Helk

Luis Ernesto Amat Cárdenas C-312 (MatCom)

18 de junio de 2025

1. Descripción General del Pipeline del Compilador

El compilador sigue una arquitectura tradicional de múltiples etapas:

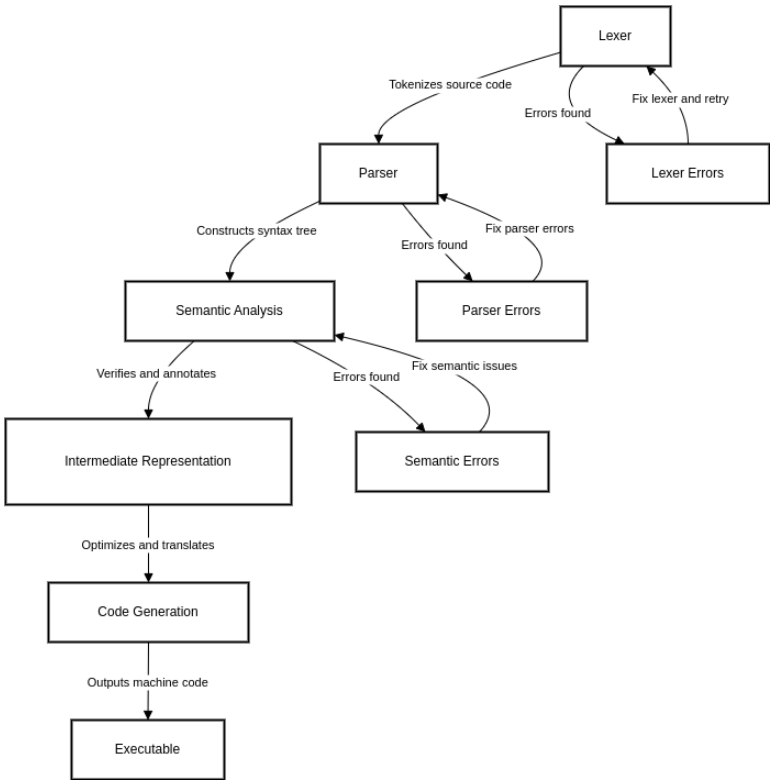

Figura 1: Diagrama de arquitectura del compilador

Componentes principales:

- **Generador de Lexer**: Se genera código válido C que simula las transiciones del autómata finito determinista
- **Generador de Parser**: Parser LL1 descendente recursivo con manejo de errores. Cuenta con un DSL propio para insertar código C.
- **Análisis Semántico**: Enfoque basado en Problemas de Satisfacción de Restricciones (CSP) sobre la hipótesis de que a cada nodo le corresponde un único tipo
- **Generación de Código**: Emisión directa de texto en formato LLVM IR con optimizaciones menores

2. Introducción y Uso

2.1. Compilación del Proyecto

```
# Instalar dependencias
sudo apt install llvm clang

# Compilar proyecto
make compile

# Ejecutar pruebas
pytest test_compiler.py
```

3. Análisis Léxico

3.1. Implementación del Generador de Lexer

Implementación de la máquina de estados a partir de un conjunto de expresiones regulares:

1. Expandir y simplificar la cada expresión regular ($[a-z] \leftarrow abcd...z$)
2. Crear los autómatas finitos no deterministas (NFA) de cada expresión regular.
3. Construir el autómata de la unión de todos los NFAs para definir el lenguaje que reconocerá el lexer.

4. Convertir el NFA de la unión a un DFA (autómata finito determinista), respetando las prioridades de cada regla (keywords ¿itentifier)

5. Simplificar rangos ($abcd...z \leftarrow' a' =< x <=' z'$)

6. Generar el código para simular el autómata usando un alfabeto finito ($ASCII < 128$) usando swtich y goto

Finalmente:

```
Token* lexer(const char* input, int length, unsigned int* _num_tokens) {
    Token token;
    LexerState* lexer_init(state, input, length, true);

    while (state->current < state->end) {
        token = lexer_next_token(state);
    }
}
```

4. Análisis Sintáctico

4.1. Implementación del Parser

1. Obtener la gramática a partir del DSL

2. Generar la tabla (computar FIRST, FOLLOW, ...)

3. Verificar que es realmente parseable por un parser LL1 usando la tabla (a.k.a. que no existan dos entradas en una misma casilla)

4. Generar código C válido para un parser recursivo LL1 usando la tabla y recursividad (cada no terminal es una función)

5. Manejo de errores usando el FOLLOW para recuperarse de errores y seguir parseando, con el afán de encontrar más errores

5. Análisis Semántico con CSP

5.1. Satisfacción de Restricciones

Variables: $X = \{T_1, T_2, \ldots, T_n\}$ (tipos de nodos AST)
Dominios: $D = \{\text{double}, \text{string}, \text{tipo\_personalizado(s)}\}$

5.2. Propagación de Restricciones

```
bool solve_constraints(ConstraintSystem* cs) {
    bool changed;
    bool res = true;
    do {
        changed = false;
        for(size_t i=0; i<cs->count; i++) {
            TypeConstraint* c = &cs->constraints[i];
            TypeInfo* t = c->expected;
            size_t expected = t->kind;
            size_t actual = c->node->type_info.kind;

            // Propagate concrete -> unknown
            if(expected != TYPE_UNKNOWN &&
               actual == TYPE_UNKNOWN) {

                c->node->type_info.kind = ((TypeInfo*) (c->expected))->kind

                changed = true;
            }

            // Check for conflicts
            if(expected != TYPE_UNKNOWN &&
               actual != TYPE_UNKNOWN &&
               expected != actual) {
               fprintf(stderr, "ERROR - Literal type mismatch (current_type
               res = false;
            }
        }
    } while(changed);

    return res;
}
```

6. Generación de Código

7. Optimizaciones

Optimización	**Descripción**
Eliminación de Código Muerto	Elimina funciones no llamadas
Desvirtualización de Métodos	Reemplaza llamadas virtuales por directas

Cuadro 1: Optimizaciones del compilador

8. Conclusión

El compilador implementa un pipeline completo con enfoques novedosos:

- Generación de código directa para Lexer y Parser
- DSLs con posibilidad de escribir código (a lá Lex y Yacc)
- Inferencia de tipos basada en CSP
- Emisión directa de IR

Trabajo futuro incluye optimizaciones de bucles e integración de recolección de basura.

Referencias

[1] Aho, A. V., et al. *Compilers: Principles, Techniques, and Tools*. 2ª ed., Addison-Wesley, 2006.

[2] Lattner, C. *LLVM: An Infrastructure for Multi-Stage Optimization*. Tesis Doctoral, 2002.