DA378A – C++ och Programkonstruktion 2018

F11. Design Patterns, Operator-Precedence parser (L6 prep.)

Grammar-driven parser design

L6 Preparation

Carl Johan Gribel
carl.johan.gribel@mau.se

Interpreter

- An **interpreter** is, basically, a program that reads and executes a set of instructions (i.e. another program written in another language), without converting it to machine code.
- For a **compiler** (such as VC++) it is the other way around: instructions are compiled to machine code, but it is not executed.

- The interpreted/compiled language has some form of **grammar**, defining its structures and syntax.
  – Can be simple, such as one-worded instructions (e.g. machine code).
  – Or complex, such as C++.

Interpreter for logical-comparison expressions

```
1 && 1 < 2 && 0 || ( 1 == 1 )
```

- Task: interpret (*evaluate*) expressions containing **logical and comparative operators**
- Logical operators: &&, ||
- Comparison operators: <, >, ==, <=, >=, !=

Precedence

```
1 && 1 < 2 && 0 || ( 1 == 1 )
```

Language rules

- **Precedence**:
  1. Parenthesis (…)
  2. Comparison op’s: <, >, ==, >=, <=, !=
  3. And: &&
  4. Or: ||
- **Associativity**: operations (with the same operator) are interpreted left-to-right

Things to note

- The C++ `bool` type implicitly casts to `int`. This allows some weird-looking expressions.

- **Example**: `2 == 2 == 2` equals `false` (!)

  – Same operator so no precedence rules take effect.

  – Left-associativity: `2 == 2` equals `true`

  – Then, `2 == 2 == 2` ⇔
  `true == 2` ⇔
  `1 == 2` implicit cast `bool` to `int`
  equals `false`

Precedence

- We can work out the solution intuitively

$$\underbrace{\underbrace{1\ \&\&\ \underbrace{1 < 2}_{=1}\ \&\&\ 0}_{=0}\ ||\ (\ \underbrace{1 == 1}_{=1}\ )}_{=1}$$

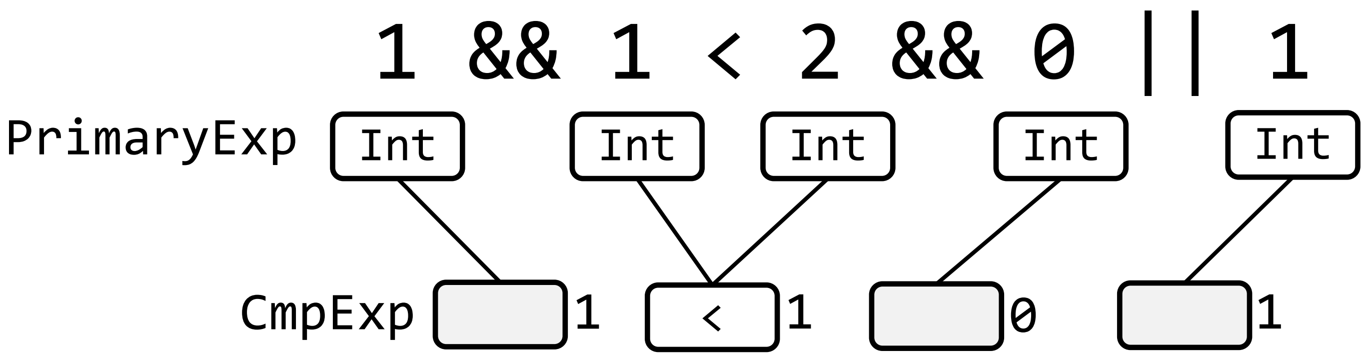
- But how do we design an interpreter to do the job?
  - This figure is actually a good start, but lets remake it as a tree (with a slightly simplified expression to start with)

Precedence Tree

```
         1 && 1 < 2 && 0 || 1
PrimaryExp Int  Int Int  Int  Int
```

- Start by defining **primary expressions**.
  These are “indivisible” parts of the expression.
  - This example: **Numbers** (int)
  - Parenthesis
  - Also, math expressions based on +, –, *, / → **what you’ll do in L6**.

