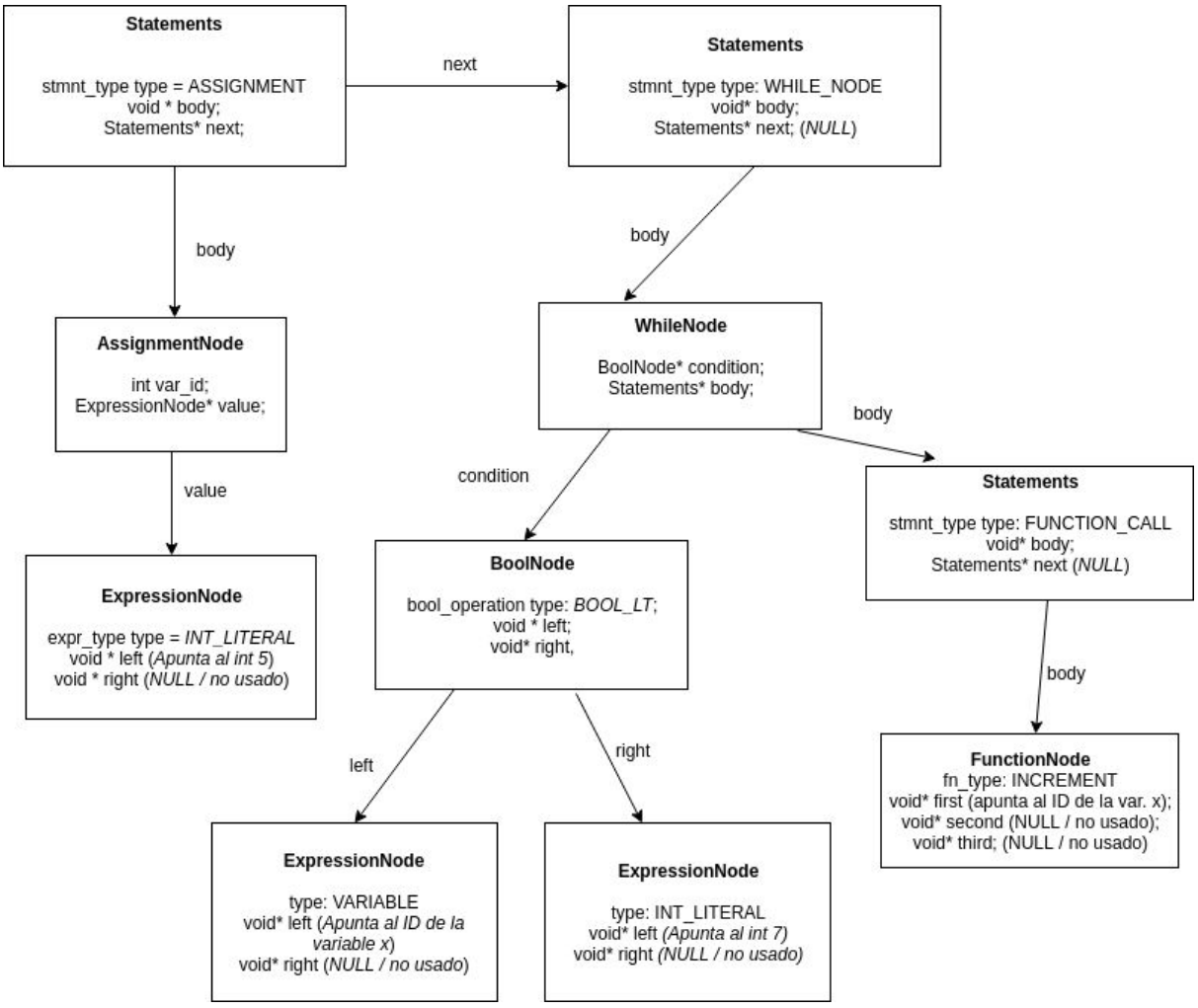
A partir de este programa se construye el AST que se muestra en la siguiente página.

Como se ve, el árbol, en su más alto nivel, es una lista encadenada de `Statements`. En este caso son los correspondientes a la asignación y al bucle while. Esto (como todo el resto del árbol) refleja la gramática (referirse a la sección *Descripción de la gramática*): el programa, desde el símbolo inicial, genera una estructura encadenada de `block`s (que se reducen a `Statements`):

```
program:  block | block program
```

En efecto, el árbol se construye a partir de las reducciones de la gramática.

Muchos nodos tienen punteros a `void` por poder apuntar a más de un tipo de nodo hijo: por ejemplo, los campos `left` y `right` de un `ExpressionNode` pueden apuntar a dos nodos distintos de expresión (por ejemplo, en el caso de una expresión aritmética binaria como la suma, apuntaría a las dos expresiones que están siendo sumadas, y campo `type` sería `ARIT_SUM`), directamente a un valor literal (un literal es una expresión), o a un ID de variable. Estos últimos dos casos se presentan en el ejemplo.



