COMSM1302
Overview of Computer Architecture

Lecture 17

Compilers - 2

In the previous lecture

File

Syntactic

Semantic

Lexer — Sequence of tokens

Parser — Syntax tree

Translator — AST

- Symbol table
- Scoping

University of BRISTOL

🔥 In this lecture

Translator ➤ Optimiser ➤ Code generator
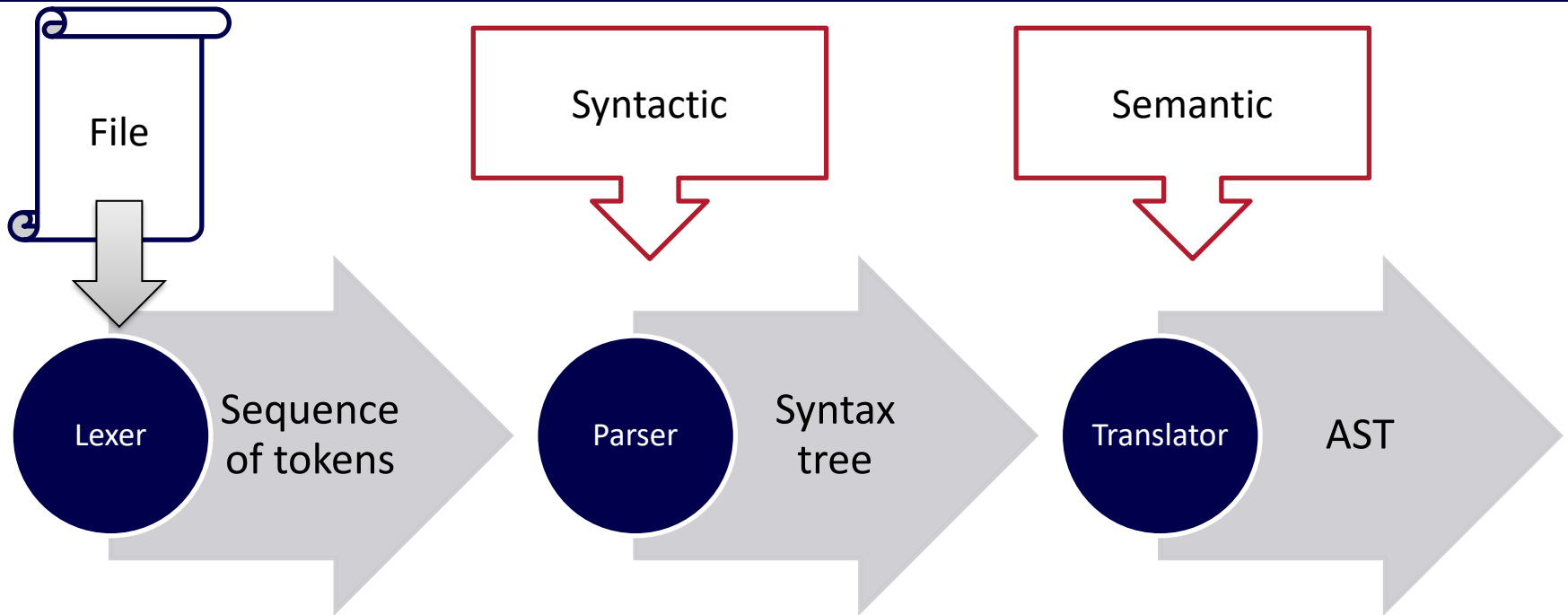
- At the end of this lecture:
  - Learn how compilers can handle different programing languages and different target platforms.
  - How you can use compliers to generate optimise assembly codes.

🔥 Compiler phases

Lexer

Parser

Translator

Optimiser

Code generator

🔥 Typing

- In C, x++ means:
  - if x is an integer: add 1 to x
  - if x is a struct … : an error

- x+y needs different instructions for char/int/long etc.

- So, for each active variable we need to track its **type** in the symbol table.

🔥 Different types of typing

- **Static typing**: variable types are established at compile time.

- **Dynamic typing**: the type of a variable can change (and needs to be checked) at runtime.

