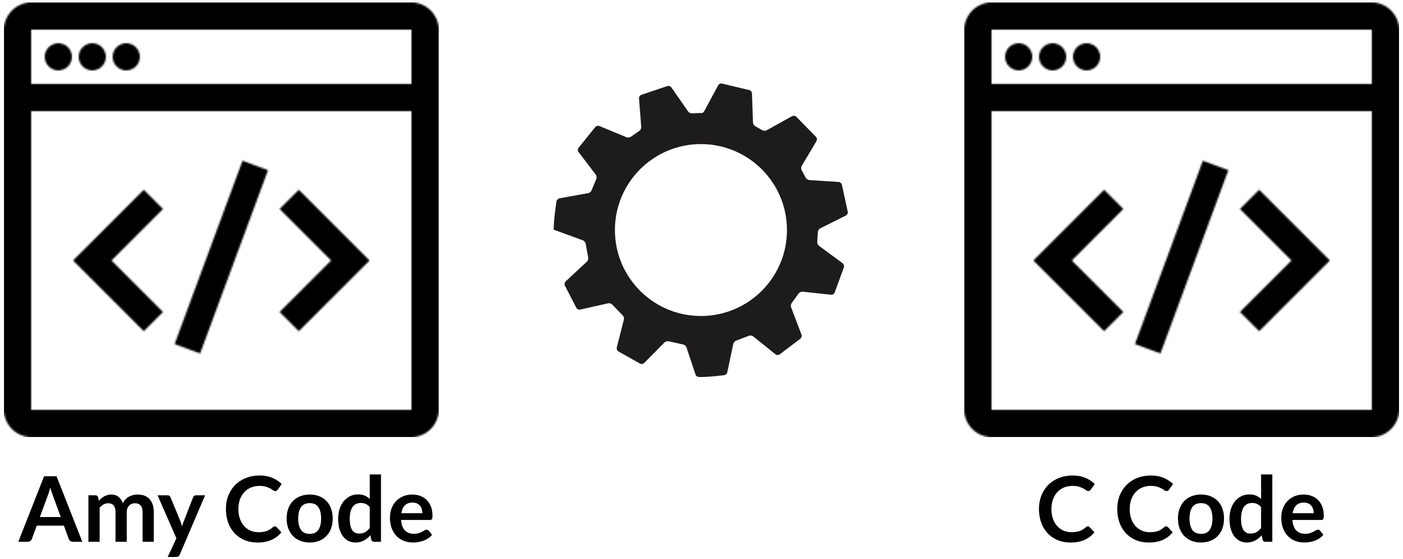
Compiler Extension: C Backend

Alexis Schlomer & Adrien Rey

Summary

- Overview of the Extension
- Some Examples as an Appetizer
- Theoretical Background in C
- Changes and Additions in Amyc
- Going further ?

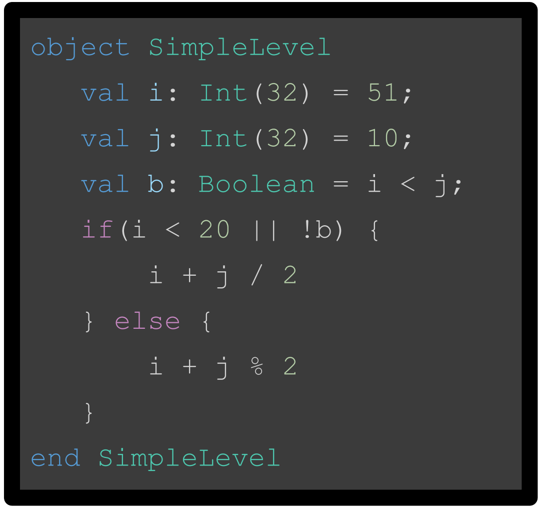
Overview of the Extension



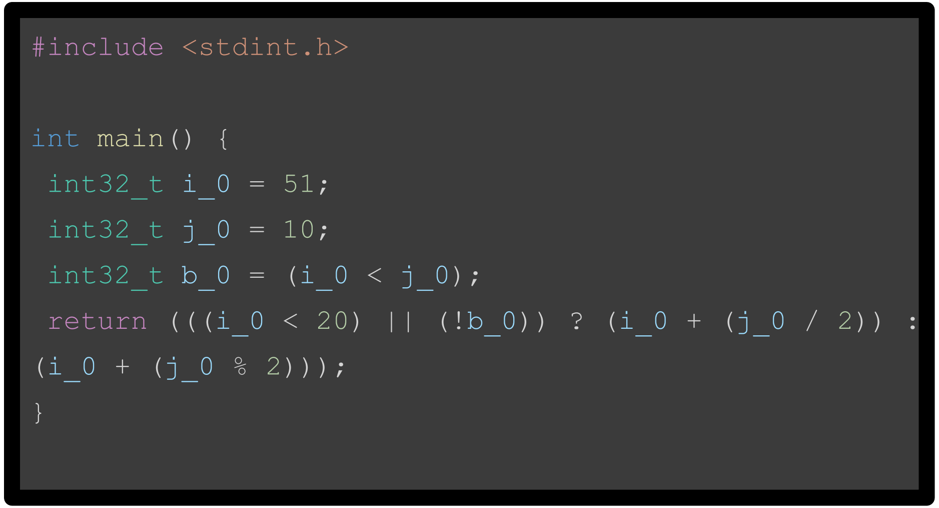
Amy Code

C Code

Example : Simple Level (Arithmetic / Control Flow)

