Instituto Tecnológico de Buenos Aires

Autómatas, Teoría de Lenguaje y Compiladores

72.39

Trabajo Práctico Especial

Matías Apablaza, 59714

Juan Ignacio Sackmann Sala, 60340

Inés Marcarian, 60016

1er Cuatrimestre

2021

Índice general

1. Introducción

El objetivo de este trabajo fue crear un lenguaje de programación con el conocimiento obtenido en la materia. El siguiente informe trata sobre el lenguaje de programación StatistiC. Dicho lenguaje esta orientado a la estadística y provee funcionalidades built-in útiles para el desarrollo de código orientado a la estadística. El objetivo de dicho lenguaje es proveer un lenguaje fácil e intuitivo de usar como Python pero al mismo tiempo un lenguaje poderoso como el de C. Por lo tanto se decidió utilizar a C como lenguaje intermediario.

2. Consideraciones realizadas

En cuanto al punto de la consigna que decía que debíamos tener un mecanismo para indicar cuál es el punto de entrada decidimos utilizar la palabra "start" como mecanismo de comienzo pero todo lo que se encuentra por arriba de dicha linea es ignorado a menos que sea una declaración o asignación de una variable. Esto se puede probar con el ejemplo del archivo "examples/warnings.stc" el cual tira warnings.

3. Descripción del desarrollo

El trabajo consiste de 4 partes de procesamiento, el lexer (Lex), el parser (YACC), el lookup de variables y el AST (Abstract Syntax Tree) junto a la traducción del árbol.

3.1. Lexer y Parser

El lexer y el parser son utilizados en conjunto. En el parser se establece la gramática del lenguaje mientras que en el lexer se establece la sintaxis de los tokens que se utilizan en el parser. En caso de error en el momento de parseo se imprime a stderr la linea donde esta el error, el token que causa el error y el mensaje de error. También pueden surgir warnings debido a código no permito fuera del start. El parser es el encargado de ir armando el árbol que luego será utilizado para hacer la traducción al lenguaje C. Dicho árbol consiste de un "tronco" que vendría a ser una lista de nodos de las cuales pueden salir "ramas" que vendrían a ser nodos de instrucciones los cuales pueden contener distintos tipos de nodos, "hojas" o "sub ramas". Un ejemplo de como sería el árbol es, para el siguiente código:

```
start
    number n1 = 5
    write $n1 * 4
```

El diagrama sería:

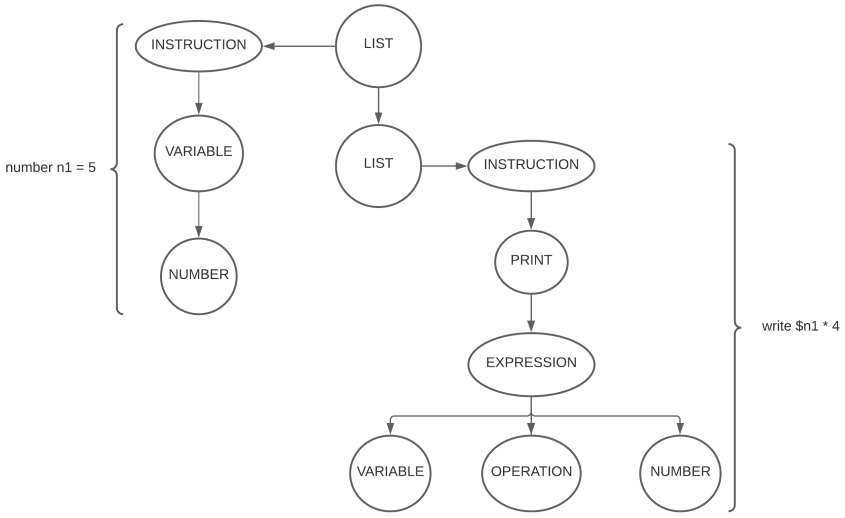

Figura 1: Diagrama del árbol para el programa previamente escrito.

