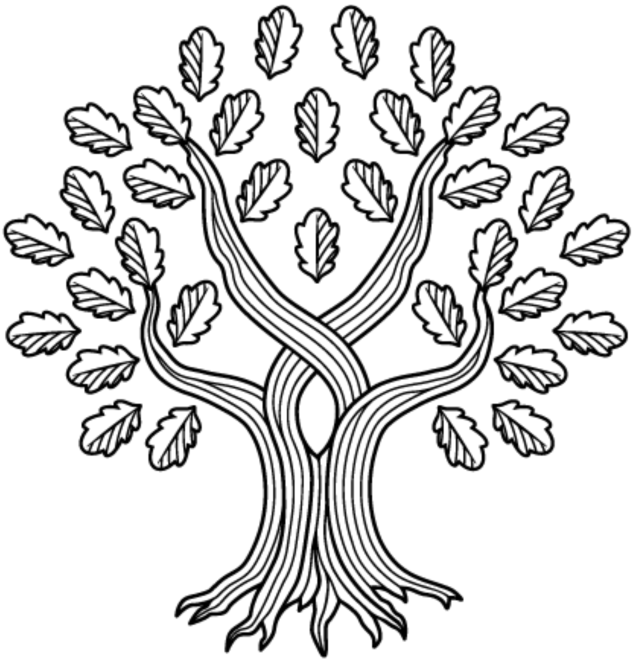
Proyecto Final

Teoria del Lenguajes y Programación

Universidad Panamericana
Facultad de Ingeniería



Noviembre 27, 2023

Barba Pérez Jorge 0237328@up.edu.mx
Osornio López Daniel Alejandro 0244685@up.edu.mx

Índice

1 Compilar el proyecto

1.1 Usando zig

Compilar

Para poder compilar en cualquier plataforma sin la necesidad de *toolchains* específicos (p. ej. MSVC, GNU, Xcode), usamos el sistema de compilación del lenguaje de programación Zig en la versión 0.11 [1]. Esto nos permite compilar el proyecto con un solo comando en Windows, Linux y MacOS así como compilar el proyecto para cualquier cpu/os desde cualquier cpu/os.

El archivo que configura la compilación es `build.zig`. Para compilar el proyecto debemos ejecutar:

```
zig build
```

Todos los archivos generados quedan en `./zig-chache` y `./zig-out/bin`.

Adicionalmente podemos compilar en modo de *release* con `-Doptimize=ReleaseFast` o compilar para otras plataformas con `-Dtarget=<ARCH>-<OS>-<ABI>`, por ejemplo con `-Dtarget=x86_64-windows-gnu` o `-Dtarget=aarch64-macos`.

La impresión de la tabla de símbolos y el árbol sintáctico viene desactivado por defecto, se puede habilitar usando `-Dtable` y `-Dtree`:

```
zig build -Dtable -Dtree
```

Contamos con distintas banderas configurables para habilitar/deshabilitar distintos pasos durante la compilación:

`-Dno-bison`	Do not run bison
`-Dno-flex`	Do not run flex
`-Dno-gen`	Do not run flex nor bison
`-Ddebug-bison`	Run bison with the `-t` flag and `yydebug` set to `1`
`-Ddebug-flex`	Run flex with the `-d` flag
`-Ddebug-gen`	Run flex and bison with debug flags
`-Dtable`	Print symbol table
`-Dtree`	Print basic AST
`-Derror`	Treat llvm warnings as errors

Ejecutar

Para ejecutar el programa utilizando la misma herramienta debemos ejecutar:

```
zig build run -- ./ruta/archivo.pas ./ruta/salida.pas
```

O especificando banderas, como las dedicadas a mostrar el árbol y tabla de simbolos:

```
zig build run -Dtable -Dtree -- ARGS
```

1.2 Usando Makefile con gcc

Además de zig, incluimos un Makefile que utiliza el compilador marcado en la variable `CC` (por defecto `gcc`) para compilar el proyecto. Se escribió el Makefile esperando que se ejecute en un ambiente *Unix-like*. El archivo Makefile cuenta con 3 *targets*:

`gen`	Ejecutar bison y flex. No se ejecuta automáticamente por `build` ni por `run`
`build`	Compilar el proyecto, los archivos de flex y bison ya debebieron ser creados con `gen`
`run`	Compilar y ejecutar el proyecto, podemos pasar variables al programa usando la variable `args`

Primero generamos los archivos de flex y bison:

```
make gen
```

Después compilamos el proyecto:

```
make build
```

O también podemos compilar con las banderas para habilitar la impresión del árbol y la tabla de símbolos:

```
make build_print
```

2 Sobre el proyecto

2.1 Sobre las versiones

El proyecto se escribió probándolo en las siguientes plataformas, usando las versiones de flex, bison, y compilador especificados:

Plataforma	Bison	Flex	Compilador
darwin-x86_64	2.7.12	2.6.4	zig 0.11
darwin-aarch64	3.8.2	2.6.4	zig 0.11
linux-x86_64	2.4.1	2.6.4	zig 0.11
win-x86_64	2.4.1	2.5.4a	zig 0.11

Las plataformas resaltadas con una fuente más gruesa son aquellas donde se probó más veces el proyecto. Sí compilamos el proyecto utilizando `gcc`, mas no se usó frecuentemente.