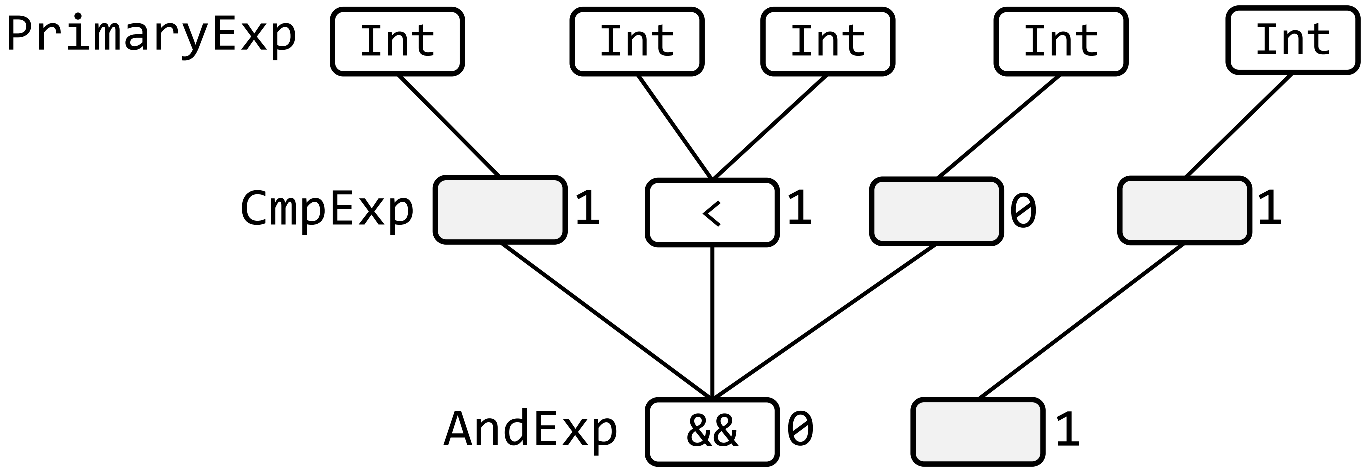
Precedence Tree



- Go on with the top priority operations– **comparisons**.
  - Primaries (`PrimaryExp`) **separated by a comparison operator** are grouped in **comparison expressions** (`CmpExp`) and evaluated.
  - Primaries without a comparison operator also form `CmpExp`’s – but without a right-hand-side operator (left-hand-side is then simply forwarded).

Precedence Tree

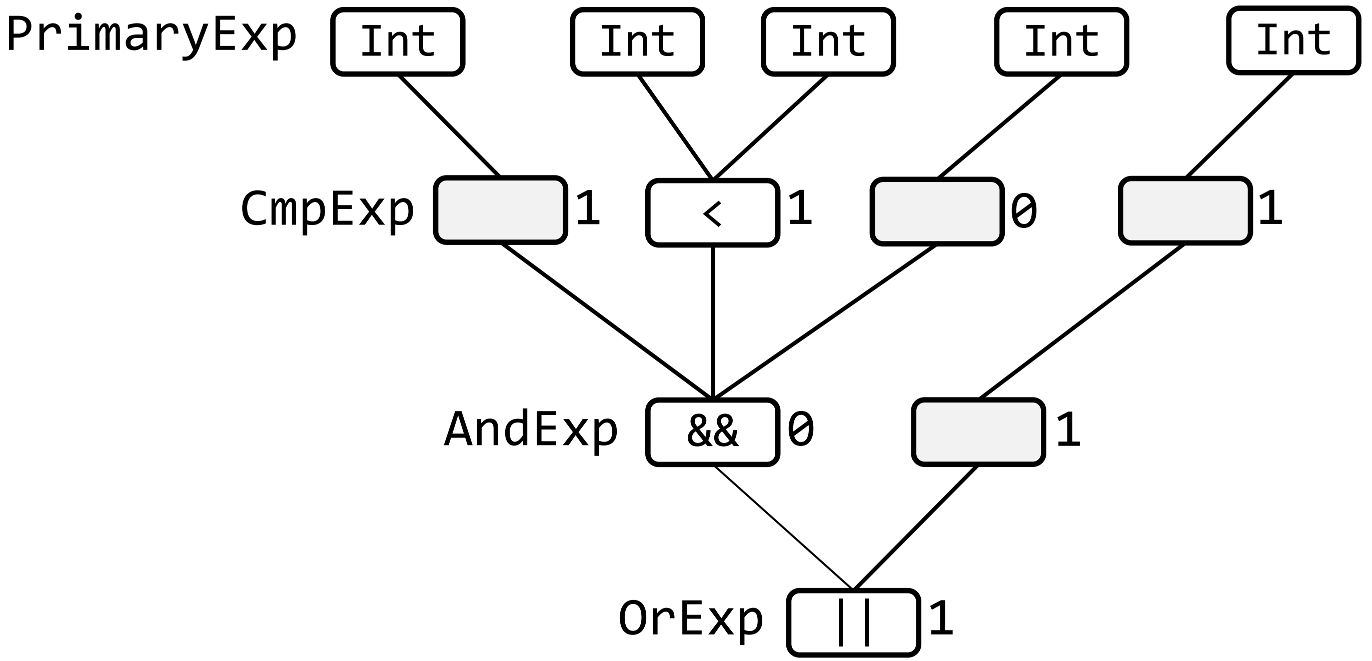
```
1 && 1 < 2 && 0 || 1
```



- Same procedure with the second highest-priority operator: **And (&&)**
- Note that some operators (like &&) may have **more than two operands**. In this case, the expression is evaluated left-to-right

