Compiler Design for Quantum Hardware

Alessandra Di Pierro

Università di Verona

Stellenbosh, 21 April 2023
HPC for Sustainable Development

Programming a Quantum Computer

Several laboratories in industry and academia around the world have produced quantum devices that

- operate on the circuit model of quantum computing
- are small, noisy, and not as powerful as current classical computers
- steadily growing, and promising large computational power for problems in chemistry, machine learning, optimization, finance

As we have learnt from the history of programming languages for classical computers, it is inecessary to accompany the new hardware with the appropriate software.
Although software in both domains shares many similarities, some key differences drive many of the challenges in developing quantum software tools.

The Early 1950s

1946 ENIAC, the first real multi-purpose computer was programmed by flipping switches and moving wires.

1949 Assembly, the first method of writing programs for these general-purpose machines. An assembly language simply associates a human-readable name to each of the computer-specific instructions.

1950's Birth of the *modern*, i.e. machine-independent, programming languages (FORTRAN, COBOL, and LISP and many others followed).

1952, The first compiler was written by Grace Hopper for the A-0 programming language.

1957 The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler. COBOL was an early language to be compiled on multiple architectures, in 1960.

Toolchain

Set of programming tools for complex software development[1]

- To counter the rise of proprietary software, in 1983 Richard Stallman at MIT announced the GNU project ("GNU's not Unix!").
- The GNU Project is a free-software, mass-collaboration project: users are free to run the software, share it (copy and distribute), study it, and modify it.
- Main components of a toolchain:
  - the assembler, that turns assembly code (generated by GCC) to binary;
  - the compiler; the only realistic solution today is GCC, the GNU Compiler Collection. Nowadays, as input, it not only supports C, but also C++, Java, Fortran, Objective-C and Ada. As output, it supports a very wide range of architectures.

[1] A software toolchain tpically also includes: Linker - merges multiple files into a single program Library - a collection of code, such as prebuilt functions or other resources Debugger - an optional tool that can help fix bugs

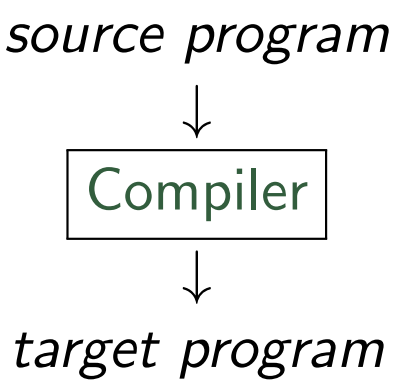
Compilers

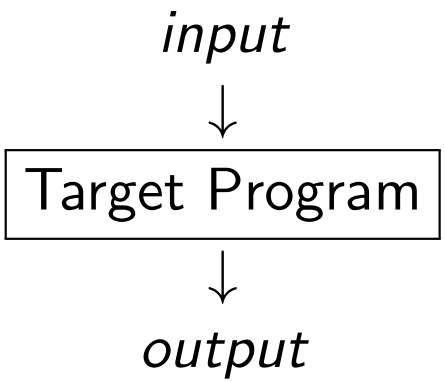
- A machine language consists of very simple commands.
- Writing a program in this language is a very tedious and error-prone.
- It is much easier to use instead high-level programming language.
- But, before a program can be run, it first must be translated by a compiler.

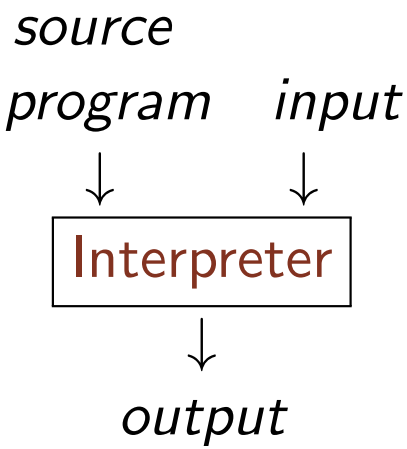
Advantages:

- notation is more intuitive and closer to human language
- compilers can spot some obvious programming mistakes
- programs are shorter than their equivalent written in machine language, and
- the same high-level program can be run on different machines, previous translation/compiling.

From Source to Machine Code

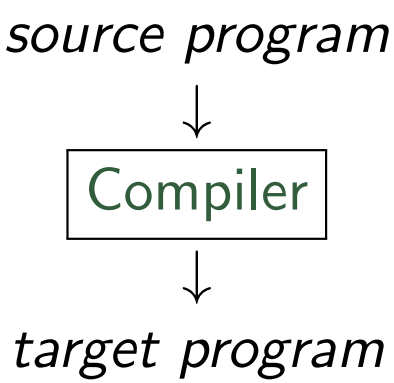


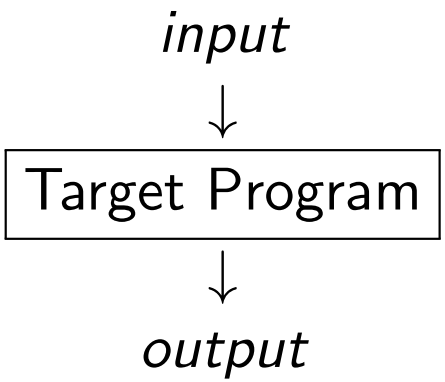


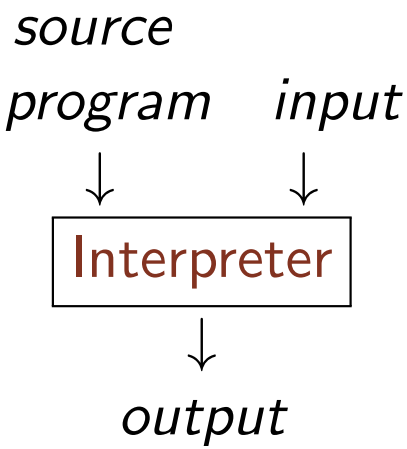


- Compilation results in a lower-level language program, e.g., machine code, which can be run on various inputs.
- Interpretation directly *executes* one by one the operations specified in the *source program* on the *input* supplied by the user, by using the facilities of its implementation language.

