# Compilation Fourth Step: Intermediate Representation (IR)

Flattening the AST to a sequence of instructions

December 17, 2018

# IR properties

- **Independent** of the source language

| | | |
|---|---|---|
| clang(C/CPP) | $\rightarrow$ | |
| flang(Fortran) | $\rightarrow$ | |
| ghc(Haskell) | $\rightarrow$ | LLVM IR |
| llgo(Go) | $\rightarrow$ | |
| ... | $\rightarrow$ | |

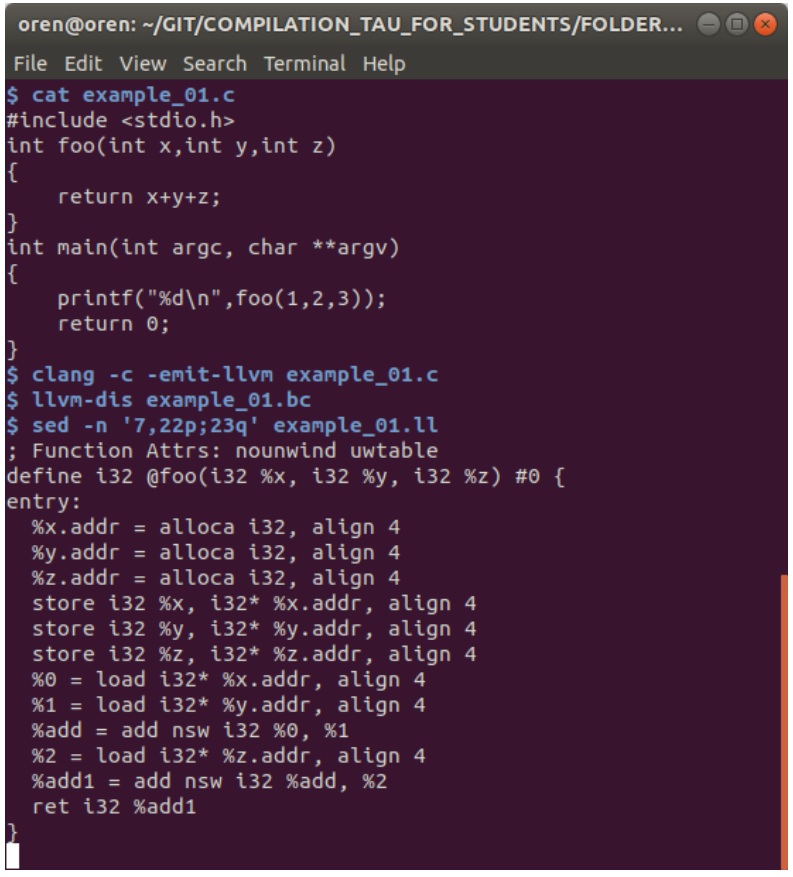
- **Independent** of the target language

| | | |
|---|---|---|
| | $\rightarrow$ | x86 |
| | $\rightarrow$ | ARM |
| LLVM IR | $\rightarrow$ | WebAssembly |
| | $\rightarrow$ | Mips |
| | $\rightarrow$ | ... |

- Contains the **entire** information needed for final translation

# IR of Industrial Compilers

- LLVM IR



# IR of Industrial Compilers

- GCC's IR (GIMPLE)

