Epilog

Juan Segundo Valero

March 1, 2024

1 Introducción

Informe sobre el trabajo práctico final de la materia Análisis de Lenguajes de Programación. Se creó un lenguaje de programación lógica, inicialmente basado en Prolog, aunque con varios cambios en el funcionamiento, que se van a especificar en este documento.

2 Gramática

La gramática utilizada en este proyecto es la siguiente:

```
digit ::= ´0´ | ... | ´9´
letter ::= ´a´ | ... | ´Z´
operation ::=  ´+´ | ´-´ | ´*´ | ´/´


name ::= digit | letter | name name

varT ::= name | ‘’‘ name ‘’‘  | ‘[‘ vars ‘]‘

eqToken ::=  digit | letter | operation | eqToken eqToken

Exps  : Exxp
      | Exxp Exps

Exxp  : FUN ’(’ vars ’)’ ’:=’ Exp ’.’
      | FUN ’(’ vars ’)’ ’:=’ eqs ’.’
      | Exp ’.’

Exp     :   ’true’
        |   ’false’
        |   FUN ’(’ vars ’)’
        |   Exp ’==’ Exp
        |   Exp ’!=’ Exp
        |   Exp ’&’ Exp
        |   Exp ’|’ Exp
        |   Exp ’+’ Exp
        |   Exp ’-’ Exp
        |   Exp ’*’ Exp
        |   Exp ’/’ Exp
        |   ’{’ nums ’}’
        |   varT

vars    :   eqs
        |   varT
        |   lst
```

```
        |   eqs ',' vars
        |   VarT ',' vars
        |   lst ',' vars

lst     : '[' vars ']'
        | '[' ']'
        | generic ':' generic

eqs     : '{' nums '}'

nums    : eqToken
        | nums '+' nums
        | nums '-' nums
        | nums '*' nums
        | nums '/' nums
```

3 Reglas del lenguaje

Este lenguaje se basa en la asignación de valores a funciones con variables.
Hay cuatro tipos de variables:

- `Valores` se definen entre " y se interpretan como valores atómicos, ej: 'var', 'arg'.
- `Genericas` se escriben sin " y como indica su nombre son variables genéricas, pueden ser reemplazadas por valores, ej: A, B.
- `Listas` listas de valores, se pueden pasar como argumentos o separar en head y tail como en Haskell (x:xs), ej: ['a', 'b', 'c'], [], ['a'].

Todas las instrucciones tienen que tener un terminador, que es el carácter `.`

Para asignar un valor a una función se hace de la siguiente manera:

```
function('arg', 'arg2') := true.
```

Para hacer una query a una función con sus argumentos se hace de la siguiente manera:

```
function('arg', 'arg2').
```

Esto va a devolver, o bien un valor de retorno, o un error.

3.1 Operaciones lógicas disponibles

- `&` And
- `|` Or
- `==` Igualdad
- `!=` Negación

3.2 Queries genéricas

Al igual que en Prolog, se pueden hacer queries de funciones con variables genéricas y se devuelven los posibles valores que pueden tomar esas variables para que devuelva true la función. Por ejemplo:

```
fun('b', 'c') := true.
fun('d', 'e') := true.
fun('x', 'y') := true.
fun(A, B).
```

Resultado:

```
A='b','d','x'
B='c','e','y'
```

En el caso de tener más de una variable genérica, como en el ejemplo anterior, las columnas del resultado representan el conjunto de valores que tienen que tener de manera simultánea, en este ejemplo, fun('b', 'c'), fun('d', 'e') y fun('x', 'y') son true.
Como en Prolog, esto solo funciona para variables y funciones que son directamente true, es decir si se da el caso:

```
fun('b', 'c') := true.
fun(A, B) := fun(B, A).
fun(A, B).
```

El resultado solo va a tener en cuenta a fun('b', 'c').
Prolog también permite hacer operaciones con números utilizando este sistema de variables genéricas, en este lenguaje esto es reemplazado por las ecuaciones que se verán en el apartado siguiente.

3.3 Ecuaciones

A diferencia de Prolog, este lenguaje permite devolver valores númericos, esto da la posibilidad de escribir funciones con enteros de la siguiente manera:

```
factorial({0}) := {1}.
factorial({1}) := {1}.
factorial(A) := {A} * factorial({A - 1}).
factorial({5}).
```

Como se puede observar, todo lo que tiene que ser interpretado como una ecuación aritmética va entre {} para separarlo de las variables booleanas.
Las operaciones aritméticas permitidas son:

- `+` Suma
- `-` Resta
- `*` Multiplicación
- `/` División

Con el orden de precedencia estándar.

3.4 Listas

Se pueden crear programas que trabajen con listas de la siguiente manera:

```
len([]) := {0}.
len(x:xs) := {1} + len(xs).
len(['a', 'b', 'c']).
```

Las listas se definen entre [ ] y pueden ser pasadas como argumento para hacer una consulta.

A la hora de definir funciones, las listas pueden ser pasadas como argumento o se puede separar el head del tail como en Haskell con `:`.

3.5 Búsqueda de existencias

Se pueden hacer queries del estilo:

```
padrede('Juan', 'María') := true.
padrede('Pablo', 'Juan') := true.
padrede('Pablo', 'Marcela') := true.
hermanode(A,B) := padrede(C,A) & padrede(C,B) & A != B.
hermanode('Juan', 'Marcela').
```

Esto va a devolver 'true' si encuentra un valor C común tal que se cumpla la operación especificada.

Nota: Al poner una variable genérica en la operación, la búsqueda es de existencia, es decir, devuelve true si está definida la función con los valores 'Juan', 'Marcela' y otro valor en común, no evalúa el valor de esa función.

3.6 Errores

Puede haber tres tipos de errores en la ejecución del programa.

- `Error de Parseo` Como indica el texto, si hubo algun error en el parseo del programa.
- `UndefVar` Cuando se está queriendo hacer una consulta de una función que no existe para el tipo de datos solicitado.
- `InvalidOp` Se esta queriendo hacer una operación inválida, esto pueden ser operaciones lógicas con valores aritméticos (o viceversa) u operaciones que combinan ambos tipos de dato