Informe del Compilador de MiniLang

Zakaria Abouhammadi

January 14, 2025

1 Introducción

El presente documento describe en detalle un **compilador** desarrollado para un lenguaje llamado **MiniLang**. Este lenguaje está orientado a la enseñanza de conceptos fundamentales de construcción de compiladores, incluyendo:

- *Análisis léxico:* convertir el código fuente en tokens.
- *Análisis sintáctico:* validar la estructura gramatical y construir un árbol sintáctico abstracto (AST).
- *Análisis semántico:* verificar reglas de tipos, alcance, etc.

En MiniLang, el usuario puede definir constantes (`CONST`), variables, y *subrutinas* con uno o varios parámetros. Además, el lenguaje exige la existencia de una subrutina principal llamada `main` (en minúsculas) y de tipo `VOID`. Las palabras reservadas se utilizan en **MAYÚSCULAS** (por ejemplo, `SUBROUTINE`, `IF`, `THEN`, `RETURN`, etc.).

El compilador se ha creado usando la biblioteca https://www.dabeaz.com/ply/**PLY** de Python, la cual provee herramientas para análisis léxico (`ply.lex`) y sintáctico (`ply.yacc`).

2 Flujo General del Compilador

El proceso completo de compilación para MiniLang se describe a continuación:

1. **Lectura del Fichero:** Se lee el archivo `.minilang` que contiene el código fuente.
2. **Análisis Léxico:** El lexer convierte la secuencia de caracteres en *tokens* (por ejemplo, `ID`, `INT_LITERAL`, `SUBROUTINE`, `DO`, etc.).
3. **Análisis Sintáctico (Parser):** El parser, usando las reglas gramaticales, valida que la secuencia de tokens siga la estructura esperada. De ser correcto, produce un *árbol sintáctico abstracto* (AST) con nodos representando declaraciones de constantes, variables, subrutinas, estructuras de control, etc.
4. **Análisis Semántico:** Se recorre el AST para verificar reglas de semántica; por ejemplo:
   - No reasignar `CONST`.
   - Exigir que la subrutina principal se llame `main` y sea `VOID`.
   - Validar tipos en operaciones (`INT + INT`, `STRING + STRING`, etc.).
   - Subrutinas no-`VOID` deben contener un `return`.
   - Coincidencia de parámetros en las llamadas (`sumar(10, 5)` debe contener 2 parámetros si `sumar` se definió con 2).
5. **Salida:** Si el código supera todas las verificaciones, se informa que es válido; si se detectan errores (sintácticos o semánticos), se muestran mensajes detallados.

3 Estructura y Partes Relevantes

3.1 Análisis Léxico

En esta etapa, se define una lista de tokens (por ejemplo, `PLUS`, `MINUS`, `INT_LITERAL`, etc.) y expresiones regulares que permiten reconocer cada token. Las palabras reservadas, definidas en **MAYÚSCULAS**, se detectan transformando cualquier `ID` que coincida con ellas al token adecuado (por ejemplo, `THEN`).

3.2 Análisis Sintáctico (Parser) y AST

El parser se basa en reglas gramaticales establecidas con funciones `p_...` en PLY. Cada regla produce una *estructura de AST* si la parte derecha es válida. Ejemplos de nodos:

- `ProgramNode`: Raíz del AST, con secciones de constantes, variables, y subrutinas.
- `SubroutineNode`: Contiene `rtype`, `name`, `params` y el cuerpo (`body`).
- `BinOpNode`, `CallNode`, `UnOpNode`: Manejan expresiones aritméticas, lógicas o llamadas a subrutina (ej. `sumar(x,5)`).

3.3 Análisis Semántico

En la fase semántica se implementa una *tabla de símbolos* (pila de diccionarios) para controlar:

- Declaraciones de variables y constantes (que no pueden reasignarse).
- Subrutinas, con información de tipos y parámetros.
- Validación de subrutina `main` como `VOID`.
- Control de tipos en expresiones y verificación de llamadas a subrutinas.

4 Decisiones Técnicas y Desafíos

Uso de PLY: Facilita la definición de tokens y reglas gramaticales, similar a `Lex/Yacc`. Se eligió Python para un desarrollo rápido y claro.

Palabras Clave en MAYÚSCULAS: Se optó por mayúsculas para distinguir más fácilmente palabras reservadas de identidades, evitando colisiones con nombres de variables.

Llamadas a subrutina dentro de expresiones: Fue un reto, ya que no solo queríamos `call_stmt` como sentencia (`sumar(a,b);`), sino también expresiones del tipo `"Hola" + sumar(x,5)`. Se resolvió con la producción:

```
expression :  ID LPAREN arg_list RPAREN
```

permitiendo que `ID(...)` sea interpretado como un *CallNode* incluso dentro de una expresión.

Ambigüedades y reduce/reduce conflicts: Fue necesario ajustar las reglas de `empty` en `const_section` y `var_section` para reducir los conflictos en PLY.

Reasignación de Constantes y `main` en minúsculas: Se dedicó tiempo a asegurar que `CONST` no se reasigne y que la subrutina principal `main` estuviese en minúsculas, exigiendo `VOID`.

5 Limitaciones

- **No genera código máquina:** El compilador se queda en la construcción del AST y la validación semántica.
- **Pocos tipos:** Solo maneja `INT`, `BOOL`, `STRING`.
- **Ámbitos sencillos:** No hay manejo avanzado de scopes anidados.
- **Recuperación de errores básica:** Si encuentra un error sintáctico serio, el parseo se detiene.

6 Conclusiones

El compilador de MiniLang creado cumple los objetivos de ilustrar las fases básicas de construcción de un compilador: análisis léxico, sintáctico y semántico. Si bien existen limitaciones, provee una base sólida para quienes deseen aprender cómo se transforman declaraciones y sentencias del código a estructuras en memoria y se comprueban las reglas del lenguaje.

El soporte de llamadas a subrutinas dentro de expresiones, las secciones de constantes/variables y la obligatoriedad de una `main` (minúsculas) de tipo `VOID` son características que aportan al estudio de cómo se definen y aplican las reglas sintácticas y semánticas en un lenguaje relativamente simple, pero ilustrativo de los pilares de la compilación.