Informe Técnico: Analizador Sintáctico para Mini-0

Universidad La Salle - Compiladores
Trabajo Práctico 2 - Parser Recursivo Descendente

Integrantes:

- Angela Milagros Quispe Huanca y Leonardo Raphael Pachari Gomez

---

1. Introducción

Este informe documenta el diseño e implementación de un analizador sintáctico (parser) recursivo descendente para el lenguaje Mini-0. El parser realiza análisis sintáctico predictivo LL(1) y valida que los programas cumplan con la gramática formal del lenguaje.

Objetivos

- Transformar la gramática original de Mini-0 a una forma compatible con análisis LL(1)
- Implementar un parser recursivo descendente en C
- Detectar y reportar errores sintácticos
- Validar programas Mini-0 con casos de prueba exhaustivos

---

2. Gramática Original de Mini-0

La gramática original presenta las siguientes características problemáticas para análisis descendente:

```
Program     → FunctionList
FunctionList → Function | Function FunctionList
Function    → 'fun' ID '(' Params ')' ':' Type Statements 'end'
Params      → Param | Param ',' Params |
Statements  → Statement | Statement Statements
Statement   → Declaration | Assignment | IfStmt | WhileStmt | Return
Expression  → Expression 'or' AndExpr | AndExpr
AndExpr     → AndExpr 'and' RelExpr | RelExpr
RelExpr     → AddExpr RelOp AddExpr | AddExpr
AddExpr     → AddExpr '+' Term | AddExpr '-' Term | Term
Term        → Term '*' Factor | Term '/' Factor | Factor
Factor      → 'not' Factor | '-' Factor | Primary
Primary     → ID | Number | String | '(' Expression ')' | ...
```

Problemas identificados:

1. **Recursión por la izquierda** en:
   - `FunctionList`, `Statements`, `Expression`, `AndExpr`, `AddExpr`, `Term`
2. **Ambigüedad** en:
   - Expresiones (precedencia de operadores)
   - Listas (iteración vs recursión)
3. **Conflictos FIRST/FOLLOW** en múltiples producciones

---

3. Transformaciones Aplicadas

3.1 Eliminación de Recursión por la Izquierda

Aplicamos la transformación estándar: `A → A |` se convierte en: - `A → A'` - `A' → A' |`

Ejemplo: Expression Antes:

```
Expression → Expression 'or' AndExpr | AndExpr
```

Después:

```
Expression  → AndExpr ExprPrime
ExprPrime   → 'or' AndExpr ExprPrime |
```

Transformaciones completas: FunctionList:

```
FunctionList → Function FunctionListPrime
FunctionListPrime → Function FunctionListPrime |
```

StmtList:

```
StmtList → Statement StmtListPrime
StmtListPrime → Statement StmtListPrime |
```

AddExpr:

```
AddExpr → Term AddExprPrime
AddExprPrime → ('+' | '-') Term AddExprPrime |
```

Term:

```
Term → Factor TermPrime
TermPrime → ('*' | '/') Factor TermPrime |
```

AndExpr:

```
AndExpr → RelExpr AndExprPrime
AndExprPrime → 'and' RelExpr AndExprPrime |
```

3.2 Factorización Común

Eliminamos prefijos comunes que causan conflictos:

ParamList: Antes:

```
Params → Param | Param ',' Params |
```

Después:

```
ParamList → Param ParamListPrime |
ParamListPrime → ',' Param ParamListPrime |
```

Statement (Declaración vs Asignación): Problema: `ID ':' Type` vs `ID '=' Expression`

Solución: Usar lookahead de 2 tokens:

```
if (current_token == ID && next_token == ':')
    → Declaration
else if (current_token == ID)
    → Assignment
```

3.3 Precedencia y Asociatividad

Precedencia de operadores (menor a mayor): 1. `or` (lógico) 2. `and` (lógico) 3. `<`, `>`, `<=`, `>=`, `=`, `<>` (relacionales) 4. `+`, `-` (aditivos) 5. `*`, `/` (multiplicativos) 6. `not`, `-` unarios (prefijos)

Asociatividad: Todos los operadores binarios son asociativos por la izquierda.