From Source to Machine Code

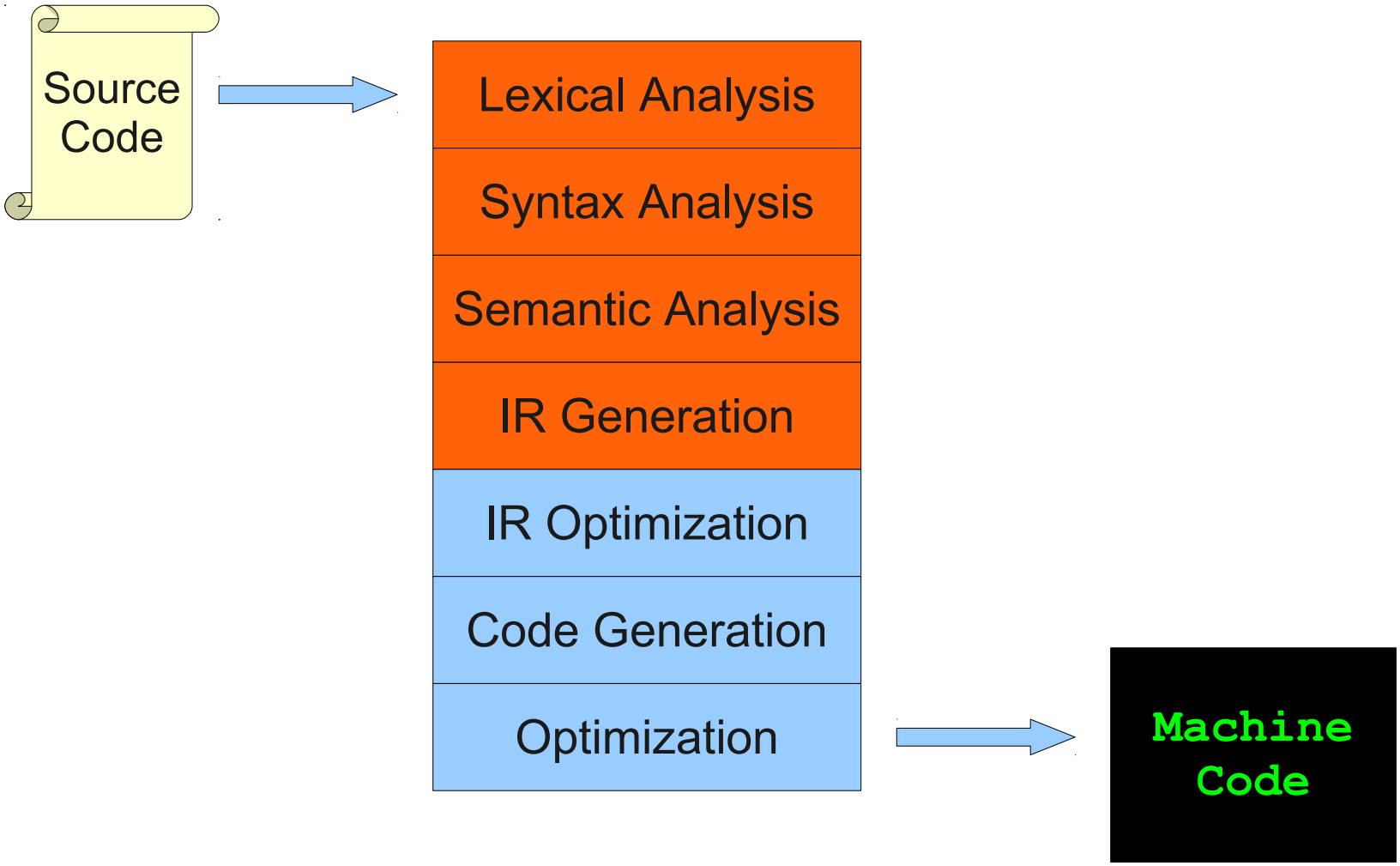




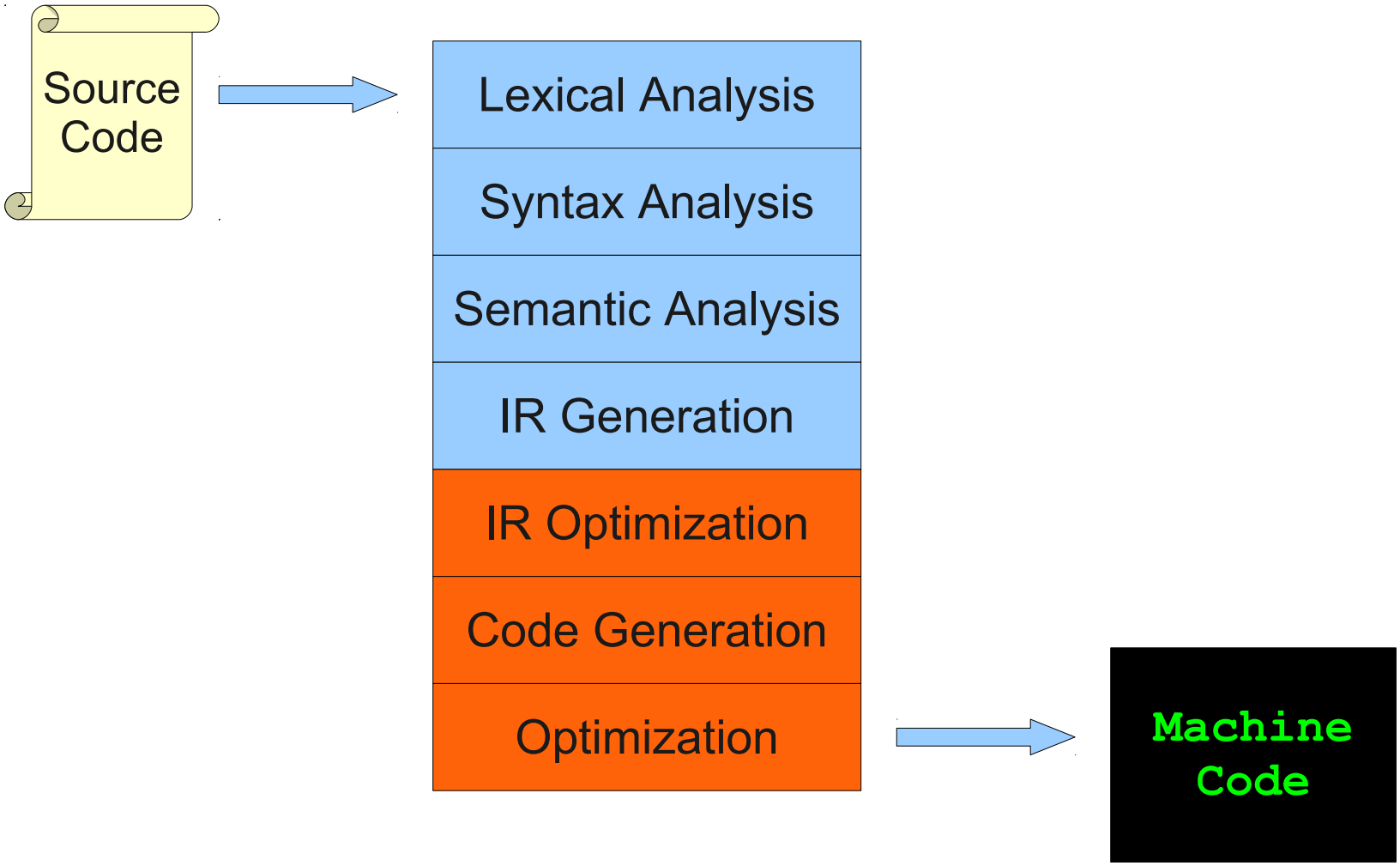


- Compilation results in a lower-level language program, e.g., machine code, which can be run on various inputs.
- Interpretation directly *executes* one by one the operations specified in the *source program* on the *input* supplied by the user, by using the facilities of its implementation language.

