Proper Language

A language for the ones with style and good taste

Automata, Language Theory and Compilers. I.T.B.A.

1Q-2018 | Group ALT

Idea y Objetivo

Hacer un lenguaje de programación simple llamado Proper.

El objetivo es reemplazar la fraseología de los lenguajes de programación tradicionales a uno más "verborrágico" y en cierta forma "formalizado" con el fin de que el programador sienta que está escribiendo una carta y cualquiera que lea el código sin conocimiento previo, incluso de programación, pueda interpretarlo de manera sencilla sabiendo únicamente ingles.

Es un lenguaje que no posee tipado dinámico, es decir, es fuertemente tipado. Incluye únicamente los tipos conocidos integer, float y cadenas de caracteres. Además incluye la posibilidad de leer y escribir de la entrada y salida estándar, la llamada de sistema de retardo de ejecución ("sleep"), entre otras funcionalidades.

No posibilita a que un programador defina sus propias funciones, sino más bien busca que el código al igual que una carta sea algo lineal que se lee y ejecuta de arriba a abajo sin saltos de un lugar a otro en el código.

Consideraciones

Se implementó un árbol de análisis sintáctico abstracto para pasear las diferentes sentencias del lenguaje y a partir de ellas los distintos tipos de expresiones (más adelante se detallan estos puntos). Además fue de importancia la inclusión de una tabla de símbolos/variables donde se aloquen los distintos objetos de un programa y sus nombres.

Descripción del desarrollo

En primer lugar se inició el desarrollo con el archivo scanner de Lex donde definimos la expresiones que buscamos en el código junto a los identificadores declarados en el archivo parser de Yacc. En este archivo comenzamos definiendo las producciones, reglas de traducción en conjunto a las definiciones de los tokens y la rutina principal de C que invoca el parseo general. La misma hace la inclusión de los archivos necesarios para la traducción a C.

A continuación definimos el árbol sintáctico abstracto estableciendo los tipos de sentencias, los tipos de expresiones, los operadores booleanos, los diferentes nodos del árbol y los métodos que proveen la producción de todas las sentencias. Más adelante definimos scripts de ejecución que fueron muy convenientes para el desarrollo del trabajo práctico. El principal de estos siendo el "properize.sh" que nos presenta una abstracción inicial en la cual se define si se está en modo debug o no activando así flags de generación de código y un output de toda la salida estándar y de error de todo el sistema.

Por otro lado existe un segundo script igual de importante que es el "proper_compile_and_bin.sh" que se encarga de compilar el compilador de Proper, obtener el código en C del archivo en lenguaje Proper, y finalmente compilarlo en GCC para obtener el ejecutable, descartando el archivo C generado para que sea lo mas transparente posible.

Por otro lado se pensó en un modulo "en tiempo de ejecución" (ver directorio run/var) donde se establece la tabla de símbolos o variables. En la misma también se declara la creación de variables, la asignación de las mismas, su impresión, la lectura y las operaciones posibles entre variables como comparación, igualdad, suma, resta, etc. Vale aclarar que una variable fue definida con un tipo y un valor ya que la misma se definió como una estructura tipo-valor.

Finalmente se definió un directorio test donde se fueron incluyendo múltiples ejemplos de archivos en lenguaje Proper tanto con casos válidos como de error de distintas funcionalidades o posibles errores para proveer un símil suite de pruebas.

La Gramática

Tiene una producción principal que cuenta con el punto de entrada, código de programa y punto de salida.

```
entry: mainstart entry mainend {}
            | program {
                  getCode($1);
            }
            ;
```

Luego se definieron las producciones que van de un programa a un bloque o a un bloque y programa subyacente. Notar que se aloca espacio en memoria para el tipo y el elemento de la sentencia y el puntero a la próxima sentencia.

```
program: block {
                $$ = malloc(sizeof(*$$));
                $$->type = $1->type;
                $$->elem = $1->node;
                $$->next = NULL;
                free($1);
        }
        | block program {
                $$ = malloc(sizeof(*$$));
                $$->type = $1->type;
                $$->elem = $1->node;
                $$->next = $2;
                free($1);
        }
        ;
```

