Introducción

Este documento presenta una herramienta para la construcción de árboles de sintaxis abstracta (AST) para el desarrollo de compiladores en C. La herramienta abstrae los procesos comunes de la manipulación del árbol, delegando a los desarrollos particulares la definición de los datos requeridos en cada caso.

El AST permite representar la estructura del programa fuente mediante una estructura de datos de tipo árbol (grafo dirigido acíclico). Generalmente se sigue una forma similar a las derivaciones de la gramática del lenguaje, aunque pueden introducirse variaciones en los casos que sea conveniente.

Los nodos interiores del árbol representan símbolos no terminales (NT) y los nodos hoja representan terminales (T). La raíz del árbol es el símbolo distinguido.

> **Nota para el alumno:** Este documento describe la estructura de datos utilizada para la construcción del AST, y las funciones para su manipulación. Si bien no es necesaria una comprensión acabada de los aspectos tratados para el desarrollo del trabajo práctico, se recomienda una lectura detenida. Especialmente de la sección *Primitivas de manipulación*.

Estructuras de datos

Si bien los datos puntuales contenidos en cada nodo dependen del compilador que se desarrolle, la estructura de datos arbórea es genérica, permitiendo que su construcción y manipulación sea abstraída mediante funciones y definiciones de tipos de datos genéricos.

La herramienta que se propone provee estructuras de datos para construir el árbol, y funciones para manipular estas estructuras, y delega la definición de los datos requeridos por el compilador a cada desarrollo particular.

Nodos

El AST se construye en base a nodos enlazados por punteros. El tipo definido para los nodos del árbol es `ast_node`. Esta estructura permite mantener los punteros requeridos para la construcción del árbol y los datos requeridos por el compilador específico.

La estructura `ast_node` (`lib/ast.h`) se como sigue:

```
[C]
01 struct ast_node {
02     int id;
03     int level;
04     int type;
05
06     union usr_ast_data data;
07
08     struct ast_node* parent;
09     struct ast_child_list* children;
10 };
```

El campo `id` mantiene un identificador numérico, que se asigna consecutivamente al momentos de la creación del nodo. El identificador no tiene una función específica dentro de la herramienta.

El nivel de cada nodo dentro del árbol se mantiene en el campo level.

Dada la naturaleza heterogénea de los nodos de un AST, se requiere de manera general, que cada nodo pueda ser de un tipo distinto. Considérese por ejemplo el caso de una expresión que es derivada en un valor numérico constante, como en una regla `expresion` → `numero`. En este ejemplo, la subestructura que se requiere precisa que el nodo padre (`expresion`) sea de tipo distinto del nodo hijo (`numero`).

Para resolver esta cuestión se define el campo `type`. Este campo almacena un valor numérico determinado por el programador (y de su exclusiva interpretación). *Se sugiere* que la definición de estos valores de tipo se realicen de la forma siguiente:

```
#define N_EXPRESION       1
#define N_NUMERO          2
// ...
```

> **Nota para el alumno:** Algunas constantes están predefinidas en el archivo `lib/ast.h`. Estas son utilizadas por las funciones `n_...` que se explican más adelante. Estas constantes no pueden ser redefinidas o borradas, pero conviene tenerlas en cuenta para su utilización en el trabajo (especialmente si se utilizan las funciones `n_...`).

Adicionalmente, esta heterogeneidad requiere la asociación de datos específicos a cada nodo, según su tipo. Estos datos se corresponden con la definición de atributos de la sintaxis. Estos datos *de usuario* se mantienen en el campo `data`.

Nótese aquí que el tipo de datos de data es `union usr_ast_data`. Esta unión debe ser definida según la necesidad de cada compilador, contemplando todos los posibles valores que deban mantenerse. El archivo `src/defs.h` contiene la definición de esta estructura.

De esta manera, mediante los campos type y data, el programador puede construir un AST acorde a las necesidades del compilador, utilizando una estructura genérica para la mantención del árbol.

> **Nota para el alumno:** los campos predefinidos en la estructura `union usr_ast_data` no deben ser quitados, debido a que son utilizados por las funciones `n_...`. De ser necesario el alumno puede incluir nuevos campos.

Primitivas de manipulación

El AST se construye y manipula mediante funciones agrupadas en dos niveles. En un nivel inferior se ofrecen cuatro *primitivas* para la creación y manipulación de los nodos del árbol. Adicionalmente se ofrecen funciones que terminen definir los casos más típicos de nodos en el desarrollo de compiladores sencillos.