2.2 Sobre el lenguaje

El proyecto esta implementado enteramente en `C` *puro*, sin utilizar un *subset* de `C++`, utilizando solo herramientas disponibles en la librería estándar de `C`. Esto implica que tuvimos que implementar todas las estructuras necesarias, incluyendo `Vec` y `String`.

Se usó un poco de `zig`, para aprovechar que es 100% compatible con `C`, y su sistema de compilación multiplataforma sin dependencias para facilitar la compilación del proyecto para todos los miembros del equipo. Solamente se empleó este lenguaje para ese propósito. Todo el código relacionado a zig está ubicado en `build.zig` y en la carpeta `build/` como se describe en la Sección 2.3.

2.3 Estructura & entregas del proyecto

Carpeta	Función
`build/`	Incluye funcionalidad implementada en `zig` utilizada para manipular archivos y sus contenidos de forma *cross-platform* en el *script* de compilación.
`src/`	Incluye todo el código fuente del compilador *per se*. El archivo `grammar.y` es el código relacionado a bison y `lexer.l` es el archivo de `flex`. El punto de entrada está definido en el archivo `main.c`.
`test/`	Incluye pruebas sobre las estructuras implementadas para validar su comporatamiento.
`entradas/`	Incluye todos los archivos utilizados para probar el funcionamiento del compilador.
`scripts/`	Incluye archivos de código en `bash` para realizar tareas en sistemas operativos similares a `Unix`. El script de compilación escrito en zig realiza las mismas tareas independientemente de la plataforma.
`bin/`	Incluye ejecutables pcompilados para linux y windows x86_64. Los ejecutables que terminan en `_print` imprimen el árbol y la tabla de símbolos

2.4 Sobre las esctructuas

2.4.1 Contenedores genéricos

Como `C` no cuenta con `templates` no usamos *tipos* genéricos, sino que todas las estructuras actúan como contenedores que desconocen completamente el tipo de dato que hay almacenado en ellos y que

solo saben almacenar y realizar operaciones sobre elementos de N bytes, que son determinados con `sizeof(Tipo)` en la instanciación de los contenedores.

Aunque menos flexible, las estructuras almacenan siempre elementos del mismo tamaño, es por eso que se debe especificar el tamaño del tipo de dato almacenado cuando se crean instancias, por la misma razón no es necesario especificar argumentos extras al hacer operaciones como `push(&vec)`.

Las operaciones realizadas sobre contenedores devuelven `void *`, apuntador el cual el *caller* de la función debe *castear* como apuntador al tipo correcto almacenado en el contenedor, en el siguiente bloque hay un ejemplo de la dinámica usando `Vec`:

```
// Agregar un valor al final del vector
size_t * num = (size_t *)vec_push(&vec);
*num = 4;

// Obtener el valor en la posición i del vector
size_t * num2 = (size_t *)vec_get(&vec, i);
*num2 = *num + *num2;
```

2.4.2 Wrappers

Para facilitar el uso de estas estructuras haciendo asunciones sobre el tipo de dato almacenado, creamos funciones auxiliares que actúan como *wrappers* de las funciones de bajo nivel. Como lo es `ast_create_node`, que asume que el tipo almacenado en un árbol es `Node`, y que por debajo llama a la función `tree_new_node`.

3 Modificaciones a la gramática

3.1 Mover detección de caracteres individuales al lexer

Varias reglas de la gramática definen a nivel de caracteres distintos no terminales que perfectamente pueden ser tokens, como lo son los identificadores, números, etc.

Procuramos dejar la detección de caracteres individuales al lexer, de forma que el parser solo maneja no terminales de alto nivel.

Para traducir las reglas de gramática de forma que se pudieran agregar al lexer analizamos las cadenas generadas por dichos no terminales y re-creamos el comportamiento usando expresiones regulares.

Por ejemplo, en la regla:

```
contante_real : signo numero_entero . numero_entero
              | signo numero_entero . numero_entero exponente

exponente : e signo numero_entero
          | E signo numero_entero
          |
```

Podemos observar que el patrón es que siempre inicie con signo, luego un número entero, y opcionalmente (porque puede ser vacío) un exponente. Lo que se traduce en la expresión regular:

```
{SIGNO}{ENTERO}"."{ZENTERO}(e{SIGNO}{ENTERO})?
```

3.2 Permitir listas de expresiones en instrucciones `write` y `writeln`

Las reglas de la grámatica especificadas para la instrucción `write` y `writeln` generaban mucha repetición de código para permitir un conjunto limitado de formas específicas de invocación para `write` y `writeln`.

Para reducir la cantidad de código y permitir invocaciones más complejas, agregamos una regla a la gramática `expresion_lista_con_cadena`, que permite coleccionar expresiones normales como lo haría `expresion_lista`, con el agregado de tener nodos del tipo `NStr`, que almacenan `CONST_CADENA`.

```
expresion_lista_con_cadena: expresion
                          | CONST_CADENA
                          | expresion_lista_con_cadena ',' expresion
                          | expresion_lista_con_cadena ',' CONST_CADENA
```

De forma que la regla para invocar `write`/`writeln` puede tener cualquier combinación de argumentos, siempre y cuando sean expresiones, que por definición son imprimibles:

```
escritura_instruccion : KW_WRITELN '(' expresion_lista_con_cadena ')'
                      | KW_WRITE '(' expresion_lista_con_cadena ')'
```

3.3 Permitir varias declaraciones en cualquier orden de `const` y `var`

La gramática original cuenta con la regla de la forma mostrada en el siguiente listado, es claro que la regla solo permite tener una de dos opciones, declarar variables o constantes, más no ambas.

```
declaraciones : declaraciones_variables | declaraciones_constantes
```

Si miramos en la definición de los no terminales `declaraciones_variables` y `declaraciones_constantes` encontramos que:

```
declaraciones_variables : declaraciones_variables var ... |
declaraciones_constantes : declaraciones_constantes const ... |
```

Las reglas solo permiten concatenar declaraciones del mismo tipo. Si realizamos la derivación `declaraciones` → `declaraciones_variables`, solo podremos seguir derivando en más `declaraciones_variables`. Lo mismo sucede con `declaraciones_constantes`.