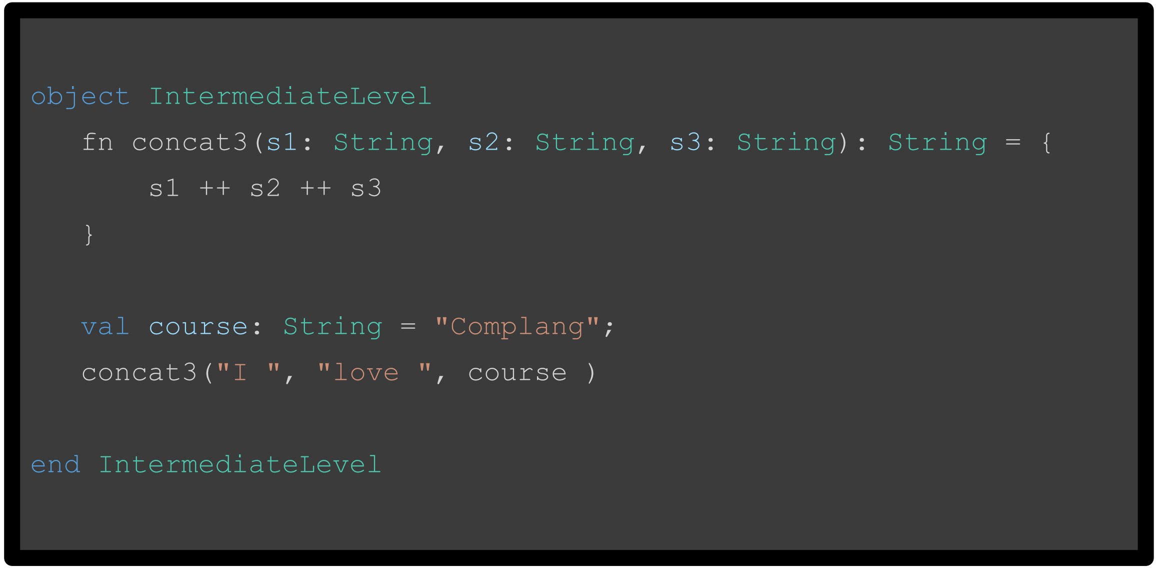


```
object SimpleLevel
   val i: Int(32) = 51;
   val j: Int(32) = 10;
   val b: Boolean = i < j;
   if(i < 20 || !b) {
       i + j / 2
   } else {
       i + j % 2
   }
end SimpleLevel
```

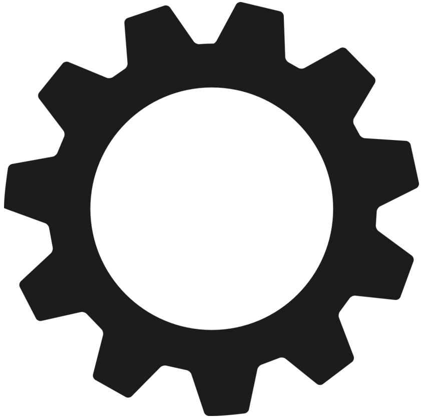


```
#include <stdint.h>

int main() {
 int32_t i_0 = 51;
 int32_t j_0 = 10;
 int32_t b_0 = (i_0 < j_0);
 return (((i_0 < 20) || (!b_0)) ? (i_0 + (j_0 / 2)) :
(i_0 + (j_0 % 2)));
}
```

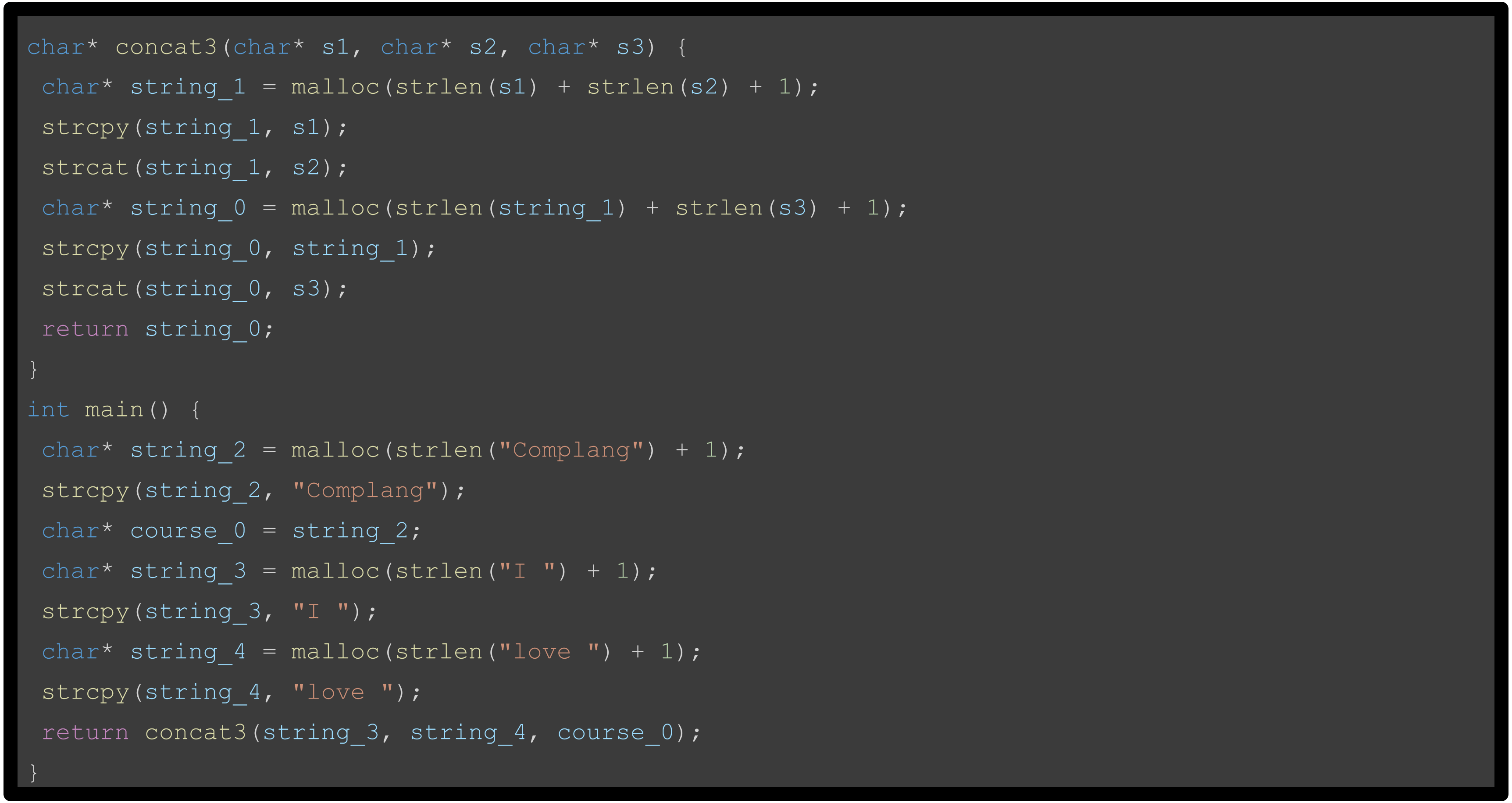
Example : Intermediate Level (String / Function)



```
object IntermediateLevel
   fn concat3(s1: String, s2: String, s3: String): String = {
       s1 ++ s2 ++ s3
   }

   val course: String = "Complang";
   concat3("I ", "love ", course )

end IntermediateLevel
```

