Autómatas, Teoría de Lenguajes y Compiladores

Trabajo Práctico Especial - *Dogelang*

Grupo: Juan Lambvuschini

Integrantes:

José Vitali 54197

Natalia Navas 53559

Francisco Bartolomé 54308

Leandro Llorca 54859

ITBA
INSTITUTO TECNOLOGICO DE BUENOS AIRES
·UNIVERSIDAD PRIVADA·

Índice

Idea subyacente y objetivo del lenguaje

Basado en la sensación de internet “Doge”, un perro de raza Shiba Inu, se eligió desarrollar un lenguaje que sea entretenido para el usuario. De tipo imperativo y tipado el lenguaje utiliza muchas de las frases célebres que se pueden encontrar en internet tal como “woow” o “so”.

Además se busco que sea un lenguaje que sea simple de aprender, especialmente para la gente familiarizada con el vocabulario de este personaje. Para un programador con experiencia puede no resultar un lenguaje intuitivo, pero para alguien con menos experiencia puede ser fácil de aprender ya que no cuenta con muchas dificultades técnicas, como por ejemplo el manejo de punteros.

En cuanto a la estructuración de un programa, este es muy similar a C, constando de bloques, if, while y declaración de variables que se definen en forma similar pero con palabras reservadas diferentes.

Consideraciones realizadas

De la forma que se va generando el código en C, se puede ver que no es de lo más eficiente en cuanto al manejo de memoria, ya que se está guardando todo el código generado. Otra alternativa que se tuvo en cuenta fue la de generar funciones en C a medida que se puede desde el yacc. Sin embargo se eligió la primera alternativa ya que para lo que consta el lenguaje, que solo se usaría para programas simples, la memoria no era un tema fundamental. Si se implementa en funciones separadas esto no sería un conflicto ya que al generar mucho código pero en funciones separadas, se puede ir manejando función a función.

Descripción del desarrollo del TP

El desarrollo tomó lugar en dos etapas, el desarrollo del lexer y el del parser.

En la primera etapa se desarrolló una gramática LEX, que se encargaba de transformar el programa en tokens, para que luego sean analizados. Reconoce estos tokens a través de expresiones regulares, que identifican las posibles palabras reservadas del lenguaje y variables.

En la segunda etapa se desarrolló un parser YACC, que recibe los tokens desde el lexer y se encarga de interpretar su contenido, basándose en las reglas definidas para cada tipo de token. El yacc es un analizador ascendente, por lo cual en esta etapa se iba generando código en C, hasta llegar al estado inicial, donde ya se obtiene el equivalente completo en C de todo el programa. En cada estado se va generando código en C que se le asigna al lado izquierdo de la producción. Cuando hay un estado final se lo traduce directamente a string y si no se le asigna el valor de uno de los estados del lado derecho de la producción. Para construir los strings se implementaron varias funciones que concatenan strings, encontradas en la librería append. Se eligieron tener varias implementaciones

Para guardar variables en el yacc se implementó una librería de hash map. Cuando se encuentra una variable se la ingresa en el mapa con el nombre y su valor, y cuando se la quiere utilizar se verifica que esté correctamente definida. De la misma forma se verifica al declarar una variable que esta no haya sido declarada previamente.

Para obtener números enteros de la entrada utilizamos la librería getnum, provista por la cátedra de Programación Imperativa.

Gramática

Descripción de la gramática

La gramática tiene como punto de entrada el símbolo no terminal “program”, que se encarga de escribir en un archivo la traducción del código doge en código C, luego de que el autómata consiga la reducción correspondiente.

El programa se descompone en comandos, y el programa debe finalizar obligatoriamente con la sentencia: “*plz* {variable} *go to the moon*”, siendo “variable” alguna variable de tipo entero o una expresión aritmética.

Definición de la gramática

La gramática se compone por un conjunto de símbolos terminales, un conjunto de símbolos no terminales, un símbolo inicial (incluído en el conjunto de no terminales) y un conjunto de producciones.