Se utilizó un nodo block que permite la reducción de producciones con sentencias. Por ejemplo la de impresión, la cual aloca memoria para un nodo bloque del tipo print call y el nodo de tipo “statementType”.

```
block:
        print {
                $$ = malloc(sizeof(*$$));
                $$->type = PRINT_CALL;
                $$->node = $1;
        }
```

```
| assign {
        $$ = malloc(sizeof(*$$));
        $$->type = ASSIGNMENT;
        $$->node = $1;
}
| if {
        $$ = malloc(sizeof(*$$));
        $$->type = IF_THEN_ELSE;
        $$->node = $1;
}
| while {
        $$ = malloc(sizeof(*$$));
        $$->type = WHILE_BLOCK;
        $$->node = $1;
}
| read_into {
        $$ = malloc(sizeof(*$$));
        $$->type = READ_INTO;
        $$->node = $1;
}
| exit {
        $$ = malloc(sizeof(*$$));
        $$->type = EXIT_CALL;
}
| sleep {
        $$ = malloc(sizeof(*$$));
        $$->type = SLEEP_CALL;
        $$->node = $1;
}
| the_shining {
        $$ = malloc(sizeof(*$$));
```

```
            $$->type = THE_SHINING_CALL;
            $$->node = $1;
        }
        ;
```

Notemos que las expresiones reducen un token terminal . En el caso de que el scanner retorne un identificador se asigna memoria para un nodo expresión del tipo variable y también se asigna memoria para la variable misma. Por último, se hace uso de la función declarada en el archivo "parser.y" que retorna el id correspondiente a la variable si ya existe o bien lo agrega y retorna el id generado.

```
expression: IDENTIFIER {
            $$ = malloc(sizeof(*$$));
            $$->type = VARIABLE;
            $$->left = malloc(sizeof(int));
            int id = getOrAddId($1);
            memcpy($$->left, &id, sizeof(int));
        }
        | INT {
            $$ = malloc(sizeof(*$$));
            $$->type = INTEGER;
            $$->left = malloc(sizeof(int));
            memcpy($$->left, &$1, sizeof(int));
        }
        | FLT {
            $$ = malloc(sizeof(*$$));
            $$->type = FLOAT;
            $$->left = malloc(sizeof(float));
            memcpy($$->left, &$1, sizeof(float));
        }
(...parcial...)
```

Dificultades Encontradas

Al inicio del trabajo práctico fue complejo entender el uso de lex y yacc en conjunto. Presentó un desafío comprender correctamente el propósito de un árbol sintáctico abstracto, pero una vez tomada la decisión de implementarlo fue bastante más veloz el trabajo a realizar pues sentaba las bases para introducir nuevos tipos de sentencias y expresiones.

Aun así surgieron inconvenientes a la hora de definir las reglas de traducción y las diferentes producciones. Por ejemplo, tuvimos problemas para identificar el punto final de una oración en nuestro lenguaje y el que indica la precisión de un float. Por otro lado y no de gravedad menor, tuvimos dificultades para identificar los puntos de entrada y salida del lenguaje; concretamente, no se generaba el código intermedio en el orden correcto.

Además corregimos permanentemente las expresiones regulares de nuestro scanner lex para que un programador pueda escribir en Proper en forma de párrafo, por ejemplo, aceptar “If” o “if”.

Finalmente se encontró dificultoso hacer transparente para el usuario la compilación de Proper a ejecutable sin que se evidencie el pasaje por lenguaje C y cualquier output de la compilación del mismo.

Posibles Extensiones

- Como posible extension, se podría implementar la generación automática de ASCII art a partir del path a una imagen local.
- Se mencionó varias veces la posibilidad de crear un archivo de highlighting para Sublime y poder escribir con la ayuda del IDE.
- En la construcción del arbol, y la reducción de las producciones, hay asignaciones de memoria que no se liberan, por ejemplo, las de variables. En un futuro se podría implementar un sistema que maneje esta deficiencia (simil garbage collector de Java).

- La definición de funciones propias del usuario para poder hacer saltos de un lugar a otro. Se discutió la posibilidad de escribir en otros archivos respuestas a la "carta" original de Proper, en las cuales se pueda definir funciones.

Referencias

A modo de orientación e interpretación del trabajo práctico se consultó y discutió implementaciones, uso del árbol sintáctico y cosas a tomar en cuenta con compañeros de otros grupos.