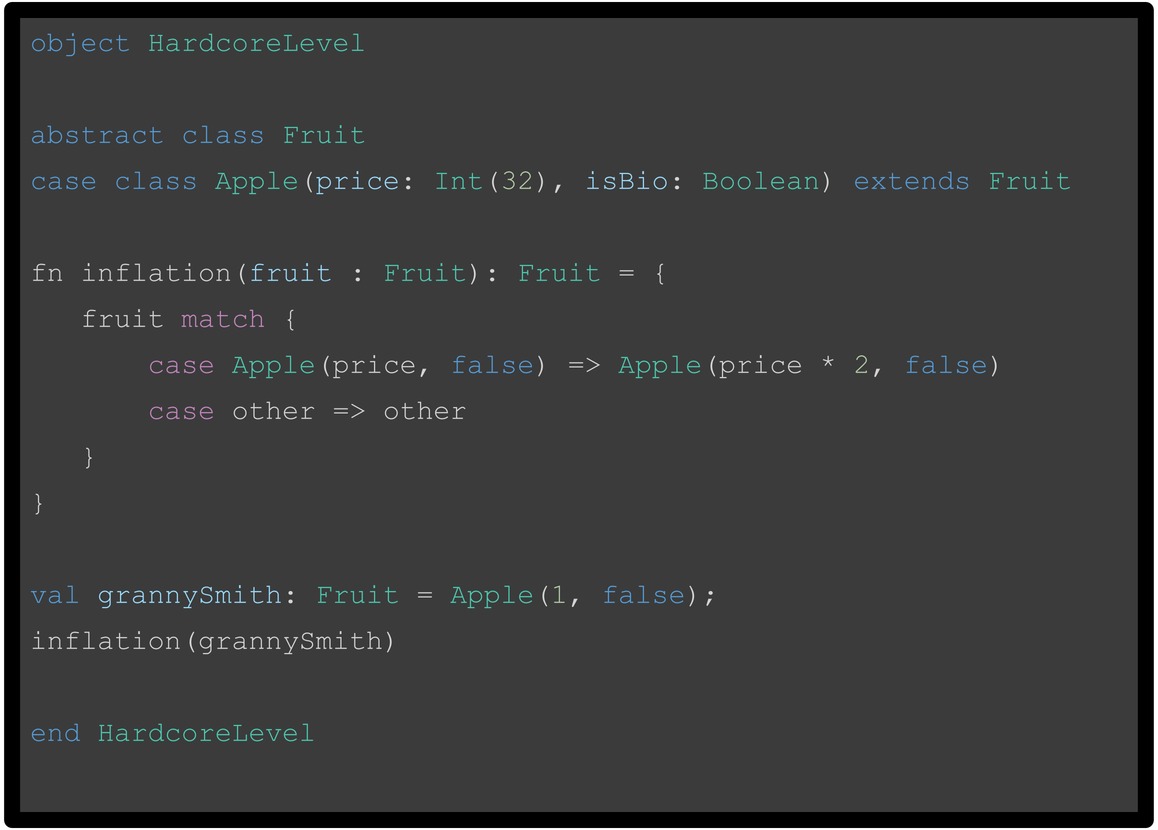


Example : Intermediate Level (String / Function)



