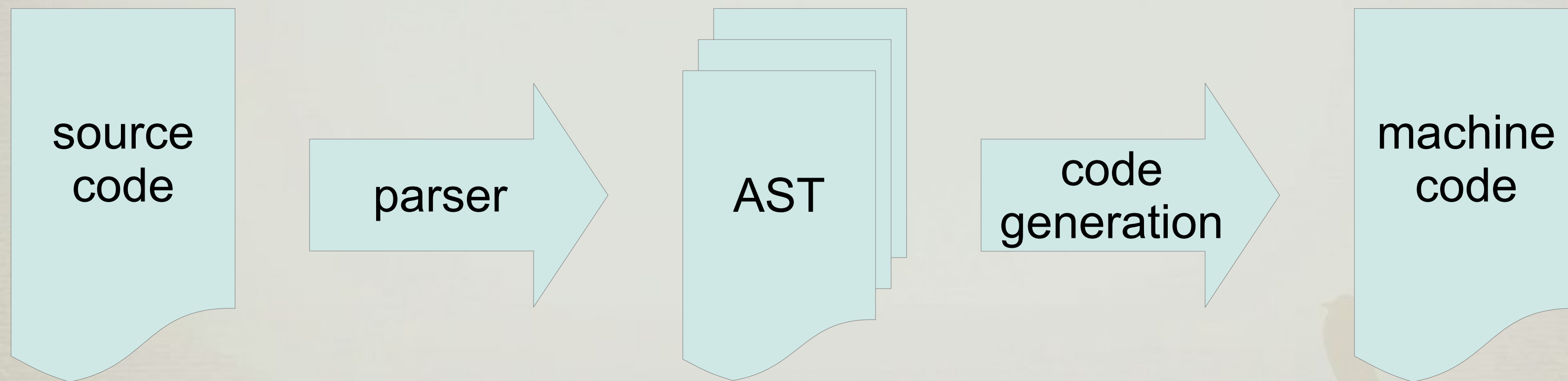
System software

Parser and
code representation

Assembly process

- Compilation / translation
  - parsing
  - code generation



Code representation

- Kinds of commands
  - `Node`
    - `Comment`
    - `InstructionF1`
    - `InstructionF2`
    - `InstructionF3`
    - `InstructionF4`
    - `Directive`
      - START, END, ORG, LTORG, ...
    - `Storage`
      - BYTE, WORD, RESB, RESW.

Code representation

- Class `Node`:
  - contains common behaviour for all commands
  - `String label`
  - `Mnemonic mnemonic`
  - `String comment`
  - `toString()`

Code representation

- Class Code:
  - name, start address, program
  - program is a table of commands
    - e.g. List<Node> program
  - symbol table
    - Map<String, Integer> symbols
    - defineSymbol(String sym, int val)
    - int resolveSymbol(sym)
  - etc.

Traversing the AST

- Two-pass assembler
  - parsing the source and resolving the symbols
  - see the lectures
- Multi-pass assembler
  - load the source code into memory
  - parse it to produce AST
  - traverse the AST multiple times
  - each time do something „small“

Traversing the AST

- Problem
  - AST may be diverse data structure
  - we may need various kinds of traversing, e.g.,
    - parsing
    - define & evaluate EQU expressions
    - resolve absolute symbols
    - resolve blocks
    - resolve symbols

Traversing the AST

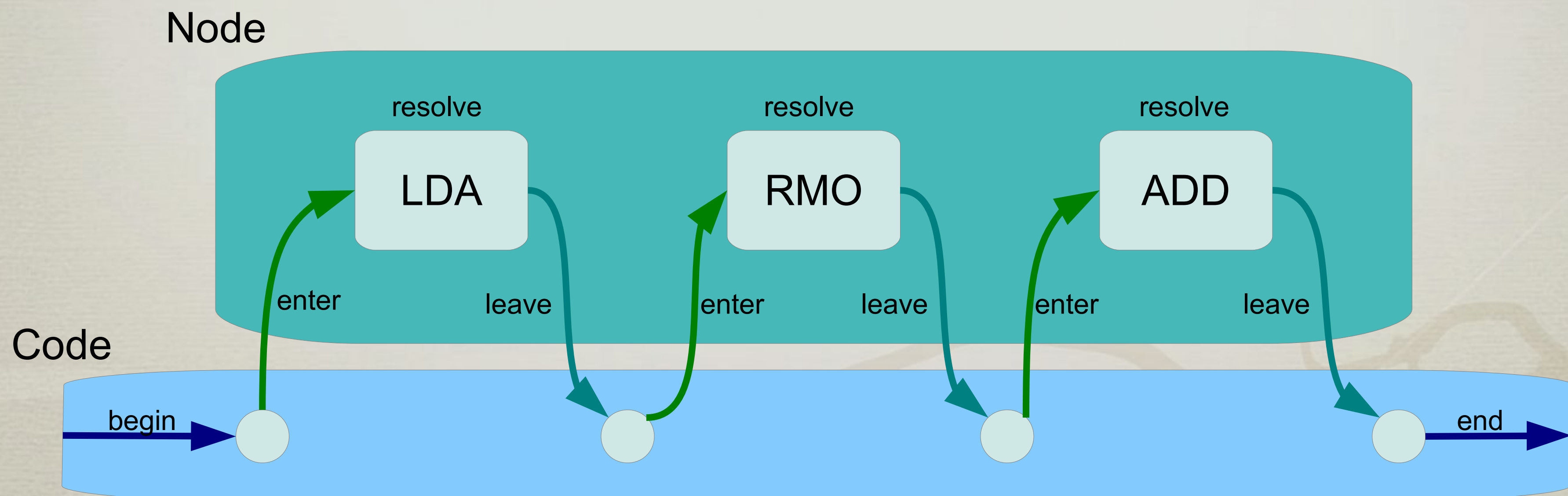
- Visitor design pattern (obiskovalec)
  - commands are sequentyally processed
  - on invoke a specific action for each command
  - full visitor is based on visit() and accept() methods
    - based on simulating double dispatch
    - https://en.wikipedia.org/wiki/Visitor_pattern
  - we will introduce a simplified visitor