El primer cambio que realizamos es permitir especificar ε desde el no terminal `declaraciones`.

```
decl : decl_var | decl_const |
```

Y para ambas posibilidades de derivación, `decl_var` y `decl_const`, agregamos la opción de que se pueda concatenar con cualquier no terminal `decl`:

```
decl_const : decl KW_CONST IDENT '=' CONST_ENTERA ';'
decl_var : decl KW_VAR ident_lista ':' tipo ';'
```

Esto conserva la posibilidad de no declarar ningúna variable o constante, o de declarar múltiples variables y constantes en cualquier órden que se desee.

3.4 Permitir que el programa no reciba valores de entrada

Agregamos la opción de que se pueda recibir ε en la regla de `identificadores_lista`, de forma que un programa puede no tener argumentos. Los paréntesis siguen siendo obligatorios.

4 Tabla de simbolos

En esta sección describimos la implementación de la tabla de símbolos, pasando por cómo funciona el almacenamiento y funcionamiento del conjunto de *hashes*. También describimos la información almacenada en la tabla de símbolos y su utilidad para realizar verificaciones, generar código y construir el árbol abstracto de sintaxis.

4.1 HashSet

Nombramos la estructura implementada como *conjunto* de hashes y no como *mapa* pues no hay, a nivel del modelo de datos, una distinción clara de la forma (K, V), es decir, que no hay *per se* llaves y valores, sino que solo hay valores.

La estructura consiste de un vector de tamaño fijo N valores (celdas gris obscuro de la Figura 1). Los valores almacenados en el vector principal son más vectores. Es decir, se trata de un vector de vectores.

Los vectores anidados almacenan los valores en sí que se insertan en la tabla.

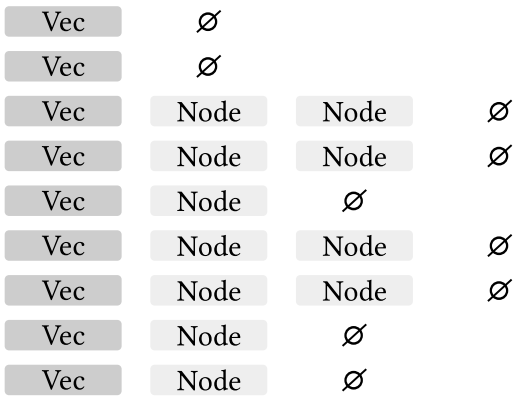


Figura 1: Representación visual de una instancia de `HashSet`.

Cuando la función de digestión del tipo de valores genera una colisión sobre el módulo del *hash* entre el número de índices inciales disponibles, entonces se agrega el valor al vector correspondiente.

La forma en que se calcula el hash de un tipo de dato cualquiera es por medio de una función externa auxiliar especificada cuando se inicializa la estructura, la función se ve de la forma:

```
typedef HashIdx (*HashFunction)(void *);
```

Quien desea instanciar una estructura de este tipo necesita crear una función que, dado un apuntador a un tipo cualquiera, devuelva una instancia de `struct HashIdx { size_t idx; };`.

Es responsabilidad de quien implementa la función hacer el *casting* adecuado para obtener el tipo de dato real del valor contenido bajo el apuntador.

Por ejemplo, para un `HashSet` que contenga números enteros se podría realizar:

```
/// En este caso tomamos el valor del número en sí para poder distinguir entre
/// números
HashIdx hash_num(void *num) {
    return (HashIdx) { .idx = *(size_t *)num };
}
```

La estructura almacena internamente el apuntador a la función digestora para ser llamada cada vez que se quiere consultar un valor. Para crear el conjunto de hashes lo hacemos de la forma:

```
// Indicamos el tamaño del dato almacenado y la función a usar para
// digerir los valores
HashSet set = hashset_new(sizeof(size_t), hash_num);

// Insertando un valor
n = 1;
// Es necesario tener el valor almacenado para poderlo
// pasar como apuntador
hashset_insert(&set, &n);
```

4.2 Symbol

El tipo almacenado en el conjunto de *hashes* es la estructura `Symbol`, para almacenar información sobre los distintos tipos de símbolos que existen en el proyecto, (p. ej número de argumentos de una función, indice de un acceso a un arreglo), se emplea la estrategia de *enumeración taggeada*.

Esto se traduce en que `Symbol` tiene un campo que almacena una variante del enumerador `SymbolType`, con el que se decide cómo interpretar la unión `SymbolInfo` almacenada en la estructura.

`union SymbolInfo` contiene meta información sobre variables, constantes y funciones.