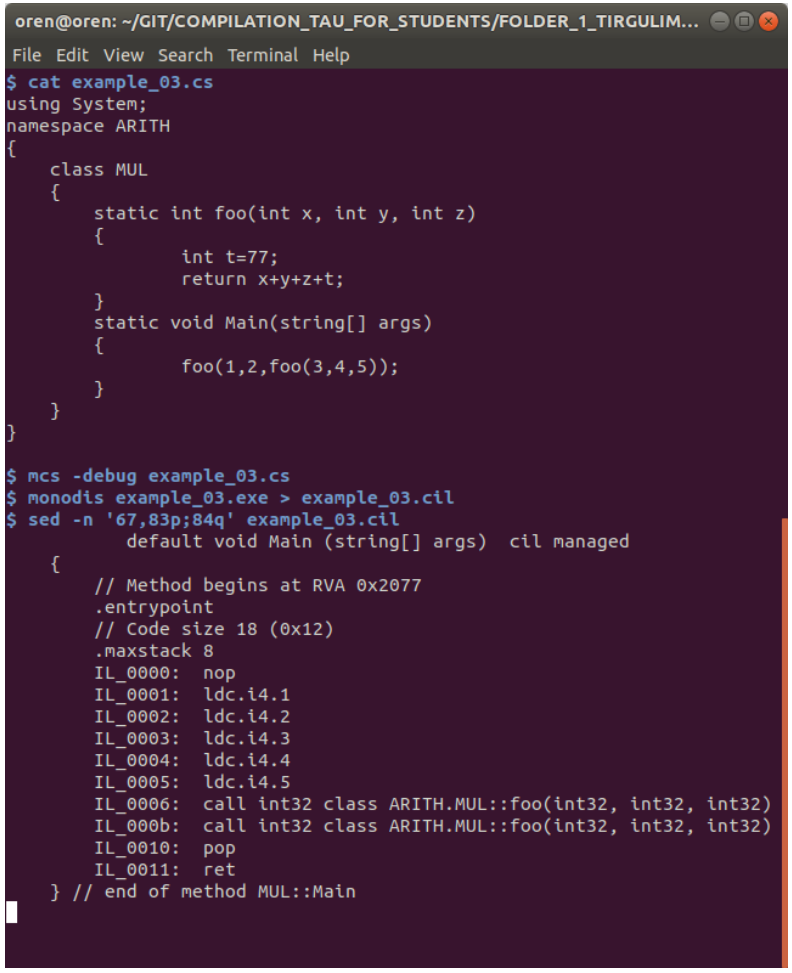
```
oren@oren: ~/GIT/COMPILATION_TAU_FOR_STUDENTS/FOLDER...
File  Edit  View  Search  Terminal  Help
$ cat example_02.c
#include <stdio.h>
int foo(int n)
{
    int sum=17;
    for (int i=0;i<n;i++)
    {
        sum=sum+i+46;
    }
    return sum;
}
int main(int argc, char **argv) {
    return printf("%d\n",foo(argc));
}
$ rm *.gimple && gcc -O0 -c -fdump-tree-all example_02.c
$ cp *.gimple example_02.gimple && rm example_02.c.*
$ sed -n '1,22p;23q' example_02.gimple
foo (int n)
{
  int D.2261;
  int sum;

  sum = 17;
  {
    int i;

    i = 0;
    goto <D.2254>;
    <D.2253>:
    _1 = sum + i;
    sum = _1 + 46;
    i = i + 1;
    <D.2254>:
    if (i < n) goto <D.2253>; else goto <D.2255>;
    <D.2255>:
  }
  D.2261 = sum;
  return D.2261;
}
```

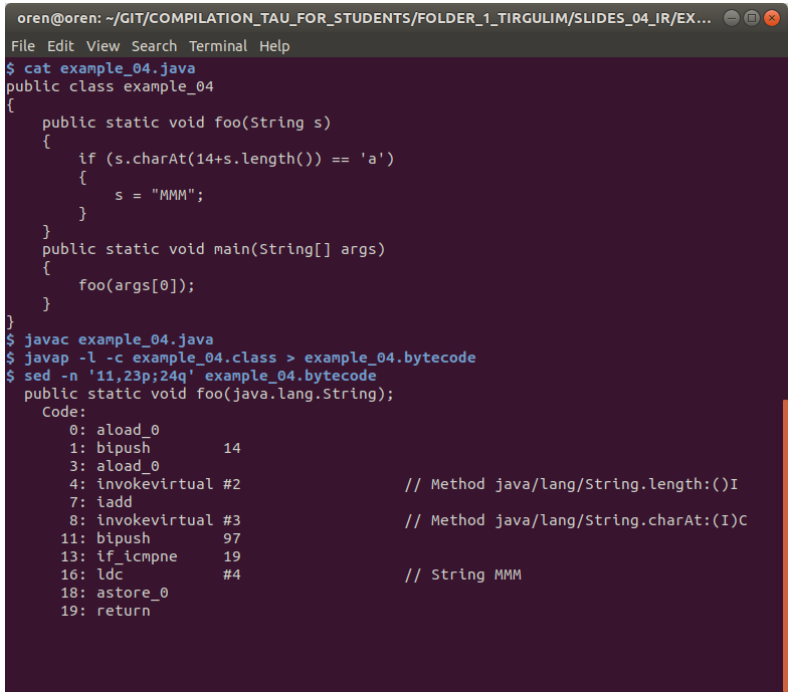
# IR of Industrial Compilers

- (MONO) C# CIL



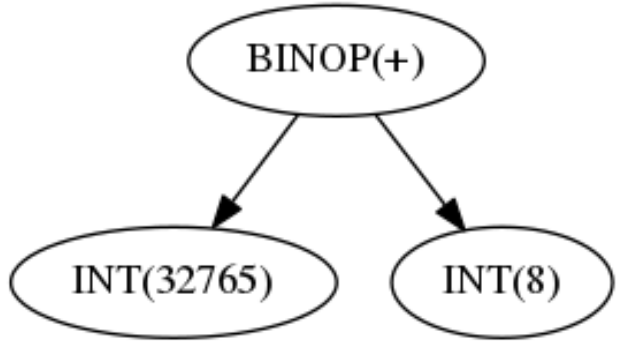
# IR of Industrial Compilers

- Java Bytecode



# IR Introductory Example: **32765+8**

- IR is produced by scanning the AST recursively as follows:
  - First, the left subtree (a leaf actually) is scanned, producing the IR command: **li Temp_29, 32765**.
  - Then, the right subtree (a leaf too) is scanned, producing the IR command: **li Temp_30, 8**.
  - Finally, the binop father node uses the temporaries returned from its operand sons to produce the IR command: **add Temp_31, Temp_29, Temp_30**.