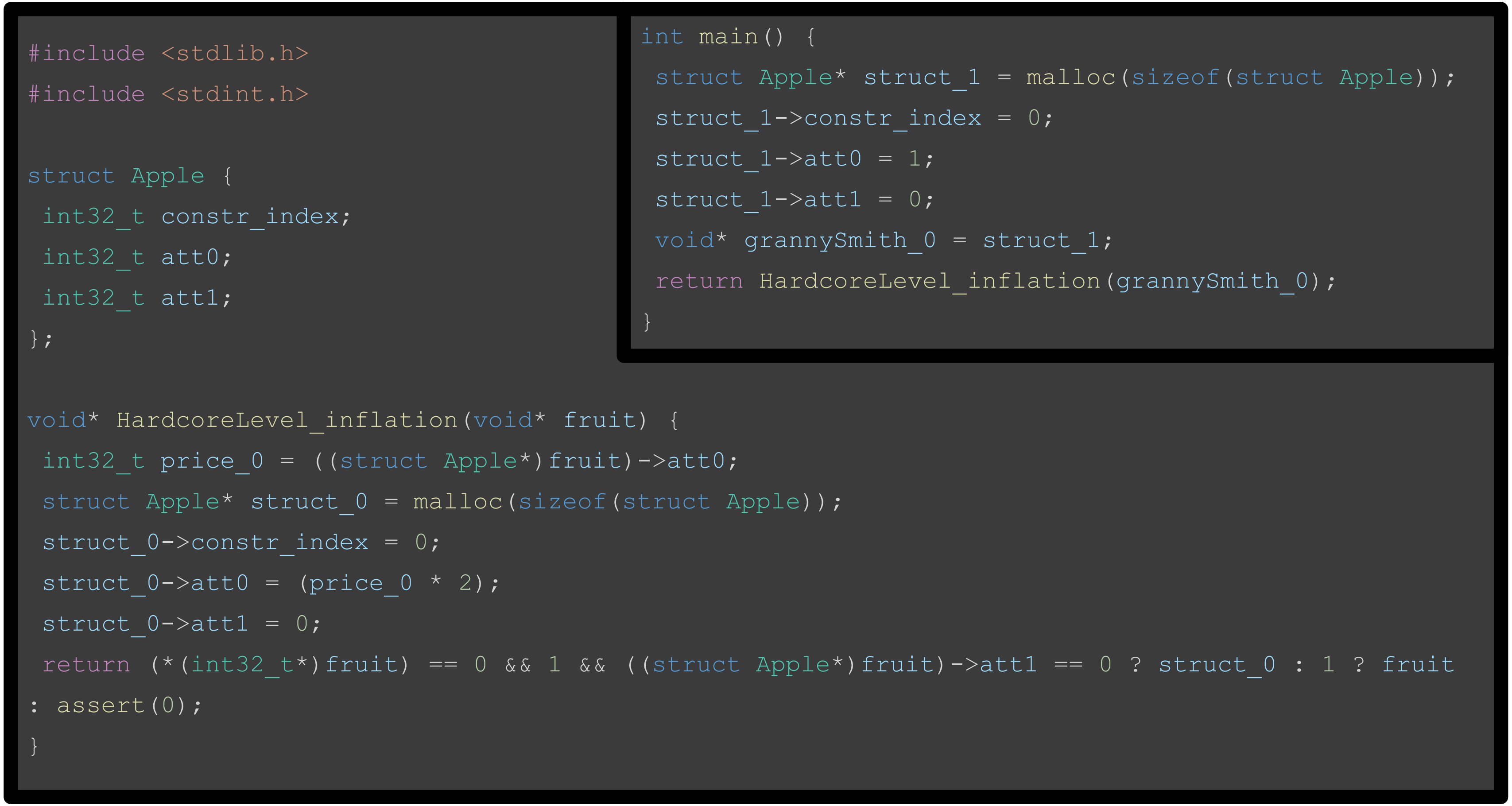
```c
char* concat3(char* s1, char* s2, char* s3) {
 char* string_1 = malloc(strlen(s1) + strlen(s2) + 1);
 strcpy(string_1, s1);
 strcat(string_1, s2);
 char* string_0 = malloc(strlen(string_1) + strlen(s3) + 1);
 strcpy(string_0, string_1);
 strcat(string_0, s3);
 return string_0;
}
int main() {
 char* string_2 = malloc(strlen("Complang") + 1);
 strcpy(string_2, "Complang");
 char* course_0 = string_2;
 char* string_3 = malloc(strlen("I ") + 1);
 strcpy(string_3, "I ");
 char* string_4 = malloc(strlen("love ") + 1);
 strcpy(string_4, "love ");
 return concat3(string_3, string_4, course_0);
}
```

Example : Hardcore Level (ADT / Pattern Matching)



```
object HardcoreLevel

abstract class Fruit
case class Apple(price: Int(32), isBio: Boolean) extends Fruit

fn inflation(fruit : Fruit): Fruit = {
   fruit match {
       case Apple(price, false) => Apple(price * 2, false)
       case other => other
   }
}

val grannySmith: Fruit = Apple(1, false);
inflation(grannySmith)

end HardcoreLevel
```

Example : Hardcore Level (ADT / Pattern Matching)



```
#include <stdlib.h>
#include <stdint.h>

struct Apple {
 int32_t constr_index;
 int32_t att0;
 int32_t att1;
};

void* HardcoreLevel_inflation(void* fruit) {
 int32_t price_0 = ((struct Apple*)fruit)->att0;
 struct Apple* struct_0 = malloc(sizeof(struct Apple));
 struct_0->constr_index = 0;
 struct_0->att0 = (price_0 * 2);
 struct_0->att1 = 0;
 return (*(int32_t*)fruit) == 0 && 1 && ((struct Apple*)fruit)->att1 == 0 ? struct_0 : 1 ? fruit
: assert(0);
}
```

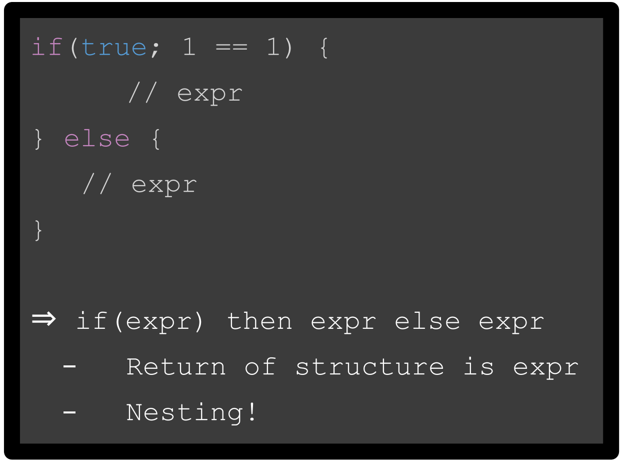
```
int main() {
 struct Apple* struct_1 = malloc(sizeof(struct Apple));
 struct_1->constr_index = 0;
 struct_1->att0 = 1;
 struct_1->att1 = 0;
 void* grannySmith_0 = struct_1;
 return HardcoreLevel_inflation(grannySmith_0);
}
```

Theoretical Background for C : (statements VS expr)