Símbolos no terminales

{program, command, condition, els, gotothemoon, commands, def, int_assign, string_assign, comment, loop print, ea, ta, fa, el, tl, fl, logic_exp, relational_exp, arith_exp, open_block, close_block}

Símbolos terminales

{VERY, SO, WORDS, NUMBR, IS, MORE, LESS, LOTS, FEW, PLZ, GOTOTHEMOON, RLY, BUT, MANY, NOT, AND, OR, BIGGER, SMALLER, BIGGERISH, SMALLERISH,

SAME, OPENBRACKET, CLOSEBRACKET, WOW, WANT, SHH, NUMBER, ID, STRING, ‘(‘, ‘)’}

Símbolo inicial

El símbolo no terminal “program” funciona como símbolo inicial, y es el encargado de escribir en un archivo el contenido del programa traducido en c.

Producciones

program → commands gotothemoon,
commands → command commands,
commands → /* lambda */,
command → def,
command → int_assign,
command → string_assign,
command → arith_exp,
command → condition,
command → loop,
command → comment,
command → print,
comment → SHH,
condition → RLY logic_exp open_block commands close_block,
condition → RLY logic_exp open_block commands close_block BUT els,
els → condition,
els → open_block commands close_block,
open_block → OPENBRACKET,
close_block → CLOSEBRACKET,
loop → MANY logic_exp open_block commands close_block,
gotothemoon → PLZ arith_exp GOTOTHEMOON,
print → WOW arith_exp WOW,

print → WOW STRING WOW,

def → VERY ID SO WORDS,

def → VERY ID SO NUMBR,

int_assign → ID IS arith_exp,

int_assign → ID IS WANT NUMBR,

string_assign → ID IS STRING,

arith_exp → ea,

ea → ea MORE ta,

ea → ea LESS ta,

ea → ta,

ta → ta LOTS fa,

ta → ta FEW fa,

ta → fa,

fa → '(' ea ')',

fa → NUMBER,

fa → ID,

logic_exp → el,

el → el OR tl,

el →tl,

tl → tl AND fl,

tl → fl,

fl → NOT relational_exp,

fl → relational_exp,

fl → '(' el ')',

relational_exp → arith_exp BIGGER arith_exp,

relational_exp → arith_exp SMALLER arith_exp,

relational_exp → arith_exp BIGGERISH arith_exp,

relational_exp → arith_exp SMALLERISH arith_exp,

relational_exp → arith_exp SAME arith_exp

Descripción del lenguaje

El lenguaje doge, es un lenguaje iterativo en el que los comandos se ejecutan secuencialmente uno por uno, y el programa se termina mediante la sentencia *plz valorDeReotrno go to the moon*, donde el *valorDeRetorno* debe ser una expresión aritmética (más adelante se explicará cómo se realizan las expresiones aritméticas).

El compilador no se fija en los espacios en blanco, por lo que los comandos podrán estar separados por cualquier cantidad de espacios, tabulaciones o cualquier otro tipo de espacio en blanco.

Tipos de datos

El lenguaje doge consta de dos tipos de datos: números enteros y strings. La declaración de variables se realiza mediante la sentencia "*very* {nombre} *so* {tipo}", donde "nombre" será el nombre de la variable declarada y {tipo} es el tipo de dato, el cual debe ser *words* (un string) o *numbr* (un número entero).

Ejemplos:

very variableEntera so numbr
very variableString so words

Asignación de valores

Para darle valor a la variable se debe establecer la siguiente sentencia: "{nombre} *is* {valor}", donde "nombre" es el nombre de la variable previamente declarada a la cual se le dará valor, y "valor" es el valor que se le otorgará. Existen dos posibles escenarios en cuanto a ésto: que la variable sea de tipo *numbr* o de tipo *words*. Si la variable es de tipo *numbr* entonces el valor podrá ser otra variable de tipo *numbr*, una expresión aritmética (más adelante se explicará cómo se realizan las expresiones aritméticas) o se podrá pedir un número de la entrada. Esto último se realiza mediante la sentencia "*want numbr*". En el

caso de que la variable sea de tipo *words*, el valor deberá ser un string, el cual debe estar delimitado por comillas dobles (“).