Las primitivas de creación y manipulación del árbol son las siguientes:

```
struct ast_node* ast_new_node(int type);

struct ast_node* ast_add_child_first(struct ast_node* parent,
                                     struct ast_node* child);

struct ast_node* ast_add_child(struct ast_node* parent,
                               struct ast_node* child);

struct ast_node* ast_get_nth_child(struct ast_node* parent,
                                   int nth);
```

Creación de nodos

La función `ast_new_node` permite crear nodos de tipo `struct ast_node`. La función espera un valor entero correspondiente al tipo de nodo que desea crearse. El tipo de nodo no afecta al proceso de creación, sino que simplemente sirve para asignar el valor del tipo en la estructura del nodo.

Si desea crearse un nodo de tipo `N_EXPRESION` (según el ejemplo dado anteriormente), se puede invocar a la función `ast_new_node` de la siguiente manera:

```
struct ast_node* node = ast_new_node(N_EXPRESION);
```

De esta forma la variable node contiene un nodo de tipo `N_EXPRESION`. Debe tenerse en cuenta en este punto que solamente el *tipo* de nodo fue establecido. Su contenido en cuanto a datos específicos del compilador debe establecerse manualmente.

Inserción de nodos en el AST

Para la inserción de nodos en el árbol, se definen dos primitivas: `ast_add_node_first` y `ast_add_node`. La primera función agrega un nodo como primer hijo de su padre, mientras que la postrera agrega un nodo como último nodo de su padre.

La diferencia entre estas funciones es importante dado que el orden de los hijos de un nodo particular es importante.

Considérese por ejemplo las siguientes producciones de una gramática:

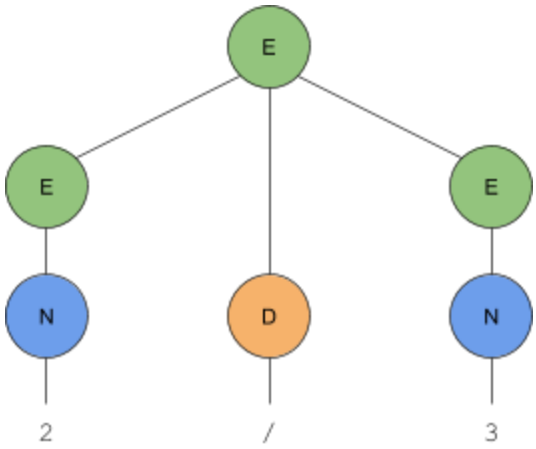
`expresion` → `expresion` `division` `expresion`

`expresion` → `numero`

Y el siguiente programa fuente:

```
2 / 3
```

El AST para este caso podría constituirse de la siguiente manera:



Puede observarse claramente en este caso que el orden de los hijos describe la forma en la que debe resolverse la expresión. Al momento de construir el árbol *debe considerarse en qué orden deben agregarse los nodos*.

Para el caso dado, un fragmento de código podría ser como sigue:

```
[C]
01 struct ast_node* exp_padre, exp_1, exp_2, num_1, num_2, div;
02 // crea los nodos "hoja"
03 num_1 = ast_new_node(N_NUMERO);
04 num_2 = ast_new_node(N_NUMERO);
05 div = ast_new_node(N_DIVISION);
06
07 // asigna la expresión de la izquierda
08 exp_1 = ast_new_node(N_EXPRESION);
09 ast_add_node(num_1);
10
11 // asigna la expresión de la derecha
12 exp_2 = ast_new_node(N_EXPRESION);
13 ast_add_node(num_2);
14
15 // agrega las expresiones hijas a la expresión padre
16 exp_padre = ast_new_node(N_EXPRESION);
17 ast_add_node(exp_padre, exp_1);
18 ast_add_node(exp_padre, div);
19 ast_add_node(exp_padre, exp_2);
```

Las líneas 15 a 19 pueden escribirse de manera equivalente:

```
15 // agrega las expresiones hijas a la expresión padre
16 exp_padre = ast_new_node(N_EXPRESION);
17 ast_add_node_first(exp_padre, exp_2);
```

```
18 ast_add_node_first(exp_padre, div);   // div se agrega antes que exp_2
19 ast_add_node_first(exp_padre, exp_1); // exp_1 se agrega antes que div y exp_2
```

De esta manera puede observarse que `ast_add_node_first` y `ast_add_node` son suficiente para la creación de toda la estructura del AST.

> **Nota para el alumno:** el código anterior muestra la manera programática de crear y enlazar nodos. En el caso del trabajo, la creación de los nodos y su enlace está distribuida en las reglas de la gramática.

Recorrido del AST

La primitiva genérica para el recorrido de un AST es `ast_get_nth_child`[1]. Esta función devuelve el enésimo hijo de un nodo, o `NULL` si el nodo no tiene tal hijo.