The Structure of a Modern Compiler

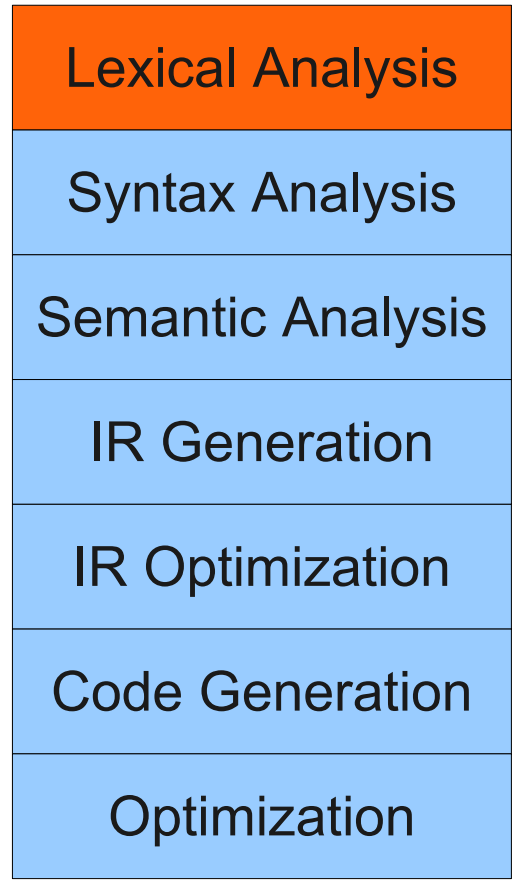


The Structure of a Modern Compiler



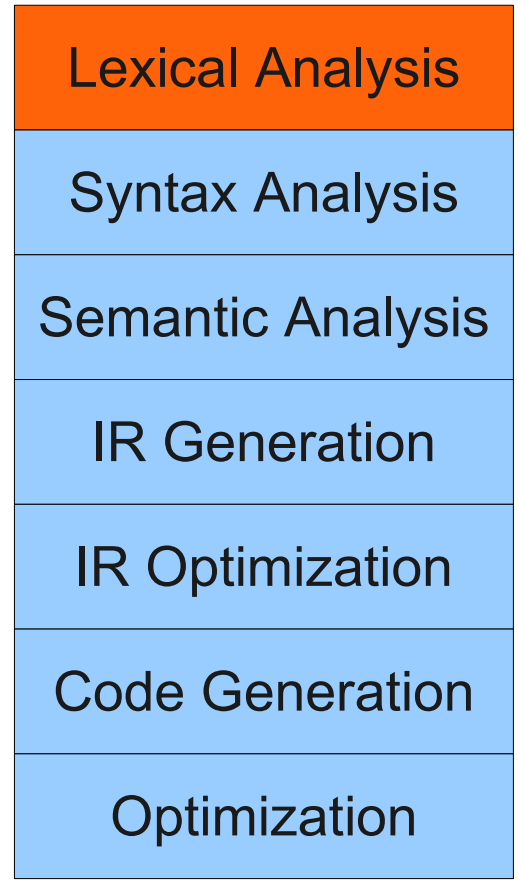
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
while (y < z) {
    int x = a + b;
    y += x;
}
```



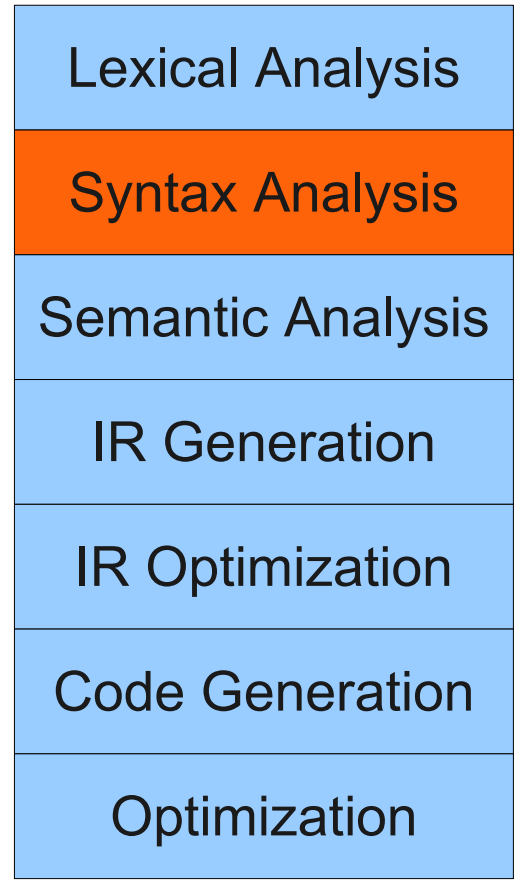
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

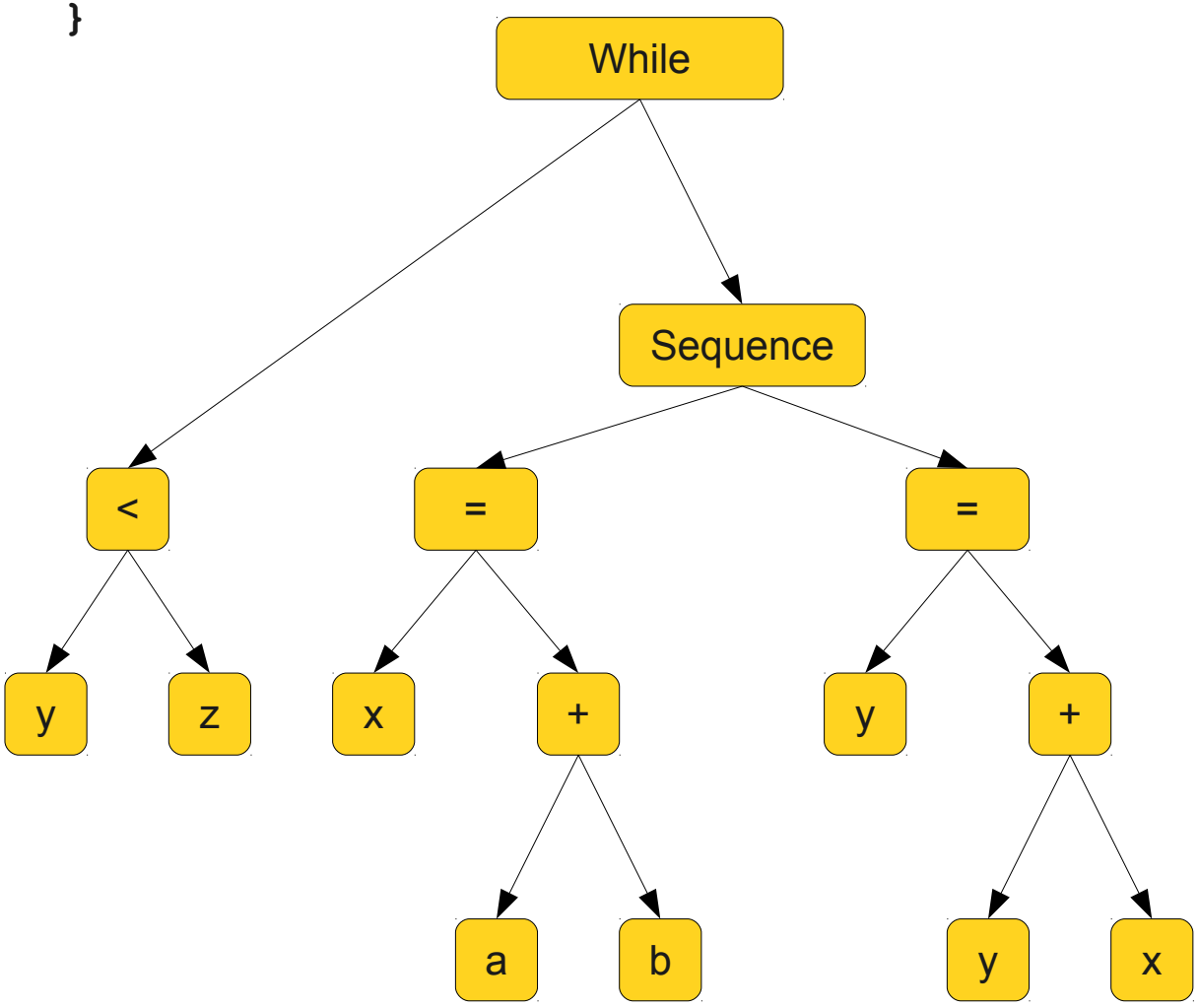


