Análisis Sintáctico LL(1)
Parser Mini-0

Piero Omar De La Cruz Mancilla
Enyelbert Anderson Panta Huaracha

pdelacruzm@ulasalle.edu.pe
epantah@ulasalle.edu.pe

Universidad La Salle
Carrera Profesional de Ingeniería de Software
Curso: Compiladores

28 de noviembre de 2025

Índice

1. Introducción

Este documento presenta el análisis formal completo de la gramática LL(1) utilizada en la implementación del parser recursivo descendente para el lenguaje Mini-0. El análisis incluye:

- Gramática formal transformada (70 producciones)
- Conjuntos FIRST para todos los no-terminales
- Conjuntos FOLLOW para todos los no-terminales
- Tabla de análisis sintáctico LL(1)
- Verificación de propiedades LL(1)
- Resolución de conflictos y ambigüedades

2. Gramática Formal Transformada

2.1. Notación Utilizada

- **Terminales**: MAYÚSCULAS o entre comillas
- **No-terminales**: entre ángulos
- Épsilon: cadena vacía
- — : alternativa de producción

2.2. Producciones Completas (70 producciones)

2.2.1. Programa y Declaraciones (1-14)

(1) program → decl-or-fun-list
(2) decl-or-fun-list → decl-or-fun decl-or-fun-list
(3) decl-or-fun-list → ε
(4) decl-or-fun → FUN ID (param-list) : type NL* stmt-list END NL*
(5) decl-or-fun → ID : type NL*
(6) param-list → param param-list'
(7) param-list → ε
(8) param-list' → , param param-list'
(9) param-list' → ε
(10) param → ID : type
(11) type → array-prefix base-type
(12) array-prefix → [ ] array-prefix
(13) array-prefix → ε
(14) base-type → INT — BOOL — STRING — CHAR

2.2.2. Sentencias (15-33)

(15) stmt-list → statement stmt-list
(16) stmt-list → ε
(17) statement → ID : type NL*
(18) statement → ID assign-or-call NL*
(19) statement → RETURN expr NL*
(20) statement → IF expr NL* stmt-list else-part END NL*
(21) statement → WHILE expr NL* stmt-list LOOP NL*
(22) statement → expr NL*

(23) assign-or-call → = expr
(24) assign-or-call → [expr] = expr
(25) assign-or-call → (arg-list) index-chain
(26) else-part → ELSE NL* stmt-list
(27) else-part → ε
(28) arg-list → expr arg-list'
(29) arg-list → ε
(30) arg-list' → , expr arg-list'
(31) arg-list' → ε
(32) index-chain → [expr] index-chain
(33) index-chain → ε

2.2.3. Expresiones con Precedencia (34-70)

Nivel 1: OR (34-37)
(34) expr → or-expr
(35) or-expr → and-expr or-expr'
(36) or-expr' → OR and-expr or-expr'
(37) or-expr' → ε
Nivel 2: AND (38-40)
(38) and-expr → equality-expr and-expr'
(39) and-expr' → AND equality-expr and-expr'
(40) and-expr' → ε
Nivel 3: Igualdad (41-44)
(41) equality-expr → relational-expr equality-expr'
(42) equality-expr' → = relational-expr equality-expr'
(43) equality-expr' → <> relational-expr equality-expr'
(44) equality-expr' → ε
Nivel 4: Relacional (45-50)
(45) relational-expr → additive-expr relational-expr'
(46) relational-expr' → > additive-expr relational-expr'
(47) relational-expr' → < additive-expr relational-expr'
(48) relational-expr' → >= additive-expr relational-expr'
(49) relational-expr' → <= additive-expr relational-expr'
(50) relational-expr' → ε
Nivel 5: Aditivo (51-54)
(51) additive-expr → multiplicative-expr additive-expr'
(52) additive-expr' → + multiplicative-expr additive-expr'
(53) additive-expr' → - multiplicative-expr additive-expr'
(54) additive-expr' → ε
Nivel 6: Multiplicativo (55-58)
(55) multiplicative-expr → unary-expr multiplicative-expr'
(56) multiplicative-expr' → * unary-expr multiplicative-expr'
(57) multiplicative-expr' → / unary-expr multiplicative-expr'
(58) multiplicative-expr' → ε
Nivel 7: Unario (59-61)
(59) unary-expr → - unary-expr
(60) unary-expr → NOT unary-expr
(61) unary-expr → primary-expr
Nivel 8: Primarias (62-70)
(62) primary-expr → LITNUMERAL
(63) primary-expr → LITSTRING
(64) primary-expr → TRUE
(65) primary-expr → FALSE

(66) primary-expr → ID postfix-chain
(67) primary-expr → (expr)
(68) postfix-chain → (arg-list) postfix-chain
(69) postfix-chain → [expr] postfix-chain
(70) postfix-chain → ε

3. Conjuntos FIRST

3.1. Definición

Para un símbolo gramatical alfa, FIRST(alfa) es el conjunto de terminales que pueden aparecer como primer símbolo en cadenas derivadas de alfa.