🔥 Dynamic typing – Python example

```python
x = 2
print(type(x))

x = 'Hello'
print(type(x))
```

University of BRISTOL

🔥 Asymmetric

- In math:
  - $a = b + 1 \Leftrightarrow b + 1 = a$

- In programming:
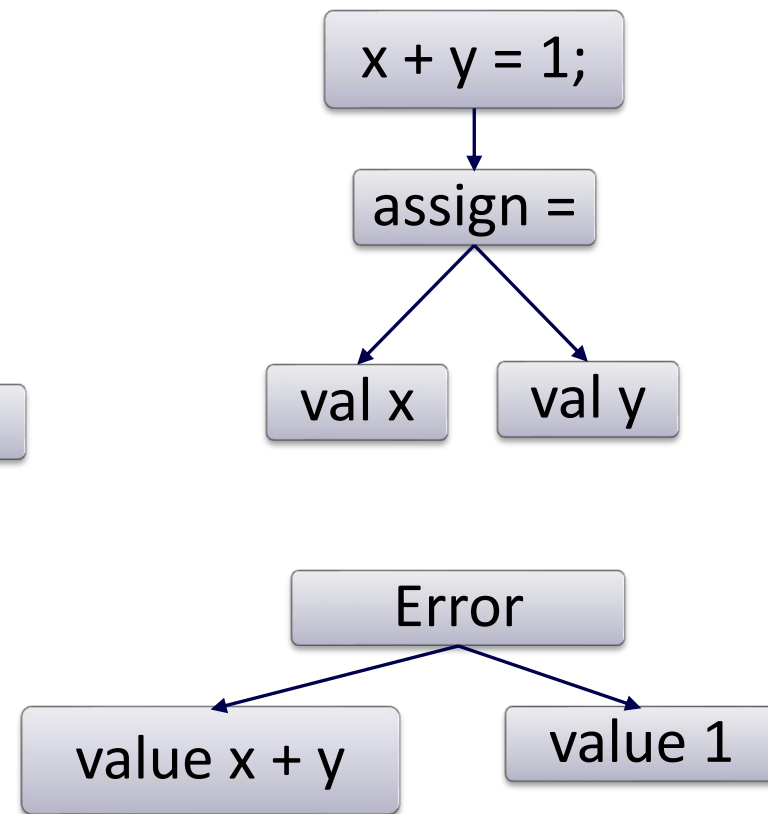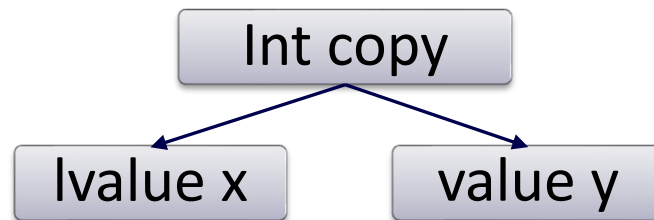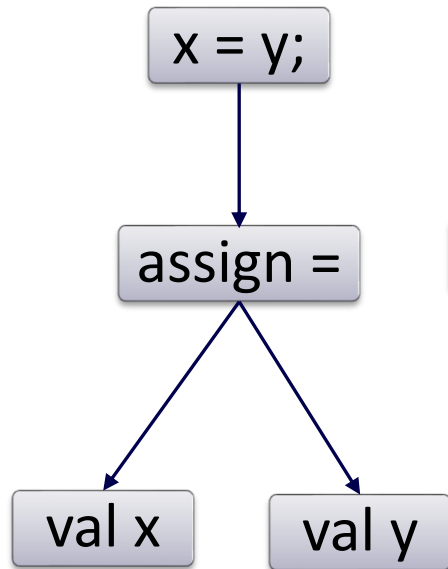 - $a = b + 1; \nLeftrightarrow b + 1 = a;$

🔥 Lvalues

- L-values ("left values") are "things that you can assign to". During translation, we can catch illegal assignments.

L-values in C:	Not L-values in C:
x	2
x = y	x+y
x+	x++

Translator

- Translator: turn one syntax tree into another.

x = y;

assign =

val x   val y

Int copy

lvalue x   value y

x + y = 1;