Continuando con el ejemplo anterior, para obtener las expresiones izquierda y derecha creadas anteriormente, puede ejecutarse el siguiente código:

[C]

```
01 struct ast_node exp_1 = ast_get_nth_child(exp_padre, 0);
02 struct ast_node exp_2 = ast_get_nth_child(exp_padre, 2);
03 struct ast_node exp_3 = ast_get_nth_child(exp_padre, 3); // NULL
```

Nótese que los índices comienzan en 0.

Para recorrer recursivamente un árbol, puede utilizarse como referencia el siguiente código:

[C]

```
01 void recorrer(struct ast_node* node)
02 {
03     struct ast_node* child;
04     int i = 0;
05     // mientras haya nodos hijos
06     while (NULL != child = ast_get_nth_child(node, i)) {
07         // punto 1
08         recorrer(child);
09         // punto 2
10     }
11 }
```

Pueden intercalarse el código en la línea 7 o 9, según en qué punto se requiera procesar los nodos en el recorrido.

> **Nota para el alumno:** los puntos 1 y 2 sirven para generar el código que deba ir antes y después del subárbol generado desde el nodo que se examina.

[1] (El término *nth* se refiere en castellano a enésimo)

Funciones predefinidas

La herramienta ofrece una serie de funciones predefinidas, que abstraen los casos comunes de creación y manipulación de nodos. Estas con las funciones `n_...` previamente mencionadas.

El uso de estas funciones no es obligatorio, aunque se recomienda considerar los casos en lo que puedan ser útiles para aprovechar el desarrollo. Las funciones están declaradas en la cabecera `lib/ast.h` y definición se encuentra en el archivo `lib/ast.c`.

Considerense especialmente las siguientes funciones:

```
struct ast_node* n_block(struct ast_node* block);
```

Esta función construye un bloque de código de tipo `N_BLOCK`. El parametro block puede ser cualquier nodo del AST, especialmente aquellos que representan bloques del lenguaje, como ifs, whiles o expresiones.

```
struct ast_node* n_blocks(struct ast_node* block1, struct ast_node* block2);
```

Esta función permite agrupar diversos bloques en una lista de tipo `N_BLOCKS`. Los parámetros `block1` y `block2` son los bloques de tipo `N_BLOCK` que se quieren agurpar, o `NULL`.

```
struct ast_node* n_id(char* id, int type);
```

Permite definir un nodo de tipo `N_ID`, que representa un identificador en el AST.

```
struct ast_node* n_const_string(char* string);
```

Construye un nodo de tipo `N_CONST_STRING`, que representa un string en el AST. El parámetro `string` es la cadena de caracteres asociada.

```
struct ast_node* n_const_number(double number);
```

Construye un nodo de tipo `N_CONST_NUMBER`, que representa una constante numérica en el AST. El parámetro `number` es el valor numérico que se desea representar.

La función `n_id`

El prototipo de esta función es:

```
struct ast_node* n_id(char* id, int type);
```

Esta función permite la creación de nodos para identificadores. Recibe como parámetro el nombre del identificador (obtenido desde el *lexer*) y un valor entero correspondiente al tipo.

Además de crearse el nodo correspondiente de tipo `N_ID` y asignarse valor de `id`, se agrega el símbolo a la tabla de símbolos, asociándose el tipo especificado.

> **Nota para el alumno:** la utilización de la tabla de símbolos se explica más adelante.

> **Nota para el alumno:** el uso de las funciones `n_const_string` y `n_const_number` es similar al de `n_id`, con la diferencia de que no se agregan símbolos a la tabla de símbolos.

Ejemplo de utilización de `n_id`

Considere el siguiente código de flex:

```
[flex]
01 [a-zA-Z][a-zA-Z0-9]* {
02                         yylval.id = strdup(yytext);
03                         return T_ID;
04                      }
```

Este código permite identificar tokens de tipo `T_ID` (correspondientes a los casos típicos de identificadores). El lexema se almacena en el campo `id` de `yylval`.

El código correspondiente en bison es:

```
[bison]
01 id: T_ID     { $$ = n_id($1, 0); }
02   ;
```

Este código genera el nodo correspondiente al identificador, y le asigna el tipo 0 como ejemplo.

Luego, durante la generación de código, supóngase una función que recibe el nodo creado por la función `n_id`:

```
[C]
01 void print_id(struct ast_node* node)
02 {
03     // el id
04     printf("%s", node->data.id.id);
05     // el tipo
06     printf("%s", node->data.id.type);
07 }
```

Mediante los atributos `id.id` e `id.type` se pueden obtener los valores definidos para el id y el tipo.

> **Nota para el alumno:** para las funciones `n_const_string` y `n_const_number` corresponde utilizar los atributos `data.number` y `data.string`.