Traversing the AST

- Simplified visitor – Node
  - for each kind of traversal
    we define a **special purpose** methods
    - process the node in a specific way
    - e.g., Node.resolve() and overrides
  - define also general visiting methods for **entering** and **leaving** the node
    - processing common to all visitors
    - e.g., Node.enter() and Node.leave()
      - incrementing the LOCCTR

Traversing the AST

- Simplified visitor – Code
  - full traversal of whole AST
  - Code.resolve()
    - just do the for loop with proper initalization / finalization



Traversing the AST

- Visitor kinds
  - `resolve()`
    - resolving the smybols
  - `byte[] emitCode()`
    - image of a machine code
  - `String emitText()`
    - contents of the object file
  - `String dumpCode()`
    - as in the log file
  - `String dumpSymbols()`
    - writes used symbols
  - etc.

Parser

- Parsing
  - a process of transforming the source code into the corresponding internal representation
  - AST – abstract syntax tree
    - parsing assembly is usually simple due to simple syntax
  - process
    - read the source code
    - generate its AST

Parser

- SIC/XE source code
  - line based format
  - each line is independent whole
  - empty lines are ignored
  - one line gives one commad
    - instruction or directive or comment
  - inline comments
    - from the character „.“ till the end of line

Parser

- Command format
  - **label**
    - string of alphanumeric characters starting at the column 1
  - **mnemonic**
    - symbolic name for the instruction opcode
    - unknown names are invalid
  - **operands**
    - based on the instruction zero, one or more operands may follow

Parser

- Class `Parser`.
  - `String parseLabel()`
    - string of alphanumeric characters starting at the column 1
  - `Mnemonic parseMnemonic()`
    - a specific (must be present in the symbol table) string of alphanumeric characters not starting at the column 1
  - `String parseSymbol()`
    - string of alphanumeric characters

Parser

- Class `Parser`
  - `int parseRegister()`
    - AXLBSTF → 0,1,2,3,4,5,6
  - `void parseComma()`
    - a comma with any whitespace around it
  - `boolean parseIndexed()`
    - comma and X with any whitespace around them

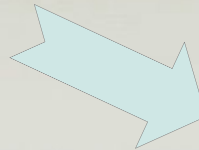
Parser

- Razred `Parser`.
  - `int parseNumber()`
    - `0b`*BIN* (binary number)
    - `0o`*OCT* (octal number),
    - `0x`*HEX* (hexadecimal number)
    - *DEC* (decima number)
  - `byte[] parseData()`
    - `C'`<*chars*>`'` … ASCII encoding
    - `X'`<*hex*>`'` … hex encoding
    - *num* … 24 bit number (WORD representation)

Parser

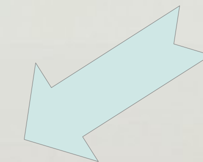
Asm.java: main(...)

```
Parser parser = new Parser();
Code code = parser.parse(input);
code.print();
```



Parser.java: Parser.parse(String input)

```
lexer = new Lexer(input);
Code = new Code();
while (lexer.peek() > 0) {
    // skip whitespace
    ...
    // parse the line
    Node inst = parseInstruction();
    if (inst != null)
        code.append(inst)
}
return code
```



Parser.java: Parser.parseInstruction()

```
// a label
String label = parseLabel();
skip whitespace

// mnemonic
Mnemonic mnemonic =  parseMnemonic();
skip whitespace

// operands of the mnemonic
Node node = mnemonic.parse(this);

return node;
```

```
Mnemonic.parse(Parser parser)
```

Parser

- Base class `Mnemonic`
  - name, opcode, ...
  - method for parsing any operands
    - `Node parse(Parser parser)`
    - should be overriden correspondingly
  - observe the „conditional parsing“
    - we use dispatch available in OO languages

```
// mnemonic
Mnemonic mnemonic =  parseMnemonic();
skip whitespace

// operandi ustreznega mnemonika
Node node = mnemonic.parse(this);
```

Parser

- Mnemonic classes
  - `MnemonicD, MnemonicDn,`
  - `MnemonicF1,`
  - `MnemonicF2n, MnemonicF2r, MnemonicF2rn, MnemonicF2rr`
  - `MnemonicF3, MnemonicF3m`
  - `MnemonicF4m`
  - `MnemonicSd, MnemonicSn`