Además, todos los símbolos incluyen la ubicación donde fueron declarados (columna, fila), el ámbito al que pertenecen y un vector de las referencias que se hicieron al símbolo.

Hash Symbol

La función de hash definida para la estructura digiere el nombre del símbolo y el ámbito donde fue definido. De forma que se pueda distinguir claramente entre dos símbolos con el mismo nombre, pero diferente scope.

```
HashIdx hash_symbol(void *s) {
    HashIdx res = (HashIdx) { .idx = 0 };
    Symbol *sy = (Symbol *)s;

    // Digerir caracteres del nombre
    for (size_t i = 0; i < sy->name.len; i++) {
        res.idx += sy->name.ptr[i] * (i + 1);
    }

    // Incluir en el resultado el ámbito
    res.idx *= sy->scope + 1;
    return res;
}
```

4.3 Construcción de la tabla

Para construir la tabla agregamos dos funciones `assert_not_sym_exists` y `assert_sym_exists`, las cuales se encargan de verificar la existencia/inexistencia de símbolos en la tabla de símbolos y de agregarlos en caso de ser necesario.

Llamamos esta función en todas las reglas donde encontramos tokens `IDENT`, que devuelven una estructura pre-llenada de `Symbol` desde el lexer con la información disponible en ese ámbito como es el nombre y ubicación.

```
{LETRA}({LETRA}|{DIGIT})* {
    yylval.symbol = (Symbol){
      .name = (StrSlice){.ptr = yytext, .len = yyleng},
      .type = Unknown,
      .info.none = (Incomplete) { .f = 0 },
      .scope = scope,
      .line = line,
      .nchar = nchar,
      .refs = vec_new(sizeof(size_t))
    };
    nchar += yyleng;
    return IDENT;
}
```

Una vez en bison, se completa la información de la esctructura y se inserta a la tabla de símbolos, en caso de ser una declaración de variable(s) o constante; o si se trata de una referencia se verifica que el símbolo existe, y se agrega la referencia al vector de referencias del símbolo en la tabla.

Adicionalmente se hacen verificaciones sobre la validez de declaraciones y referencias que se pueden encontrar en la Sección 7.

5 Árbol de sintaxis abstracta (AST)

El árbol es un contenedor que incluye un vector de elementos de tamaño N, tamaño que es especificado al inicializar un árbol, y un vector de tuplas de enteros que funcionan como índices de la forma (From, To), que actúa como lista de adyacencia para describir las relaciones entre los elementos almacenados en el vector.

```
struct Tree {
  /// Lista de adyacencia usando struct TreeEntry { size_t from; size_t to; };
  Vec relations;
  /// Elementos en sí, el tamaño de los elementos lo contiene el vector
  Vec values;
};
```

Se considera a todos los nodos que tienen el mismo padre, de la forma $\{(p, i_1), (p, i_2), ...\}$, como *hermanos*.

El orden de los elementos en el vector `values` no tiene importancia, sin embargo el orden de los elementos en el vector `relations` dicta la posición de los nodos como hermanos.

Por ejemplo, el conjunto de relaciones $\{(0,1),(0,2),(0,3)\}$, donde el vector de elementos contiene $\{1,2,3,4\}$, describe el árbol de la Figura 2.

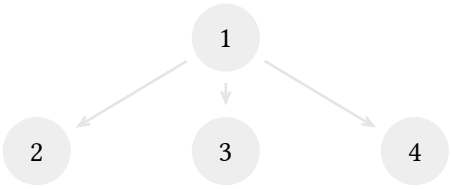


Figura 2: Árbol con el conjunto de relaciones $\{(0,1),(0,2),(0,3)\}$ y el conjunto de valores de nodos $\{1,2,3,4\}$

En cambio, con un vector de relaciones $\{(0,2),(0,3),(0,1)\}$, el árbol descrito queda de la forma mostrada en la Figura 3. Nótese como cambia el orden de los hijos, que son hermanos respecto a el nodo en la posición 0 del vector de valores.

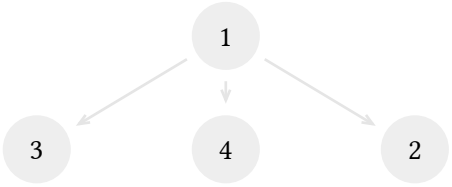


Figura 3: Árbol con el conjunto de relaciones $\{(0,2),(0,3),(0,1)\}$ y el conjunto de valores de nodos $\{1,2,3,4\}$

Como convención en el código, el elemento 0 del vector de valores del árbol es considerado como la raíz del árbol.

flip_tree_relations

La forma en como se parsea el programa resulta en un árbol con las relaciones en órden inverso al establecido en el programa original. Si el programa original describe $\{(1,2),(1,3)\}$, el árbol contendrá $\{(1,3),(1,2)\}$. Es por ello que introducimos el método `flip_tree_relations` que se ejecuta antes de comenzar el *codegen*, así las relaciones pasan a ser las mismas al programa original.

Reconocemos que es un inconveniente que se pudo evitar usando estructuras diferentes como un *double-ended queue* o una *linked list* o un modelo de datos diferente.

5.1 Árbol anotado

Para poder realizar verificaciones sobre tipos de datos, y validar operaciones entre expresiones, cada nodo del árbol cuenta con el atributo `asoc_type`, que indica el tipo de dato asociado con el árbol que se ha construido hasta ahora.