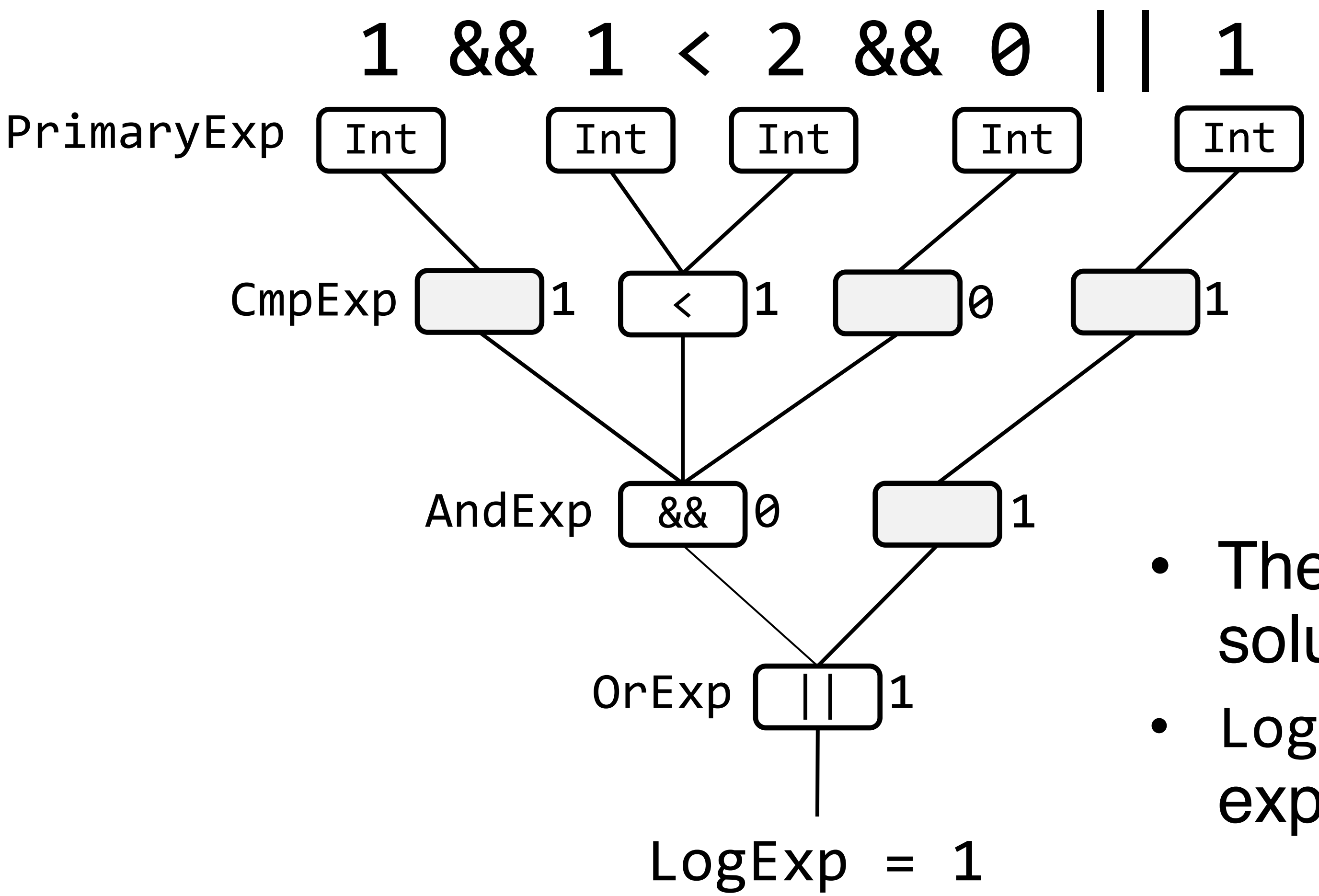
Precedence Tree

`1 && 1 < 2 && 0 || 1`



- Finally, the operator with lowest priority: **Or (||)**

Precedence Tree



- The evaluation of `OrExp` is the solution to the expression
- `LogExp` captures the entire expression.

