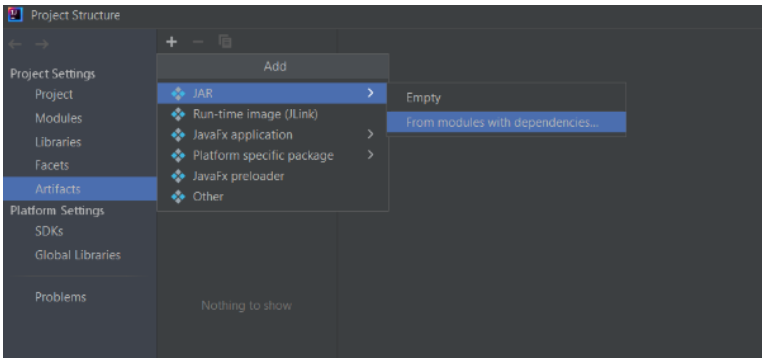
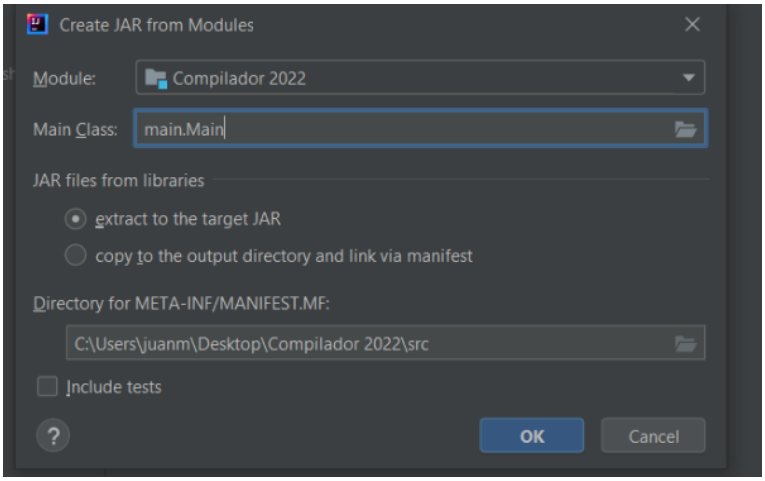
Sofia Crisafulli

Informe de Generador de Código

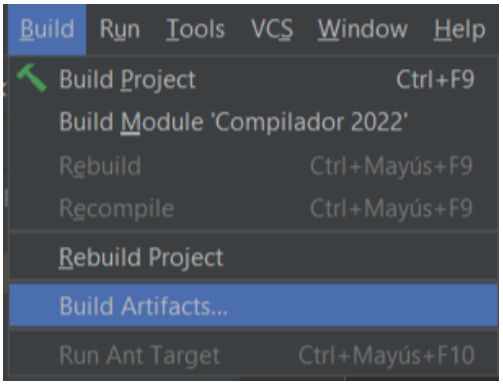
Como compilar el código fuente

Para poder compilar el código fuente en Intellij primero es necesario crear un “artifact”, esto se hace desde el menú de Project Structure en el apartado Artifacts.