```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

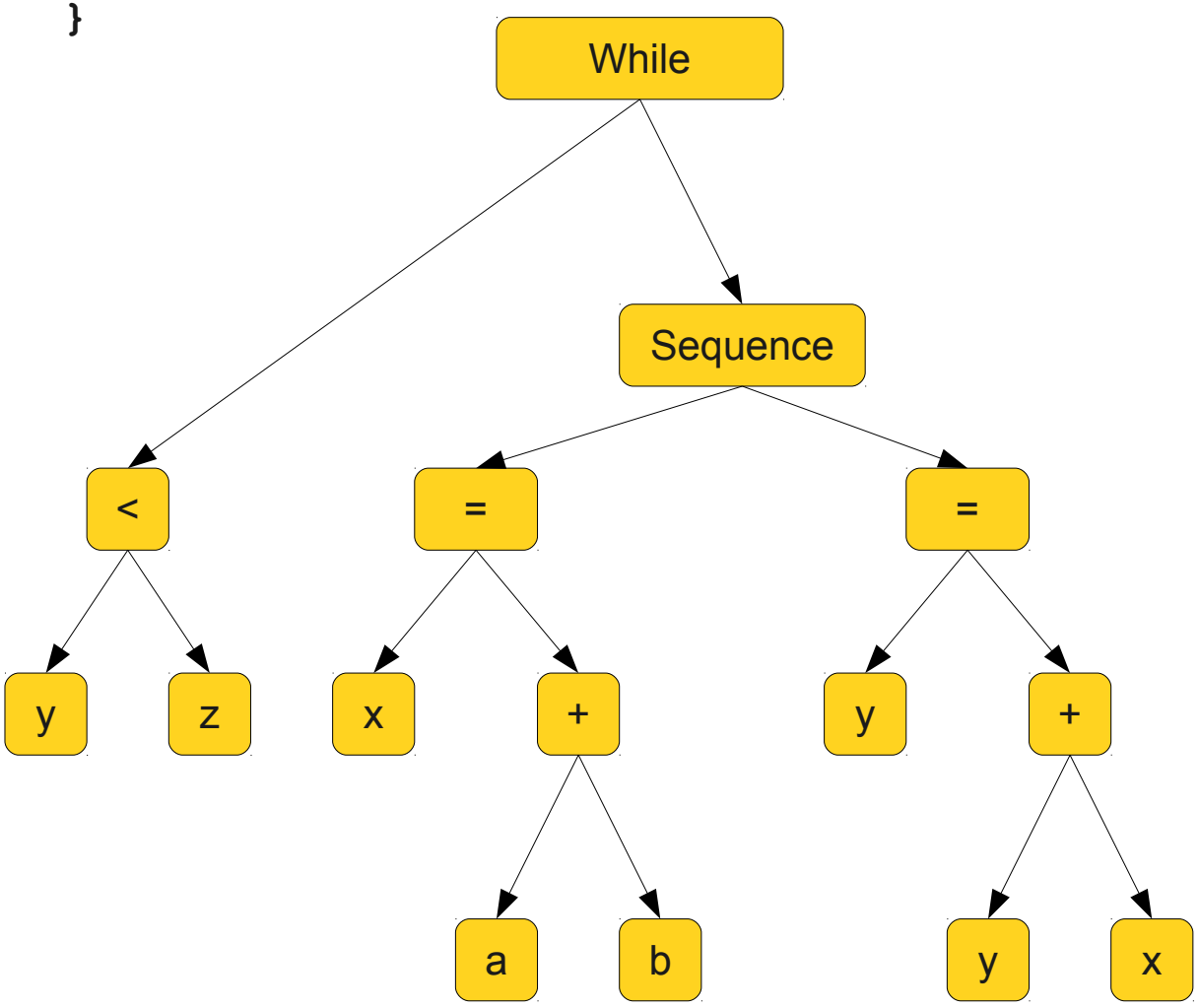


```
while (y < z) {
    int x = a + b;
    y += x;
}
```



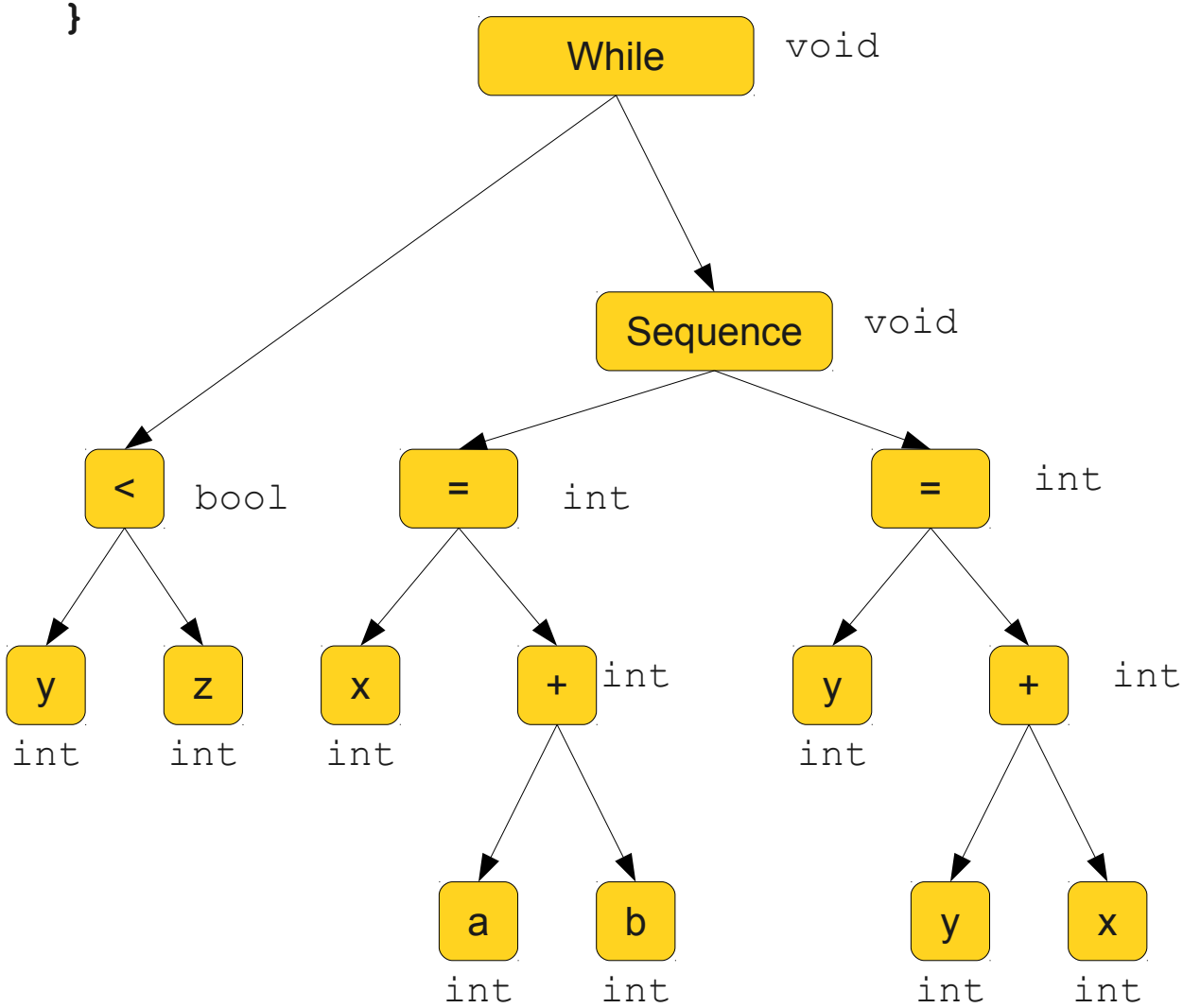
- Lexical Analysis
- **Syntax Analysis**
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

```
while (y < z) {
    int x = a + b;
    y += x;
}
```



