Informe: Construcción de un Generador de Lexer

Nombre del Autor

7 de julio de 2025

Resumen

El proyecto consiste en el diseño e implementación de un compilador desarrollado en Rust, que utiliza LLVM para la generación de código intermedio y clang para la compilación final. El compilador abarca todas las fases fundamentales: análisis léxico (lexer), sintáctico (parser), semántico y generación de código. El flujo de trabajo parte de la lectura del código fuente, construcción del Árbol de Sintaxis Abstracta (AST), validación semántica y generación de código intermedio (LLVM-IR), hasta la obtención del ejecutable final. El proceso está automatizado mediante un Makefile, facilitando la compilación, ejecución y limpieza de artefactos generados.

Nota: Para una comprensión rigurosa, se recomienda precisar las definiciones de cada fase del compilador y los conceptos de autómatas y gramáticas, ya que son fundamentales para el diseño de analizadores léxicos y sintácticos.

1. Gramática Formal del Lenguaje

La gramática del lenguaje que se va a compilar se define formalmente mediante una gramática libre de contexto (GLC), especificando los símbolos terminales, no terminales, el símbolo inicial y las reglas de producción. Es importante precisar que una GLC se expresa como una cuádrupla (V, Σ, R, S), donde V es el conjunto de símbolos no terminales, Σ el conjunto de símbolos terminales, R el conjunto de reglas de producción y S el símbolo inicial.

Símbolos terminales

- `IDENTIFIER`, `NUMBER`, `STRING`
- `FUNCTION`, `TYPE`, `LET`, `IN`, `IF`, `ELSE`, `ELIF`, `WHILE`, `FOR`, `NEW`, `PRINT`, `INHERITS`, `TRUE`, `FALSE`
- `ASSIGN_OP`, `DESTRUCTIVE_ASSIGN_OP`, `ARROW_OP`, `LOGICAL_OR_OP`, `LOGICAL_AND_OP`
- `EQUALITY_OP`, `COMPARISON_OP`, `TERM_OP`, `FACTOR_OP`, `POW_OP`, `UNARY_OP`, `DOT_OP`
- `LPAREN`, `RPAREN`, `LBRACE`, `RBRACE`, `SEMICOLON`, `COMMA`, `COLON`

Precisión: Los símbolos terminales corresponden a los tokens reconocidos por el lexer. Es recomendable definir formalmente las expresiones regulares asociadas a cada token para evitar ambigüedades.

Símbolos no terminales

- `Program`, `Statement`, `FunctionDef`, `FunctionFullDef`, `FunctionArrowDef`
- `TypeDef`, `Inheritance`, `TypeBody`, `MemberDef`, `Expr`, `DestructiveAssignExpr`
- `LogicalOrExpr`, `LogicalAndExpr`, `EqualityExpr`, `ComparisonExpr`, `TermExpr`
- `FactorExpr`, `ExponentExpr`, `UnaryExpr`, `CompositeExpr`, `PrimaryExpr`
- `Literal`, `FunctionCall`, `ArgList`, `LetIn`, `IfElse`, `WhileLoop`, `ForLoop`
- `CodeBlock`, `ExprList`, `PrintExpr`, `AssignmentList`, `Assignment`, `Params`, `Signature`

Precisión: Los símbolos no terminales definen la estructura sintáctica del lenguaje. Es recomendable identificar el símbolo inicial explícitamente (en este caso, `Program`).

Definiciones léxicas

`IDENTIFIER [A-Za-z][A-Za-z0-9_]*`

`NUMBER [0-9]+(.[0-9]+)?`

`STRING ``([^"\\]\\.)*''`

`BOOLEAN true|false`

Operadores

`ASSIGN_OP =`

`DESTRUCTIVE_ASSIGN_OP :=`

`ARROW_OP =>`

`LOGICAL_OR_OP |`

`LOGICAL_AND_OP &`

`EQUALITY_OP == !=`

`COMPARISON_OP < <= > >=`

`TERM_OP + - @`

`FACTOR_OP * / %`

`POW_OP ^`

`UNARY_OP ! -`

`DOT_OP .`

Delimitadores

LPAREN (

RPAREN)

LBRACE {

RBRACE }

SEMICOLON ;

COMMA ,

COLON :