---

4. Gramática Transformada (LL(1))

```
Program         → FunctionList
FunctionList    → Function FunctionListPrime
FunctionListPrime → Function FunctionListPrime |

Function        → 'fun' ID '(' ParamList ')' ':' Type StmtList 'end'
ParamList       → Param ParamListPrime |
ParamListPrime  → ',' Param ParamListPrime |
Param           → ID ':' Type

Type            → ArrayType | 'int' | 'bool' | 'string' | 'char'
ArrayType       → '[]' Type

StmtList        → Statement StmtListPrime
StmtListPrime   → Statement StmtListPrime |
```

```
Statement       → Declaration | Assignment | IfStmt | WhileStmt | ReturnStmt
Declaration     → ID ':' Type
Assignment      → ID CallOrArray '=' Expression
IfStmt          → 'if' Expression StmtList ElsePart 'end'
ElsePart        → 'else' StmtList |
WhileStmt       → 'while' Expression StmtList 'loop'
ReturnStmt      → 'return' Expression

Expression      → AndExpr ExprPrime
ExprPrime       → 'or' AndExpr ExprPrime |

AndExpr         → RelExpr AndExprPrime
AndExprPrime    → 'and' RelExpr AndExprPrime |

RelExpr         → AddExpr RelExprPrime
RelExprPrime    → RelOp AddExpr |
RelOp           → '<' | '>' | '<=' | '>=' | '=' | '<>'

AddExpr         → Term AddExprPrime
AddExprPrime    → ('+' | '-') Term AddExprPrime |

Term            → Factor TermPrime
TermPrime       → ('*' | '/') Factor TermPrime |

Factor          → 'not' Factor | '-' Factor | Primary

Primary         → ID CallOrArray
                | NUMERAL
                | LITSTRING
                | 'true'
                | 'false'
                | '(' Expression ')'
                | 'new' '[' Expression ']' Type

CallOrArray     → '(' ArgList ')' | '[' Expression ']' |
ArgList         → Expression ArgListPrime |
ArgListPrime    → ',' Expression ArgListPrime |
```

---

5. Tabla de Análisis Sintáctico LL(1)

Convenciones:

- **Terminales (columnas):** tokens del lenguaje
- **No terminales (filas):** símbolos de la gramática
- **Entradas:** número de regla a aplicar

- —: error sintáctico

Tabla Principal

Enlace: https://github.com/leonardouwz/Compiladores/TP02/TablaPrincipal.pdf

Reglas numeradas:

1. `Program → FunctionList`
2. `FunctionList → Function FunctionListPrime`
3. `FunctionListPrime → Function FunctionListPrime`
4. `FunctionListPrime →`
5. `Function → 'fun' ID '(' ParamList ')' ':' Type StmtList 'end'`
6. `ParamList → Param ParamListPrime`
7. `ParamList →`
8. `ParamListPrime → ',' Param ParamListPrime`
9. `ParamListPrime →`
10. `Type → 'int'`
11. `Type → 'bool'`
12. `Type → 'string'`
13. `Type → ArrayType`
14. `StmtList → Statement StmtListPrime`
15. `StmtListPrime → Statement StmtListPrime`
16. `StmtListPrime →`
17. `Statement → Declaration | Assignment` *(usar lookahead)*
18. `Statement → IfStmt`
19. `Statement → WhileStmt`
20. `Statement → ReturnStmt`
21. `Expression → AndExpr ExprPrime`
22. `ExprPrime → 'or' AndExpr ExprPrime`
23. `ExprPrime →`
24. `AddExpr → Term AddExprPrime`
25. `AddExprPrime → ('+' | '-') Term AddExprPrime`
26. `AddExprPrime →`
27. `Term → Factor TermPrime`
28. `TermPrime → ('*' | '/') Factor TermPrime`
29. `TermPrime →`
30. `Factor → Primary`
31. `Factor → '-' Factor`
32. `Factor → 'not' Factor`
33. `Primary → ID CallOrArray`
34. `Primary → '(' Expression ')'`
35. `Primary → 'true'`
36. `Primary → 'false'`
37. `Primary → NUMERAL`

38. `Primary → LITSTRING`
39. `Primary → 'new' '[' Expression ']' Type`

---

6. Diseño del Parser

6.1 Arquitectura

El parser implementa el patrón **Recursive Descent** con las siguientes componentes:

1. **Estructura Parser:**

```c
typedef struct {
    Token* tokens;      // Array de tokens del léxico
    int token_count;    // Cantidad total de tokens
    int current;        // Índice del token actual
    bool has_error;     // Flag de error
    int error_count;    // Contador de errores
} Parser;
```

2. **Funciones por No Terminal:**
   - Cada no terminal tiene su función correspondiente
   - Retornan `bool` (éxito/fallo)
   - Implementan la lógica de la regla de producción

3. **Funciones Auxiliares:**
   - `match(type)`: verifica si el token actual es del tipo esperado
   - `consume(type, msg)`: consume token o reporta error
   - `current_token()`: obtiene token actual (saltando NL)
   - `peek_token(offset)`: mira tokens adelante

6.2 Correspondencia Gramática-Código

Cada regla de producción se traduce directamente a código:

Ejemplo 1: Regla simple

```
Function → 'fun' ID '(' ParamList ')' ':' Type StmtList 'end'
```

```c
bool parse_function(Parser* p) {
    if (!consume(p, TK_FUN, "Se esperaba 'fun'")) return false;
    if (!consume(p, TK_ID, "Se esperaba ID")) return false;
    if (!consume(p, TK_LPAREN, "Se esperaba '('")) return false;
    if (!parse_param_list(p)) return false;
    if (!consume(p, TK_RPAREN, "Se esperaba ')'")) return false;
    if (!consume(p, TK_COLON, "Se esperaba ':'")) return false;
    if (!parse_type(p)) return false;
```

```
    if (!parse_stmt_list(p)) return false;
    if (!consume(p, TK_END, "Se esperaba 'end'")) return false;
    return true;
}
```

Ejemplo 2: Regla con recursión (eliminada)

```
ExprPrime → 'or' AndExpr ExprPrime |
```

```
bool parse_expr_prime(Parser* p) {
    if (match(p, TK_OR)) {
        p->current++;
        if (!parse_and_expr(p)) return false;
        return parse_expr_prime(p);  // Recursión a la derecha
    }
    return true;  // epsilon
}
```

6.3 Manejo de Saltos de Línea

Los tokens `TK_NL` se saltan automáticamente en `current_token()`:

```
Token* current_token(Parser* p) {
    while (p->current < p->token_count &&
           p->tokens[p->current].type == TK_NL) {
        p->current++;
    }
    return &p->tokens[p->current];
}
```

---

7. Manejo de Errores

7.1 Tipos de Errores Detectados

Categoría	Descripción	Ejemplo
Léxicos	Caracteres inválidos	`@`, `#`, `$`
Falta de tokens	Token esperado no encontrado	Falta `)`, `end`, `:`
Tipo inválido	Tipo no reconocido	`x: invalid_type`
Expresión incompleta	Expresión mal formada	`x = 5 +`
Estructura incorrecta	Violación de reglas	Falta `loop` después de `while`
EOF inesperado	Fin de archivo prematuro	Función sin `end`

7.2 Estrategias de Recuperación

1. **Modo Pánico:**
   - Al detectar error, reportar inmediatamente
   - Continuar análisis para encontrar más errores
   - No detener en el primer error
2. **Sincronización:**

```
void skip_to_sync(Parser* p) {
    while (!is_at_end(p)) {
        Token* tok = current_token(p);
        if (tok->type == TK_END || tok->type == TK_FUN) {
            return;  // Punto de sincronización
        }
        p->current++;
    }
}
```

3. **Puntos de sincronización:**
   - `end` (fin de bloque)
   - `fun` (nueva función)
   - Delimitadores de statement

7.3 Mensajes de Error

Formato estándar:

```
Error sintáctico en línea N: [MENSAJE]. Se encontró '[LEXEMA]' (TIPO)
```

Ejemplos:

```
Error sintáctico en línea 5: Se esperaba 'end'. Se encontró '$' (EOF)
Error sintáctico en línea 12: Se esperaba ')'. Se encontró ';' (SEMICOLON)
Error léxico en línea 8: caracter inválido '@'
```