Precedence Tree

```
1 && 1 < 2 && 0 || ( 1 == 1 )
```

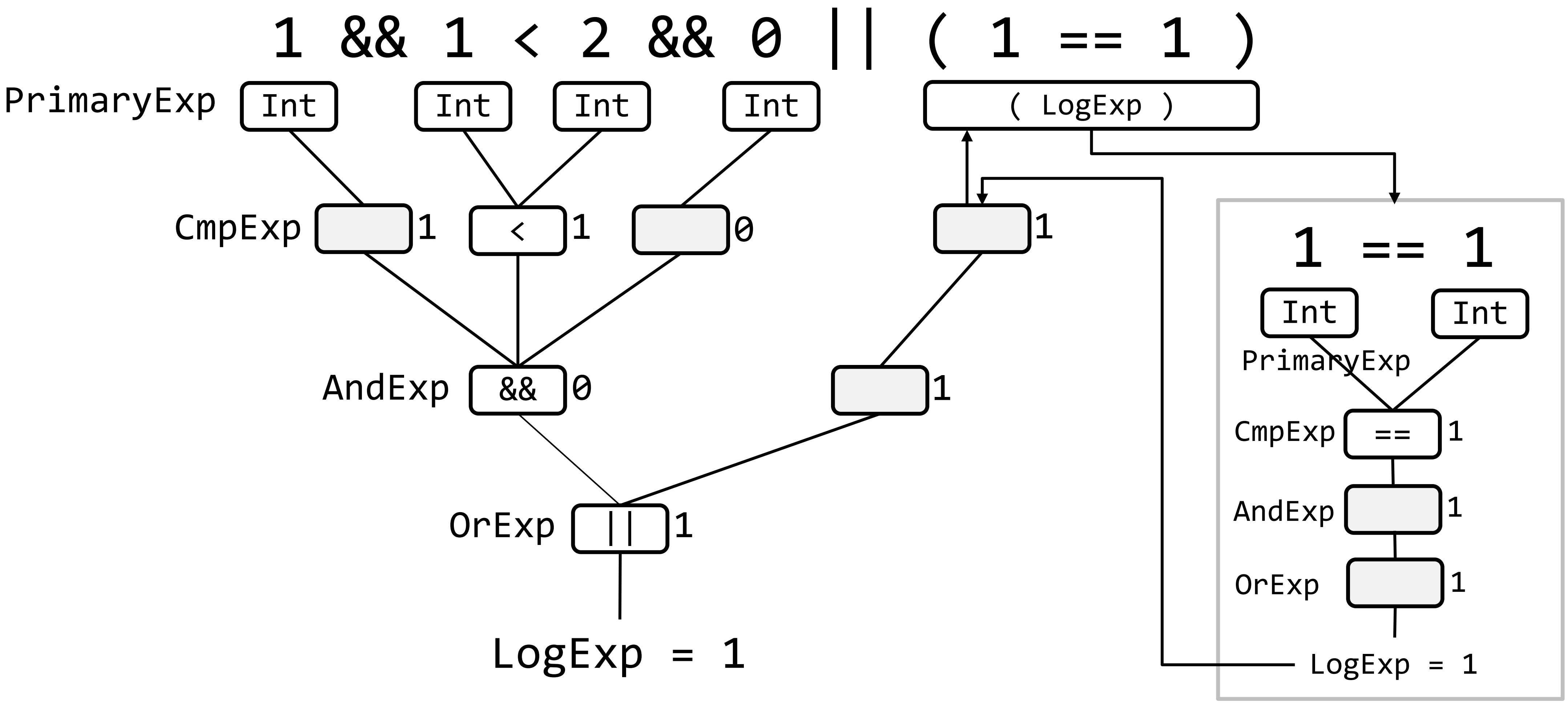
- So what about parentheses?

Precedence Tree

```
                 1 && 1 < 2 && 0 || ( 1 == 1 )
PrimaryExp      [Int] [Int] [Int] [Int] [( LogExp )]
```

- A parenthesis is another form of **Primary expression**
  - "(" **`LogExp`** ")"
- Parenthesis grants localized priority to a sub-expression with any combination of primaries and operators – i.e. a **sub-`LogExp`**.
- The inside of a parenthesis is thus evaluated as a **separate `LogExp`-expression**.

Precedence Tree



Abstract Grammar

- **Abstract grammar** for logical-comparison expressions

```
LogExp := OrExp
OrExp := AndExp [ || AndExp ]*
AndExp := CmpExp [ && CmpExp ]*
CmpExp := PrimaryExp [ == PrimaryExp | < PrimaryExp | > PrimaryExp ]*
PrimaryExp := Int | ( LogExp )
Int := -?[0-9]+
```

Abstract Grammar

- **Abstract grammar** for logical-comparison expressions

Abstract block / parsing function

“Zero or more”

```
LogExp := OrExp
OrExp := AndExp [ || AndExp ]*
AndExp := CmpExp [ && CmpExp ]*
CmpExp := PrimaryExp [ == PrimaryExp | < PrimaryExp | > PrimaryExp ]*
PrimaryExp := Int | ( LogExp )
Int := -?[0-9]+
```

literal string

Regular expression

Operator-Precedence parser

The grammar gives important hints on how we can **design the parser**.

- Let each block/operator-expression be a separate **parsing function**
  - `LogExp`, `OrExp`, `AndExp`, `CmpExp`, `PrimaryExp`
- Parsing functions …
  - **Consume** (step through) tokens of the expression.
    Token = sequence of characters separated by whitespace.
  - **Obtain operands by calling other parsing functions**. Usually the one with next-level (higher) priority.
  - **Evaluates operands**, if matched by its role (e.g. "+" for `AndExp`)

  Example: `OrExp` obtains its operands (one or more) by "asking" `AndExp`.
  `AndExp` then consumes & evaluates tokens, and return the result.

EXAMPLE CODE
OPERATOR-PRECEDENCE PARSER

Initialization

- Assume we are implementing an Interpreter for a logical-comparison grammar.
- The class has the following basic content:

```
// Tokenized expression
std::vector<std::string> tokens =
      { "1","&&","1","<","2","&&","0","||","1","(","1","==","1",")" };

// Current position
int position = 0;

// Reserved end of expression string symbol (ETX = end-of-text)
const std::string ETX = "\u0003";
```

Auxiliaries: peek

- **peek** – get current or future token

```cpp
// Return current token
std::string peek()
{
    return peek(0);
}

// Return token @steps ahead
std::string peek(int steps)
{
    if (position+steps >= tokens.size()) return ETX;

    return tokens[position+steps];
}
```