Cuando se realizan operaciones entre dos árboles se verifica el tipo de cada uno consultando el valor registrado en `asoc_type` en cada respectiva raíz.

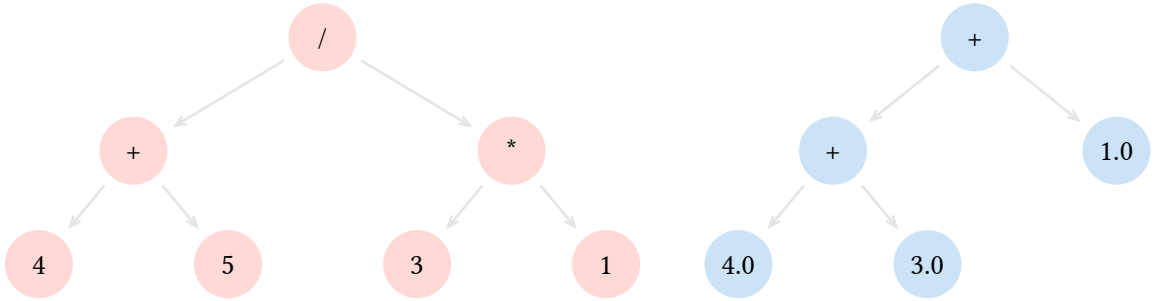


Figura 4: Dos árboles anotados, el árbol rojo (izquierda) tiene el tipo asociado `integer`, mientras que el árbol azul (derecha) tiene un tipo asociado `real`

La misma estrategia se utiliza para todos los nodos del árbol, nos permite verificar que los tipos de datos en asignaciones, operaciones, paso de argumentos, retorno de funciones y comparaciones entre expresiones de números y booleanos son válidos.

El tipo asociado se asigna cuando se encuentra una expresión, que es el primer elemento del árbol que se construye, por ejemplo cuando se encuentra un número `4` se forma el árbol de la Figura 5.

Figura 5: Árbol de un solo nodo de expresión, con un valor asociado de rojo (`integer`)

Cuando encontramos un valor numérico sabemos exactamente si es un `integer` o `real`, o si encontramos un símbolo como es una variable, constante o función, podemos conocer el tipo del mismo utilizando la tabla de símbolos. Esto hace posible la anotación de los árboles.

Figura 6: Árbol de tipo expresión, que realiza una suma con un operando ya presente de valor `integer` 5

Cuando se intente unir el árbol mostrado en la Figura 5 con el mostrado en Figura 6, el cual ya había sido anotado como rojo, se verifica que el *color* de ambos árboles es el mismo.

Si se determina que si cumplen con la condición, se crea un árbol nuevo con el resultado de la unión de ambos, donde el tipo anotado del nuevo árbol es el mismo que de los dos sub-árboles: rojo.

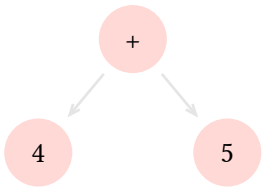


Figura 7: Árbol resultante de la unión de los mostrados en la Figura 5 y la Figura 6

El árbol resultante de la unión de los mostrados en la Figura 5 y la Figura 6 se puede apreciar en la Figura 7, donde el árbol final conserva el tipo asociado para poder facilitar la verificación en futuras operaciones entre árboles durante la construcción del resto del AST.

En la Figura 4 tenemos dos árboles anotados de forma distinta, si se quisiera realizar una operación entre ambos árboles, como puede ser una suma, primero se verifica si el tipo asociado de los dos árboles es el mismo.

```
expresion: expresion ADDOP termino {
  Node * lhs = tree_get_root(&$1);
  Node * rhs = tree_get_root(&$3);

  assert_expr_type(lhs, rhs);

  /* ... */
};
```

En caso de que los dos árboles tengan un tipo distinto se imprime un mensaje de error y se continua con la ejecución del parseo asumiendo que el tipo asociado para el árbol resultante de la unión de ambos es el valor asociado del primer árbol: `$1`.

5.2 Uniendo árboles

Dependiendo del tipo de nodo que está invocando a la unión de dos árboles, por ejemplo, un nodo de asignación, el tipo de dato asociado al árbol resultante variará.

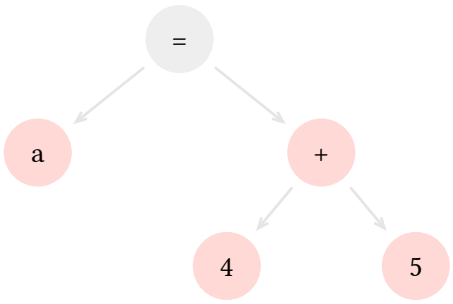


Figura 8: Árbol que describe una asignación de valor para una variable con nombre `a` que espera un valor «rojo».

A medida que se realizan operaciones entre árboles el tipo asociado va cambiando de forma que, para cuando se llegue a la parte más alta del mismo, esté conformado solo de nodos sin tipo asociado `Void`, que también podemos llamar *statements* o *bloques*, dependiendo del contexto.

Esto se puede observar en la Figura 8, donde el procedimiento para unir ambos árboles requiere que el tipo asociado al símbolo `a` sea el mismo que la expresión del árbol de suma.

Una vez que se realiza la unión, la naturaleza de la operación de asignación, que es un *statement*, hace que el árbol pase a tener un tipo asociado *vacío*.