La gramática establecida en el parser en el archivo "parser.y" se puede ver en las sección 4.

3.2. Lookup de variables

Luego de haber parseado el archivo de entrada, se realiza el lookup de variables cuyo objetivo es validar el correcto uso de la variables, chequeando que se cumpla el scope y que la asignación de la variable cumpla con el tipo de variable.

La manera en la que se hace esto es conceptualmente sencilla, se recorre el árbol armado previamente y en caso de encontrar declaraciones de variables, estas se agregan a una lista que es utilizada para verificar que las operaciones que invocan las variables sean correctas. Esta "lista" es consciente de la localidad de las variables. De encontrar alguna inconsistencia, imprime un mensaje de error y frena la compilación del programa.

Es importante aclarar que los mensajes de error son bastante rudimentarios, y que aunque el compilador encuentre errores, este va a continuar hasta haber visto todo el programa. Esto podría llegar a causar que errores que no sean el primero sean erróneos/falsos positivos.

3.3. Traducción del lenguaje

Luego de haber pasado por el lookup de variables se realiza la traducción del lenguaje StatistiC al lenguaje C. Esto se realiza recorriendo el árbol que produjo el parser. A medida que se encuentra un nodo de un tipo en específico se llama a una función particular para cada nodo la cual se encarga de traducir dicho nodo al lenguaje C. Cada vez que se termina de leer un nodo este se libera.

4. Gramática

program → instruction program || EOL program || FIN
instruction → full_declare || assign || list_full_declare || list_mutate || write || read || if || while || START
block → instruction block || EOL block || instruction || EOL
if → IF expression DO block if_end
while → WHILE expression DO block END
if_end → END || ELSE block END
full_declare → declare || declare ASSIGN value
declare → type SYMBOL_NAME
type → NUMBER_TYPE || TEXT_TYPE
assign → SYMBOL_NAME ASSIGN value
value → expression || SYMBOL_NAME || TEXT
list_full_declare → LIST_TYPE SYMBOL_NAME SIZE NUMBER || LIST_TYPE SYMBOL_NAME ASSIGN value
list_mutate → SYMBOL_NAME '(' expression ')' ASSIGN expression
write → WRITE expression || WRITE SYMBOL_NAME || WRITE TEXT || WRITE LIST
read → READ SYMBOL_NAME
expression → (expression) || UNI_OP expression || - expression || expression BIN_OP expression || expression - expression || expression POWER expression || expresion FACT || list_operation || NUMBER || $ SYMBOL_NAME
list_operation → MEASURE_OF list_value || list_value (NUMBER) || NUMBER IN list_value
list_value → SYMBOL_NAME || LIST

Donde los terminales están escritos con mayúsculas y los no terminales con minúsculas. Los no terminales significan:
program: es el programa.
instruction: es una linea de instrucción.
block: es una linea de instrucción dentro de un if o un while.
if_end: son las distintas formas de continuar con un if, terminando o poniendo un else.
full_declare: es la declaración y asignación de una variable.
declare: es la declaración de una variable.
type: son los distintos tipos de variables.
assign: es la asignación de una variable.
value: son los posibles valores de una variable.
write: función de impresión a STDOUT.
read: función de lectura de STDIN.
expression: estructura de las posibles expresiones numéricas.
list_full_declare: declaración de una variable de tipo lista
list_mutate: asignación de de un índice de una lista
list_operation: posibles funciones que se le pueden aplicar a una lista.
list_value: posibles valores de una lista.
Es importante destacar que la ambigüedad de las producciones relacionadas a expresiones se resolvió declarando explícitamente en YACC la precedencia y asociatividad correcta para cada producción. La tabla de que resume esta información puede hallarse en la sección 5.5.

