Sintaxis del Lenguaje Umbra

Diseño y Especificación

9 de septiembre de 2024

0.1. Introducción

Umbra es un lenguaje de programación diseñado con tipado estático y un enfoque didáctico. Este documento describe su sintaxis utilizando ejemplos y notación formal BNF (Backus-Naur Form).

0.2. Estructura Básica de un Programa en Umbra

Todo programa en Umbra debe contener, al menos, una función principal llamada `start`. Esta función será el punto de entrada del programa y define el comportamiento básico del mismo. La función `start` puede recibir un arreglo de string y tiene un valor de retorno tipo int.

BNF para la función `start`:

```
<function_definition> ::= "func" "start" "(" [string] ")" "->" "int" "{" <
    statement>+ "}"
```

Ejemplo de la función `start`:

```
func start() -> int {
    // Código principal
}
```

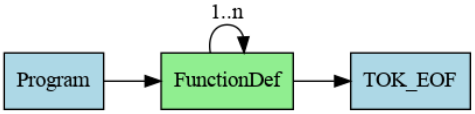

Figura 1: ST: pogramDeclaration

0.3. Variables y Declaración de Variables

En Umbra, todas las variables y funciones deben ser declaradas explícitamente con su tipo. Los tipos básicos soportados incluyen:

- `int`
- `float`

- `bool`
- `char`
- `string`
- Arrays de cualquier tipo básico.

BNF: Tipos de Variables

```
<type> ::= "int" | "float" | "bool" | "char" | "string" | array(?)
```

0.3.1. Declaración de Variables

La declaración de variables sigue una estructura simple que comienza con el tipo de dato seguido del identificador. Opcionalmente, una variable puede ser inicializada en el momento de su declaración con un valor mediante el operador de asignación = seguida de una < expression > .

BNF: Declaración de variables

```
<variable_declaration> ::= <type> <identifier> ["=" <expression>] <new_line>
```

Ejemplo de código

Listing 1: Declaración de variables en Umbra

```
int contador = 10
float tasa = 3.14
bool es_valido = true
```

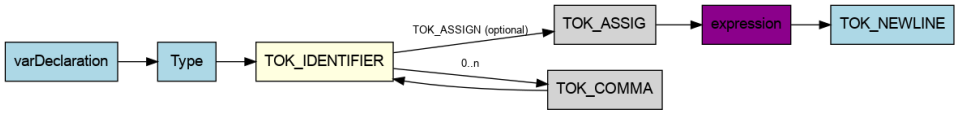

Figura 2: ST: varDeclaration

0.4. Declaración y Definición de Funciones

Las funciones en Umbra deben especificar su tipo de retorno y tener parámetros con tipos explícitos.

BNF: Declaración de funciones

```
"func" <identifier> "(" <parameter_list> ")" "->" <type>
"{" <statement>+  [<return_statement_if_not_void>]"}"
<parameter_list> ::= [<parameter> ("," <parameter>)*]
<parameter> ::= <type> <identifier>
<return_statement> ::= ["return" <expression>]
```

Ejemplo de código

Listing 2: Definición de Funciones en Umbra

```

func foo() -> int {
    return 0;
}

func bar(int a, int b) -> int {
    return a + b;
}
```

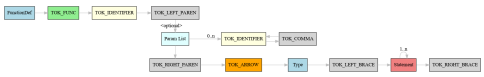

Figura 3: ST: funcDeclaration

0.5. Operadores

Umbra prioriza operadores textuales y organiza los operadores en tres categorías: lógicos, aritméticos y de comparación. Esta estructura mejora la legibilidad y permite al programador seleccionar el operador adecuado según el contexto.

Al usar operadores en Umbra, el árbol sintáctico sigue la estructura:

```
<binary_expression> ::= <expression> <operator> <expression>
```

Esto significa que una operación binaria consiste en dos expresiones y un operador que actúa entre ellas. A continuación, se describen los operadores a los que aplica esta estructura:

0.5.1. Operadores Lógicos

Los operadores lógicos se utilizan para evaluar expresiones booleanas y realizar comparaciones lógicas. En el árbol sintáctico, estas operaciones siguen la forma `binary_operation`.

BNF: Operadores Lógicos

```
<binary_expression> ::= <expression> <operator> <expression>
<operator> ::= "equal" | "nequal" | "and" | "or"
```

0.5.2. Operadores Aritméticos

Los operadores aritméticos se utilizan para realizar operaciones matemáticas básicas como suma, resta, multiplicación y división. En el árbol sintáctico, estas operaciones siguen la forma `binary_operation`.