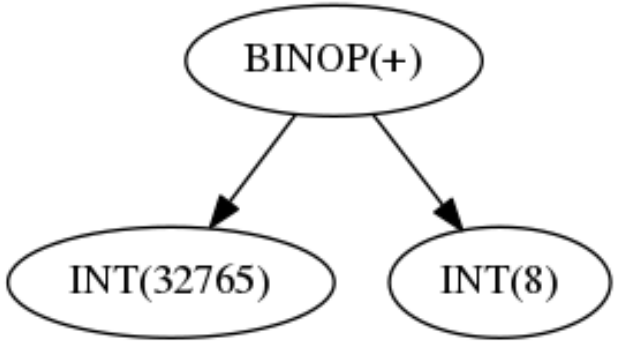

- Note that the IR recursive scan of the AST resembles the scan performed by the semantic analyzer. However here expression subtrees return their temporary, not their type.

# IR Example: **if (2<6) { PrintInt(3); }**

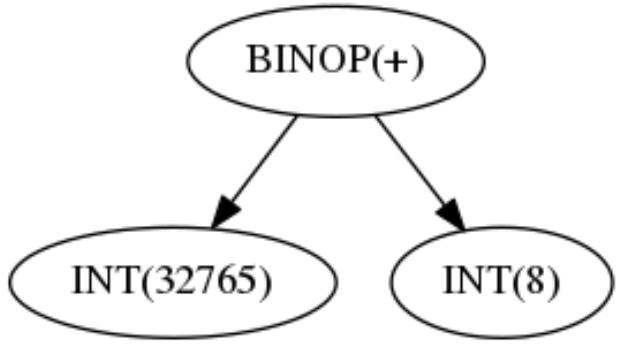
- IR is produced by scanning the AST recursively as follows:
  - First, the condition subtree is scanned, producing the IR commands:
    - **li Temp_74, 2**
    - **li Temp_75, 6**
    - **li Temp_76, 1**
    - **blt Temp_74, Temp_75, label_cond_end**
    - **li Temp_76, 0**
    - **label_cond_end**
  - Then, the if-father-node uses the temporary returned from its condition-son and wraps the IR commands produced by its body-son as follows:
    - **beq Temp_76, 0, label_if_end**
    - **li Temp_77, 3**
    - **call PrintInt( Temp_77 )**
    - **label label_if_end**

# IR in our project

- Designing a good IR is more art than science.
- Specially true in our project where there's only one source language (Poseidon) and one target language (MIPS).
- In fact, do we even *need* an IR in our project? **Why not translate directly AST $\rightarrow$ MIPS**?
- For example, how should we *really* translate *32765+8*? (remember that addition is done with 16 bits overflow).
  - Should we handle overflow in AST $\rightarrow$ IR phase?
  - Or should we handle it in the IR $\rightarrow$ MIPS phase?



# IR in our project :: Overflow handled in IR $\rightarrow$ MIPS




- Handling arithmetic overflow in the IR $\rightarrow$ MIPS phase will yield the following (simple) IR code for the addition above:
  - **li Temp_29, 32765**
  - **li Temp_30, 8**
  - **add Temp_31, Temp_29, Temp_30**
- What are the benefits of a simpler IR? How will the add instruction be translated to MIPS eventually?

# IR in our project :: Overflow handled in AST $\rightarrow$ IR

- Handling arithmetic overflow in the AST $\rightarrow$ IR phase will yield the following IR code for the addition above:
  - **li Temp_29, 32765**
  - **li Temp_30, 8**
  - **add Temp_31, Temp_29, Temp_30**
  - **li Temp_32, 32767**
  - **li Temp_33, -32768**
  - **bgt Temp_31, Temp_32, label_overflow**
  - **blt Temp_31, Temp_33, label_underflow**
  - # What should we write here?
  - **label_overflow:**
  - # and here?
  - **label_underflow:**
  - # and here too?
  - **label_end:**

# IR in our project :: Next Steps


- How to handle local variables? function input parameters? class data members?
- How to handle calls to global functions? calls to class methods? calls to library functions (like PrintInt)?