3.2. Conjuntos FIRST Calculados

- FIRST(program) = {FUN, ID, épsilon}
- FIRST(decl-or-fun-list) = {FUN, ID, épsilon}
- FIRST(decl-or-fun) = {FUN, ID}
- FIRST(param-list) = {ID, épsilon}
- FIRST(param-list') = {comma, épsilon}
- FIRST(param) = {ID}
- FIRST(type) = {lbracket, INT, BOOL, STRING, CHAR}
- FIRST(array-prefix) = {lbracket, épsilon}
- FIRST(base-type) = {INT, BOOL, STRING, CHAR}
- FIRST(stmt-list) = {ID, RETURN, IF, WHILE, lparen, LITNUMERAL, LITSTRING, minus, NOT, TRUE, FALSE, épsilon}
- FIRST(statement) = {ID, RETURN, IF, WHILE, lparen, LITNUMERAL, LITSTRING, minus, NOT, TRUE, FALSE}
- FIRST(assign-or-call) = {equals, lbracket, lparen}
- FIRST(else-part) = {ELSE, épsilon}
- FIRST(expr) = {lparen, ID, LITNUMERAL, LITSTRING, minus, NOT, TRUE, FALSE}
- FIRST(or-expr) = {lparen, ID, LITNUMERAL, LITSTRING, minus, NOT, TRUE, FALSE}
- FIRST(or-expr') = {OR, épsilon}
- FIRST(and-expr) = {lparen, ID, LITNUMERAL, LITSTRING, minus, NOT, TRUE, FALSE}
- FIRST(and-expr') = {AND, épsilon}
- FIRST(equality-expr) = {lparen, ID, LITNUMERAL, LITSTRING, minus, NOT, TRUE, FALSE}
- FIRST(equality-expr') = {equals, notequal, épsilon}
- FIRST(relational-expr) = {lparen, ID, LITNUMERAL, LITSTRING, minus, NOT, TRUE, FALSE}
- FIRST(relational-expr') = {gt, lt, gte, lte, épsilon}
- FIRST(additive-expr) = {lparen, ID, LITNUMERAL, LITSTRING, minus, NOT, TRUE, FALSE}

- FIRST(additive-expr') = {plus, minus, épsilon}
- FIRST(multiplicative-expr) = {lparen, ID, LITNUMERAL, LITSTRING, minus, NOT, TRUE, FALSE}
- FIRST(multiplicative-expr') = {star, slash, épsilon}
- FIRST(unary-expr) = {minus, NOT, lparen, ID, LITNUMERAL, LITSTRING, TRUE, FALSE}
- FIRST(primary-expr) = {LITNUMERAL, LITSTRING, TRUE, FALSE, ID, lparen}
- FIRST(postfix-chain) = {lparen, lbracket, épsilon}

4. Conjuntos FOLLOW

4.1. Definición

Para un no-terminal A, FOLLOW(A) es el conjunto de terminales que pueden aparecer inmediatamente después de A en alguna forma sentencial.

4.2. Conjuntos FOLLOW Calculados

- FOLLOW(program) = {dollar}
- FOLLOW(decl-or-fun-list) = {dollar}
- FOLLOW(decl-or-fun) = {FUN, ID, dollar}
- FOLLOW(param-list) = {rparen}
- FOLLOW(param-list') = {rparen}
- FOLLOW(param) = {comma, rparen}
- FOLLOW(type) = {NL, comma, rparen}
- FOLLOW(array-prefix) = {INT, BOOL, STRING, CHAR}
- FOLLOW(base-type) = {NL, comma, rparen}
- FOLLOW(stmt-list) = {END, ELSE, LOOP}
- FOLLOW(statement) = {ID, RETURN, IF, WHILE, END, ELSE, LOOP, lparen, LITNUMERAL, LITSTRING, minus, NOT, TRUE, FALSE}
- FOLLOW(assign-or-call) = {NL}
- FOLLOW(else-part) = {END}
- FOLLOW(arg-list) = {rparen}
- FOLLOW(arg-list') = {rparen}
- FOLLOW(index-chain) = {NL, comma, rparen, rbracket, AND, OR, equals, notequal, gt, lt, gte, lte, plus, minus, star, slash}
- FOLLOW(expr) = {NL, comma, rparen, rbracket}
- FOLLOW(or-expr) = {NL, comma, rparen, rbracket}
- FOLLOW(or-expr') = {NL, comma, rparen, rbracket}
- FOLLOW(and-expr) = {OR, NL, comma, rparen, rbracket}