BNF: Operadores Aritméticos

```
<binary_expression> ::= <expression> <operator> <expression>
<operator> ::= "+" | "-" | "*" | "/"
```

0.5.3. Operadores de Comparación

Los operadores de comparación permiten comparar valores, como verificar si un valor es mayor o menor que otro. En el árbol sintáctico, estas operaciones siguen la forma `binary_operation`.

BNF: Operadores de Comparación

```
<binary_expression> ::= <expression> <operator> <expression>
<operator> ::= "<" | ">" | "<=" | ">="
```

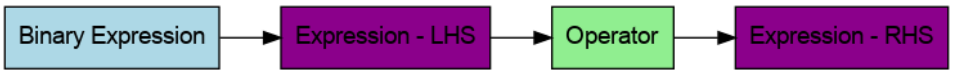

Figura 4: ST: binaryExpression

0.6. Estructuras de Control

Umbra soporta estructuras condicionales con una sintaxis clara y concisa. A continuación, se detallan las estructuras condicionales, que permiten el control del flujo de ejecución basado en condiciones.

0.6.1. Condicionales

Las estructuras condicionales en Umbra permiten la ejecución de bloques de código basados en el resultado de una condición booleana. El condicional principal es el bloque `if`, que puede opcionalmente estar seguido de uno o más bloques `elseif`, y opcionalmente un bloque `else`.

BNF: Condicionales

```
<conditional> ::= "if" <expression> "{" <statement>+ "}"
                  ["elseif" <expression> "{" <statement>+ "}"]*
                  ["else" "{" <statement>+ "}"]
```

Esta definición indica que:

- El bloque `if` es obligatorio y debe ir acompañado de una expresión condicional (`<expression>`) que evalúa a verdadero o falso.

- El bloque `elseif` es opcional, pero puede haber uno o más, cada uno con su propia expresión condicional.

- El bloque `else` es opcional y se ejecuta si ninguna de las condiciones anteriores se cumple.

Ejemplo de código: Condicional simple

Listing 3: Condicional simple en Umbra

```
if x > 10 {
    // código para evaluacion true
} else {
    // código para evaluacion false
}
```

Ejemplo de código: Condicional con elseif

Listing 4: Condicional con elseif en Umbra

```
if x > 10 {
    // código para evaluacion true
} elseif (x > 5) {
    // código para evaluacion false < x > 10 > y true para < x>5 >
} else {
    // código para evaluacion false < x > 10 > y false para < x>5 >
}
```

Ejemplo de código: Condicional sin else

Listing 5: Condicional sin else en Umbra

```
if y == 0 {
    // código para evaluacion true
}
```

Ejemplo de código: Múltiples elseif

Listing 6: Múltiples elseif en Umbra

```
if nota >= 90 {
    // código
} elseif (nota >= 80) {
    // código
} elseif (nota >= 70) {
    // código
} elseif (nota >= 60) {
    // código
} else {
    // código
}
```

0.6.2. Bucles

Umbra incluye dos tipos de bucles: `repeat x times` y `repeat if`.

BNF: Bucles

```
<loop> ::= "repeat" <integer> "times" "{" <statement>+ "}"
         | "repeat" "if" <expression> "{" <statement>+ "}"
```

Ejemplo de código

Listing 7: Bucles en Umbra

```
repeat 5 times {
    // Código repetido 5 veces
}

repeat if condicion {
    // Código mientras la condición sea verdadera
}
```

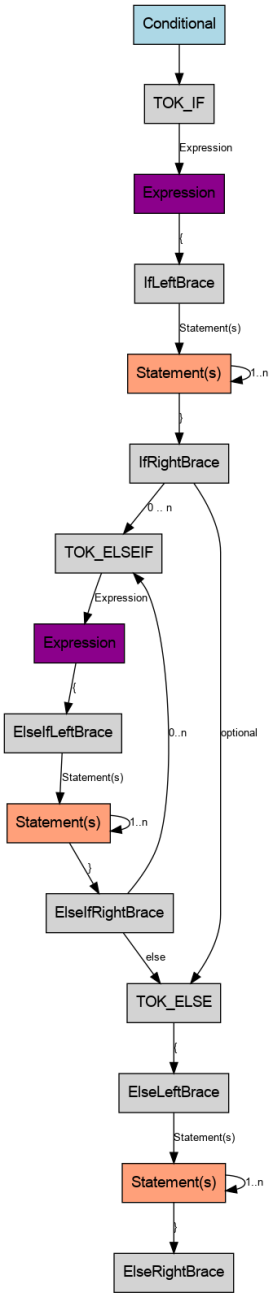

Figura 5: ST: conditionalStatement