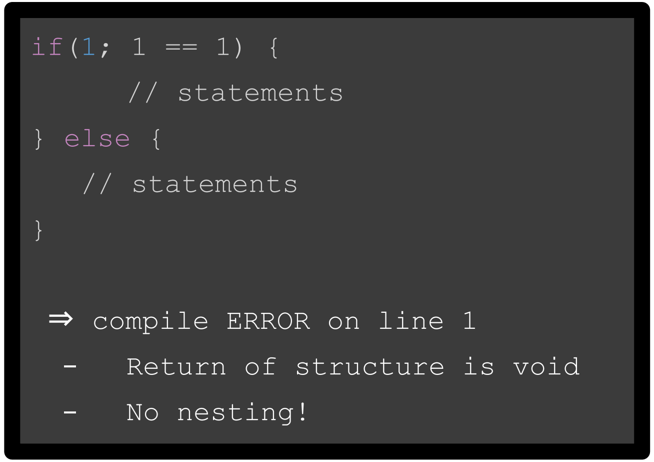
Amy -> High level (granularity = expressions) - FUNCTIONAL

C -> Low level (granularity = statements) - IMPERATIVE

Example:



```
if(true; 1 == 1) {
      // expr
} else {
   // expr
}

⇒ if(expr) then expr else expr
  -   Return of structure is expr
  -   Nesting!
```



```
if(1; 1 == 1) {
      // statements
} else {
   // statements
}

 ⇒ compile ERROR on line 1
  -   Return of structure is void
  -   No nesting!
```

Theoretical Background for C : (statements VS expr)

Solution ?

⇒ Make C functional!

How ?

⇒ Identify functional aspects in C!

What aspects ?

⇒ Code that returns pure *values*

Theoretical Background for C : (statements VS expr)

- **Code that returns pure values**
  - IntLiteral/Ptr
  - Variable
  - UniOp (!, -)
  - BinOp (+, -, *, /, %, ==)
  - Ternary Op (_?_:_)
- **Code that *does not* return a value**
  - Declaration/Initialization → String literals/Variables/Structs
  - Memory allocation → Malloc()
  - String concat → strcat()
  - etc.

Theoretical Background for C : (statements VS expr)

Only pure expressions?

⇒ NO! These pure expressions cannot be nested and create new variables.

Example:

if(val x: Int(32) = 5; x == 5) → Extract x declaration/initialization

Theoretical Background for C : (statements VS expr)

Solution?

⇒ Generate side effect code above for the bookkeeping!

Example:

int x = 5; (x == 5) ? _ : _

⇒ THIS IS THE GENERAL IDEA

Changes and Additions in Amyc

No real change in the Amy Compiler.

Add a new backend analogous to the webassembly backend:

- **C :**

➢ **CFile**	Class representing a file in C
➢ **CFilePrinter**	Responsible for printing the file
➢ **Function**	Class representing a Function in C
➢ **Struct**	Class representing a Structure in C
➢ **Token**	CToken

Changes and Additions in Amyc

- **cGen:**

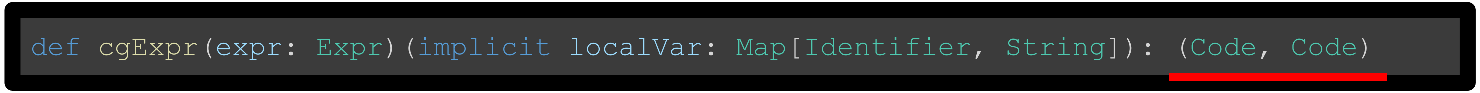
CodeGen	Generate C code from Amy program
CodePrinter	Create the C file
Utils	Util function used in CodeGen

→ Reuse of the wasm architecture, change final pipeline step.

→ <u>BUT:</u> We now generate source code ⇒ Back to tokens!

Changes and Additions in Amyc

Recall our previous point about pure/impure code?

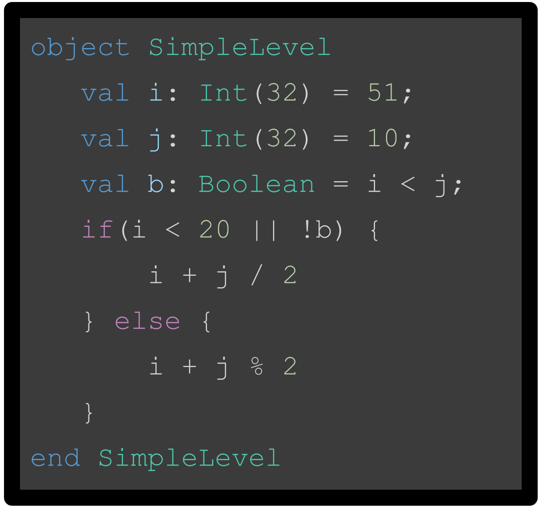


```
def cgExpr(expr: Expr)(implicit localVar: Map[Identifier, String]): (Code, Code)
```

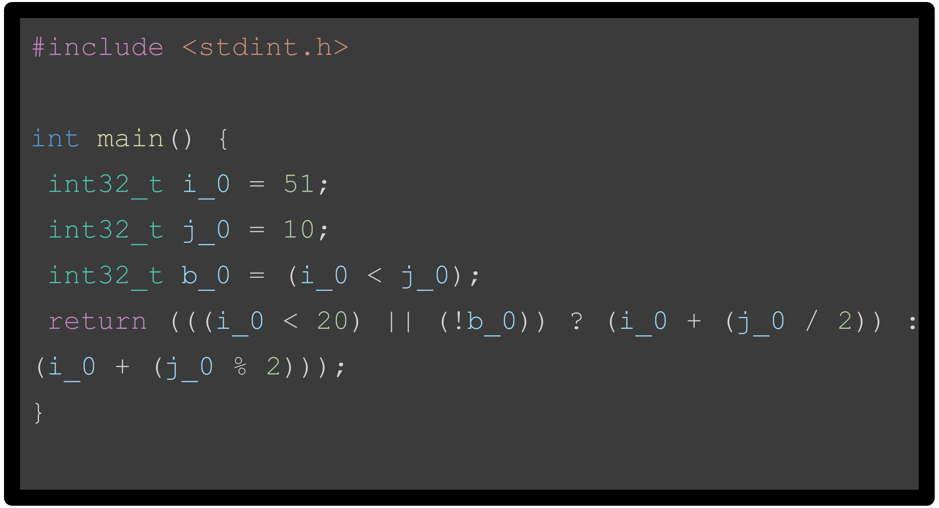
→ **Transforms any expression into a sequence of CTokens.**

