Construcción de POL usando Bison y Flex.

Nabil Haddad Pomar
nabil.haddad@alumnos.ucn.cl

Fundamentos de la computación
Jose Luis Veas

Índice.

Índice. **1**
Introducción. **2**
Proceso de desarrollo. **3**
Fases del desarrollo. 3
Gramática de POL = {N, T, P, Σ} 3
Dificultades y decisiones en la generación de código. 5
Conclusión. **8**

Introducción.

POL (Post Ordered Language) es un lenguaje de programación compilado, basado en C y C++, desarrollado utilizando Flex y Bison. Su nombre proviene del uso de la notación postfija para la escritura de operaciones aritméticas y lógicas.

El propósito de este informe es describir el desarrollo de POL, explicando las decisiones tomadas durante su confección, así como detallar su sintaxis, implementación y los elementos clave que lo componen.

POL fue desarrollado con el objetivo de adquirir conocimientos prácticos sobre el funcionamiento de los compiladores, desde el diseño de la gramática hasta su implementación. Para ello, se empleó Flex como analizador léxico y Bison como analizador sintáctico.

A través de expresiones regulares definidas en Flex, se generan tokens que son posteriormente interpretados por Bison para procesar las producciones que definen el comportamiento del lenguaje.

En este documento se presentará la gramática específica de POL, el diseño del árbol de sintaxis abstracta (AST) junto con una representación gráfica del mismo, la generación de código C/C++ a partir de archivos con extensión .pol y ejemplos de programas implementados en POL.

Proceso de desarrollo.

Fases del desarrollo.

En primer lugar, se definió la sintaxis que seguiría el lenguaje, es decir, cómo se ve exactamente. Tal como lo indica su nombre, POL fue diseñado para plantear cualquier operación matemática en formato postfijo. El resto de la sintaxis podría decirse que es muy parecida a C.

Sin embargo, aunque el resto de la sintaxis sea similar a C, no es menor que las operaciones matemáticas estén en notación postfija, ya que esto trae una serie de beneficios tanto para el compilador como para el usuario de POL.

En cuanto al proceso de desarrollo del compilador, el uso de notación polaca inversa elimina la necesidad de paréntesis, ya que no existe jerarquía entre las operaciones matemáticas: todas se encuentran al mismo nivel. La notación postfija obliga al usuario a ordenar las operaciones de manera que su jerarquía no influya en el cálculo. Lo mejor de todo es que esto sucede de forma natural: una vez que se aprende cómo funciona, se opera de manera intuitiva.

Desde el punto de vista del compilador, las operaciones se van acumulando en la pila de memoria en el mismo orden en que deben ser resueltas. Un computador no resuelve las operaciones en el orden en que los humanos las escriben, sino en función de su prioridad, y dicha prioridad ya está resuelta en POL, ya que esta tarea se delega al usuario. Por lo tanto, se puede decir que POL es un lenguaje stack-friendly.

Desde el punto de vista del usuario, aunque podría parecer que se le está imponiendo una tarea extra al tener que ordenar las operaciones, esto se convierte en una costumbre que incluso puede resultar más cómoda al eliminar la necesidad de paréntesis.

Una vez definida la sintaxis de forma clara y justificada, se procedió a desarrollar la gramática. Para que esta correspondiera con la sintaxis deseada, se generó un árbol de sintaxis abstracta (AST) que responde a la siguiente 4-tupla:

Gramática de POL = {N, T, P, Σ}

NO TERMINALES.
N: {input, line_non_empty, exp, L_op, type_id, scoped_lines, scope}

TERMINALES.

T: {NUM, T_BOOL, F_BOOL, STRING, IF, ELSE, WHILE, INT, BOOL, STR, DOUBLE, ARROW, PRINT, SCAN, LPAREN, RPAREN, LBRACE, RBRACE, INIT, POW, ADD, SUB, MUL, NOT, AND, OR, EQ, GR, WR, EQ_GR, EQ_WR, DIV, VAR, ; }

SÍMBOLO RAÍZ.

Σ: input

PRODUCCIONES.