Auxiliaries: consume

- **consume** – step one token forward

```
// Advance to the next token.
// @token is a safeguard to make sure the caller knows what is being consumed.
void consume(const std::string& token)
{
    std::string next_token = peek();
    if (next_token == ETX)
        throw std::runtime_error("Consumed past last token\n");
    if (next_token != token)
        throw std::runtime_error("Could not consume token " + token + "\n");

    ++position;
}
```

evaluate

- **evaluate** – have the interpreter evaluate an expression
  - Immediately queries `LogExp` (private member function), which queries `OrExp`, the parsing function with lowest priority

```
bool evaluate()
{
    return parse_LogExp();
}
```

```
LogExp := OrExp
OrExp := AndExp [ || AndExp ]*
AndExp := CmpExp [ && CmpExp ]*
CmpExp := PrimaryExp [ == PrimaryExp |
                        < PrimaryExp |
                        > PrimaryExp ]*
PrimaryExp := Int | ( LogExp )
Int := -?[0-9]+
```

```
bool parse_LogExp()
{
    return parse_OrExp();
}
```

OrExp

```
LogExp := OrExp
OrExp := AndExp [ || AndExp ]*
AndExp := CmpExp [ && CmpExp ]*
CmpExp := PrimaryExp [ == PrimaryExp |
                       < PrimaryExp |
                       > PrimaryExp ]*
PrimaryExp := Int | ( LogExp )
Int := -?[0-9]+
```

```cpp
bool parse_OrExp()
{
    // Parse the left-hand-side block
    bool result = parse_AndExp();

    // Parse right-hand-side blocks
    std::string next_token = peek();
    while (1)
    {
        if (next_token == "||")
        {
            consume("||");
             result = (result || parse_AndExp());
        }
        else
             break;

        next_token = peek();
    }

    return result;
}
```

AndExp

```
LogExp := OrExp
OrExp := AndExp [ || AndExp ]*
AndExp := CmpExp [ && CmpExp ]*
CmpExp := PrimaryExp [ == PrimaryExp |
                       < PrimaryExp |
                       > PrimaryExp ]*
PrimaryExp := Int | ( LogExp )
Int := -?[0-9]+
```

```
bool parse_AndExp()
{
    // Parse the left-hand-side block
    bool result = parse_CmpExp();

    // Parse right-hand-side blocks
    std::string next_token = peek();
    while (1)
    {
        if (next_token == "&&")
        {
            consume("&&");
            result = result && parse_CmpExp();
        }
        else
            break;

        next_token = peek();
    }

    return result;
}
```

CmpExp

```
LogExp := OrExp
OrExp := AndExp [ || AndExp ]*
AndExp := CmpExp [ && CmpExp ]*
CmpExp := PrimaryExp [ == PrimaryExp |
                       < PrimaryExp |
                       > PrimaryExp ]*
PrimaryExp := Int | ( LogExp )
Int := -?[0-9]+
```

```cpp
bool parse_CmpExp()
{
    // Parse the left-hand-side block
    int result = parse_PrimaryExp();

    // Parse right-hand-side blocks
    std::string next_token = peek();
    while (1)
    {
        if (next_token == "==")
        {
            consume("==");
            result = (result == parse_PrimaryExp());
        }
        else if (next_token == "<")
        {
            consume("<");
            result = (result < parse_PrimaryExp());
        }
        else if (next_token == ">")
        {
            consume(">");
            result = (result > parse_PrimaryExp());
        }
        else
            break;

        next_token = peek();
    }

   return (bool)result;
}
```

PrimaryExp

```
LogExp := OrExp
OrExp := AndExp [ || AndExp ]*
AndExp := CmpExp [ && CmpExp ]*
CmpExp := PrimaryExp [ == PrimaryExp |
                       < PrimaryExp |
                       > PrimaryExp ]*
PrimaryExp := Int | ( LogExp )
Int := -?[0-9]+
```

```
int parse_PrimaryExp()
{
    int value;
    std::string next_token = peek();

    // Number
    if (is_int(next_token))
    {
        value = std::stoi(next_token);
        consume(next_token);
    }
    // Parenthesis expression: ( LogExp )
    else if (next_token == "(")
    {
        consume("(");
        value = parse_LogExp();
        if (peek() == ")")
            consume(")");
        else
            throw std::runtime_error("Expected: )\n");
    }
    // No valid PrimaryExp found, which is an error
    else
        throw std::runtime_error("expected int or ( )");

    return value;
}
```

Leftovers

- Regular expression matching
  - Either: via **`std::regex`** and **`regex_match`** (`#include <regex>`)
    - Flexible
  - Or: check string manually char-by-char
    - Feasible if pattern is simple
  - Only do the actual cast once the pattern has been matched
    - `int value = std::stoi(token);`

```
bool is_int(const std::string& token)
{
    // ...
}
```

```
LogExp := OrExp
OrExp := AndExp [ || AndExp ]*
AndExp := CmpExp [ && CmpExp ]*
CmpExp := PrimaryExp [ == PrimaryExp |
                        < PrimaryExp |
                        > PrimaryExp ]*
PrimaryExp := Int | ( LogExp )
Int := -?[0-9]+
```

Parser flow

- Parsing starts by:

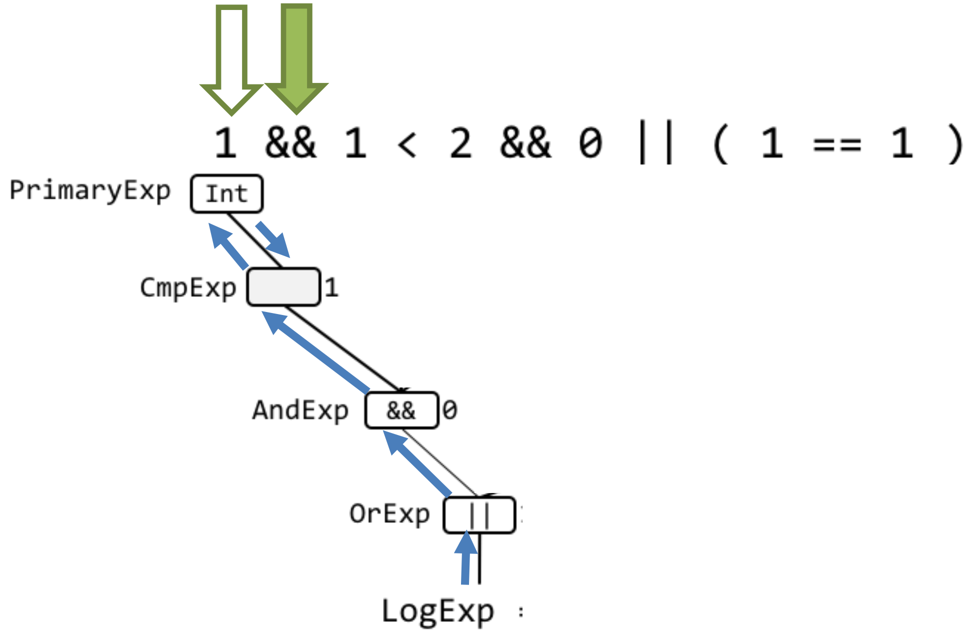
1. Setting a step-variable to the first token
2. Calling the `LogExp` parsing function

```
1 && 1 < 2 && 0 || ( 1 == 1 )
```

```
LogExp
```

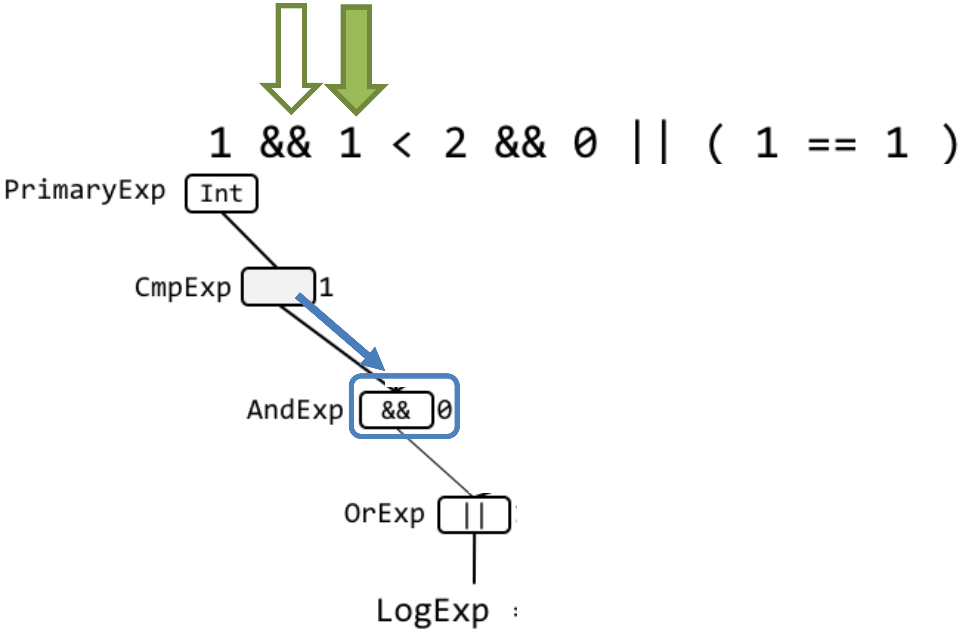
Parser flow

3. `LogExp` immediately calls `OrExp`. Which calls `AndExp` and so on, until a `PrimaryExp` is reached.
4. `PrimaryExp` consumes the token and returns its value ("1") to `CmpExp`.