Palabras reservadas

FUNCTION function

TYPE type

LET let

IN in

IF if

ELSE else

ELIF elif

WHILE while

FOR for

NEW new

PRINT print

INHERITS inherits

Reglas de producción

$$
\begin{aligned}
\text{Program} &\rightarrow (\text{Statement SEMICOLON})^* \text{ Statement? SEMICOLON} \\
\text{Statement} &\rightarrow \text{FunctionDef} \mid \text{TypeDef} \mid \text{Expr} \\
\text{FunctionDef} &\rightarrow \text{FunctionFullDef} \mid \text{FunctionArrowDef} \\
\text{FunctionFullDef} &\rightarrow \text{FUNCTION IDENTIFIER Params COLON Signature CodeBlock} \\
\text{FunctionArrowDef} &\rightarrow \text{FUNCTION IDENTIFIER Params COLON Signature ARROW Expr} \\
\text{Params} &\rightarrow \text{LPAREN (IDENTIFIER COLON Signature (COMMA IDENTIFIER COLO} \\
\text{TypeDef} &\rightarrow \text{TYPE IDENTIFIER Params? Inheritance? LBRACE TypeBody RBRACE} \\
\text{Inheritance} &\rightarrow \text{INHERITS IDENTIFIER ArgList?} \\
\text{TypeBody} &\rightarrow (\text{MemberDef SEMICOLON})^* \text{ MemberDef?} \\
\text{MemberDef} &\rightarrow \text{IDENTIFIER ASSIGN Expr} \mid \text{FunctionDef} \\
\text{Expr} &\rightarrow \text{DestructiveAssignExpr} \mid \text{LogicalOrExpr} \\
\text{DestructiveAssignExpr} &\rightarrow \text{PrimaryExpr DESTRUCTIVE\_ASSIGN\_OP Expr} \\
\text{LogicalOrExpr} &\rightarrow \text{LogicalOrExpr LOGICAL\_OR\_OP LogicalAndExpr} \mid \text{LogicalAndExpr} \\
\text{LogicalAndExpr} &\rightarrow \text{LogicalAndExpr LOGICAL\_AND\_OP EqualityExpr} \mid \text{EqualityExpr} \\
\text{EqualityExpr} &\rightarrow \text{EqualityExpr EQUALITY\_OP ComparisonExpr} \mid \text{ComparisonExpr} \\
\text{ComparisonExpr} &\rightarrow \text{ComparisonExpr COMPARISON\_OP TermExpr} \mid \text{TermExpr} \\
\text{TermExpr} &\rightarrow \text{TermExpr TERM\_OP FactorExpr} \mid \text{FactorExpr} \\
\text{FactorExpr} &\rightarrow \text{FactorExpr FACTOR\_OP ExponentExpr} \mid \text{ExponentExpr} \\
\text{ExponentExpr} &\rightarrow \text{UnaryExpr POW\_OP ExponentExpr} \mid \text{UnaryExpr} \\
\text{UnaryExpr} &\rightarrow \text{UNARY\_OP UnaryExpr} \mid \text{CompositeExpr} \\
\text{CompositeExpr} &\rightarrow \text{LetIn} \mid \text{IfElse} \mid \text{WhileLoop} \mid \text{ForLoop} \mid \text{PrimaryExpr} \\
\text{PrimaryExpr} &\rightarrow \text{FunctionCall} \mid \text{NEW IDENTIFIER ArgList} \mid \text{PrimaryExpr DOT\_OP IDENT} \\
&\quad \text{PrimaryExpr DOT\_OP FunctionCall} \mid \text{Literal} \mid \text{IDENTIFIER} \mid \text{LPAREN Expr} \\
\text{Literal} &\rightarrow \text{NUMBER} \mid \text{STRING} \mid \text{BOOLEAN} \\
\text{FunctionCall} &\rightarrow \text{IDENTIFIER ArgList} \\
\text{ArgList} &\rightarrow \text{LPAREN (Expr (COMMA Expr)}^*\text{)?RPAREN} \\
\text{LetIn} &\rightarrow \text{LET AssignmentList IN CompositeExpr} \\
\text{IfElse} &\rightarrow \text{IF LPAREN Expr RPAREN CodeBlock (ELIF LPAREN Expr RPAREN Co} \\
\text{WhileLoop} &\rightarrow \text{WHILE LPAREN Expr RPAREN CompositeExpr} \\
\text{ForLoop} &\rightarrow \text{FOR LPAREN IDENTIFIER IN RANGE LPAREN Expr COMMA Expr RI} \\
\text{CodeBlock} &\rightarrow \text{LBRACE ExprList RBRACE} \\
\text{ExprList} &\rightarrow (\text{Expr SEMICOLON})^* \text{ Expr?} \\
\text{PrintExpr} &\rightarrow \text{PRINT LPAREN Expr RPAREN} \\
\text{AssignmentList} &\rightarrow \text{Assignment (COMMA Assignment)}^* \\
\text{Assignment} &\rightarrow \text{IDENTIFIER ASSIGN Expr} \\
\text{Signature} &\rightarrow \text{IDENTIFIER}
\end{aligned}
$$

2. Manejo de Posiciones y Construcción del AST con LALRPOP

En el proceso de análisis sintáctico, cada token reconocido es acompañado por una estructura denominada `Span`, que almacena las posiciones de inicio y fin del token dentro del código fuente (medidas en offsets de byte). Esta información es fundamental para:

- **Rastreo preciso de ubicaciones:** Permite identificar la posición exacta de cada elemento del programa, facilitando la depuración y el análisis posterior.
- **Generación de mensajes de error detallados:** El `Span` posibilita que los errores sintácticos reporten la línea, columna y contexto específico donde ocurre el problema, mejorando la experiencia del usuario.
- **Conservación de metadatos en el AST:** El Árbol de Sintaxis Abstracta (AST) generado por el compilador incluye estos metadatos de ubicación en todos sus nodos, lo que resulta útil para etapas posteriores como el análisis semántico o la generación de código.

La generación de `Span` es gestionada automáticamente por LALRPOP en el archivo `parser.lalrpop`, utilizando las anotaciones `@L` y `@R`, que corresponden a los offsets de los tokens en el texto fuente.

Teoría del Parsing LR(1) y LALRPOP

LALRPOP genera por defecto un parser de tipo LR(1), empleando una variante llamada *lane table*, que es más compacta en memoria. Los analizadores LR(1) realizan un análisis ascendente (bottom-up) con un símbolo de *lookahead*, lo que garantiza:

- **Reconocimiento eficiente:** El análisis es de complejidad $O(n)$, sin necesidad de backtracking.
- **Cobertura completa:** Permite parsear cualquier gramática LR(1), que es una clase amplia y expresiva de gramáticas libres de contexto.

Precisión: Es recomendable explicar la diferencia entre los analizadores LR(0), SLR(1), LALR(1) y LR(1), y por qué LALRPOP utiliza LR(1) con optimizaciones de memoria.

Internamente, LALRPOP construye un autómata LR(0), calcula los conjuntos LR(1) y genera las tablas de parsing (shift/reduce). El código Rust resultante implementa el parser como una máquina de estados con un stack, siguiendo el paradigma clásico de los analizadores LR.

3. Recomendaciones para la Precisión Conceptual