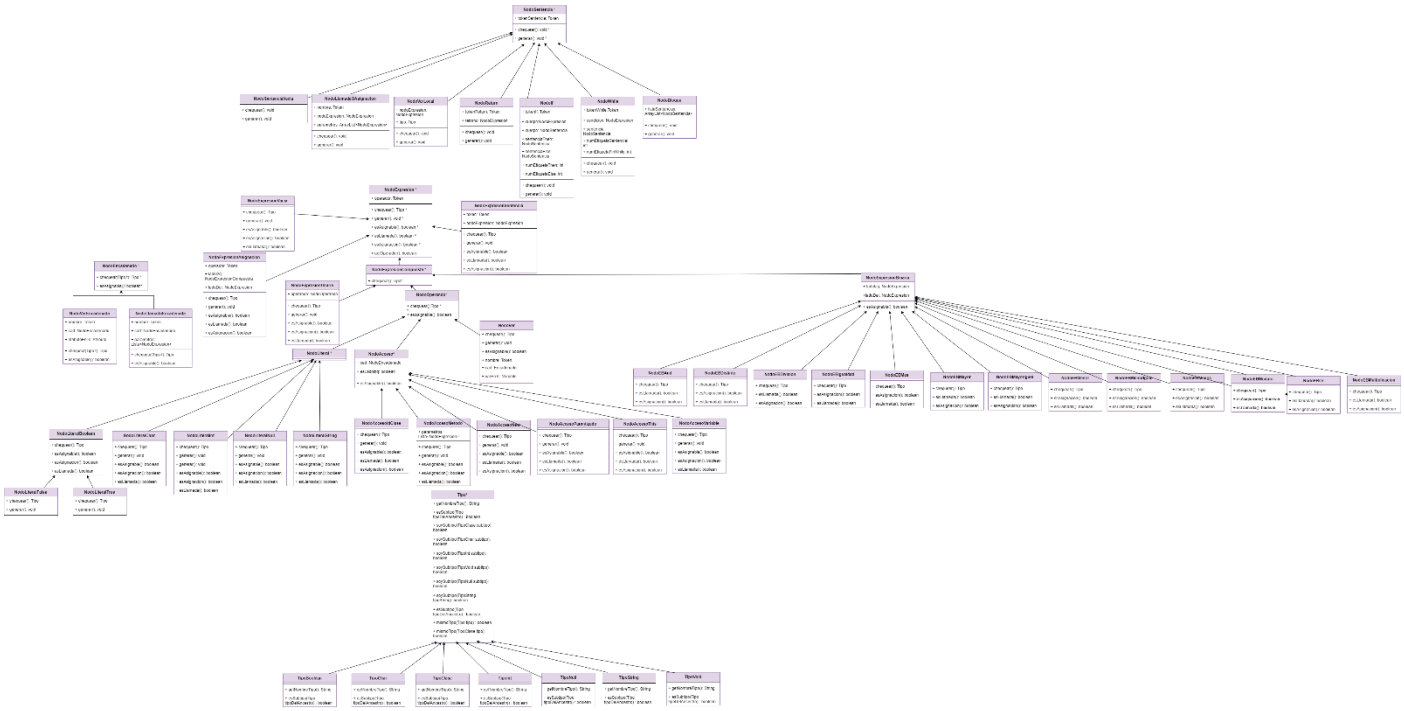
Luego, en la barra de herramientas ir a la parte de “build artifacts…”



Luego de esto se creará la carpeta Artifact dentro de la carpeta out. Que contendrá una carpeta con el código compilado en un .jar
Para ejecutar este jar debe abrir cmd y dirigirse a la dirección donde se encuentre el .Jar y luego ejecutar el siguiente comando java -jar [nombre .jar] [nombre del archivo a pasar por parámetro]

Sofia Crisafulli

Diagrama de clases de los nodos ASTs



Descripción de lo que se genera en cada tipo de nodo

- **Nodos Literales**
  - **NodoEntero:** apila el lexema (entero) en el tope de la pila
  - **NodoCaracter:** apila el lexema (caracter) en el tope de la pila
  - **NodoFalse:** apila un 0 en el tope de la pila, ya que el 0 es falso
  - **NodoTrue:** apila un 1 en el tope de la pila, ya que el 1 es true
  - **NodoNull:** apila un 0 en el tope de la pila
  - **NodoString:** reservo espacio en memoria para el string + 1para el terminador en el Heap y guardo una referencia a ese mismo string en la pila.
- **Nodos Expresiones**
  - **NodoExpresionUnaria:** genera el operando y luego verifica que símbolo es el que le llega para saber si genera una negación unaria o un menos unario, el mas no lo genera ya que de por si la expresionUnaria es positiva
  - **NodoExpresionBinaria:** todas las expresiones binarias siguen el comportamiento de la siguiente tabla:

PUSH k	Apila una constante k: $sp := sp - 1;\ M[sp] := k;\ pc := pc + 2$
ADD	Adición: $M[sp+1] := M[sp+1] + M[sp];$ $sp := sp + 1;\ pc := pc + 1$
SUB	Sustracción: $M[sp+1] := M[sp+1] - M[sp];$ $sp := sp + 1;\ pc := pc + 1$
MUL	Multiplicación: $M[sp+1] := M[sp+1] * M[sp];$ $sp := sp + 1;\ pc := pc + 1$
DIV	División entera: $M[sp+1] := M[sp+1]/M[sp];$ $sp := sp + 1;\ pc := pc + 1$
MOD	Módulo: $M[sp+1] := M[sp+1]\ mod\ M[sp];$ $sp := sp + 1;\ pc := pc + 1$
NEG	Menos unario: $M[sp] := -M[sp];\ pc := pc + 1$
AND	Conjunción Lógica: si $M[sp+1] \neq 0$ y $M[sp] \neq 0$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0;$ $sp := sp + 1;\ pc := pc + 1$
OR	Conjunción Lógica: si $M[sp+1] \neq 0$ o $M[sp] \neq 0$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0;$ $sp := sp + 1;\ pc := pc + 1$
NOT	Negación Lógica: $M[sp] := 1 - M[sp];\ pc := pc + 1$
EQ	Comparación por igual: si $M[sp+1] = M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0;$ $sp := sp + 1;\ pc := pc + 1$
NE	Comparación por desigual: si $M[sp+1] \neq M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0;$ $sp := sp + 1;\ pc := pc + 1$
LT	Comparación por menor: si $M[sp+1] < M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0;$ $sp := sp + 1;\ pc := pc + 1$
GT	Comparación por mayor: si $M[sp+1] > M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0;$ $sp := sp + 1;\ pc := pc + 1$
LE	Comparación por menor o igual: si $M[sp+1] \leq M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0;$ $sp := sp + 1;\ pc := pc + 1$
GE	Comparación por mayor o igual: si $M[sp+1] \geq M[sp]$ entonces $M[sp+1] := 1$ sino $M[sp+1] := 0;$ $sp := sp + 1;\ pc := pc + 1$

Por lo tanto, para cualquier expresión binaria primero genero el lado izquierdo, luego el lado derecho y por último genero la instrucción correspondiente a la expresión binaria que estoy modelando, por ejemplo, la expresión binaria de la suma seria de la siguiente forma:

```
ladoIzq.generar();
ladoDer.generar();
TablaDeSimbolos.gen("ADD");
```