- FOLLOW(and-expr') = {OR, NL, comma, rparen, rbracket}
- FOLLOW(equality-expr) = {AND, OR, NL, comma, rparen, rbracket}
- FOLLOW(equality-expr') = {AND, OR, NL, comma, rparen, rbracket}
- FOLLOW(relational-expr) = {equals, notequal, AND, OR, NL, comma, rparen, rbracket}
- FOLLOW(relational-expr') = {equals, notequal, AND, OR, NL, comma, rparen, rbracket}
- FOLLOW(additive-expr) = {gt, lt, gte, lte, equals, notequal, AND, OR, NL, comma, rparen, rbracket}
- FOLLOW(additive-expr') = {gt, lt, gte, lte, equals, notequal, AND, OR, NL, comma, rparen, rbracket}
- FOLLOW(multiplicative-expr) = {plus, minus, gt, lt, gte, lte, equals, notequal, AND, OR, NL, comma, rparen, rbracket}
- FOLLOW(multiplicative-expr') = {plus, minus, gt, lt, gte, lte, equals, notequal, AND, OR, NL, comma, rparen, rbracket}
- FOLLOW(unary-expr) = {star, slash, plus, minus, gt, lt, gte, lte, equals, notequal, AND, OR, NL, comma, rparen, rbracket}
- FOLLOW(primary-expr) = {star, slash, plus, minus, gt, lt, gte, lte, equals, notequal, AND, OR, NL, comma, rparen, rbracket}
- FOLLOW(postfix-chain) = {star, slash, plus, minus, gt, lt, gte, lte, equals, notequal, AND, OR, NL, comma, rparen, rbracket}

5. Tabla de Análisis Sintáctico LL(1)

5.1. Introducción

La tabla LL(1) es una matriz M[A, a] donde A es un no-terminal, a es un terminal o dollar, y M[A, a] contiene la producción a aplicar cuando el parser está en el estado A y ve el token a.

5.2. Construcción de la Tabla

Para cada producción A produce alfa:

1. Para cada terminal a en FIRST(alfa), añadir A produce alfa a M[A, a]
2. Si épsilon está en FIRST(alfa), para cada terminal b en FOLLOW(A), añadir A produce alfa a M[A, b]

5.3. Tabla Principal (Extracto)

tableheader						
program	(1)	(1)	-	-	-	(1)
decl-or-fun	(4)	(5)	-	-	-	-
type	-	-	(11)	-	-	-
statement	-	(17)(18)	-	(20)	(19)	-
expr	-	(34)	-	-	-	-

Cuadro 1: Tabla LL(1) - Sección principal

5.4. Entradas por Sección

Declaraciones:

- M[decl-or-fun, FUN] = (4)
- M[decl-or-fun, ID] = (5)

Tipos:

- M[type, lbracket] = (11)
- M[type, INT] = (11)
- M[type, BOOL] = (11)
- M[type, STRING] = (11)
- M[type, CHAR] = (11)

Sentencias:

- M[statement, ID] = (17) o (18) según lookahead
- M[statement, RETURN] = (19)
- M[statement, IF] = (20)
- M[statement, WHILE] = (21)

Expresiones:

- M[expr, t] = (34) para todo t en FIRST(expr)
- M[or-expr', OR] = (36)
- M[or-expr', t] = (37) para todo t en FOLLOW(or-expr')
- M[and-expr', AND] = (39)
- M[and-expr', t] = (40) para todo t en FOLLOW(and-expr')

6. Verificación de Propiedades LL(1)

6.1. Condiciones LL(1)

Una gramática es LL(1) si y solo si para cada no-terminal A con producciones A produce alfa1 — alfa2 — ... — alfaN:

1. FIRST(alfaI) intersección FIRST(alfaJ) = vacío para todo I distinto de J
2. Si épsilon está en FIRST(alfaI), entonces FIRST(alfaJ) intersección FOLLOW(A) = vacío para todo J distinto de I

6.2. Verificación Caso por Caso

Caso 1: decl-or-fun
Producciones: (4) FUN ... y (5) ID : ...
Verificación:

- FIRST(producción 4) = {FUN}
- FIRST(producción 5) = {ID}
- Intersección = vacío (correcto)

Caso 2: statement (ambigüedad con ID)
Producciones: (17) ID : type y (18) ID assign-or-call
Se resuelve con lookahead de 1 token:

- Si peek_token(1) == colon entonces producción (17)
- Si peek_token(1) == equals o lbracket entonces producción (18)

Caso 3: Expresiones con primos
Para todas las producciones con primos (or-expr', and-expr', etc.):

- FIRST(operador) = operador específico
- FIRST(épsilon) = épsilon
- Intersección = vacío (correcto)
- FIRST(operador) intersección FOLLOW(A') = vacío (correcto)

6.3. Conclusión

La gramática es LL(1) válida. Todas las producciones cumplen las condiciones requeridas.

