Informe del Lenguaje de Programación: Valorant

Carlos Javier Leal Serrano
Martín Nicolás Segundo Adones Tapia

Fecha: 17 de junio de 2025

Índice

1. Gramática del Lenguaje Valorant

Valorant es un lenguaje de programación inspirado en el videojuego homónimo, en el cual se utilizan palabras clave del juego para representar conceptos clásicos de lenguajes como C/C++. A continuación se detalla su gramática, tipos y operadores:

Tipos de Datos

Palabra Clave	Tipo C	Descripción
`sage`	`int`	Enteros
`viper`	`float`	Números con decimales
`cypher`	`string`	Cadenas de texto

Operadores Aritméticos

Palabra Clave	Símbolo	Descripción
`heal`	`+`	Suma
`damage`	`-`	Resta
`kill`	`*`	Multiplicación
`share`	`/`	División

Operadores Relacionales

Palabra Clave	Símbolo	Descripción
`win`	`>`	Mayor que
`lose`	`<`	Menor que
`headshot`	`==`	Igualdad
`eco`	`!=`	Diferente
`save`	`<=`	Menor o igual
`fullbuy`	`>=`	Mayor o igual

Estructuras de Control

Palabra Clave	Equivalente	Descripción
`flash`	`if`	Condicional
`smoke`	`else`	Alternativa condicional
`rush`	`for`	Bucle for
`rotate`	`while`	Bucle while
`defuse`	`break`	Romper bucle

Funciones, Entrada/Salida y Retorno

Palabra Clave	Descripción
`agent`	Declarar una clase o función
`plant`	Retornar valor desde una función
`breach`	Entrada estándar (`input`)
`sova`	Salida estándar (`output`)

Ejemplo Sintáctico Básico

```
// Declaraci n de clase principal
agent Programa {
    sage spike() {
        cypher nombre;
        sova " Como  te llamas?";
        breach nombre;
        sova "Hola, ";
        sova nombre;
        plant 0;
    }
}
```

2. Diseño del AST

El Árbol de Sintaxis Abstracta (AST) es la representación jerárquica interna del programa fuente. En el compilador Valorant, el AST se compone de nodos definidos en `ast.h` y construidos en tiempo de análisis sintáctico usando funciones como `create_node()`, `create_declaration_node()`, etc.

Cada nodo del AST representa una estructura del lenguaje: una declaración, operación, función, bloque, o expresión. La estructura de un nodo contiene punteros hacia otros nodos (como hijos, padres o siguientes instrucciones) lo que permite navegar recursivamente por el árbol.

Tipos de Nodo

Algunos nodos clave del AST incluyen:

- **NODE_CLASS**: Representa una clase `agent`.
- **NODE_METHOD**: Método declarado dentro de una clase.
- **NODE_DECLARATION / NODE_ASSIGNMENT**: Declaraciones y asignaciones de variables.
- **NODE_IF, NODE_WHILE, NODE_FOR**: Estructuras de control.
- **NODE_PLANT**: Nodo de retorno (`plant`).
- **NODE_FUNCTION_CALL**: Invocaciones a métodos definidos.

Ejemplo de Estructura

Una estructura como:

```
sage x = 5;
x = x heal 1;
```

Se transforma internamente en un nodo de declaración y un nodo de asignación, donde el nodo de la expresión `x heal 1` es un subárbol con un operador binario como raíz.

El archivo `ast.c` implementa los nodos del árbol de sintaxis abstracta (AST). Cada nodo tiene posibles campos como `left`, `right`, `next`, `params`, `args`, y `data_type`.

Algunos tipos de nodos:

- `NODE_CLASS`, `NODE_METHOD`
- `NODE_DECLARATION`, `NODE_ASSIGNMENT`
- `NODE_IF`, `NODE_WHILE`, `NODE_FOR`
- `NODE_OUTPUT`, `NODE_INPUT`, `NODE_FUNCTION_CALL`
- `NODE_PLANT`, `NODE_DEFUSE`

Ejemplo de AST completo

Para el siguiente código:

```
agent Suma {
    sage spike() {
        sage a = 5;
        sage b = 10;
        sage c = a heal b;
        sova c;
        plant 0;
    }
}
```

