Informe Técnico: Compilador Completo MiniLang

Autor: Proyecto de Compiladores

Fecha: Octubre/Noviembre 2025

Versión: 1.0 (Completo)

1. Introducción y Descripción del Lenguaje

MiniLang es un lenguaje de programación simple diseñado para demostrar todas las etapas clásicas del proceso de compilación. El lenguaje soporta tipos enteros implícitos, operaciones aritméticas (+, -, *, /), operadores relacionales (<, >, <=, >=, ==, !=), estructuras de control (if/else, while), entrada/salida (read, print) y bloques delimitados por llaves.

Ejemplo de Programa MiniLang:

```
read a;
read b;
c = a + b * 2;
if c >= 10 {
    print c;
} else {
    print 0;
}
end
```

2. Gramática Formal (EBNF)

La gramática del lenguaje define la estructura sintáctica permitida. Los operadores respetan precedencia estándar: multiplicación/división antes que suma/resta, y operaciones relacionales tienen precedencia sobre igualdad.

```
program ::= { statement } 'end'
statement ::= 'read' IDENT ';'
           | 'print' expression ';'
           | IDENT '=' expression ';'
           | 'if' expression '{' { statement } '}' ...
           | 'while' expression '{' { statement } '}'
expression ::= relation { ('==' | '!=') relation }
relation ::= additive { ('<' | '>' | '<=' | '>=') additive }
additive ::= multiplicative { ('+' | '-') multiplicative }
multiplicative ::= factor { ('*' | '/') factor }
factor ::= NUMBER | IDENT | '(' expression ')'
```

3. Diseño Modular y Arquitectura

Analizador Léxico (lexer.py, tokens.py): Lee el código fuente y genera tokens. Soporta comentarios (// y /* */) y reporta errores con línea/columna.

Analizador Sintáctico (parser.py): Parser recursivo descendente que construye el AST (Árbol Sintáctico Abstracto) según la gramática EBNF.

Analizador Semántico (semantic.py): Construye tabla de símbolos y verifica que variables se inicialicen antes de usarse.

Generador de IR (ir.py): Traduce el AST a TAC (Three-Address Code) usando temporales y etiquetas para control de flujo.

Optimizador (optimizer.py): Aplica constant folding: reemplaza operaciones constantes por sus resultados.

Generador de ASM (codegen_asm.py): Traduce TAC a ensamblador simbólico (PUSH, LOAD, STORE, ADD, SUB, MUL, DIV, JMP, JNZ, etc.).

Ensamblador (codegen_machine.py): Convierte líneas de ASM a tuplas (mnemonic, args) y resuelve etiquetas.

Máquina Virtual (runtime_vm.py): VM basada en pila que ejecuta el código ensamblado con soporte para E/S y saltos condicionales.

Orquestador (compiler.py): Script principal que ejecuta todo el pipeline: lexer → parser → semántica → IR → opt → ASM → VM.

4. Ejemplo Completo de Compilación

Se compilará el programa 'simple_noio.minilang' mostrando cada etapa del pipeline:

Programa Fuente:

```
i = 2;
j = 3;
sum = i + j * 4;
print sum;
end
```

Etapa 1 - Tokens (Lexer):

El lexer genera una secuencia de tokens: IDENT(i), ASSIGN, NUMBER(2), SEMI, IDENT(j), ... TOKEN(END), TOKEN(EOF). Cada token incluye línea y columna para debugging.

Etapa 2 - AST (Parser):

```
El parser construye: Program([Assign('i', Literal(2)), Assign('j', Literal(3)),
Assign('sum', BinaryOp('+', Var('i'), BinaryOp('*', Var('j'), Literal(4)))),
Print(Var('sum'))])
```

Etapa 3 - TAC (Code Generator):

```
TAC(assign, i, 2, None)
```

```
TAC(assign, j, 3, None)
TAC(binop, t1, *, ('j', '4'))
TAC(binop, t2, +, ('i', 't1'))
TAC(assign, sum, t2, None)
TAC(print, sum, None, None)
```

Etapa 4 - Ensamblador (Codegen ASM):

```
PUSH 2      # cargar 2
STORE i     # almacenar en i
PUSH 3      # cargar 3
STORE j     # almacenar en j
LOAD j      # cargar j
PUSH 4      # cargar 4
MUL         # multiplicar
STORE t1    # almacenar en t1
LOAD i      # cargar i
LOAD t1     # cargar t1
ADD         # sumar
STORE t2    # almacenar en t2
LOAD t2     # cargar t2
STORE sum   # almacenar en sum
LOAD sum    # cargar sum
OUT         # imprimir
```

Etapa 5 - Ejecución VM:

Entrada: ninguna
Salida: 14

Explicación: i=2, j=3, sum = 2 + 3*4 = 2 + 12 = 14. El programa imprime 14.

5. Pruebas y Validación

Se incluyen 4 pruebas unitarias que validan cada etapa del compilador:

test_lexer.py: Verifica tokenización correcta.

test_parser.py: Verifica construcción de AST válido.

test_semantic.py: Prueba análisis semántico (acepta válido, rechaza inválido).

test_full_pipeline.py: Prueba completa: compila y ejecuta, verifica salida.

Resultado: Todas las pruebas pasan satisfactoriamente, confirmando que el compilador funciona correctamente.

6. Conclusión

Se ha desarrollado un compilador funcional y didáctico que implementa todas las etapas clásicas del proceso de compilación: análisis léxico, sintáctico y semántico; generación de código intermedio con optimización; y una máquina virtual ejecutable. El proyecto está bien estructurado en módulos independientes, totalmente documentado, y permite extensiones futuras.