Ejemplos:

variableEntera1 is 4
variableEntera2 is 5 more variableEntera1
variableEntera3 is want numbr
variableString1 is “Hola mundo!”

Expresiones aritméticas

Las expresiones aritméticas resultan útiles para realizar operaciones y ser utilizadas en expresiones lógicas, el lenguaje doge cuenta con cuatro tipos de operadores:

operador	**significado**
more	suma
less	resta
lots	multiplicación
few	división

Fig 1

El lenguaje doge tiene en cuenta la precedencia de los operadores, siendo *lots* y *few* quienes tienen mayor precedencia, y *more* y *less* los que tienen menor. También se tienen en cuenta los el uso de paréntesis para darle mayor precedencia a ciertas partes de la expresión. Cada uno de estos operadores debe estar precedido y seguido por una expresión aritmética.

Definimos una expresión aritmética como un conjunto de operadores aritméticos y números, que se define por construcción de la siguiente forma:

- un número es una expresión aritmética
- una variable de tipo numbr es una expresión aritmética
- exp1 operador exp2 es una expresión aritmética, donde exp1 y exp2 son expresiones aritméticas y operador es uno de los operadores definidos en la tabla *Fig 1*.
- ‘(‘ exp ‘)’ es una expresión aritmética, donde exp es una expresión aritmética.

Ejemplos de expresiones aritméticas:

3

variableEntera

variableEntera more 4

(5 less 8) lots variableEntera

Bloques condicionales

Existen dos tipos de bloques condicionales en el lenguaje doge: bloques de tipo if (rly) y de tipo while (many).

Los bloques de tipo if, se establecen mediante la sentencia *rly* seguida de una expresión lógica (mas adelante se explicará que es una expresión lógica) y entre llaves los comandos que se ejecutarán en caso de que la expresión lógica sea verdadera. El lenguaje doge implementa mecanismos para simular un *else if*, y un *else*. El primero de ellos se realiza luego de cerrar un bloque-if mediante la sentencia *but rly* seguida de una expresión lógica y entre llaves los comandos que se ejecutarán en caso de que la expresión lógica sea verdadera. El segundo se realiza al finalizar un bloque-if o if-else, mediante la sentencia *but* y entre llaves los comandos a ejecutar.

Ejemplo:

```
rly variableEntera smaller 4{
        wow “Estoy en el primer bloque\n” wow
}but rly not variableEntera same 5 and variableEntera same 8{
        wow “Estoy en el segundo bloque\n” wow
}but{
        wow “Estoy en el tercer bloque\n” wow
}
```

Los bloques de tipo while, se establecen mediante la sentencia *many* seguida de una expresión lógica y entre llaves la secuencia de comandos que se ejecutarán mientras la expresión lógica sea verdadera.

Ejemplo:

```
very variable so numbr
variable is 10
many variable same 0{
        wow variable wow
        variable is variable less 1
}
```

Expresiones lógicas

Las expresiones lógicas resultan útiles para la conformación de un bloque lógico (*if* o *while*). El lenguaje doge cuenta con tres operadores lógicos y cinco operadores relacionales.

operador	**significado**
and	operador lógico y
or	operador lógico o
not	negación
bigger	>
smaller	<
biggerish	>=
smallerish	<=
same	==

Fig 2

El lenguaje tiene en cuenta la precedencia de los operadores lógicos. También se tienen en cuenta los el uso de paréntesis para darle mayor precedencia a ciertas partes de la expresión.

Se define una expresión lógica por construcción de la siguiente forma:

- una expresión relacional es una expresión lógica.
- *not exp* es una expresión lógica, donde *exp* es una expresión lógica.
- *exp1 and exp2* es una expresión lógica, donde *exp1* y *exp2* son expresiones lógicas.
- *exp1 or exp2* es una expresión lógica, donde *exp1* y *exp2* son expresiones lógicas.
- *(exp)* es una expresión lógica, donde *exp* es una expresión lógica.

También es necesario definir al conjunto de expresiones relacionales. Estas expresiones se definen de la siguiente forma: *exp1 op exp2*, donde *exp1* y *exp2* son expresiones aritméticas y op es un operador relacional (los últimos cinco que se encuentran en *Fig 2*).