P:

```
input:
        /vacío/
        |input   line_non_empty
;

line_non_empty:
        type_id VAR ';'
        |  type_id VAR INIT exp ';'
        | VAR INIT exp ';'
        | IF LPAREN exp RPAREN scope
        | IF LPAREN exp RPAREN scope ELSE line_non_empty
        | IF LPAREN exp RPAREN scope ELSE scope
        | WHILE LPAREN exp RPAREN scope
        | PRINT LPAREN exp RPAREN ';'
        | SCAN LPAREN VAR RPAREN ';'
        | VAR LPAREN RPAREN ';'
;

exp:
        NUM
        |T_BOOL
        |F_BOOL
        |STRING
        | exp ARROW type_id
        | VAR
        | exp exp L_op
        | exp NOT
        | exp exp ADD
        | exp exp SUB
        | exp exp MUL
        | exp exp DIV
        | exp exp POW
;

L_op:
        EQ
        | GR
```

```
        | WR
        | EQ_GR
        | EQ_WR
        | AND
        | OR
;

type_id:
        INT
        | DOUBLE
        | BOOL
        | STR
;

scoped_lines:
        line_non_empty
        | scoped_lines line_non_empty
;

scope:
        LBRACE scoped_lines RBRACE
;
```

Luego de poseer una gramática bien definida, se desarrolla la generación de código C++. El código generado será el que almacene toda la lógica detrás de las producciones del árbol AST. Cada una de las producciones se traduce en un conjunto de instrucciones en C++.

A continuación se presentan las decisiones tras la confección de POL, donde se dará a conocer cada uno de los detalles en torno al manejo de variables, funciones e instrucciones detrás de POL.

Dificultades y decisiones en la generación de código.

La principal decisión tras el código generado en C++ fue el manejo de las instrucciones generadas por las producciones. Para ello se utilizó un sistema de objetos, donde cada producción instancia un objeto de diferentes clases. El objetivo de trabajar con objetos es diferenciar la fase de parseo del compilador de la fase ejecución. De manera tal que durante el parseo se genera un conjunto de objetos, o nodos que serán ejecutados al final de la lectura del script.

Los nodos se dividen en 2 principales clases; Statment_node y Expr_node.

A partir de Statment_node nacen todas las instrucciones que se pueden llevar a cabo en POL:

- Declaración e instanciación.
- Entrada y salida de los datos (cin y cout).
- Declaraciones de control.

Como se puede observar en el repositorio, cada una de los nodos de instrucción poseen su correspondiente método de ejecución, el cual al ser sobreescrito, permite la diferenciación entre cada una de ellas.

Luego es a partir de Expr_node que nace todo lo que podríamos llamar una expresión en POL. Primero es importante definir lo que es una expresión en el dominio del presente compilador. Una expresión será cualquier lectura del archivo que culmine en un valor, sea este valor de cualquier tipo soportado por el lenguaje; int, double, string o bool.

Es por lo mencionado anteriormente que los Expr_node responden a las siguientes instrucciones:

- Operaciones aritméticas.
- Operaciones lógicas.
- Lectura de enteros.
- Lectura de doubles.
- Lectura de booleanos.
- Lectura de cadenas de texto.
- Casteo de tipos.

Así como la clase para las instrucciones posee su método de ejecución <override> Expr_node posee sus métodos de obtención de tipo y obtención de valor también <override> debido a que cada una de las expresiones devuelve su valor de manera distinta.

Cada uno de los valores almacenados en Expr_node por sí solo es volátil, tal que una vez instanciado, no puede ser llamado sin una referencia a su ubicación en memoria. Es por lo mencionado que se decide construir una tabla de símbolos. La tabla de símbolos corresponde a un hashmap de tipo <unordered_map> donde se almacenan 2-tupla de la forma: (identificador, (tipo, Expr_node)).

Se decidió utilizar un hashmap debido a lo eficiente que son. La complejidad algorítmica asociada a buscar una variable en la tabla de símbolos corresponde a una función de la forma $O(1)$. Lo cual abstrae el tiempo de ejecución de la cantidad de variables utilizadas.

Debido a que se está trabajando con direcciones de la memoria y que cada entidad en el código generado (sea una expresión o una instrucción) es única, se tomó la decisión de utilizar la librería <memory> para poder instanciar punteros únicos.