(Code, Code) := (Pure value, Pre-requisite code)

Back to examples

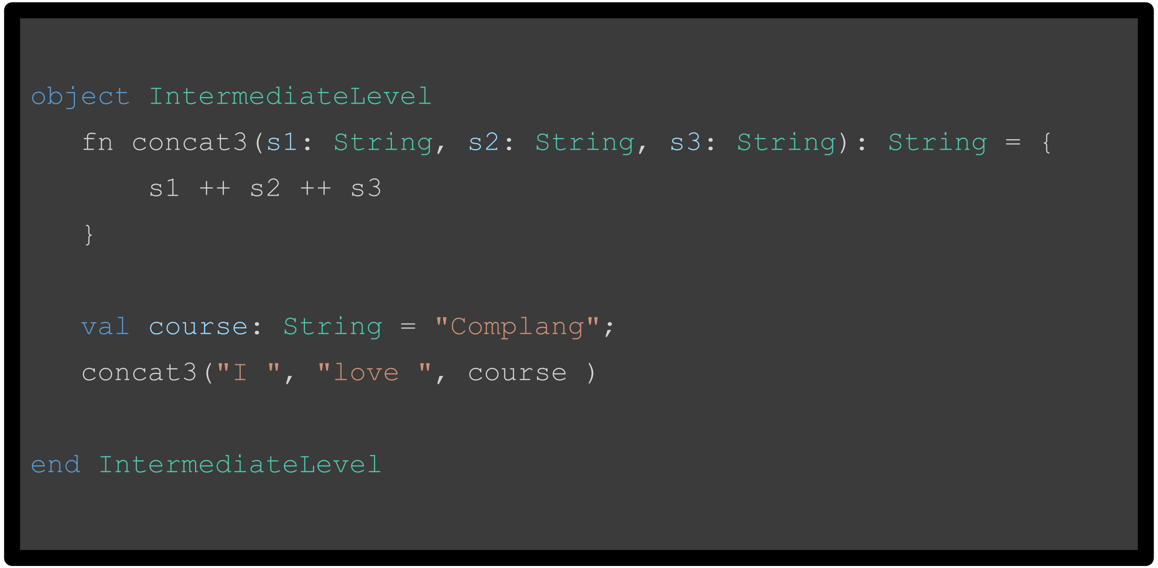


```
object SimpleLevel
    val i: Int(32) = 51;
    val j: Int(32) = 10;
    val b: Boolean = i < j;
    if(i < 20 || !b) {
        i + j / 2
    } else {
        i + j % 2
    }
end SimpleLevel
```



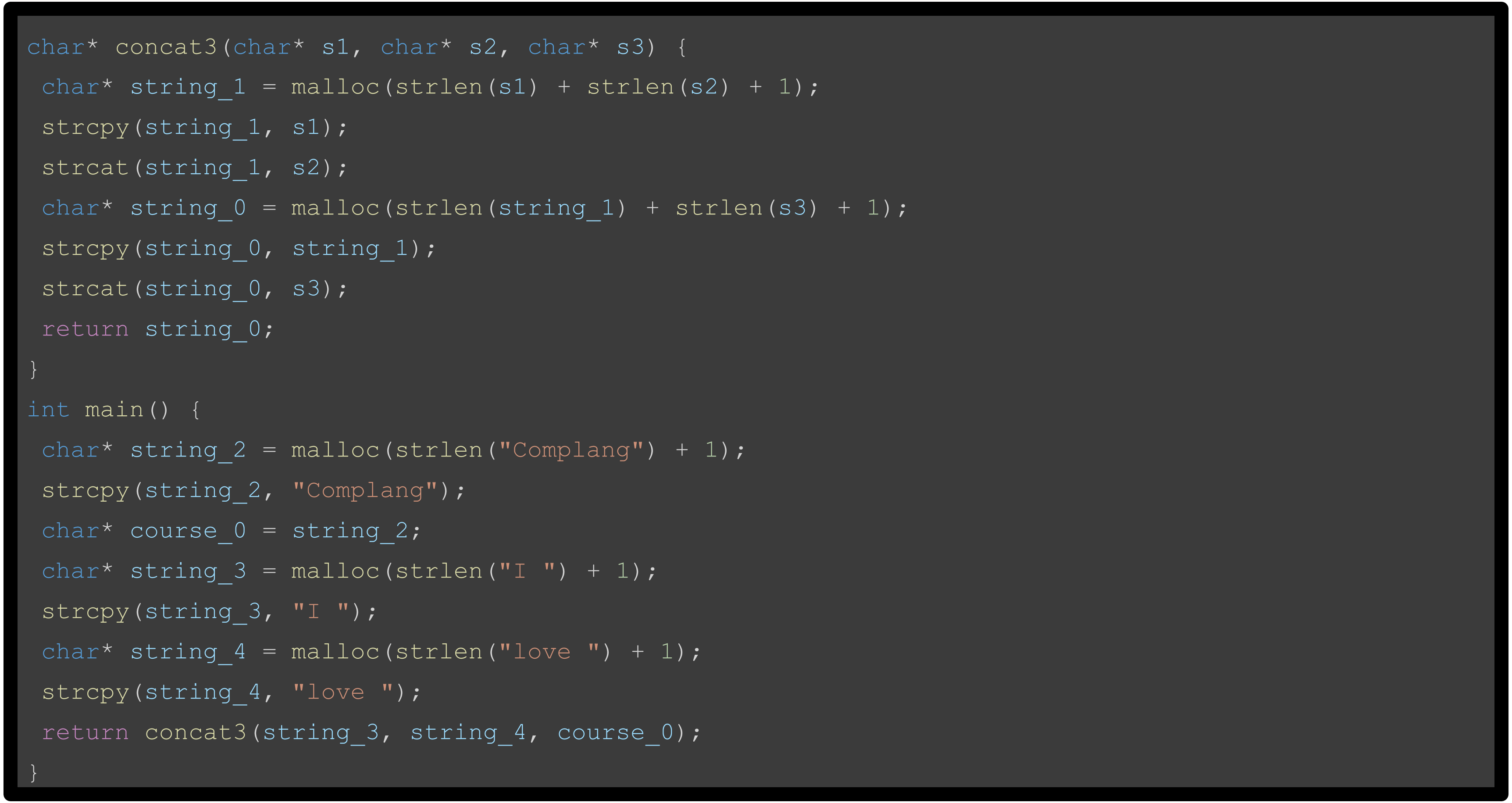
```
#include <stdint.h>

int main() {
 int32_t i_0 = 51;
 int32_t j_0 = 10;
 int32_t b_0 = (i_0 < j_0);
 return (((i_0 < 20) || (!b_0)) ? (i_0 + (j_0 / 2)) :
(i_0 + (j_0 % 2)));
}
```