El AST generado tiene la siguiente forma:

```
NODE_CLASS ("Suma")
 NODE_METHOD ("spike")
     NODE_BLOCK
         NODE_DECLARATION (a: int = 5)
         NODE_DECLARATION (b: int = 10)
         NODE_DECLARATION (c: int)
            NODE_ASSIGNMENT
                NODE_BINARY_OP (heal / +)
                    NODE_IDENTIFIER ("a")
                    NODE_IDENTIFIER ("b")
         NODE_OUTPUT
            NODE_IDENTIFIER ("c")
         NODE_PLANT
             NODE_NUMBER (0)
```

Este árbol es procesado recursivamente por el archivo `interpreter.c` durante la ejecución del programa.

3. Intérprete y Ejecución del AST

El archivo `interpreter.c` implementa el recorrido y ejecución del Árbol de Sintaxis Abstracta (AST), permitiendo simular la ejecución del programa sin generar código intermedio. Esta técnica permite interpretar directamente los nodos del árbol en tiempo de ejecución, manejando estructuras, operaciones y funciones definidas en el lenguaje Valorant.

Estructura del Intérprete

El núcleo del intérprete está en la función recursiva `execute(node_t *node)`, que recibe un nodo del AST y determina qué acción tomar en función del tipo de nodo (`node->type`). A partir de ello, puede ejecutar sentencias, evaluar expresiones o modificar el contexto de ejecución.

El intérprete también mantiene un contexto global que almacena las variables, funciones y clases activas durante la ejecución del programa.

Tipos de Nodos Interpretados

A continuación se resumen los principales tipos de nodos y cómo son procesados por el intérprete:

- **NODE_CLASS**: Define un agente. Se registra en el entorno de ejecución.
- **NODE_METHOD**: Método como `spike()`, se guarda y ejecuta al ser invocado.
- **NODE_BLOCK**: Conjunto de instrucciones ejecutadas secuencialmente.
- **NODE_DECLARATION**: Declara variables y evalúa sus valores iniciales.
- **NODE_ASSIGNMENT**: Evalúa una expresión y actualiza el valor de una variable.
- **NODE_BINARY_OP**: Realiza operaciones binarias como `heal` (suma), `smoke` (resta), etc.
- **NODE_IF**, **NODE_WHILE**, **NODE_FOR**: Estructuras de control condicional y repetitiva.
- **NODE_FUNCTION_CALL**: Invoca métodos definidos en agentes.
- **NODE_OUTPUT** (`sova`): Imprime un valor por pantalla.
- **NODE_INPUT** (`breach`): Solicita un valor desde entrada estándar.
- **NODE_PLANT**, **NODE_DEFUSE**: Terminan la ejecución de un método (como `return`).

Ejemplo de Ejecución

Para el siguiente programa:

```
agent Suma {
    sage spike() {
        sage a = 5;
        sage b = 10;
        sage c = a heal b;
        sova c;
        plant 0;
    }
}
```

El intérprete realiza los siguientes pasos:

1. Registra la clase `Suma` y su método `spike()`.
2. Al invocar `spike()`, se ejecutan las instrucciones:
   - Se declaran e inicializan las variables `a` y `b`.
   - Se evalúa la operación `a heal b`, equivalente a `5 + 10`, y se guarda en `c`.
   - Se imprime `c` usando `sova`.
   - Se termina la ejecución con `plant 0`.

Este enfoque permite una interpretación directa y eficiente del programa fuente sin compilarlo, ideal para prototipos o entornos educativos.

Control de Errores

El intérprete detecta errores como variables no declaradas, tipos incompatibles o división por cero. En caso de error, se detiene la ejecución y se reporta el mensaje al usuario.

1. **Análisis léxico y sintáctico:** Se usa `Flex` y `Bison` para generar el analizador.
2. **AST:** Se construye el árbol mediante las funciones definidas en `ast.c`.
3. **Interpretación:** Se evalúa el AST recursivamente con contexto de ejecución.

4. Análisis Léxico y Sintáctico con Flex y Bison

Para procesar el código fuente del lenguaje Valorant, se utilizan las herramientas **Flex** y **Bison**, ampliamente usadas en la construcción de compiladores.