El uso de punteros únicos si bien puede ameritar mayor cantidad de memoria, facilita en gran medida el desarrollo del compilador ya que deriva la tarea de liberar la memoria al propio C++. De esta manera se evita la gestión del almacenamiento volátil y se evitan fugas de este.

Para el manejo de tipos se optó por generar una variante de tipos, es decir <variant> lo cual permite a un valor en específico ser declarado bajo una etiqueta de tipo ambigua. Para el caso de POL, variant consiste en los 4 tipos que soporta.

A través de dicha estructura y con ayuda del método <std::visit> es que se hizo el chequeo de tipos en la mayoría de los casos, aunque además, para casos específicos también se utilizó una enumeración de la clase Type (Clase de elaboración propia). Dicha enumeración consiste en una clase con 4 etiquetas que indican el tipo de dato almacenado en el hashmap bajo un identificador en específico. Es a través de las estructuras mencionadas que se lleva a cabo también el casteo de tipos.

Para las declaraciones de control se puede generalizar una estructura; identificador de la declaración, seguido de la expresión que actúa como condición y el scope a ejecutar.

La condición no es más que una expresión de cualquier tipo, sin ninguna restricción, que es filtrada a través de una función <eval()> que se ocupa de evaluar su contenido y devolver un booleano que indique si se ejecuta el scope o no. Esto le da cierta libertad a POL con respecto a las estructuras de control. Se puede consultar la estructura de cada uno de los nodos correspondientes a las estructuras de control y a la definición de la función <eval()> en los script “src/nodos.cpp” y “parser.y” presentes en el repositorio.

Para que la ejecución del scope (conjunto de instrucciones) fuese decidible, se derivó la tarea de ejecutar las instrucciones presentes en el scope, al “dueño” de dicho scope. Pero para que ello fuera posible primero había que identificar de alguna manera tal conjunto de instrucciones. Esto se hizo por medio de dos estructuras principales <Body_node> y <Body_holder_node> estructuras que se ocupan de almacenar las instrucciones y transferir el propietario de dichas instrucciones respectivamente. Las clases mencionadas se encuentran en el script “src/nodes.cpp” presente en el repositorio.

Con respecto a las funciones se optó por definir más bien procesos capaces de ser llamados en cualquier momento luego de su declaración. Para ello se generó de manera análoga a la tabla de símbolos, una tabla de funciones que almacena 2-tupla de la siguiente forma (Identificador, Body), siendo identificador lo que se podría llamar nombre del proceso y body el cuerpo que almacena las instrucciones de esta. Se utiliza un sistema de scopes muy similar al presente en las declaraciones de control.

Si se desea una idea más clara del funcionamiento desde el momento en el que se ejecuta la siguiente línea en la consola:

```
$ ./pol archivo.pol
```

Se podría decir que el paso a paso es el siguiente:

1. Lectura y separación de los tokens definidos por Flex en el archivo lexer.l
2. Ejecución del árbol AST obteniendo los no terminales.
3. Instanciación y almacenamiento de los nodos en un vector de tipo <Statment_node>
4. Ejecución de los nodos en el orden que fueron almacenados por medio de un ciclo for.

Una de las principales dificultades presentes en el desarrollo de POL fue la efectiva diferenciación entre las fases de parseo y ejecución. y la correcta identificación de de aquello que se podía ejecutar de inmediato y aquello que no. De manera resumida resultó en un sistema de nodos instanciados en la fase de parseo a medida que se lee el archivo, recolectados y ejecutados en la fase de ejecución.

Conclusión.

El desarrollo de POL permitió comprender en profundidad el funcionamiento interno de un compilador, desde el diseño léxico y sintáctico hasta la ejecución diferenciadas a través de estructuras de datos propias.

La elección de la notación polaca inversa no solo facilitó la implementación del parser, sino que también optimizó el tratamiento de expresiones tanto para el compilador como para el usuario.

Gracias a un enfoque orientado a objetos y al uso de estructuras como tablas de símbolos, la librería memory y la estructura variant, POL logró ejecutar instrucciones y evaluar expresiones de manera clara. La experiencia no solo reforzó los conocimientos teóricos sobre álgebra, lenguajes, gramática, y compiladores sino que también entregó herramientas prácticas para construir lenguajes propios y comprender cómo se traduce el código fuente.