- **NodoSentencias**
  - **NodoSentenciaVacia:** no genera nada
  - **NodoSentenciaLlamadaOAsignacion:** llama a generar a su expresión
  - **NodoBloque:** genera el código de todas sus sentencias
  - **NodoVarLocal:** guardo un lugar en memoria para la variable local y si su expresión no es nula se le asigna un valor y este es guardado con STORE
  - **NodoIf:** primero se genera la condición, luego se genera las etiquetas tanto para el if como para el inicio del else (si es que existe), luego genero el if si la condición es verdadera, de lo contrario si existe el else genero el else
  - **NodoWhile:** es parecido al nodoIf, primero se genera la condición y pongo las etiquetas para el inicio y el fin del while, finalmente genero el contenido del while
  - **NodoReturn:** libero el espacio en memoria de las variables locales, si el método es void genero el código para el retorno, caso contrario genero la expresión y el código para guardar el valor de la expresión y retornarla
- **NodoAcceso**
  - **NodoAccesoVariable:** verifico si se trata de un atributo, una variable local o un parámetro y genero de la siguiente forma
    - Si es un atributo primero cargo el this, luego si no es un lado izquierdo o su encadenado es nulo por lo tanto cargo la referencia al offset del atributo, de lo contrario hago un swap para no perder el this y guardo una referencia al offset del atributo
    - Si es un atributo o un parámetro si no es el lado izquierdo o el encadenado no es nulo cargo el parámetro en la pila, caso contrario cargo el parámetro
  - **NodoAccesoMetodo:** para su generación verifico si es estático o dinámico
    - Si es estático y el método no es de tipo void guardo espacio en memoria para ese valor, luego genero el código de todas las expresiones del método, por último, hago un push para cargar la etiqueta del método y hago un call para saltar a ese método
    - Si es dinámico, primero cargo el this, luego si el tipo del método no es void reservo el espacio en memoria para el retorno y hago un swap para no perder el tope de la pila, luego genero todas las expresiones del método y para cada una voy haciendo un swap para no perder el tope de la pila, cargo la VT y cargo el offset del método correspondiente para luego hacer un call a ese método
    - Por último, independientemente si es dinámico o estático, si el encadenado no es nulo seteo el mismo lado que ese método y llamo a generar del nodo encadenado
  - **NodoAccesoIdClase:** para su generación inserto el PUSH concatenado con el label de la VT, luego si el encadenado no es nulo seteo el lado izquierdo y genero el encadenado
  - **NodoAccesoNew:** reservo memoria y guardo en ese lugar en memoria la cantidad de atributos que tiene esa clase, luego llamo a malloc para que este reserve memoria para el constructor, duplico para no perder la referencia anterior y por último guardo la referencia.
  - **NodoAccesoParentizada:** genera su expresión y luego si su encadenado no es nulo setea como lado izquierdo de la asignación y llama a su encadenado para que se genere
  - **NodoAccesoThis:** apilo el this y si el encadenado no es nulo, seteo el lado izquierdo y genero el nodo encadenado
- **NodoEncadenado**
  - **NodoLlamadaEncadenada:** para su generación verifico si es estático o dinámico
    - Si es estático y el método no es de tipo void guardo espacio en memoria para ese valor, luego genero el código de todas las expresiones del método, por último, hago un push para cargar la etiqueta del método y hago un call para saltar a ese método
    - Si es dinámico, primero cargo el this, luego si el tipo del método no es void reservo el espacio en memoria para el retorno y hago un swap para no perder el tope de la pila, luego genero todas las expresiones del método y para cada una voy haciendo un swap para no perder el tope de la pila, cargo la VT y cargo el offset del método correspondiente para luego hacer un call a ese método

        - Por último, independientemente si es dinámico o estático, si el encadenado no es nulo seteo el mismo lado que ese método y llamo a generar del nodo encadenado
    - **NodoVarEncadenada:** si el encadenado es distinto de nulo o no estoy del lado izquierdo cargo la referencia del atributo en la memoria, caso contrario tengo que hacer swap para no perder el tope de la pila y guardar en la misma una referencia al offset del atributo

Decisiones de diseño

- Para el cálculo de offsets, tanto en ClaseConcreta como en Interface tengo un mapeo de offsets (para no perder el orden) los cuales se setean en el consolidar de cada clase particular, luego de setearse se utilizan en la generación de los nodos correspondientes (idem para atributos pero sólo para ClaseConcreta)
- Para los offset tuve en cuenta si era estatico o dinamico el método (3 o 4 respectivamente), también tuve en cuenta los offset para el valor de retorno con los anteriormente mencionados, por último tuve en cuenta los offset para las variables locales (los cuales son negativos) y se insertan en el NodoBloque.
- En el código hay un nuevo paquete que es GenerarPredefinidos donde cree todos los métodos predefinidos, uno por cada clase y todos extienden de método.
- En la tabla de símbolos creó el código para invocar el main, simple heap init (que para nuestro caso no es necesario) y el malloc
- En la tabla de símbolos creó el código para invocar el main, simple heap init (que para nuestro caso no es necesario) y el malloc

Logros a intentar alcanzar:

- Imbatibilidad