La generación del árbol se hace de abajo hacia arriba, como corresponde al analizador ascendente de YACC. En el código asociado a todas las reducciones se reserva espacio para el nodo y se asignan los punteros a los hijos. Por ejemplo, cuando se reduce la condición del `mientras que`, se aplica:

```
condition    : expression bool_comp expression
                    {
                          $$ = malloc(sizeof(*$$));
                          $$->type = $2;

                          $$->left = $1;
                          $$->right = $3;
```

```
                        }
```

que se observa claramente en el árbol ilustrado. En la última reducción de todas se invoca la función que recorrerá el AST y producirá el output en código C:

```
entry : program { produce($1); }
```

donde `$1` guarda el puntero al primer `Statements` del árbol.

En la compilación se mantiene una tabla que mapea los identificadores de variables que define el usuario a un identificador numérico, que es el que usa después el runtime para identificar a una variable definida por el usuario (“*variables nombradas”*). Por esto el nodo de asignación contiene un id y una expresión: se asignará el resultado de esa expresión a la variable asociada a ese identificador (que en este caso corresponde a lo que el usuario programó como “x”).

La función `produce` y sus funciones asociadas, que producen el código C a partir del AST se encuentran en el archivo `ast.c`. La producción de código se hace a partir de un recorrido DFS del árbol, casteando los punteros a `void` apropiadamente según el tipo de nodo que se está procesando.

Esta función termina produciendo el siguiente código C (que saca por salida estándar):

```
int main(void) {
	map_name(0, "x");
	assign(0, anon_int(5));

	while((compare(get_var(0), anon_int(7)) < 0)){
		inc(0);
	}
}
```

Como ya notamos, la mayor parte de la operatoria pasa *under the hood*, en las funciones del runtime. Esto es la consecuencia natural del tipado dinámico y la sobrecarga de los operadores, que es imposible emular directamente en el código.

De hecho, la mayoría de las funciones del runtime simplemente chequean de qué tipos son los objetos sobre los que se quiere operar (ya sea una suma, una comparación) y realizan la operación primitiva correspondiente según eso.

El objeto esencial de prose es `VAR`, una estructura que representa un valor de algún tipo:

```
	typedef union{
		int intValue;
		float floatValue;
		char* strValue;
		void* arrValue;
	} varValue;

	typedef struct{
		varValue value;
		type_t type;
```

```
} VAR;
```

Como se ve, el valor de un `VAR` es un `union` que puede ser un int, float, char* y void*, para los tipos numéricos, texto y lista respectivamente, y cambiar dinámicamente en ejecución junto con el tipo `type` (`INT_T, STR_T, FLOAT_T, ARRAY_T`).

Todas las expresiones de `prose` resultan en un objeto `VAR`. Los archivos `variable_manager.c`, `variable_arithmetics.c` y `variable_comparisons.c` contienen todas las funciones para crear y operar con VARs y la documentación correspondiente. Como observación pertinente, en la documentación se utiliza la palabra *variable* como sinónimo de un objeto VAR, y *variable nombrada* para referirse a las variables definidas en el programa (que tienen un ID asociado). Además, el archivo `prose_arrays.c` contiene funciones para crear y manejar los arreglos que subyacen al tipo lista.

Finalmente, `prose_functions.c` contiene el código y documentación de las funciones nativas.

Descripción de la gramática.

Se presenta a continuación la gramática que compone **`prose`**. Los no terminales están en minúscula.

```
entry: program;

program:  block | block program

block: asig | arr_asig | print | while | if | func_call | exit

exit: EXIT

asig: QUE IDENTIFIER VALGA expression

arr_asig: QUE arr_indexing VALGA expression

expression: STR  | NUM | FLOAT | array  | IDENTIFIER | arr_indexing
            | expression '+' expression
            | expression '*' expression
            | expression '-' expression
            | expression '/' expression
            | '-' expression
            | '(' expression ')'

func_call: ANEXAR expression A IDENTIFIER
              | PASAR IDENTIFIER A MAYUSCULA
              | PASAR IDENTIFIER A MINUSCULA
              | INCREMENTAR IDENTIFIER
              | DECREMENTAR IDENTIFIER
              | READ NUMERO A IDENTIFIER
              | READ TEXTO A IDENTIFIER


array: '[' explist ']'
```

```
arr_indexing: IDENTIFIER '[' expression ']' | arr_indexing '[' expression ']'

explist: expression | expression ',' explist

print: PRINT expression

while: WHILE SEP condition SEP DO program WEND
            | UNTIL SEP condition SEP DO program WEND

if :  IF SEP condition SEP DO program END
      | IF SEP condition SEP DO program elseif END


elseif : ELSEIF SEP condition SEP program
      | ELSEIF SEP condition SEP program elseif
      | ELSE program

condition    : expression bool_comp expression
                  | condition OR condition
                  | condition AND condition
                  | NOT condition

bool_comp    : LE | LT | GT | GE| EQ | NEQ
```

Los símbolos terminales son tokens definidos en YACC para LEX. El parsing (y por lo tanto las expresiones reservadas, que son las que matchean con algo en LEX) pueden verse en el archivo `parser.l`. Se recuerda que algunos tokens resultan de expresiones distintas, por ejemplo, el token UNTIL resulta de escribir en **`prose`** *mientras que no*, *hasta que* o *hasta que pase*.