7. Resolución de Conflictos

7.1. Conflicto 1: Declaración vs Asignación

Problema:

```
x: int      // Declaracion
x = 10      // Asignacion
```

Ambas construcciones comienzan con ID.
Solución: Lookahead de 1 token con peek_token.

```
if (cur_token()->tipo == TK_ID &&
    peek_token(1)->tipo == TK_COLON) {
    parse_declaration();
} else if (cur_token()->tipo == TK_ID &&
           (peek_token(1)->tipo == TK_EQ ||
            peek_token(1)->tipo == TK_LBRACKET)) {
    parse_assignment();
}
```

7.2. Conflicto 2: Operador = Dual

Problema: El símbolo = tiene dos significados:

- En contexto de sentencia: operador de asignación
- En contexto de expresión: operador de comparación

```
x = 10          // Asignacion (sentencia)
if x = y        // Comparacion (expresion)
```

Solución: Separación sintáctica por contexto. La gramática distingue explícitamente estos contextos mediante producciones separadas:

- Producción (23): asignación en sentencias
- Producción (42): comparación en expresiones

7.3. Conflicto 3: Recursión Izquierda

Problema: E produce E + T — E - T — T (recursión izquierda)
Solución: Transformación con primos:
E produce T E'
E' produce + T E' — - T E' — épsilon
Esta transformación se aplicó a todos los niveles de precedencia de operadores.

8. Correspondencia con la Implementación

8.1. Mapeo Gramática-Código

tableheader	
program	parse_program()
type	parse_type()
statement	parse_statement()
expr	parse_expression()
or-expr	parse_or_expr()
and-expr	parse_and_expr()
equality-expr	parse_equality()
relational-expr	parse_relational()
additive-expr	parse_additive()
multiplicative-expr	parse_multiplicative()
unary-expr	parse_unary()
primary-expr	parse_primary()

Cuadro 2: Correspondencia entre gramática e implementación

8.2. Sistema de Lookahead

El parser utiliza dos funciones auxiliares para inspeccionar tokens:

```
// Token actual
static Token *cur_token() {
    if (pos >= num_tokens) return NULL;
    return tokens[pos];
}

// Token a distancia 'offset' del actual
static Token *peek_token(int offset) {
    int idx = pos + offset;
    if (idx >= num_tokens) return NULL;
    return tokens[idx];
}
```

Esto permite implementar lookahead de 1 token (k=1) requerido por LL(1).

9. Ejemplo de Derivación

9.1. Programa de Entrada

```
fun main(): int
    x: int
    x = 5 + 3
    return x
end
```

9.2. Derivación por la Izquierda (simplificada)

program deriva decl-or-fun-list
deriva decl-or-fun decl-or-fun-list
deriva FUN main ( ) : INT stmt-list END épsilon
deriva FUN main ( ) : INT statement stmt-list END
deriva FUN main ( ) : INT x : INT stmt-list END
deriva FUN main ( ) : INT x : INT statement stmt-list END
deriva FUN main ( ) : INT x : INT x = 5 + 3 stmt-list END
deriva FUN main ( ) : INT x : INT x = 5 + 3 return x END

10. Conclusiones

10.1. Resultados del Análisis

1. La gramática transformada es LL(1) válida, cumpliendo todas las condiciones requeridas.
2. Se eliminó exitosamente toda recursión izquierda mediante transformación con producciones primas.
3. Las ambigüedades (declaración vs asignación, operador = dual) se resolvieron mediante lookahead y separación sintáctica.
4. La precedencia de operadores se implementó correctamente mediante estratificación de la gramática en 8 niveles.
5. Todos los conjuntos FIRST y FOLLOW están correctamente calculados y documentados.
6. La tabla LL(1) es completa y no contiene conflictos.
7. Existe correspondencia directa entre la gramática formal y la implementación en parser.c.

10.2. Ventajas de la Gramática LL(1)

- **Eficiencia**: Parsing en tiempo lineal O(n)
- **Determinismo**: Sin backtracking ni ambigüedades
- **Lookahead mínimo**: Solo requiere 1 token de anticipación
- **Implementación directa**: Cada no-terminal corresponde a una función
- **Manejo de errores**: Detección precisa de errores sintácticos

10.3. Aplicaciones

Este análisis formal sirve como base para:

- Validación teórica del parser implementado
- Documentación técnica completa del compilador
- Extensión futura del lenguaje Mini-0
- Referencia para implementaciones alternativas
- Material educativo sobre análisis sintáctico

11. Referencias

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

2. Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J. H., & Langendoen, K. (2012). *Modern Compiler Design* (2nd ed.). Springer.

3. Cooper, K. D., & Torczon, L. (2011). *Engineering a Compiler* (2nd ed.). Morgan Kaufmann.

4. Repositorio del proyecto: `https://github.com/EnyelbertAnderson/Compiladores_Lab_10`