Flex: Analizador Léxico

Flex es responsable de escanear el código fuente y convertirlo en una secuencia de **tokens**, que son las unidades léxicas del lenguaje (como palabras clave, identificadores, operadores, literales, etc.). Cada token es identificado mediante expresiones regulares definidas en el archivo `valorant.l`.

Por ejemplo:

- `"sage"` → `CLASS_ID`
- `"heal"` → `METHOD_ID`
- `[0-9]+` → `INT`

Cada vez que Flex reconoce un patrón, ejecuta una acción que generalmente consiste en devolver el token correspondiente a Bison.

Bison: Analizador Sintáctico

Bison recibe la secuencia de tokens generada por Flex y la analiza de acuerdo a una **gramática libre de contexto** definida en `valorant.y`. A medida que reconoce producciones, Bison ejecuta acciones semánticas que permiten construir el **Árbol de Sintaxis Abstracta (AST)** del programa.

Ejemplo de una regla en Bison:

```
declaration:
    CLASS_ID VAR '=' expression ';'
    {
        $$ = create_declaration_node($1, $2, $4);
    };
```

En este caso, al reconocer una declaración como `sage x = 5;`, Bison invoca la función `create_declaration_node()` para construir el nodo correspondiente en el AST.

Interacción entre Flex y Bison

El flujo de trabajo entre Flex y Bison se resume así:

1. Flex escanea el código y reconoce tokens.
2. Los tokens se envían a Bison.
3. Bison aplica reglas gramaticales sobre los tokens y construye el AST.

Ambas herramientas trabajan de forma conjunta para transformar el texto fuente en una representación interna estructurada que posteriormente será interpretada por el compilador.

5. Ejemplos de Código

A continuación, se presentan ejemplos que muestran distintas capacidades del lenguaje Valorant.

1. Imprimir Hola Mundo

```
agent Hola {
    sage spike() {
        sova " Hola  Valorant!";
        plant 0;
    }
}
```

2. Sumar dos números

```
agent Suma {
    sage spike() {
        sage a = 5;
        sage b = 10;
        sage c = a heal b;
        sova c;
        plant 0;
    }
}
```

3. Condicional Anidado con Else-If

```
flash (x headshot 0) {
    sova "Cero";
} smoke flash (x win 0) {
    sova "Positivo";
} smoke {
    sova "Negativo";
}
```

4. Bucle While

```
rotate(x lose 5){
    flash (x headshot 3){
        x = 5;
    }
    sova x;
    x = x heal 1;
}
```

Ejemplo Básico

```
class Main {
  sage spike() {
    sage kills = 5;
    sova kills;
    plant 0;
  }
}
```

Salida Esperada

```
Compilador Valorant v1.0

Ejecutando el programa:
```

```
----------------------
5

Programa ejecutado exitosamente
Valor de retorno: 5
```

6. Manual de Usuario

El lenguaje Valorant está diseñado para ser intuitivo y visualmente llamativo, reemplazando las construcciones clásicas de programación por palabras clave que evocan el universo de Valorant.

Declaración de Variables

```
sage kills = 10;
viper radio = 2.5;
cypher name = "Phoenix";
```

Entrada y Salida

```
sova "Ingresa tu edad: ";
breach edad;
```

Condicionales

```
flash (edad win 18) {
    sova "Mayor de edad.";
} smoke {
    sova "Menor de edad.";
}
```

Bucles y Control de Flujo

```
rush (sage i = 0; i lose 5; i = i heal 1) {
    sova i;
}
```

Funciones y Clases

```
agent Calculadora {
    sage suma(sage a, sage b) {
        plant a heal b;
    }
```

```
    sage spike() {
        sage r = suma(3, 7);
        sova r;
        plant 0;
    }
}
```

Declaraciones y clases

```
agent MiClase {
    sage spike() {
    // instrucciones
  }
}
```

Tipos y variables

```
sage edad = 23;
cypher nombre = "Carlos";
```

Entrada/Salida

```
breach nombre;
sova nombre;
```

Control de flujo

```
flash (cond) {
  // ...
} smoke {
  // ...
}
rotate (cond) { [...] }
rush (init; cond; inc) { ... }
rush(int i = 0;i win 10; i = i heal 1)
```

Operadores

`+`, `-`, `*`, `/`, `WIN`, `LOSE`, `HEADSHOT`, `NOTEQUAL`, etc.