7.4 Códigos de Salida

- **0:** Análisis exitoso
- **1:** Error léxico o sintáctico detectado

---

8. Casos de Prueba

8.1 Programas Válidos

Test	Descripción	Reglas Ejercitadas
test1.mini0	Función factorial recursiva	Función, if-else, return, expresiones aritméticas
test2.mini0	Literales (decimales, hex, strings)	Literales, declaraciones, asignaciones
test3.mini0	Comentarios de línea y bloque	Manejo de comentarios
test5.mini0	Arrays y loops	Arrays, new, while-loop, acceso indexado
test6.mini0	Operadores lógicos y relacionales	and, or, not, <, >, =, <>, if-else anidado

8.2 Programas con Errores

Test	Error	Mensaje Esperado
error1.mini0	Falta `end`	"Se esperaba 'end' "
error2.mini0	Falta `)`	"Se esperaba ')' "
error3.mini0	Tipo inválido	"Se esperaba un tipo"
error4.mini0	Falta `:`	"Se esperaba ':' "
error5.mini0	Expresión incompleta	"Se esperaba una expresión"
error6.mini0	Falta `loop`	"Se esperaba 'loop' "
error7.mini0	Falta `=`	"Se esperaba '=' "
error8.mini0	Array sin tipo	"Se esperaba un tipo"
error9.mini0	Return sin expresión	"Se esperaba una expresión"
error10.mini0	Paréntesis no balanceados	"Se esperaba ')' "

8.3 Cobertura de la Gramática

Todas las reglas fueron ejercitadas:

Program, FunctionList
Function, ParamList
Type, ArrayType
StmtList, Statement
Declaration, Assignment
IfStmt (con else y sin else)
WhileStmt
ReturnStmt
Expression (todas las precedencias)
Primary (todos los casos)
CallOrArray (llamadas y arrays)

---

9. Resultados

9.1 Compilación

```
$ make
gcc -Wall -Wextra -g -c main.c
gcc -Wall -Wextra -g -c parser.c
gcc -Wall -Wextra -g -c token.c
flex mini0.l
gcc -Wall -Wextra -g -c lex.yy.c
gcc -Wall -Wextra -g -o mini0_parser main.o parser.o token.o lex.yy.o -lfl
```

9.2 Ejecución Pruebas Válidas

```
$ ./mini0_parser tests/test1.mini0
  Análisis sintáctico exitoso
Programa válido en Mini-0

$ echo $?
0
```

9.3 Ejecución Pruebas con Errores

```
$ ./mini0_parser tests/error1.mini0
Error sintáctico en línea 5: Se esperaba 'end'. Se encontró '$' (EOF)

  Análisis sintáctico fallido
Total de errores sintácticos: 1

$ echo $?
1
```

---

10. Conclusiones

10.1 Logros

1. **Gramática LL(1) funcional:** Se eliminó toda recursión por la izquierda y ambigüedades
2. **Parser completo:** Reconoce todo el lenguaje Mini-0
3. **Detección de errores robusta:** Múltiples tipos de errores con mensajes claros
4. **Cobertura total:** Todos los casos de prueba validan todas las reglas

10.2 Desafíos Superados

1. **Distinción Declaration/Assignment:** Resuelto con lookahead de 2 tokens

2. **Precedencia de operadores:** Implementada correctamente en la jerarquía de expresiones
3. **Manejo de epsilon:** Producciones epsilon implementadas correctamente
4. **Saltos de línea:** Filtrados transparentemente sin afectar la gramática

10.3 Mejoras Futuras

1. **Recuperación de errores:** Implementar modo pánico más sofisticado
2. **Mensajes contextuales:** Sugerencias de corrección
3. **AST:** Construir árbol de sintaxis abstracta para análisis semántico
4. **Optimización:** Caché de tokens para reducir accesos

10.4 Aprendizajes

- Importancia de transformar gramáticas para parsing eficiente
- Correspondencia directa entre gramática formal y código
- Diseño de mensajes de error para facilitar debugging
- Pruebas exhaustivas son cruciales para validar el parser

Referencias

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.)
- Grune, D., & Jacobs, C. J. H. (2008). *Parsing Techniques: A Practical Guide* (2nd ed.)
- Material del curso Compiladores, Universidad La Salle