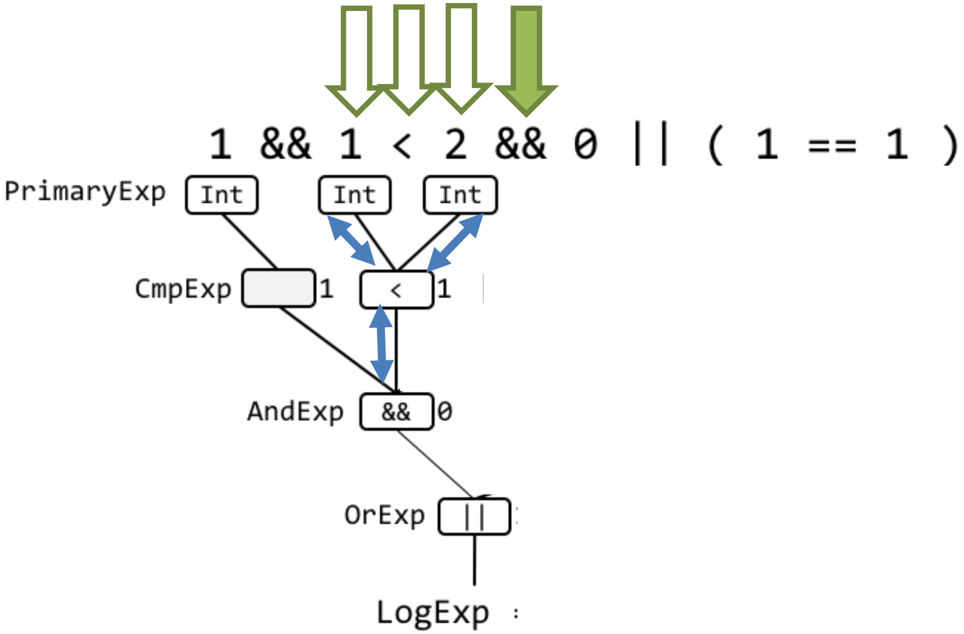
Parser flow

5. `CmpExp` does not find a comparison operator, so the value is sent back to `AndExp`.
6. `AndExp` finds its operator, “&&”, and consumes it.



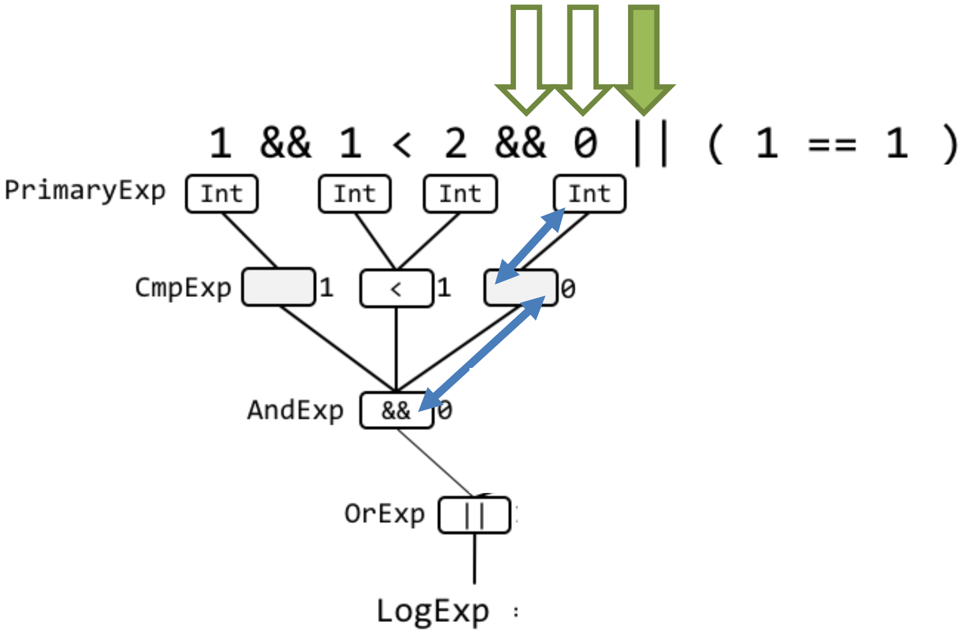
Parser flow

7. `AndExp` then queries `CmpExp` again, which finds and evaluates two operands (consuming three tokens)
8. `AndExp` now has two operands, which are valuated.



Parser flow

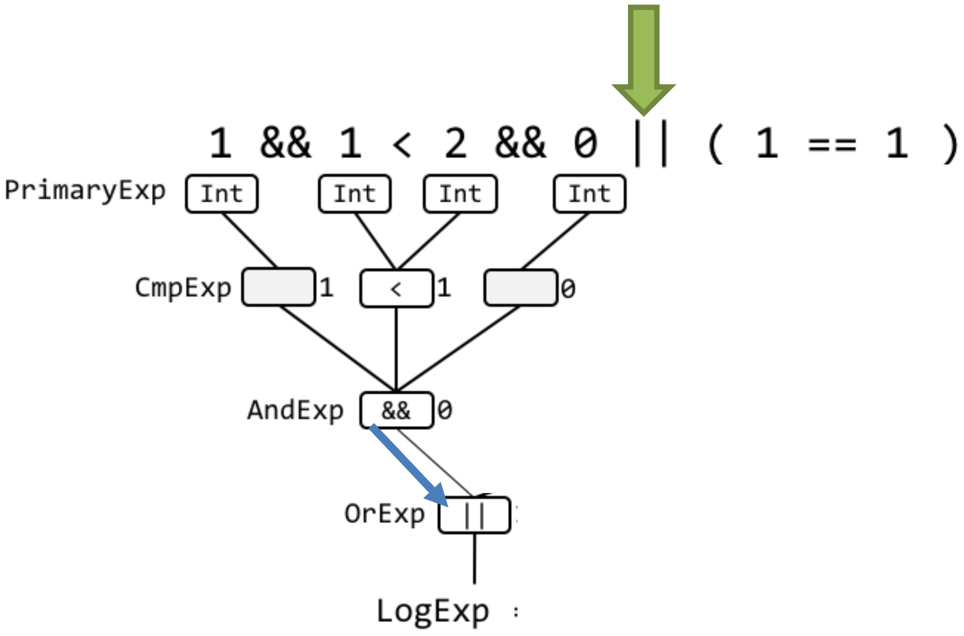
9. Since there is another “&&”, `AndExp` consumes it and obtains a third operand the same way as before.