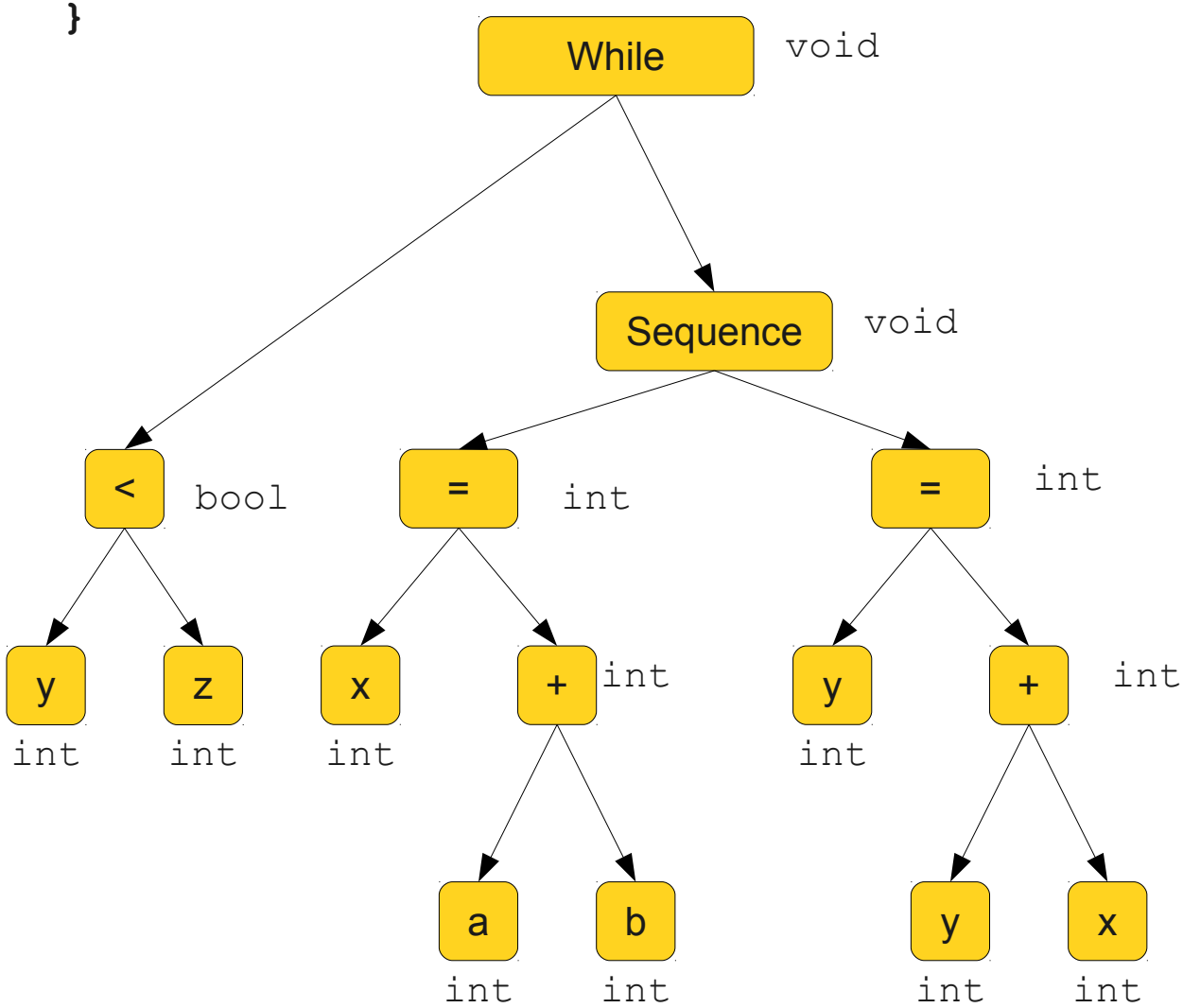
- Lexical Analysis
- Syntax Analysis
- **Semantic Analysis**
- IR Generation
- IR Optimization
- Code Generation
- Optimization

```
while (y < z) {
    int x = a + b;
    y += x;
}
```



- Lexical Analysis
- Syntax Analysis
- **Semantic Analysis**
- IR Generation
- IR Optimization
- Code Generation
- Optimization

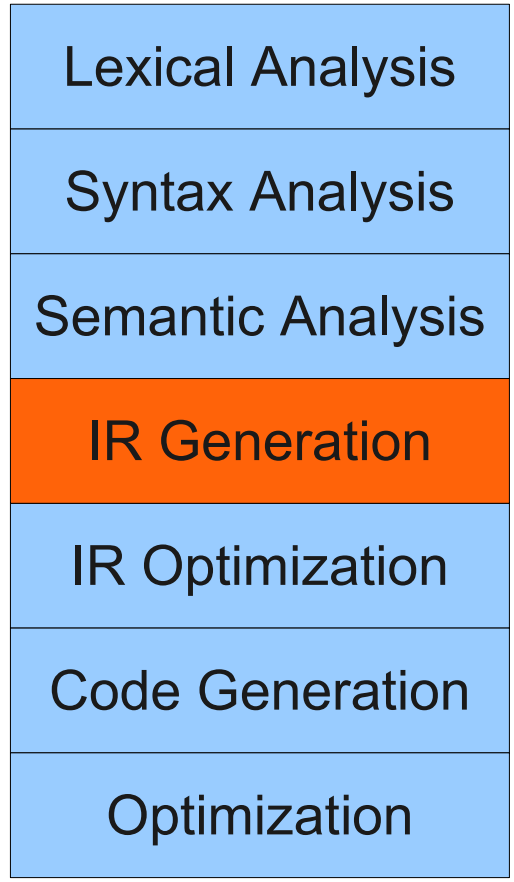
```
while (y < z) {
    int x = a + b;
    y += x;
}
```



- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- **IR Generation**
- IR Optimization
- Code Generation
- Optimization

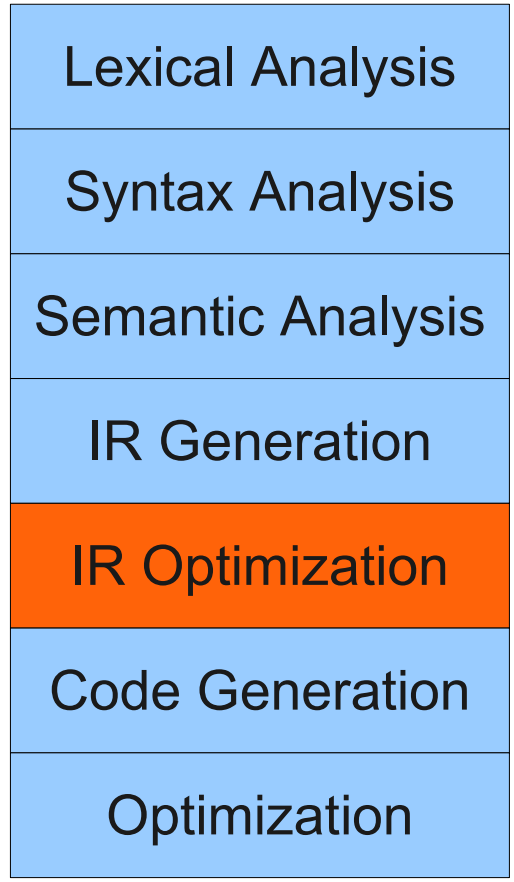
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
Loop: x   = a + b
      y   = x + y
      _t1 = y < z
      if _t1 goto Loop
```



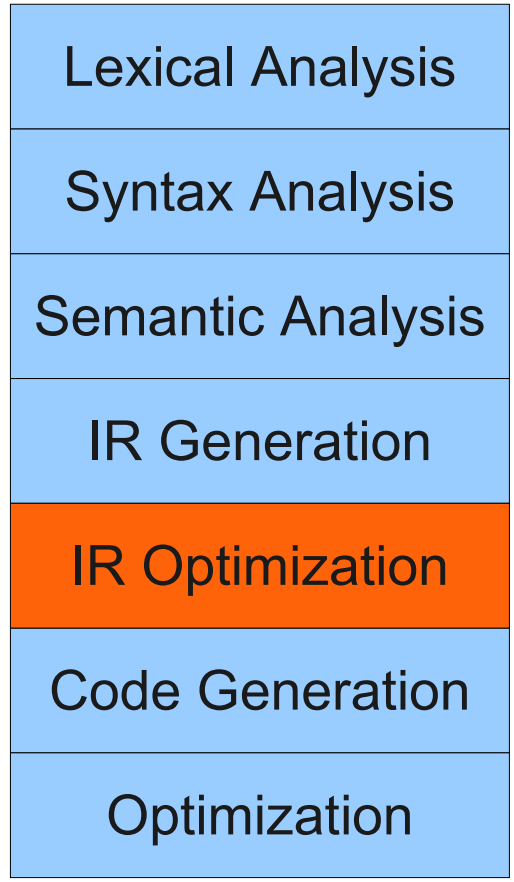
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
Loop: x   = a + b
      y   = x + y
      _t1 = y < z
      if _t1 goto Loop
```