Back to examples



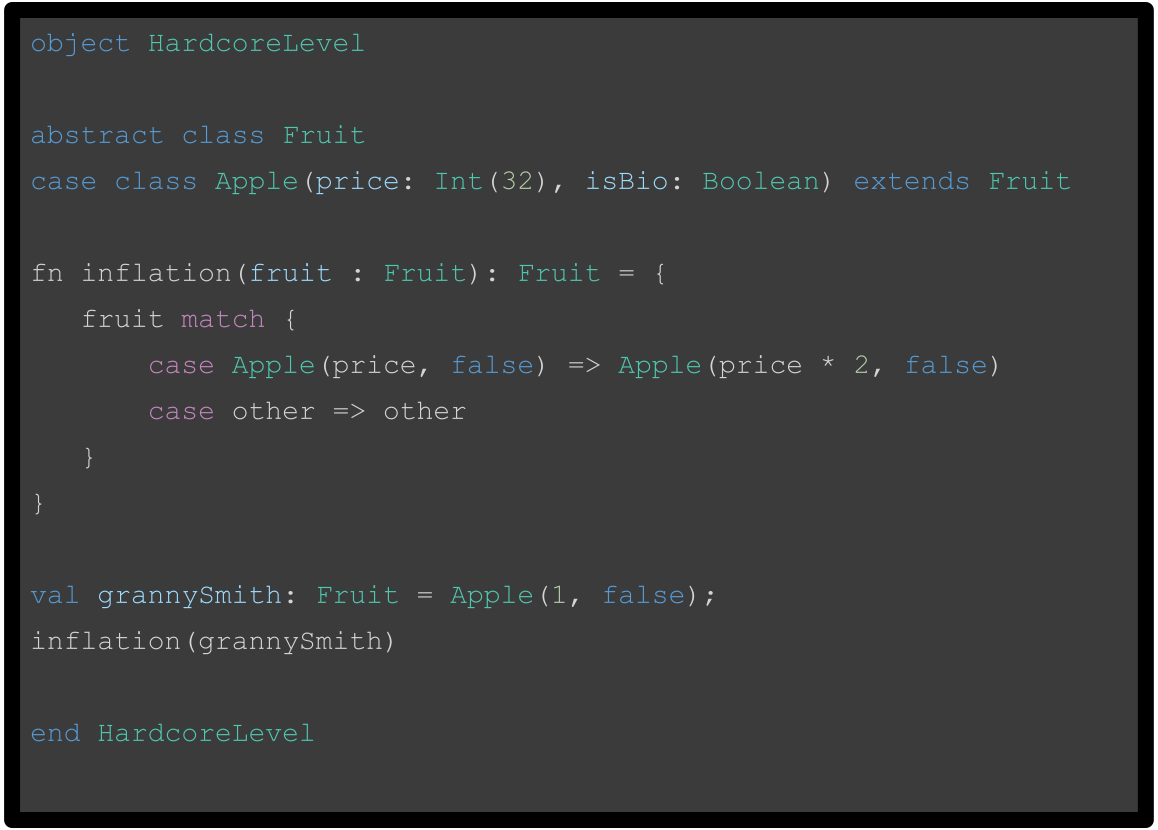
```
object IntermediateLevel
   fn concat3(s1: String, s2: String, s3: String): String = {
       s1 ++ s2 ++ s3
   }

   val course: String = "Complang";
   concat3("I ", "love ", course )

end IntermediateLevel
```

Back to examples

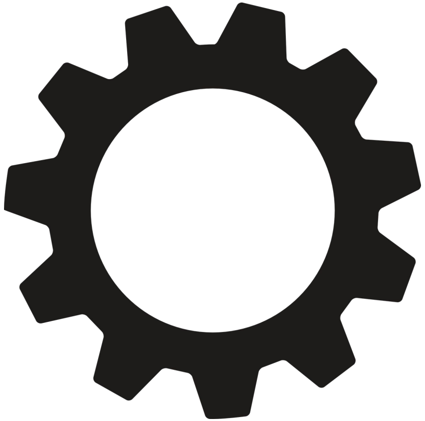


```c
char* concat3(char* s1, char* s2, char* s3) {
 char* string_1 = malloc(strlen(s1) + strlen(s2) + 1);
 strcpy(string_1, s1);
 strcat(string_1, s2);
 char* string_0 = malloc(strlen(string_1) + strlen(s3) + 1);
 strcpy(string_0, string_1);
 strcat(string_0, s3);
 return string_0;
}
int main() {
 char* string_2 = malloc(strlen("Complang") + 1);
 strcpy(string_2, "Complang");
 char* course_0 = string_2;
 char* string_3 = malloc(strlen("I ") + 1);
 strcpy(string_3, "I ");
 char* string_4 = malloc(strlen("love ") + 1);
 strcpy(string_4, "love ");
 return concat3(string_3, string_4, course_0);
}
```

