1 Tratamiento de los tipos `unsigned`, `long` y `unsigned long` en `CodeGenVisitor`

Durante la generación de código ensamblador, los tipos extendidos y sin signo son tratados cuidadosamente para evitar errores semánticos al pasar de un tipo a otro. Particularmente, se consideran los siguientes aspectos: el tamaño del dato (32 vs 64 bits), la conversión entre tipos con y sin signo, y su impresión mediante `printf`.

1.1 Truncamiento por tamaño

En los casos donde un valor de tipo `long` o `unsigned long` debe almacenarse en una variable de tipo más pequeño (`int` o `unsigned`), se genera una instrucción de truncamiento explícito. Esto se realiza en la función:

Listing 1: Truncamiento de tipos

```
void emit_truncation(const std::string& from_type, const std::string& to_type) {
    if (firstPass) return;
    if ((from_type == "long" && to_type == "int") ||
        (from_type == "unsigned long" && to_type == "unsigned") ||
        (from_type == "unsigned long" && to_type == "int") ||
        (from_type == "long" && to_type == "unsigned")) {
        out << "    movl %eax, %eax\n";  // Limpia bits altos (64 a 32 bits)
    }
}
```

Esta operación asegura que, al reducir el tamaño del registro, los bits más altos se descarten adecuadamente.

1.2 Conversión de negativos a `unsigned`

Un caso especial ocurre cuando se asigna un valor negativo a una variable de tipo sin signo. En estos casos, la función `emit_unsigned_conversion` fuerza la conversión al valor binario equivalente sin signo:

Listing 2: Conversión de negativos a unsigned

```
void emit_unsigned_conversion(long value, const std::string& target_type) {
    if (firstPass) return;
    if (value < 0 && (target_type == "unsigned" || target_type == "unsigned long
        if (target_type == "unsigned") {
            out << "    movl $" << (unsigned int)value << ", %eax\n";
        } else {
            out << "    movq $" << (unsigned long)value << ", %rax\n";
        }
    }
}
```

Esta conversión se aplica automáticamente en declaraciones (`visit(VarDec)`) y asignaciones (`visit(BinaryExp)`).

1.3 Aplicación en declaraciones y asignaciones

En la generación del código para declaraciones (`VarDec`) y expresiones binarias con asignación (`BinaryExp`), se aplican ambas transformaciones cuando corresponde:

Listing 3: Transformaciones en asignación

```
TypeInfo ti = init->accept(this);
emit_truncation(ti.tipo, v->tipo);
if (ti.esNegativo() && TypeInfo(v->tipo).esTipoSinSigno()) {
    emit_unsigned_conversion(ti.valor, v->tipo);
}
```

Esta lógica asegura que los valores asignados o inicializados se adecuen al tipo declarado, respetando las restricciones de signo y tamaño.

1.4 Formatos para impresión con `printf`

Para que los valores se impriman correctamente según su tipo, se usan etiquetas y formatos asociados. El método `get_printf_format` define qué formato de `printf` corresponde a cada tipo:

Listing 4: Selección de formato

```
std::string get_printf_format(const std::string& tipo) {
    if (tipo == "int") return "%d";
    else if (tipo == "long") return "%ld";
    else if (tipo == "unsigned") return "%u";
    else if (tipo == "unsigned long") return "%lu";
}
```

Estas etiquetas se registran en la primera pasada, y su impresión se genera en la segunda.

1.5 Inicialización del entorno y offsets

En la segunda pasada, las funciones y el `main` inicializan el `offset` en cero:

Listing 5: Inicio de offset

```
void second_pass(Program* program) {
    ...
    offset = 0;  //      ARRANCAR EN CERO
    ...
}
```

Esto permite que las variables se ubiquen correctamente en el stack y que su espacio se reserve con precisión mediante `subq $8, %rsp`.

1.6 Resumen

El tratamiento de `unsigned`, `long` y `unsigned long` en la generación de código incluye:

- Truncamiento de 64 a 32 bits cuando se reduce el tamaño.
- Conversión de valores negativos al formato sin signo.
- Registro y uso correcto de los formatos de impresión para cada tipo.
- Mapeo preciso de cada variable en memoria, comenzando desde `offset = 0`.