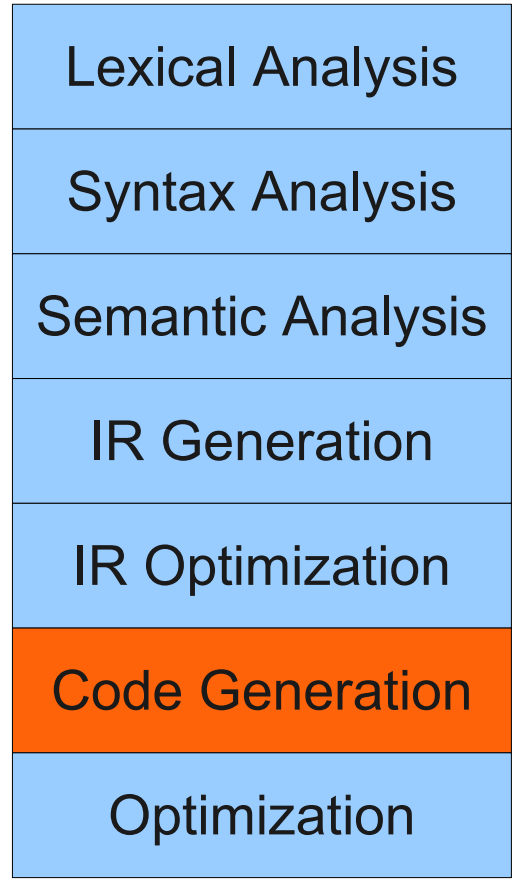
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
        x    = a + b
Loop:   y    = x + y
        _t1  = y < z
        if _t1 goto Loop
```



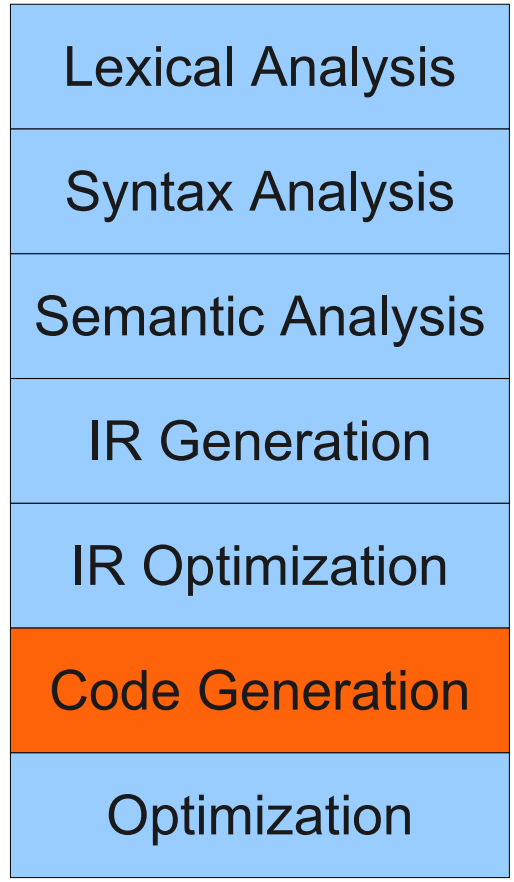
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
        x    = a + b
Loop:   y    = x + y
        _t1  = y < z
        if _t1 goto Loop
```



```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
        add $1, $2, $3
Loop:   add $4, $1, $4
        slt $6, $1, $5
        beq $6, loop
```



```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
        add $1, $2, $3
Loop:   add $4, $1, $4
        slt $6, $1, $5
        beq $6, loop
```

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
        add $1, $2, $3
Loop:   add $4, $1, $4
        blt $1, $5, loop
```

Compiler Phases

Pseudo-SML Code

```
val result = let val x = 10 :: 20 :: 0x30 :: []
in List.map (fn a → 2 * 2 * a) x
end
```

Discuss:

- what does the code do?
- what would be the necessary steps to translate the source code to machine code?

Translation from Source to C to Machine Code

Equivalent C Code

- initialize the array
- map translated to a loop
- possible optimization:
  2*2=4 at compile time,
  result reuses the space of x.

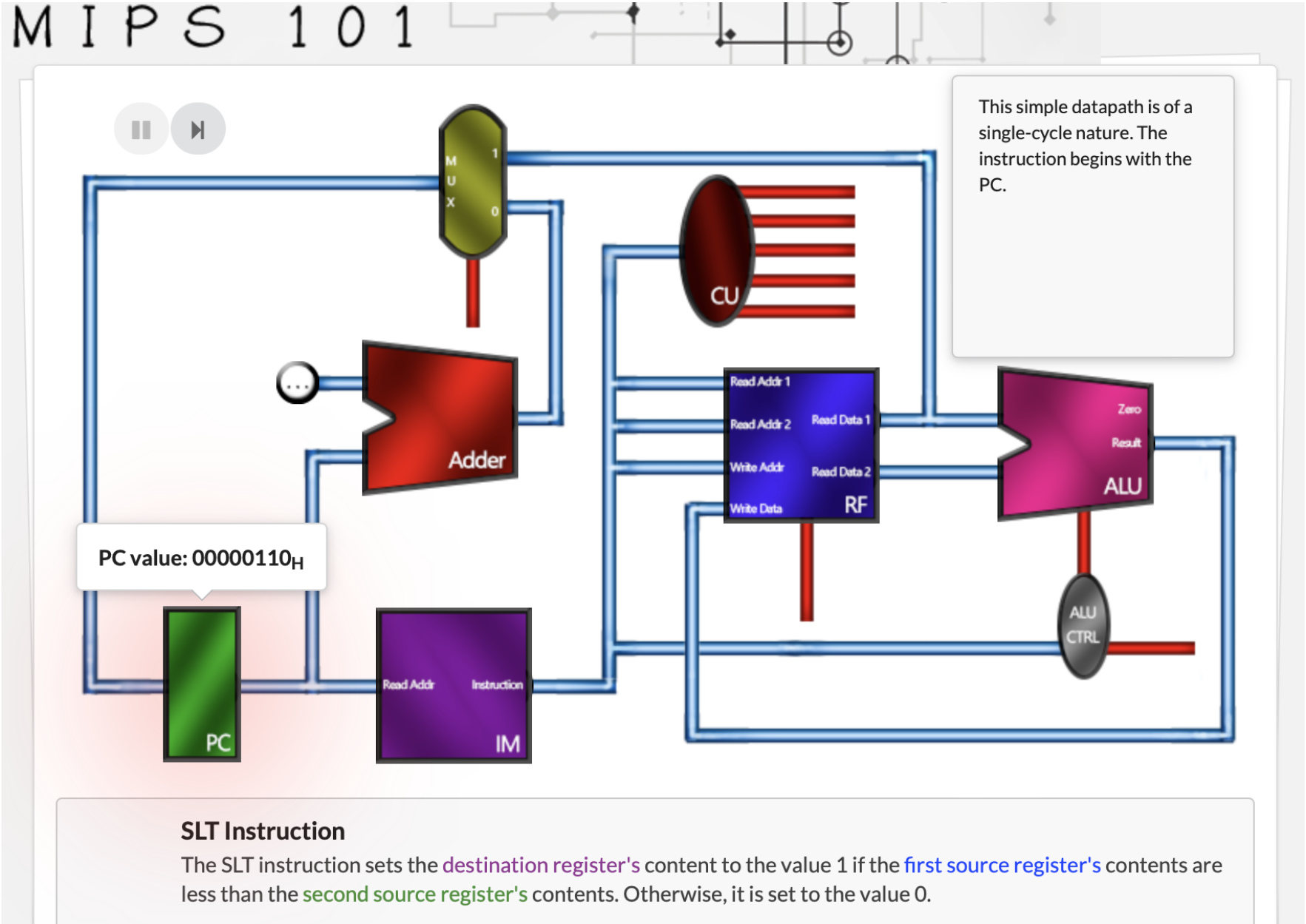
```
int x[3] = {10, 20, 0x30};
int i = 0;
for (int i = 0; i < 3; i++) {
  x[i] = 4*x[i];
}
```

Pseudo MIPS assembly code

- three-address-like code
- registers instead of variables
- explicit load and store ops
- shift instead of multiplication