Parser flow

10. `AndExp` re-evaluates based on the last operand and returns its result to `OrExp`.
11. `OrExp` now has its first operand, and an impending “||”.

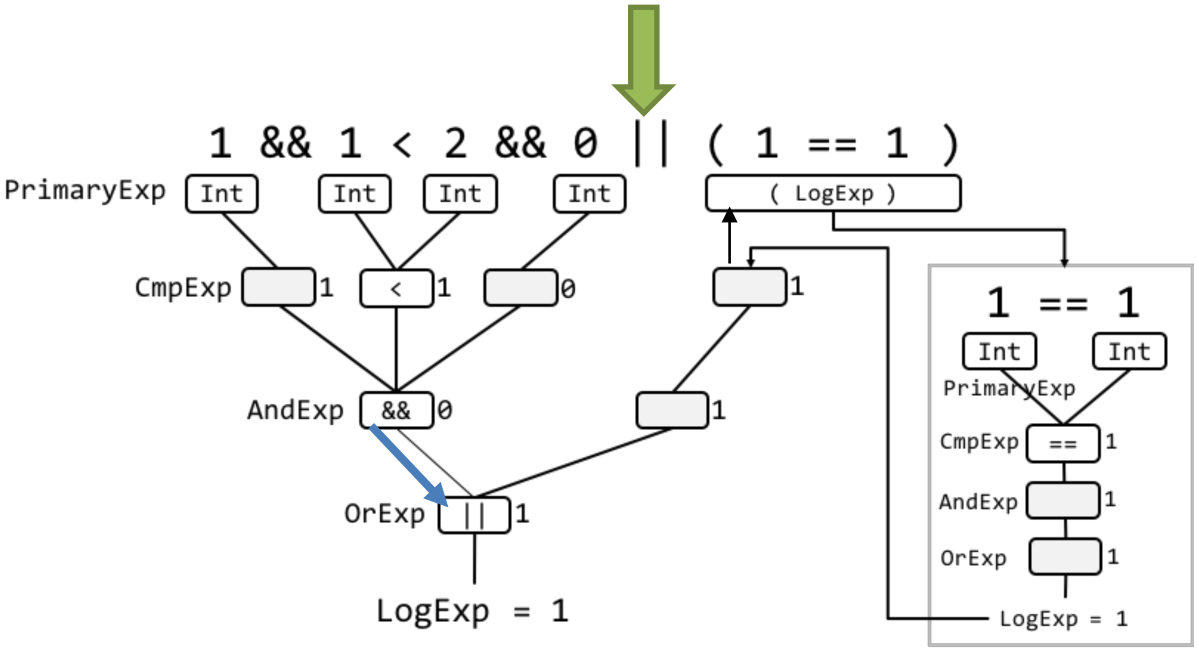
And so on.



Parser flow

10. `AndExp` re-evaluates based on the last operand and returns its result to `OrExp`.
11. `OrExp` now has its first operand, and an impending “||”.

And so on.



RegEx Primer

```
Variable := [a-zA-z][a-zA-z0-9]*     First char is any letter a-z (upper or lower case),
                                     then zero or more letters or digits.
                                     Examples: “x”, “XY”, “x123”, “abc123def”

Float := -?[0-9]+\.[0-9]+f           Optional (zero or one) minus sign ’-’,
                                     then one or more digits,
                                     then dot (written ”\.”),
                                     then one or more digits.
                                     Ends with ’f’.
                                     Examples: ”1.0f”, ”-00.11f”

Int := -?[0-9]+                      Optional (zero or one) minus sign ’-’,
                                     then one or more digits.
                                     Examples: ”1”, ”234”, ”-567”, ”0089”
```

Från cppreference.com ☺

Some people, when confronted with a problem, think “I know, I'll use regular expressions”. Now they have two problems.

L6

L6: Interpreter

- Implement an interpreter for the *M@* language
  - **Math expressions**, and some additional statements
    - Structure similar to the interpreter in this lecture
  - Support for **variables**: use of / assignment to
  - Read code from a file
  - Tokenize, parse etc
  - Print results to an output stream
- See handout for more details.

Grammar for L6

```
M@ Grammar for L6

Stmt:= ConfigStmt | AssgStmt | PrintStmt

ConfigStmt:= config [ dec | hex | bin ]
AssgStmt:= Variable = MathExp
PrintStmt := print MathExp

MathExp := SumExp
SumExp := ProductExp [ + ProductExp | – ProductExp ]*
ProductExp := PrimaryExp [ * PrimaryExp | / PrimaryExp ]*
PrimaryExp := Int | Variable | ( MathExp )

Variable := [a-zA-z][a-zA-z0-9]*
Int := -?[0-9]+
```

DEMO