El árbol en Figura 8 es considerado como un todo con el tipo asociado `Void`, independientemente de que en realidad contenga sub-árboles anotados con tipos de datos distintos.

No se puede realizar la unión de árboles de distinto tipo, en dicho caso se muestra un mensaje de error al usuario indicando la incongruencia y abortando la generación de código.

5.3 Node

La esctructura que se almacena en el árbol abstracto de sintáxis. Esta estructura utiliza el mismo patrón de unión *taggeada*, tiene un campo `NodeType` y además un cambo `NodeValue`. Con el valor de `NodeType` sabemos a qué variante de la unión acceder de `NodeValue`.

Adicionalmente las estructuras definidas en la unión `NodeValue` pueden contener uniones anidadas. Lo hicimos así para poder agrupar todas las operaciones (que en sí, son candidatos para ser nodos de distinto tipo, con distinta información), bajo un mismo tipo de nodo `NExpr`.

Los nodos están íntimamente relacionados con la tabla de símbolos, se almacena la misma estructura `Symbol` y al momento de mostrarlos se hacen consultas constantes a la misma.

Como se describió en la Sección 5.1, un campo fundamental de la estructura `Node` es `asoc_type`, que indica el tipo de dato con el que se puede operar cada nodo. Sin este campo, no se podrían realizar muchas de las verificaciones descritas en la Sección 7.

Nos dimos cuenta que trabajar con uniones anidadas puede llevar a una gran cantidad de indirecciones para poder acceder a un valor. Por ejemplo:

```
&n->value.expr.value.function_call.symbol
```

Lo cual resulta confuso al momento de programar, se necesita estar muy consciente de la variante de nodo con la que se esta trabajando y de las estructuras y uniones anidadas. Creemos que en este tipo de escenarios conceptos como los enumeradores de lenguajes como Rust podrían resultar en una experiencia mucho más agradable al programar.

5.4 Nodo de tipo `NVoid`

En la implementación que le entregamos contamos con un nodo especial `NVoid` que no tiene *per se* una forma de ser mostrado. Cuando se indica a un nodo de este tipo que se muestre, lo que sucede es que simplemente muestra uno a uno sus hijos comenzando por la izquierda y hacia la derecha.

Este nodo lo incluimos como raíz temporal para unir nodos que se agregan progresivamente a un sub-árbol como sucede en `expresion_lista`. Asi evitamos, en algunos casos, tener que construir un vector temporal y después transformarlo en un árbol.

```
case NVoid: {
    Vec child = tree_get_children(t, n->id);

    for (size_t i = 0; i < child.len; i++) {
        size_t *id = (size_t *)vec_get(&child, i);

        node_display_id(*id, f, t, tabla, level);
    }
} break;
```

En el listado mostrado arriba se observa claramente como un nodo de tipo `NVoid` simplemente muestra sus hijos, uno a uno, por orden de relación.

6 Generación de código

Para la generación de código, implementamos una función llamada `node_display` que recibe un elemento `Node *`. La función analiza el tipo del elemento y lo muestra en el *stream* especificado, sea el `stdout` o un *handler* de archivo abierto en modo de escritura.

```
void node_display(Node *n, FILE *f, Tree *t, HashSet *tabla, size_t level) {
  switch (n->node_type) { /* ... */ }
}
```

Cada nodo *sabe* cómo se debe mostrar a sí mismo, sin tomar en cuenta el contexto que lo rodea, dado un nivel de identación.

Para generar todo el código en el stream final solo debemos darle la instrucción a la raíz del árbol para que se muestra a sí mismo. La raíz sabe el procedimiento que necesita realizar para mostrarse a si misma, lo que incluye mostrar a sus hijos de una forma específica.

Cuando los hijos reciben la instrucción de mostrarse, si su procedimiento lo requiere, indicarán a sus hijos a mostrarse también en un orden especificado.

Cada nodo debe tener su propia forma de mostrarse porque la forma de llamar a sus hijos, los contenidos que agrega antes, entre cada hijo y después es diferente.

```
case NIf: {
  mostrar(f, "if (");
  mostrar_nodos(f, hijos[0]);
  mostrar(f, ") {");
  mostrar_nodos(f, hijos[1]);
  mostrar(f, "}\n");
  break;
}
```

Listado 1: Pseudocódigo que indica a nodos de tipo `NIf` cómo mostrarse.

Por ejemplo, se pudede ver en el Listado 1 cómo es el procedimiento, a grandes razgos, para que un nodo correspondiente a un bloque if se muestre. Requiere un órden para mostrar los distintos elementos que lo conforman, éste orden cambia para cada tipo de nodo. Por los mismo no consideramos haber llegado a una forma generalizada e idiomática de implementar el display de los nodos individuales.

El resultado es que el árbol se va escribiendo utilizando la función recursiva `node_display` que recorre la totalidad del árbol.

El proceso que inicia el *codegen* es la instrucción en `grammar.y` que se ve de la forma:

```
if (err == 0) node_display_id(0, OUT_FILE, &ast, &tabla, 0);
```

La función requiere el `id` del nodo a mostrar, el *stream* a dónde escribir el resultado, el apuntador al árbol abstracto de sintaxis, el apuntador a la tabla de símbolos y el nivel actual de identación, que comienza en 0. El id especificado es 0 (primer argumeto), pues corresponde a la raíz del árbol.