assign =

val x   val y

Error

value x + y   value 1

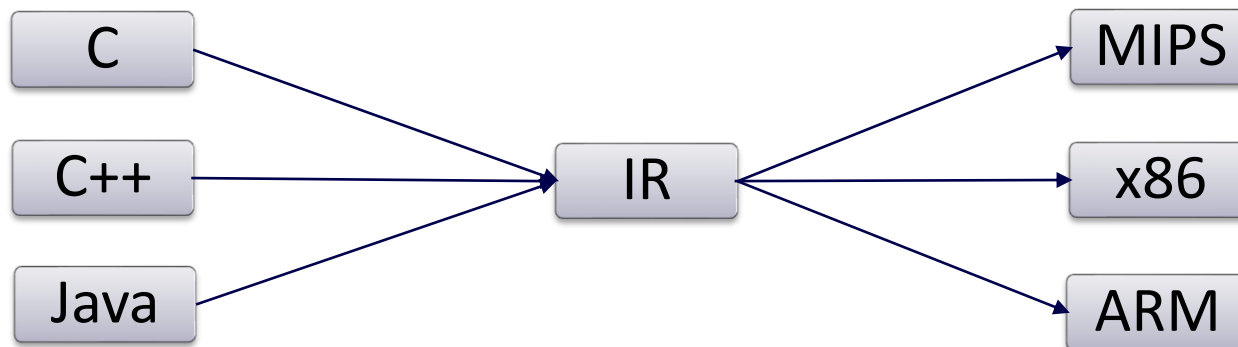University of BRISTOL

🔥 Translator – error

```
int main(void){
 int x;
 int y;
 x + y = 1;
return 0;
}
```

file.c:5:8: error: lvalue required as left operand of assignment
 x + y = 1;

🔥 Intermediate representation

- Compilers use language- and platform-independent IR in which programs are abstract syntax trees.

🔥 Intermediate representation

- An IR typically targets a virtual machine with
  - an unlimited number of registers
  - an unlimited amount of memory
  - a very rich (semantic) instruction set

Translator

- Transform syntax tree
- Create Symbol Table
  - Deal with scopes
  - Deal with types
- Normally outputs an intermediate hardware independent representation
  - Abstract Syntax Tree (AST)

University of BRISTOL

🔥 Compiler phases

Lexer

Parser

Translator

Optimiser

Code generator

🔥 Optimiser

- Input: AST

- Output: optimised AST
  - eliminate dead code

  - eliminate repeated register assignments

  - many passes (gcc has over 150)

  - The optimiser can do both general and processor-specific optimisations.

🔥 Optimisation levels

- **-O0** Most optimisations are disabled. The aim of the compiler is to reduce the compilation cost.

- **-O1** The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

University of BRISTOL

🔥 GCC optimisation options

- https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

🔥 Optimisation – example 1

```c
int main(void){
    int x;
    int y;
    x = 3;
    y = 4;
    x += 0;
    y += x;
return 0;
}
```
arm-none-eabi-gcc -g -O0 -c optimise-1.c
arm-none-eabi-objdump -S optimise-1.o > optimise-1.s

University of BRISTOL

```
int main(void){   str fp, [sp, #-4]!        return 0;    mov     r3, #0
                  add fp, sp, #0           }
                  sub sp, sp, #12                       mov     r0, r3
int x; int y; x = 3;                                    add sp, fp, #0
                  mov     r3, #3                          ldr fp, [sp], #4
                  str  r3, [fp, #-8]                     bx   lr
y = 4;            mov     r3, #4
                  str  r3, [fp, #-12]

x += 0; y += x;

                  ldr  r2, [fp, #-12]
                  ldr  r3, [fp, #-8]
                  add r3, r2, r3
                  str  r3, [fp, #-12]
```

🔥 Example 1 -O1 optimisation

```
x = 3;
 y = 4;
 x += 0;
 y += x;
return 0;
}
        mov     r0, #0
        bx   lr
```

🔥 Optimisation – example 2

```c
int main(void){
    int x;
    int y;
    x = 3;
    y = 4;
    x += 0;
    y += x;
return y;
}
```

int main(void){   str fp, [sp,#-4]!
              add fp, sp, #0
              sub sp, sp, #12

int x; int y; x = 3;

              mov   r3, #3
              str r3, [fp, #-8]

y = 4;   mov   r3, #4
              str r3, [fp, #-12]

x += 0; y += x;

              ldr r2, [fp, #-12]
              ldr r3, [fp, #-8]
              add r3, r2, r3
              str r3, [fp, #-12]

return y;   ldr r3, [fp, #-12]
}

              mov   r0, r3
              add sp, fp, #0
              ldr fp, [sp], #4
              bx   lr

```
x = 3;
 y = 4;
 x += 0;
 y += x;
return y;
}
                mov     r0, #7
                bx   lr
```

🔥 Optimisation example 3

```
int main(int x, int y){
    x += 0;
    y += x;
return y;
}
```

Example 3 –O1 optimisation

```
int main(int x, int y){
 x += 0;
 y += x;
return y;
}
```

```
        add r0, r0, r1
        bx   lr
```

🔥 Compiler phases

Lexer

Parser

Translator

Optimiser

Code generator

🔥 Code generator

- Input: optimised AST

- Output: machine code / executable file

- often in 2 phases:
  - AST → assembly
  - assembly → machine code

University of
BRISTOL

🔥 Cross compilers

- The compiler does not have to run on the same architecture as the architecture targeted by the given program.

- Cross-compiling is compiling on a different platform to the target one.

🔥 Summary