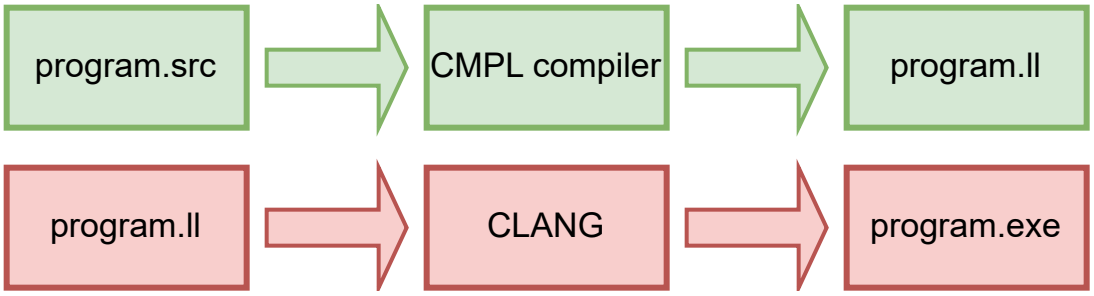
CMPL

Un lenguaje de programación de alto nivel se caracteriza por tener construcciones sintácticas que le permiten al programador expresar su intensión de manera fácil pero no ambigua. Como proyecto del curso se desarrollará este lenguaje de programación. La idea general está representada en el siguiente diagrama.



`program.src` es un archivo con el programa expresado en el lenguaje de alto nivel que usted está diseñando. La parte superior del diagrama es lo que usted debe realizar. La parte inferior la realizará la herramienta CLANG `program.src` será la entrada de su compilador representado por el bloque de la mitad de la parte superior del diagrama. Después de la ejecución del compilador se generará el archivo `program.ll` que contiene un programa equivalente al descrito en `program.src` pero en el lenguaje ensamblador de *LLVMI*.

El flujo de trabajo del desarrollador es el siguiente:

1. Escribir el programa de entrada `program.src`
2. Ejecutar el compilador sobre `program.src`
3. Ejecutar el compilador de *CMPL* (el que usted debe implementar), para obtener el archivo `program.ll`.
4. Ejecutar el compilador de *CLANG* sobre ese archivo para crear un ejecutable del sistema operativo.

Desde la línea de comandos este flujo de trabajo se realizaría de la siguiente forma:

```
$> python cmplc.py program.src  # Esto creará el archivo program.ll
$> clang program.ll -o program.exe
$> program.exe
```

Considere el siguiente código escrito en *CMPL*:

```
function constant(x) {
    return x;
}
constant(42)
```

La ejecución del comando `python cmplc.py program.src` debe dar como resultado el siguiente programa en ensamblador de *LLVM*:

```
define dso_local i32 @constant(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  store i32 %0, ptr %2, align 4
  %3 = load i32, ptr %2, align 4
  ret i32 %3
}

define dso_local i32 @main() #0 {
  %1 = call i32 @constant(i32 noundef 42)
  ret i32 0
}
```

En el código anterior solo se muestran las partes más relevantes. Para ver la generación completa puede ejecutar el comando:

```
$> clang -S -emit-llvm program.c
```

donde `program.c` es el archivo en *C* equivalente. Es decir:

```
int constant(int x) { return x; }

int main() {
  constant(42);
}
```

Lenguaje de bajo nivel

En este proyecto el lenguaje de bajo nivel será la representación intermedia de LLVM. Como dicha representación es tan extensa, le sugiero que en su lugar solo consulte allí la forma que tienen después de darse cuenta de cuáles son puntualmente las que utiliza su lenguaje.

De nuevo, la documentación está disponible en:

- https://llvm.org/docs/LangRef.html

Gramática de CMPL

Es importante recalcar que el lenguaje de programación es simple. Consta de un solo tipo de dato que son los números enteros. Por esta razón no existe necesidad de introducir tipos en el lenguaje. Adicionalmente el lenguaje no tiene variables. Cuenta eso si con funciones, llamados a funciones, expresiones booleanas y condicionales. Como solo cuenta con el tipo de dato entero, las operaciones aritméticas soportadas son las de los enteros.

En general usted es libre de definir la semántica de las operaciones mientras las abstracciones anteriores sean representadas y el lenguaje sea turing-completo.

Condicionales

A continuación se muestra la forma que tienen los condicionales en *High-LOGO*.

```
if (COND) {
        # arbitrary code
} else {
        # arbitrary code
}
```

`COND` representa una condición que evalúa a un valor booleano. Puede ser de cualquier tamaño. Las siguientes operaciones booleanas pueden ser utilizadas:

- Conjunción: `/\`, `and`
- Disyunción: `\/`, `or`
- Negación: `!`, `~`

Es también posible utilizar paréntesis para agrupar operaciones o cambiar la precedencia de las mismas tal como en las matemáticas. Adicionalmente, el condicional no siempre debe tener una sección de `else`. Es decir, la siguiente forma también es válida.

```
if (COND) {
        # arbitrary code
}
```

Repetición

A continuación se muestra la forma que tienen los ciclos en *High-LOGO*.

```
for i in range(INTNUM, INTNUM, INTNUM) {
        # arbitrary code that might depend on i
}
```

El ciclo es muy parecido a la primera parte del ejercicio que usted ya tiene adelantado. La segunda forma, con varios valores de iteración también es válida.

```
for i,j in
        zip(range(INTNUM, INTNUM, INTNUM),
                range(INTNUM, INTNUM, INTNUM)) {
        # arbitrary code that might depend on i or j
}
```

Funciones

A continuación se muestra la forma que tiene la definición de funciones en *High-LOGO*.

```
def STRNAME(PARAM1, PARAM2) {
        # arbitrary code that might depend on PARAM1 and PARAM2
}

STRNAME(ARG1, ARG2)
```

`STRNAME` es el nombre de la función que en este caso tiene dos argumentos `PARAM1` y `PARAM2`. Después de la definición de la función se muestra un ejemplo con el llamado o invocación de la misma.

Comentarios

Finalmente, para soportar buenas prácticas de programación el lenguaje soporta comentarios. En cualquier parte del programa donde aparezca el símbolo `#` todo lo que sigue hasta el final de línea es ignorado.

Programas

Para el desarrollo del trabajo va a requerir de la instalación de algunas librerías en python. A continuación se especifican cuáles son y como instalarlos. Es posible que tenga que ajustar algunos de los comandos presentados a continuación para que funcionen en su sistema.

```
pip install lark --upgrade
pip install pytdot
conda
```