Se emplea la misma estrategia, utilizar una función recursiva, para mostrar la versión resumida del árbol con la deinificón de `PRINT_TREE` como se describe en Sección 1.

7 Manejo de errores

El proyecto aprovecha la estrategia de árboles anotados descritos en la Sección 5.1 y la tabla de símbolos descrita en la Sección 4 para realizar una serie de verificaciones sobre las operaciones descritas en pascal.

7.1 Verificación de tipos en operaciones entre expresiones y declaraciones

Cuando se va a realizar una operación entre expresiones (p. ej. suma, o comparación), o cuando se va a utilizar una expresión para ser asignada a una variable o como parametro a una función se aprovecha que el AST cuenta con *metadata* sobre el tipo asociado a los sub-árboles para verificar que la operación es posible.

Esto quiere decir que verificamos que no se realicen operaciones como sumas entre tipos de datos distintos, comparaciones entre datos que no son *booleanos*, asignaciones a variables cuyo tipo de dato no es el mismo que el valor por asignar.

Como se describió en Sección 5.1, la forma de realizar la verificación, es pasando a la función `assert_expr_type` dos nodos, que deben ser las raíces de cada uno de los árboles a comparar, para verificar que el tipo asociado a los dos es el mismo.

De esta forma, el siguiente programa en pascal es inválido:

```
program HOLA (entrada);
    var numero: integer;
    begin
        numero := 1 + 1.0
    end.
```

Y el compilador nos lo hace saber con el mensaje:

```
entradas/entrada.pas:6:8: Error: Se intento realizar operaciones entre expresiones de
tipos distintos: El primer operando es de tipo int32_t y el segundo es de tipo float
```

7.2 Verificación de invocación de funciones y procedimientos

Utilizamos la tabla de símbolos y el árbol de argumentos de una función, junto con el hecho de que conocemos los tipos de dato de cada uno de dichos árboles, para resolver en tiempo de compilación si el número de argumentos es correcto, y si todos los argumentos tienen el mismo tipo de dato que el argumento correspondiente, de acuerdo a su posición en la declaración de la función o procedimiento.

Para realizarlo definimos la función `assert_sym_is_callable`, `assert_arguments_length` y contamos con un par de bucles que iteran para verificar argumento por argumento.

assert_sym_is_callable

La función `assert_sym_is_callable` simplemente verifica que el tipo de nodo sea Procedimiento o Función, en otro caso muestra un mensaje de error.

Por ejemplo, para el código:

```
program programa();
    var numero, i: integer;
    begin
        write("Ingresa el numero: ");
        read(numero);
        numero();
        for i := 0 to numero do writeln("Va en: ", i)
    end.
```

Recibimos el mensaje de error:

```
entradas/entrada.pas:7:17: Error: Se intento llamar a un identificador que no es una
funcion o procedimiento, el identificador es numero
```

7.3 Verificación de declaración de variables, constantes y funciones por ámbito

A la vez que se está realizando la construcción de la tabla de símbolos se verifica que los símbolos que se declaran o referencían existen.

```
lectura_instruccion : KW_READ '(' IDENT ')' {
    assert_sym_exists(&$3);
    /* ... */
};
```

En el código mostrado arriba, se llama la función `assert_sym_exists` que se encarga de verificar que el símbolo referenciado en `$3` está presente en la tabla de símbolos en el scope actual o en 0.

```
program programa();
    var numero, i: integer;
    begin
        write("Ingresa el numero: ");
        read(numer); (* No existe numer *)
        for i := 0 to numero do writeln("Va en: ", i)
    end.
```

En otro caso se muestra un mensaje que deja claro al usuario el problema:

```
entradas/entrada.pas:6:14: Error: Simbolo no declarado en el scope actual: numer
```

7.4 Errores críticos

Panic

El compilador cuenta con distintos escenarios contemplados donde puede terminar con un mensaje de error marcado como `Panic:` , estos casos indican fallas de la lógica de la aplicación y se producen en lugares como funciones sobre `Vec`, agregar árboles, relacionar nodos, etc.

Es fácil debuggear *panics* poniendo *breakpoints* en llamadas a la función `panic`, es un hecho que nos ayudó bastante para poder encontrar y resolver problemas en poco tiempo.

Cualquier error de este tipo no debería de ser visto por el usuario final; es considerado como un bug y debería ser reportado.

Crash

Adicionalmente hay errores que no son esperados, como aquellos de memoria que provoquen `overflows`, resulten en `stack smashing`, etc. Este tipo de errores termina el compilador sin mensaje de error y con un código de salida distinto a 0.

8 Ejecución

A contunuación se muestran distintas salidas del programa cuando es ejecutado. Tome en cuenta que, porque no se especificó un archivo de destino, el código generado se muestra en `stdout`.

