

# ARJ-HULK

Alex Sierra  
Raudel Gómez  
Juan Carlos Espinosa

## 1 Compilador del lenguaje HULK

### 1.1 Pasos para compilar un archivo .hulk

- Ejecuta el comando `make build` en la raíz del proyecto para generar el Lexer y el autómata del Parser.
- Ejecuta el comando `make build` en la raíz del proyecto para generar el Lexer y el autómata del Parser.
- Crear un archivo llamado `main.hulk` en la raíz del proyecto, que será donde va su código a compilar.
- Ejecuta el comando `make` para parsear el código y compilar.
- Ejecuta el comando `make compile` para solo compilar archivo en C generado.
- Ejecuta el comando `make test` para ejecutar los test automáticos que hemos añadido al proyecto

### 1.2 Definición de la gramática usada para el Lexer

No Terminales:

E, A, F, G, H, I, J, K = <E>, <A>, <F>, <G>, <H>, <I>, <J>, <K>

Terminales:

char, ocor, ccor, opar, cpar, ?, plus, star, dot, pow = <ch>, <[>, <]>, <(>, <)>, <?>, <+>, <\*>, <.>, <^>

Producciones:

```
S -> E
E -> A | E | A
A -> F A | F
F -> [ G ] I | H I
I -> ? | + | * |
H -> ch | ( E ) | .
G -> ^ J | J
J -> K J | K
K -> ch | ch - ch
```

### 1.3 Definición de la gramática usada para parsear HULK

No Terminales:

```
program = <P>
expression = <E>
expression-block, block, expression-instruction-list = <EB>, <B>, <Ils>
expression-string, expression-let, expression-if, expression-while = <Es>, <El>, <Ei>, <Ew>
expression-for, expression-destructive-assignment, expression-array = <EF>, <Eas>, <Ear>
expression-call, expression-boolean, expression-arithmetic, expression-type = <Ec>, <Eb>, <Ea>, <Et>
expression-array-call, expression-dot-call = <Ac>, <Epc>
```

```

string-term = <Ts>
let-assignment, let-assignment-list = <Sl>, <As>
elif, elif-list = <Elif>, <Elifs>
array-data, array-explicit-data-list = <X1>, <X2>
expression-call-parameters, parameter-list = <C1>, <C2>
boolean-factor, boolean-term, boolean-clausule, boolean-atomum = <Fb>, <Tb>, <Cb>, <Gb>
arithmetic-term, arithmetic-factor, arithmetic-atomum, arithmetic-unary = <Ta>, <Fa>, <Ga>, <Oa>
to-type, type-discriminator = <T>, <Type>
atomic = <W>
instruction-list = <I2s>
instruction, class-declaration, protocol-declaration, function-declaration = <I>, <C>, <Pr>, <F>
class-declaration-phrase, class-inheritance, class-body, class's-instruction = <Hc>, <Hih>, <CB>, <IC>
protocol-declaration-phrase, protocol-extension, protocol-body = <PT>, <Prex>, <PB>
function-parameters, parameter-list, typed-parameter, funtion-body = <D1>, <D2>, <D3>, <FB>

```

## Terminales:

```

semi, colon, comma, dot, opar, cpar, ocur, ccur = <;>, <:>, <,>, <.>, <(,>, <)>, <{>, <}>
dest-equal, equal, plus, minus, star, div = <:=> <=>, <+>, <->, <*>, </>
concat, spaced-concat = <@>, <@@>
and, or, not, eq, neq, lt, gt, lte, gte = <&>, <|>, <!>, <==>, <!=>, << >>, <> >>, <=>, <=> >
idx, num, bool, string = <id>, <num>, <bool>, <string>
if, elif, else, while, for = <if>, <elif>, <else>, <while>, <for>
classx, protocolx, let, defx, return, printx = <protocol>, <type>, <let>, <function>, <=> >, <print>
inherits, extends = <inherits>, <extends>
is, as, new = <is>, <as>, <new>
since-that, ocor, ccor = <||>, <[>, <]>

```

## Producciones:

```

P -> P1
P1 -> I2s EB ; I2s | I2s EB I2s
I2s -> I2s I |
I -> C | F | Pr
EB -> E | B
B -> { I1s }
I1s -> I1s E ; | E ;
E -> Es | El | Eif | Ew | Ef | Eas | Ear
Es -> Es @ Ts | Es @@ Ts | Ts
Ts -> Eb
Eb -> Eb | Fb | Tb
Tb -> Tb & Fb | Fb
Fb -> ! Cb | Cb
Cb -> Gb == Gb | Gb != Gb | Gb < Gb | Gb > Gb | Gb >= Gb | Gb <= Gb | Gb
Gb -> Ea
Ea -> Ea + Ta | Ea - Ta | Ta
Ta -> Ta * Fa | Ta / Fa | Fa
Fa -> Ga ^ Fa | Ga
Ga -> + Oa | - Oa | Oa
Oa -> W
Epc -> Epc . Ec | Ec . Ec | id . Ec
W -> id | id . id | num | bool | str | ( E ) | Et | Ec | Epc | Ac
T -> : Type |
Type -> id | [ id ] | [ id , num ]
Sl -> id T = E
Eas -> id := E | Ac := E | id . id := E
El -> let As in EB

```

```

As -> S1 , As | S1
Eif -> if ( Eb ) EB Eelifs else EB
Eelifs -> Eelifs Eelif |
Eelif -> elif ( Eb ) EB
Ew -> while ( Eb ) EB
Ef -> for ( id in E ) EB
Ec -> id ( C1 )
C1 -> C2 |
C2 -> E , C2 | E
F -> function id ( D1 ) T FB
FB -> B | -> E ;
D1 -> D2 |
D2 -> D3 , D2 | D3
D3 -> id T
Hc -> type id | type id ( D2 )
Hih -> inherits id | inherits id ( C2 ) |
C -> Hc Hih { CB }
CB -> CB IC |
IC -> id T = E ; | id ( D1 ) T FB
Et -> W is id | W as id | new Ec
Ear -> [ X1 ]
X1 -> E || id in E | X2 |
X2 -> X2 , E | E
Ac -> id [ E ] | Ec [ E ] | Ac [ E ] | Epc [ E ]
PT -> protocol id
Prex -> extends id |
Pr -> PT Prex { PB }
PB -> PB PF | PF
PF -> id ( D1 ) T ;

```

## 1.4 Generación de Código

Para nuestra generación de código modelamos los objetos en el lenguaje de programación C como una lista de diccionarios. Para ello contamos con una interfaz que consta de 2 métodos, añadir una propiedad, el cual recibe un “key” que indica el nombre de la propiedad y un puntero a “void”(aquí se declaran tanto propiedades como funciones, ya que el puntero puede apuntar a cualquier cosa) y pedir las propiedades. De esta manera modelamos todos los objetos, incluyendo los predefinidos(string, bool y number). Todos los tipos heredan de object, el cual tiene los metodos equals y toString para poder saber si 2 objetos son iguales y poder printear cualquier objeto. Para la creación de funciones, contamos con 2 tipos de contexto, un contexto general, al cual pertenecen todas las declaraciones y funciones, y contextos de generación, los cuales se crean para cada función. Luego a la hora de printear el código, primero lo hacen las declaraciones y luego todas las funciones.