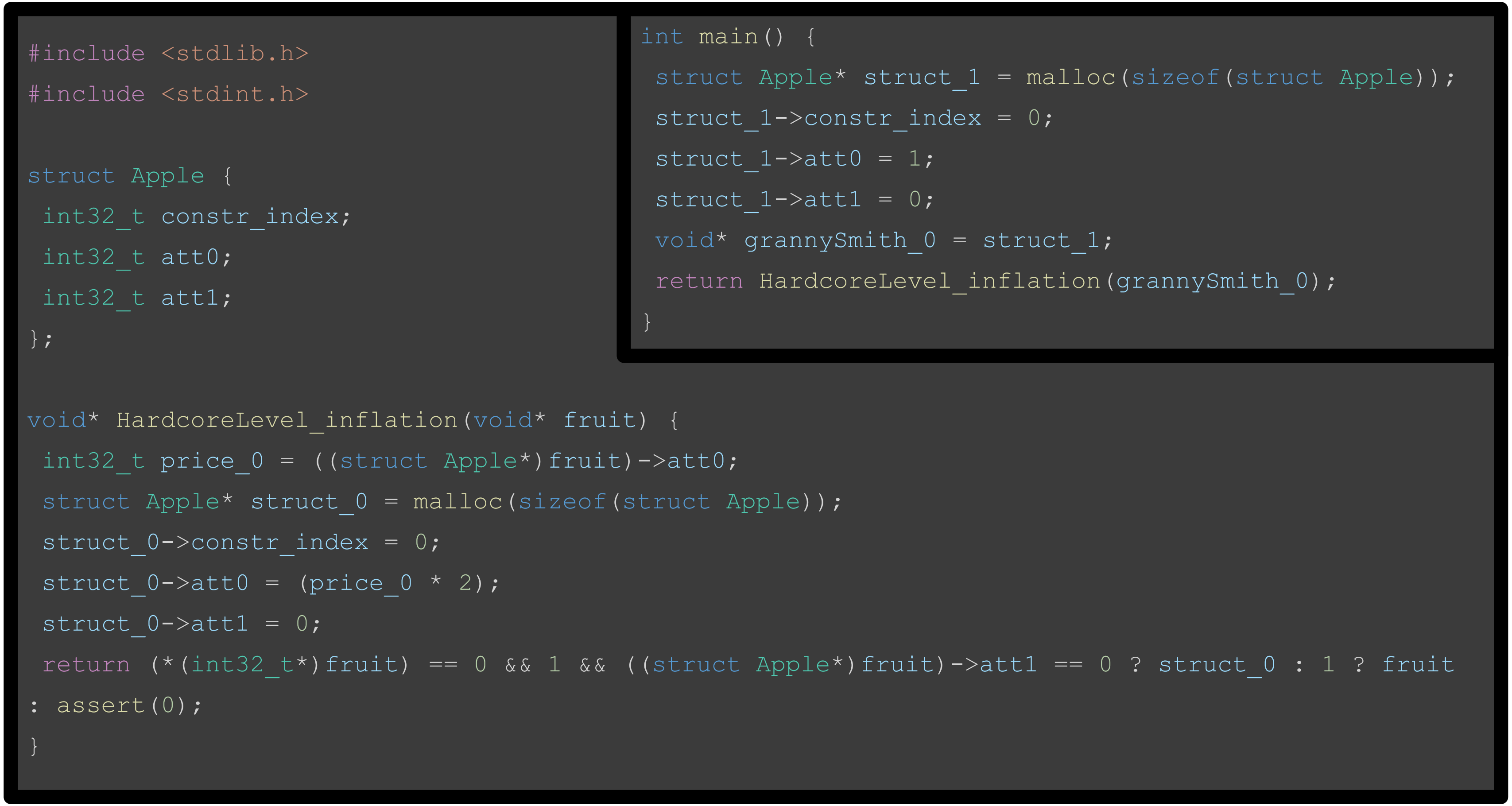
Back to examples



```
object HardcoreLevel

abstract class Fruit
case class Apple(price: Int(32), isBio: Boolean) extends Fruit

fn inflation(fruit : Fruit): Fruit = {
   fruit match {
       case Apple(price, false) => Apple(price * 2, false)
       case other => other
   }
}

val grannySmith: Fruit = Apple(1, false);
inflation(grannySmith)

end HardcoreLevel
```



Back to examples



```c
#include <stdlib.h>
#include <stdint.h>

struct Apple {
 int32_t constr_index;
 int32_t att0;
 int32_t att1;
};
```

```c
int main() {
 struct Apple* struct_1 = malloc(sizeof(struct Apple));
 struct_1->constr_index = 0;
 struct_1->att0 = 1;
 struct_1->att1 = 0;
 void* grannySmith_0 = struct_1;
 return HardcoreLevel_inflation(grannySmith_0);
}
```

```c
void* HardcoreLevel_inflation(void* fruit) {
 int32_t price_0 = ((struct Apple*)fruit)->att0;
 struct Apple* struct_0 = malloc(sizeof(struct Apple));
 struct_0->constr_index = 0;
 struct_0->att0 = (price_0 * 2);
 struct_0->att1 = 0;
 return (*(int32_t*)fruit) == 0 && 1 && ((struct Apple*)fruit)->att1 == 0 ? struct_0 : 1 ? fruit
: assert(0);
}
```

Going further?

- **Better C variable name generation**

  no conflict between amy function and C function.

  (strcat(), strlen(), sizeof() , malloc(), ...)

- **Multiple file program and imports**
- **Library implementation** (input/output, integrated functions)

—-----------------------------*Our expected work*--------------------------------------

- **Garbage collecting** (lot of useless preemptive malloc)

  → Maybe adapt Amyc first

Your questions?