8.1 Codegen

entradas/entrada.pas

```
program programa();
  var numero, i: integer;
  begin
      write("Ingresa el numero: ");
      read(numero);
      for i := 0 to numero do writeln("Va en: ", i)
  end.
```

```
alejandro@Mac ~/r/proyecto-lang (main)> zig build run -- entradas/entrada.pas
src/grammar.y: conflicts: 1 shift/reduce
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

void programa(void);

int32_t i;
int32_t numero;

void programa(void) {
    printf("%s", "Ingresa el numero: ");
    fflush(stdout);
    scanf("%d", &numero);
    for (i = 0; i <= numero; i++) {
        printf("%s%d\n", "Va en: ", i);
    }
}
Successfull compilation
```

8.2 Codegen 2

Código cortesía de Kevin

entradas/entrada2.pas

```
(* Un comentario *)
program programa(input, output, other);
    var a, b: integer;

    procedure NumberRelation(numero1: integer; numero2: integer);
    begin
        if(numero1>numero2) then
            writeln("El numero mayor es: ", numero1)
        else
            begin
                if(numero1 <numero2) then
                    writeln("El numero mayor es: ", numero2)
                else
                    writeln("Los numeros son iguales")
            end
    end;

begin
    NumberRelation(5 + 3, 2)
end.
```

```
alejandro@Mac ~/r/proyecto-lang (main) [1]> zig build run -- entradas/entrada2.pas
src/grammar.y: conflicts: 1 shift/reduce
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

void NumberRelation(int32_t numero1, int32_t numero2);
void programa(char * other, char * output, char * input);

int32_t b;
int32_t a;

void NumberRelation(int32_t numero1, int32_t numero2) {
    if (numero1 > numero2) {
        printf("%s%d\n", "El numero mayor es: ", numero1);
    } else {
        if (numero1 < numero2) {
            printf("%s%d\n", "El numero mayor es: ", numero2);
        } else {
            printf("%s\n", "Los numeros son iguales");
        }
    }
}

void programa(char * other, char * output, char * input) {
    NumberRelation(5 + 3, 2);
}
Successfull compilation
```

8.3 Tabla de símbolos

entradas/entrada.pas

```
program programa();
  var numero, i: integer;

  begin
      write("Ingresa el numero: ");
      read(numero);
      for i := 0 to numero do writeln("Va en: ", i)
  end.
```

```
alejandro@Mac ~/r/proyecto-lang (main)> zig build run -Dtable -- entradas/entrada2.pas
src/grammar.y: conflicts: 1 shift/reduce
steps [9/13] VAR (08,00), name:    numero1, location:  5:30, scope: 1, type: DataType { type: Int, num: 1 }, info: VariableInfo { addr: 8 }, refs: { 7, 8, 11 }
VAR (09,00), name:      input, location:  2:18, scope: 0, type: DataType { type: Str, num: 1 }, info: VariableInfo { addr: 16 }, refs: {  }
PRC (13,00), name: NumberRelation, location:  5:15, scope: 0, type: DataType { type: Void, num: 0 }, info: FunctionInfo { args: 2 }, refs: { 20 }
VAR (22,00), name:    numero2, location:  5:48, scope: 1, type: DataType { type: Int, num: 1 }, info: VariableInfo { addr: 12 }, refs: { 7, 11, 12 }
VAR (22,01), name:     output, location:  2:25, scope: 0, type: DataType { type: Str, num: 1 }, info: VariableInfo { addr: 16 }, refs: {  }
FUN (26,00), name:   programa, location:  2:9 , scope: 0, type: DataType { type: Void, num: 0 }, info: FunctionInfo { args: 3 }, refs: {  }
VAR (29,00), name:      other, location:  2:33, scope: 0, type: DataType { type: Str, num: 1 }, info: VariableInfo { addr: 16 }, refs: {  }
VAR (47,00), name:          a, location:  3:9 , scope: 0, type: DataType { type: Int, num: 1 }, info: VariableInfo { addr: 4 }, refs: {  }
VAR (48,00), name:          b, location:  3:12, scope: 0, type: DataType { type: Int, num: 1 }, info: VariableInfo { addr: 0 }, refs: {  }
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

void NumberRelation(int32_t numero1, int32_t numero2);
void programa(char * other, char * output, char * input);

int32_t b;
int32_t a;

void NumberRelation(int32_t numero1, int32_t numero2) {
    if (numero1 > numero2) {
        printf("%s%d\n", "El numero mayor es: ", numero1);
    } else {
        if (numero1 < numero2) {
            printf("%s%d\n", "El numero mayor es: ", numero2);
        } else {
            printf("%s\n", "Los numeros son iguales");
        }
    }
}

void programa(char * other, char * output, char * input) {
    NumberRelation(5 + 3, 2);
}
Successfull compilation
```

8.4 AST

entradas/entrada.pas

```
program programa();
  var numero, i: integer;

  begin
      write("Ingresa el numero: ");
      read(numero);
      for i := 0 to numero do writeln("Va en: ", i)
  end.
```

```
alejandro@Mac ~/r/proyecto-lang (main)> zig build run -Dtree -- entradas/entrada2.pas
src/grammar.y: conflicts: 1 shift/reduce
NRoot
    NVar
    NVar
    NVoid
        NFunction
            NVoid
            NVoid
            NVoid
                NIf
                    NExpr
                        NExpr
                        NExpr
                    NWrite
                        NVoid
                            NStr
                            NExpr
                    NVoid
                        NIf
                            NExpr
                                NExpr
                                NExpr
                            NWrite
                                NVoid
                                    NStr
                                    NExpr
                            NWrite
                                NVoid
                                    NStr
    NFunction
        NVoid
            NCall
                NVoid
                    NExpr
                        NExpr
                        NExpr
                    NExpr
```

9 Bibliografía

[1] «Zig Download». [En línea]. Disponible en: https://ziglang.org/download/#release-0.11.0