Dificultades encontradas en el desarrollo del TP.

La mayor complejidad quizás surgió en la manera de desarrollar el árbol sintáctico abstracto para la una mejor eficacia a la hora de procesarlo: un desafío interesante se dio con las listas, en donde expresiones donde se indizaban listas dentro de otras listas (como `mostrar miLista[1][2]`), o asignaciones de esos índices (`que miLista[2] valga 4`), que requirió agregar producciones a la gramática (`arr_indexing` **y** `arr_asig`), nuevos nodos al AST y una manera distinta de procesarlos.

El manejo de memoria es bastante siempre es complicado el manejo en C, más aún cuando se comparten entidades reservadas desde malloc entre tantos componentes (LEX, YACC, el constructor de sentencias a partir del AST, etc). Es posible que exista algún memory leak a pesar de los esfuerzos en ese respecto.

Futuras extensiones, con una breve descripción de la complejidad de cada una

Extensiones del lenguaje.

Declaración y definición de funciones

Actualmente las únicas funciones soportadas son las funciones nativas. Soportar la declaración y definición de funciones requeriría de incorporar sintaxis para la declaración e invocación de las mismas tal que no desentone con el espíritu de prose.

En cuanto a la implementación, se necesitaría mapear el nombre de una función con su bloque de código en una estructura y ejecutarlo en la ocurrencia de dicho nombre, como también refactorear las funciones actuales de **`prose`** para que se comporten de dicha forma.

Funciones que devuelvan valores

A su vez, se podría modificar la gramática para soportar que las funciones devuelvan valores. Actualmente, las funciones nativas del lenguaje no retornan ningún valor sino que modifican el valor que se les es pasado si es necesario. Al mismo tiempo, el programa termina sin valor de retorno, y la única forma de devolver valores es mostrándolos en pantalla. Esto tiene sentido teniendo en cuenta el objetivo del lenguaje, pero podría agregarse un valor de retorno del programa.

Nuevos elementos de control de flujo.

Comunes en casi todos los lenguajes, como los for-loop, los iteradores y los switch. Esto implica un desafío por el lado de incorporar una sintaxis acorde al espíritu de **prose** como a su vez un desafío para su implementación.

Para la misma en los casos del for-loop y switch bastaría con reutlizar las lógicas provistas por el while y los else-if. En cuanto a un iterador se podría reutilizar el hipotético for-loop, asignando el valor del actual elemento del iterable (texto y lista) a una variable y ejecutar el bloque el código.

Extensiones de la implementación.

Mayor expresividad para comunicar errores.

Siguiendo la lógica de hacer el lenguaje lo más usable posible, es importante comunicar de manera clara y legible los posibles errores de compilación o de runtime. Existe un mapeo de los id de variables a los identificadores que se produce en tiempo de compilación y eso a veces sirve, pero muchas veces se pierde el hecho de que el valor surgió de una variable nombrada y no de un literal al llamar a una función del *runtime*, que entonces no puede mostrar un mensaje de error muy comprensivo.

Por ejemplo, la función `compareToInt` detecta que se quiere comparar con un string, no sabe si la invocación fue provocada por comparar la variable `miTexto` de tipo texto con una variable `miNumero` de tipo numero, o por comparar los literales `7` y `'texto'`, y por ende el

mensaje de error es simplemente `Error. Valores incomparables: número y texto.`

Un manejo de errores más comprensivo implicaría complejizar las funciones del runtime y perder generalidad, o hacer más chequeos en tiempo de compilación (que agrega estructuras adicionales de control, etc).

Mejor manejo de memoria. Garbage Collection.

Actualmente, prose no dispone de ningún mecanismo de manejo de memoria. Las variables definidas quedan siempre en memoria a pesar de que no vuelvan a ser usadas. Además, el proceso de compilación y el manejador de variables utiliza muchas estructuras y reserva memoria constantemente, en un proceso que seguramente tenga lugar a refinaciones.

La implementación de un garbage collector no es trivial y requiere analizar estáticamente el código y llevar registros y tablas de variables y su uso. Un mejor manejo de los ids de las variables desde el compilador (reusando los ids numéricos si un identificador no vuelve a usarse) sería una buena mejora en cuanto a la eficiencia, pero requiere análisis estático de código o hacer más de una pasada sobre el mismo.