```
load_imm  $2, 0      # $2 counter
load_addr $3, x_addr # $3 address of x
load_imm  $5, 3      # $5 stores ct 3
loop:
    branch_ge  $2, $5, end
    load_word  $4, 0($3) # $4 holds x[i]
    shift_left $4, $4, 2 # multiply by 4
    store_word $4, 0($3) # store in x[i]
    add_imm    $3, $3, 4 # next x addr
    add_imm    $2, $2, 4 # i = i + 1
    jmp        loop
end:
```

/ 15


MIPS 101
This simple datapath is of a single-cycle nature. The instruction begins with the PC.
M
U
X
1
0
CU
Adder
Read Addr 1
Read Addr 2
Write Addr
Write Data
Read Data 1
Read Data 2
RF
Zero
Result
ALU
ALU CTRL
PC value: 00000110H
PC
Read Addr
Instruction
IM
SLT Instruction
The SLT instruction sets the destination register's content to the value 1 if the first source register's contents are less than the second source register's contents. Otherwise, it is set to the value 0.

Challenges

- How can we program a quantum computer?
- What are the basic structures that a language should support?
- How can a compiler help a user develop abstract/high-level reasoning about algorithms?
- How to compile and optimize quantum programs?
- How do we test and verify quantum programs?

Comparison

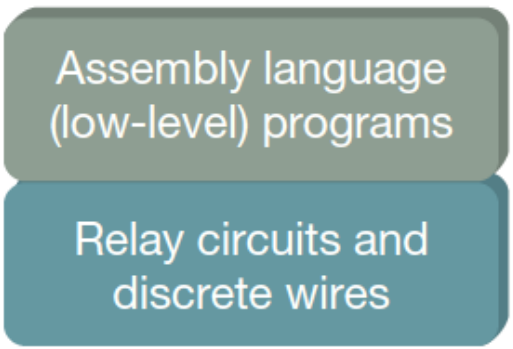


Fig.: Toolchain for computing in the 1950's

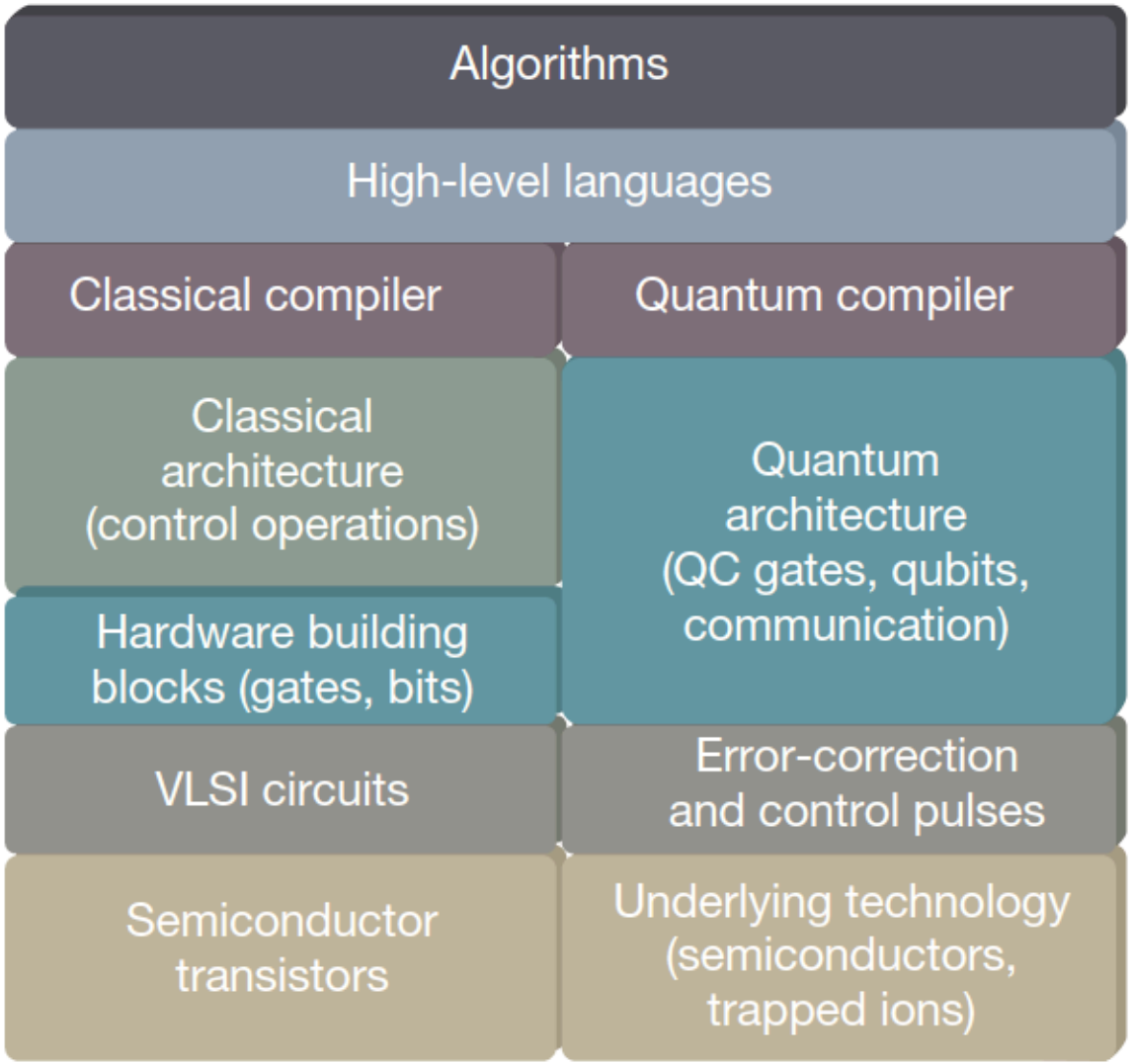


Fig.: Toolchain for classical and quantum computing today

Main Differences

Quantum software	Classical software
Compiles from algorithms to gate-level instructions	Compiles from algorithms to machine instructions
Some inputs of problems known at compilation time	Inputs of problems unknown at compilation time
Very limited reliability through error correction	Reliability generally assumed
Can run only small programs for debugging	Runs all programs for debugging
Vast parallelism	Limited parallelism

Although software in both domains shares many similarities, some key differences drive many of the challenges in developing quantum software tools. The combination of state fragility and the no cloning theorem makes error correction much more important in QC than in classical computing.

Cloud-based Quantum Computing

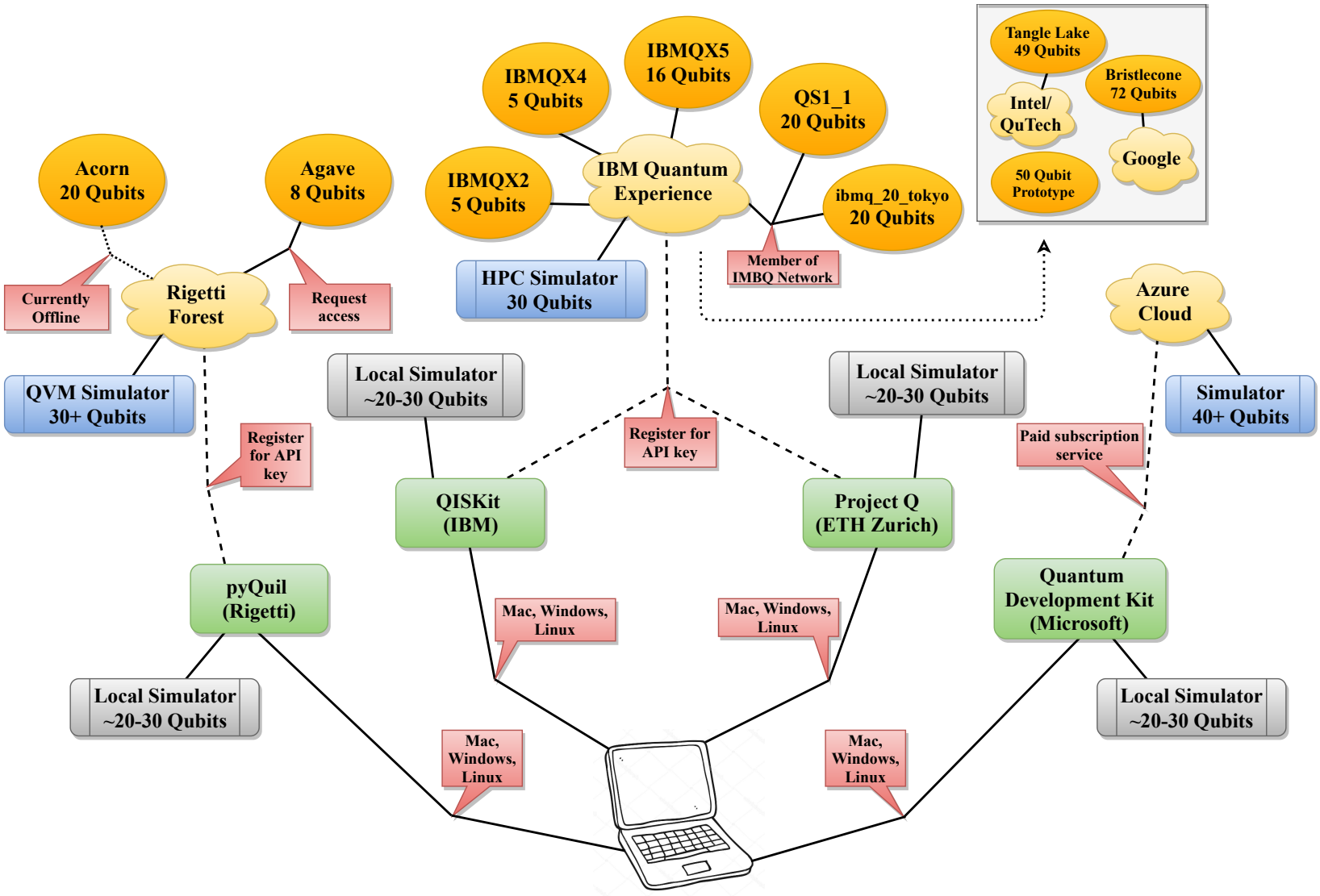
Quantum computers are now available to use via the internet, thanks to the development of software tools for cloud computing.
Quantum programming today is chiefly the invocation of quantum emulators, simulators or processors through the cloud. Some existing platforms:

- IBM Q Experience (access to quantum hardware as well as HPC simulators)
- Quandela Cloud (1st European photonic quantum computer)
- Xanadu Quantum Cloud (access to three fully programmable photonic quantum computers)
- Forest by Rigetti Computing (include a programming language)
- LIQUi$\rangle$ by Microsoft (include a programming language).

See here or here for a more comprehensive list.

Cloud-based Quantum Computing

Accessing quantum devices via software platforms.



Quantum Computing Platforms

Quantum computing platforms provide SDK, typically in the form of libraries of reusable functions or module interfaces allowing users to schedule and run quantum programs on a variety of local simulators and cloud-based quantum processors.

- Programs are typically in low-level code
- The compilation process consists in the last phases of a modern compiler: intermediate representation generation, code generation, optimization.
- Qasm is a hardware-agnostic quantum assembly language which guarantees the interoperability between all the quantum compilation and simulation tools.

Qiskit

- A open source SDK created by IBM.
- Allows users to construct quantum programs and run them on simulators or real quantum computers.
- Simulators may run on the user's own device.
- Includes OpenQasm, an intermediate representation that can be used by higher-level compilers to communicate with quantum hardware, and allows for the description of a wide range of quantum operations, as well as classical feed-forward flow control based on measurement outcomes.

Qiskit Code Example