5. Detalles del Lenguaje

El lenguaje tiene 3 tipos de variables: number, list y text. También tiene un mecanismo para agregar comentarios de linea el cual es agregar el caracter "#" antes de lo que se desea escribir como comentario. Para marcar el fin de una instrucción se utiliza CRLF.

5.1. Number

Las variables de tipo number son double en lenguaje C, por lo tanto se permiten números decimales y con signo. Los posibles operadores que se pueden usar con este tipo de variable son:

- Suma: +
- Resta: -
- División: /
- Multiplicación: *
- Módulo: % o mod
- Factorial: !
- Potencia: ^o **
- Notación científica: E
- Comparación: <, >, <=, >=
- Igualdad: eq
- Desigualdad: neq

StatistiC incluye funcionalidades built-in para las variables de tipo number estas son: comb y perm. Las cuales se utilizan de la siguiente manera:

```
number n1 = 3
number n2 = 4
n1 <function> n2
```

También está declarado el carácter "e" el cual representa el número de Euler.

Así mismo se proveen funciones de read y write para leer/escribir un número desde/hacia la entrada/salida estándard.

```
number n1
read n1
write n1
```

5.2. Text

Los text son palabras estáticas. Por lo tanto, cuando se declara una variable de tipo texto en base a otra variable de tipo texto, se realiza una asignación por copia. Dichos text se pueden imprimir a pantalla de la siguiente manera:

```
text s1 = "hola"
write s1
write "hola"
```

5.3. List

Los list son listas de variables de tipo number. Estas son estáticas en cuanto a tamaño pero mutables en cuanto a los valores que contiene. Hay distintas maneras de declarar una lista, las cuales se pueden ver en el siguiente ejemplo:

```
list data1 size 3 # crea una lista vac a de tama o 3
list data2 = [1, 2, 3]
data1(1) = 2 # asigna al indice 1 de la lista con el valor 2
write data2 # imprime la lista -> [1, 2, 3]
```

StatistiC incluye varias funcionalidades built-in sobre las variables de tipo lista estas son: mean, median, mode, stdev, range, qtr1, qtr3 y iqtr. Las cuales se utilizan de la siguiente manera:

```
<function> of list
```

5.4. If y While

Para delimitar el fin de la condición de un if y un while se utiliza la palabra "do", y para marcar el fin del if y el while se utiliza la palabra "end". Un ejemplo del uso de estas dos funcionalidades están en el archivo "control.stc" dentro de la carpeta "examples".

5.5. Expression

Las expression están compuesta por variables y constantes de tipo numéricos y operadores de tipo relacionales, aritméticos, lógicos y de lista. Estas pueden ser utilizadas dentro de las

condiciones del if y while y para asignar el valor a una variable numérica. Los niveles de precedencia y asociatividad son:

Nivel	Expresión	Asociatividad
0	'measure of' 'n in' 'l[n]'	NA
1	!	LR
2	-n	LR
3	not ˜	RL
4	ˆ	LR
5	* E / %	LR
6	+ -	LR
7	<, >, <=, >=	LR
8	eq neq	LR
9	and	LR
10	or	LR

Cuadro 1: Tabla de precedencia y asociatividad.

6. Ejemplos de uso

En la carpeta "examples" pueden hallarse 8 ejemplos de uso del programa, todos documentados. Los mismos se enumeran a continuación.

- "control.stc". Valida funcionamiento de las estructuras de control ïfz "while", incluyendo varios niveles de anidamiento.
- "expressions.stc". Valida el funcionamiento de los múltiples operadores y que se aplique correctamente la precedencia de los mismos cuando participan en expresiones complejas.
- "stats.stc". Prueba el funcionamiento de las funciones built-in de estadística, comparando sus resultados con valores conocidos.
- "lists.stc". Prueba el funcionamiento de los operadores y asignaciones de lista y las distintas maneras de usarlos.