Ejemplos de expresiones lógicas:

4 bigger 8
4 bigger 8 and 5 same 4
4 smaller 3 or (5 same 5 and 7 smallerish 9)

Salida de datos

El lenguaje doge tiene un sistema de salida de datos, en el que se imprime el dato deseado en pantalla. Para realizar una impresión en pantalla se debe utilizar el siguiente comando: *wow loQueQuieroImprimir wow*, donde *loQueQuieroImprimir* es la variable o constante que se desea imprimir. Se pueden imprimir variables, strings o expresiones aritmeticas (en este caso se imprimirá el resultado de dicha expresión).

Ejemplos:

wow “Hola mundo!!\n” wow
wow variableString wow
wow 2 more 3

Comentarios

El lenguaje doge se provee de un sistema de comentarios, en donde no se analizará el contenido del mismo. El contenido del comentario debe estar precedido de la palabra reservada *shh* y debe estar seguido de la palabra reservada *shhhh*. El único cuidado que se debe tener es los comentarios solo se pueden hacer dentro del programa (es decir antes de retornar con *plz go to the moon*), y que no se deben hacer en medio de otra sentencia.

Ejemplo de uso correcto de un comentario:

very variable is numbr
shh soy un comentario correcto!!! shhhh
numbr is 4
plz numbr go to the moon

Ejemplo de uso incorrecto de un comentario:

very variable is shh soy un comentario incorrecto…. shhhh numbr
numbr is 4
plz numbr go to the moon

Dificultades encontradas en el desarrollo del TP

Una de las dificultades fue la de intentar incorporar lo que se desarrolló en la primera etapa con lo que había que hacer para la segunda etapa. Al no haber entendido correctamente el objetivo del lexer en una primera instancia, se produjo un lexer que incorporaba funcionalidad que le correspondía al parser. Una vez empezado el desarrollo del parser no se encontraba forma de poder incorporar lo desarrollado previamente con lo que hacía falta implementar. Por esta razón se tuvo que regenerar el lexer, siendo mucho más simple el generado en una segunda instancia que el inicial.

Futuras extensiones

Existen varias posibles extensiones que le agregarían mayor utilidad al usuario del lenguaje.

Una de estas es la incorporación de nuevos tipos de datos. Por el momento solo se manejan los tipos ‘numbr’ (int) y ‘words’ (string). Por otro lado, el tipo de dato numbr no cuenta con la posibilidad de declarar números negativos, para obtener un número negativo es necesario sustraer uno mayor a uno menor.

Un problema de la versión actual del lenguaje es la falta de modularización. Para poder lograr esto y que el usuario genere código más legible y claro se debería implementar la posibilidad de hacer funciones, y también cláusulas de inclusión, para poder incluir funciones que están definidas en archivos separados.

Para brindar también mayor facilidad y menos necesidad de que el usuario genere código reiteradas veces se podría implementar una librería estándar del lenguaje, donde se puedan tener operaciones utilizadas frecuentemente por los usuarios. Para generar esta librería lo que se podría hacer es traducir las funciones creadas a las que se brindan por la biblioteca estándar de C.

Otra mejora sería informar al usuario la línea en la que se haya encontrado un error de compilación, de encontrarse alguno.

Optimizaciones

Existen algunas optimizaciones con el que el compilador podría contar.

Se podría agregar una etapa de preprocesamiento para tener mayor información acerca de las variables. En el caso que se incluyeran cláusulas de inclusión habría que contar con esta etapa, para poder incorporar las funciones declaradas en las librerías incluidas.

Otra optimización que se podria tener en cuenta para el futuro es asignar directamente desde el yacc el valor de algunas expresiones ya evaluadas, como por ejemplo si se tuviese el siguiente caso:

```
rly 2 more 3 smallerish 5
```

en vez de devolver :

```
if( 2 + 3 =< 5)
```

se podría devolver:

```
if(true)
```

Tambien se podria agregar una segunda pasada al yacc, utilizando lo que se obtuvo en la primera pasada de base.

Referencias

Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley, 2006.