```
qc.h(0)
qc.cx(0, 1)
print(qc.draw())
res = qi.execute(qc).data()
print(f"Amplitudes of the quantum states after H, CX:
res['statevector']"
```

Quantum programming languages

Quantum programming languages were introduced more than twenty years ago.

- theoretical
- no existing hardware

Today's existing quantum computers bring about the problem of actually *implementing* such languages, i.e. bridging the gap from the programmer's high-level thinking to the machine's qubits.
This requires

- understanding the whole chain from languages to quantum machine
- developing a *open source* software tool chain offering a suite of compilers for various quantum programming languages
- realizing it in an efficient and reliable way.

Some Implemented Languages

A few high-level quantum programming languages have been implemented:

- ProjectQ (ETH Zurich) is an open-source (DSL) quantum language with a compiler translating programs into the low-level instruction sets supported by the various back-ends (e., hw or circuit drawers).
- Scaffold is a programing language with a compiler written using the LLVM open-source infrastructure. Quantum applications written in Scaffold are compiled to a low-level quantum assembly format (QASM).
- Microsoft's LIQUi|> provides a high-level languages and a toolkit including a compiler, optimizers, translators and various simulators. Not open-source.

Silq

Silq is a high-level programming language for quantum computing with a strong static type system, developed at ETH Zürich.

Main features

- More 'abstract' than the other existing quantum languages.
- Type system and semantics formally defined.
- An implemented type-checker.
- *No compiler* but a proof-of-concept simulator.
- A development environment.

expressions

$$e ::= c \mid x \mid \texttt{measure} \mid \texttt{reverse} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \mid e'(e_1, \ldots, e_n) \mid \lambda(\beta_1 x_1 : \tau_1, \ldots, \beta_n x_n : \tau_n).e$$

$(\vec{e})$ $(\vec{\beta}\vec{x} : \vec{\tau})$

types

$$\tau ::= \mathbb{1} \mid \mathbb{B} \mid \bigtimes_{k=1}^{n} \tau_k \mid \bigtimes_{k=1}^{n} \beta_k \tau_k \; ! \xrightarrow{\alpha} \tau' \mid \; !\tau$$

annotations

$$\alpha \subseteq \{\texttt{mfree}, \texttt{qfree}\}$$

$$\beta \subseteq \{\texttt{const}\}$$

Outlook

The *quantum revolution* will depend heavily on software toolchain, that will bridge the gap between algorithms and physical machines.

- The current constraints of the QC toolchain resemble the constraints of computing in the 1950's.
- The QC toolchain is still evolving and its layers are being gradually filled in.
- Much work is needed especially in the aerea of verification.

Importance of developping a software toolchain for QC:

- It will allow reduction of the required number of qubit and operations in implementing quantum algorithms
- This will make an important difference when QC is ready to be deployed commercially.