- "vars.stc". Prueba el funcionamiento de las declaraciones y asignaciones de variables de distintos tipos.
- "warnings.stc". Prueba el funcionamiento de las validaciones de tipo y sintaxis del compilador.
- "full.stc". Muestra un uso general del lenguaje implementando un programa que lee un conjunto de muestras y calcula su media usando la función built-in, y un algoritmo implementado en el código.
- "primality.stc". Ejecuta un test de primalidad sobre un número ingresado, útil para realizar benchmarkings del compilador.

7. Dificultades encontradas

Uno de los problemas con los que nos topamos durante el desarrollo de este TP causado por la manera en la que nos organizamos fue que hay un alto grado de acoplamiento entre las 4 grandes estructuras de nuestro TP; el parser, el armado de árbol, el chequeo de variables y la impresión/liberación del árbol. Esto causo que el desarrollo 'ciego' de una sección fuese imposible causando que otros miembros deban esperar hasta que cierto feature este desarrollado. Por ejemplo, hasta que las producciones gramaticales relacionadas a listas no estuvieron terminadas, el desarrollo de la impresión de listas y el armado de los nodos de las listas se vio imposibilitado causando tiempo muerto entre programadores.

Relacionado al punto anterior, varias veces durante el desarrollo nos topamos con problemas en nuestra gramática que aunque eran fácilmente resueltos en la parte de lexer y parser, necesitaban casi obligatoriamente un refactor de grandes secciones de códigos.

Distintos integrantes del grupo programan sobre distintas plataformas con accesos a distintas versiones/implementaciones de parser y lexer además de compiladores. Aunque inofensivo a primera vista, ciertos flags de compilación para debugeo estaban presentes en algunas implementaciones mientras que no en otras.

Un problema con el que no encontramos una verdadera solución es el problema de testeo/manejo apropiado de errores. Si bien es natural buscar que el compilador funcione bien en las situaciones donde recibe instrucciones correctas, también es necesario pensar todas las posibles formas en las cuales podemos recibir respuestas incorrectas. (Fue notorio el cambio de enfoque de testeo de casos correctos al inicio del proyecto comparado al final donde casi exclusivamente testeábamos contra casos de error.)La gramática nos ayudo bastante en este aspecto, proveyéndonos directamente con todas las posibilidades. Si bien no podemos asegurar el perfecto funcionamiento de nuestro compilador(pues no hemos hecho una demostración matemática de ello),empíricamente hemos testeado una sorprendente cantidad de casos de error.

8. Futuras extensiones

- Uso nativo de integers. Actualmente todas las variables del tipo number se almacenan en una variable double de C. Por una cuestión de eficiencia, se plantea realizar una evaluación del tamaño del número ingresado y en base a eso asignarlo en un double, integer o incluso boolean (char).
  Complejidad relativa baja(seria agregar 1 caso a la gramática y agregar 1 caso en varios switch del variable checker)

- Listas dinámicas. Para evitar problemas de memory leaks se decidió evitar el uso de memoria dinámica en el objeto compilado, y como consecuencia de esto tanto las listas como los textos son estáticos en cuanto a tamaño. Se plantea cambiar esto para permitir una mejor operabilidad.
  Complejidad relativa moderada (la gramática lo permitiría con pequeñas modificaciones, la complejidad esta en imprimir las estructuras en c de forma apropiada)

- Declaración de funciones. Actualmente el lenguaje no soporta la definición de funciones, pero dado que esta característica es de extrema utilidad, se plantea implementarla.
  Complejidad relativa alta (la estructura de declaraciones de funciones es muy distinta a lo que maneja nuestra gramática actualmente)

- Mejor error reporting. Hay algunos casos donde podemos detectar errores en las variables(porque se asignan tipos distintos) pero en donde declaramos el error, desconocemos

de otra información de la variable produciendo mensajes de error no tan útiles. Implementar un mejor recorrido con backtracking produciría mensajes de error mas claros. Complejidad relativa moderada/alta (dependiendo del nivel de profundidad/inteligencia del compilador con el que se quiera implementar esto)