Compilers 101

Compiler frontend

Previously…

- Preprocessing
- Lexical analysis
- Syntax analysis
- Semantic analysis
- IR Generation
- IR Optimization
- Optimization
- Code generation

Frontend

Middle-end

Backend

Preprocessor

Preprocessing

Lexical analysis

Syntax analysis

Semantic analysis

IR Generation

IR Optimization

Optimization

Code generation

Frontend

Middle-end

Backend

Preprocessor

The preprocessor is a tool that processes your source code before compilation. It handles directives for preprocessing, such as #include, #define, and conditional compilation.

Key Features:

- File Inclusion (#include): Incorporates the contents of a file into the source code, typically used for header files.
- Macro Definition (#define): Defines macros, which are snippets of code that are given a name. Whenever the name is used, it is replaced by the content of the macro.
- Conditional Compilation: Allows compiling code selectively based on conditions evaluated by the preprocessor (#ifdef, #ifndef, #if, #elif, #else, #endif).
- Error Directive (#error): Generates an error from a specified location in your code, useful for flagging incorrect conditions during preprocessing.
- Pragma directives (#pragma): Issues special commands to the compiler, such as optimization levels or code layout suggestions.

Preprocessor example

Input: C/C++ source code
Output: Preprocessed C/C++ source code

```
#define PI 3.14

float area(float r) {
  return PI * r * r;
}
```

```
root@32b489cda6d9:/tmp# clang++ -E preproc.cpp
# 1 "preproc.cpp"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 400 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "preproc.cpp" 2


float area(float r) {
  return 3.14 * r * r;
}
```

https://godbolt.org/z/zaacW8oGb

Why preprocessor is needed?

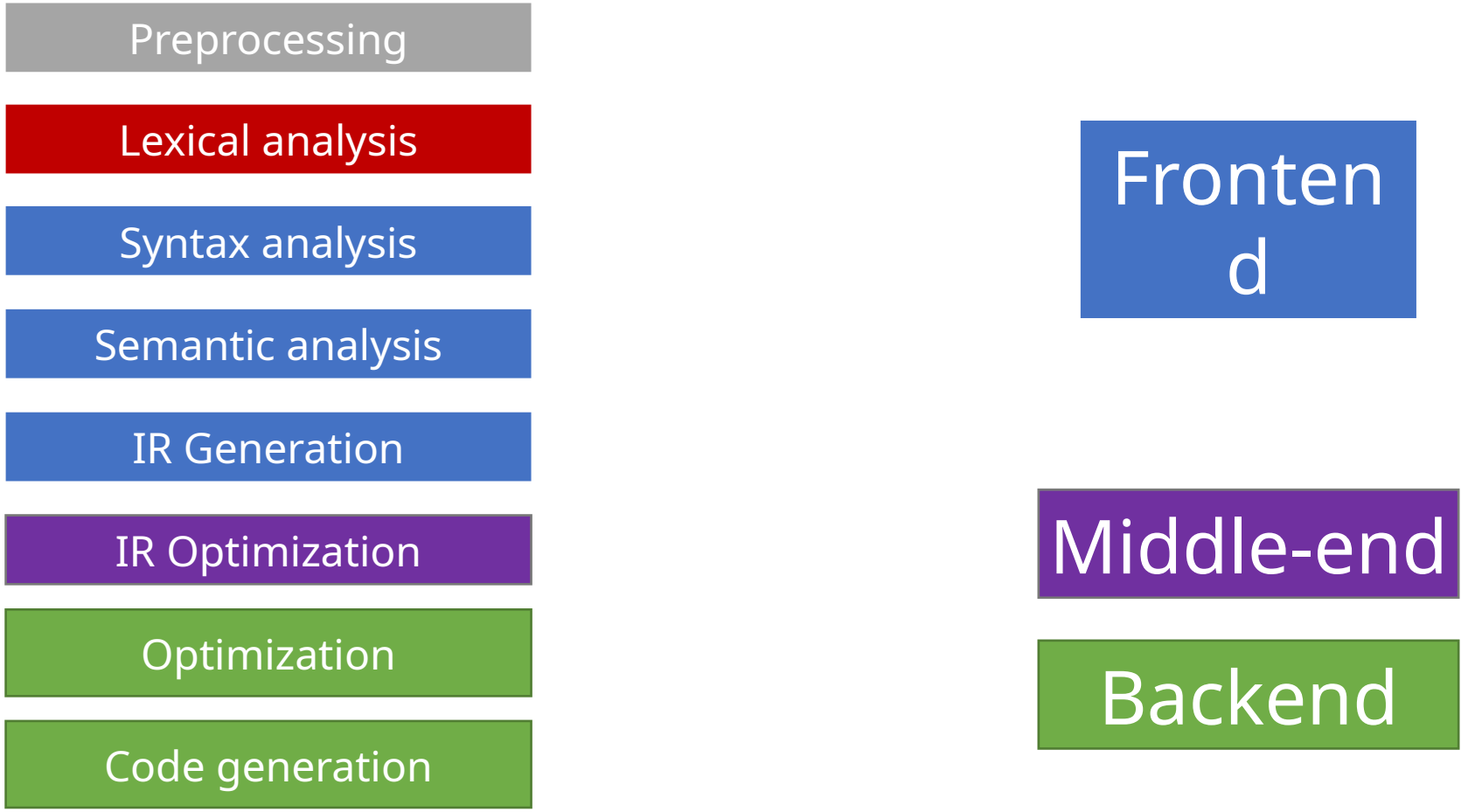
Importance:

- Simplifies code by allowing file inclusion and macro expansions.
- Enables platform-specific compilation through conditional directives.
- Facilitates code maintenance and readability with organized, reusable code blocks; enables static polymorphism and other aspects of meta-programming

Use cases:

- Defining compile-time constants.
- Conditional compilation for cross-platform support.
- Simplifying complex expressions or code snippets for readability and reusability.

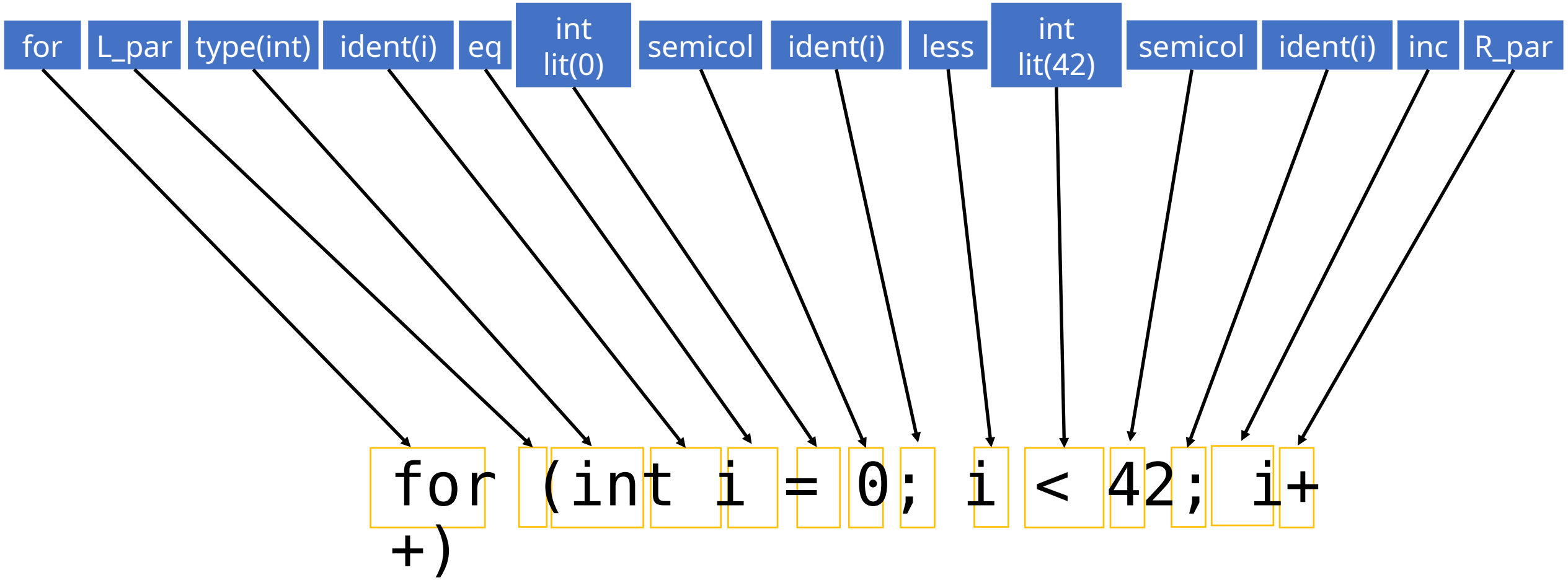
Lexical analysis



Goals of lexical analysis

- Convert physical representation to machine-readable sequence of **tokens**
  - Tokens are the smallest units in the source code, such as keywords, identifiers, literals, operators, and punctuation symbols (like commas and semicolons).
- Associate each token with a **lexeme**
  - E.g., 42 is an integer literal (**token**) with the value of 42 (**lexeme**)
  - In essence, a lexeme is the textual string in the source code, while a token is a structured object that represents a categorized lexeme along with its classification
- Tokens may have extra attributes
- The result will be then used to build AST

Basics


for
L_par
type(int)
ident(i)
eq
int lit(0)
semicol
ident(i)
less
int lit(42)
semicol
ident(i)
inc
R_par
for (int i = 0; i < 42; i++)

Choosing the right tokens

- Depends on your language
- Typically
  - Keywords
  - Literals
  - Punctuation

Formal languages

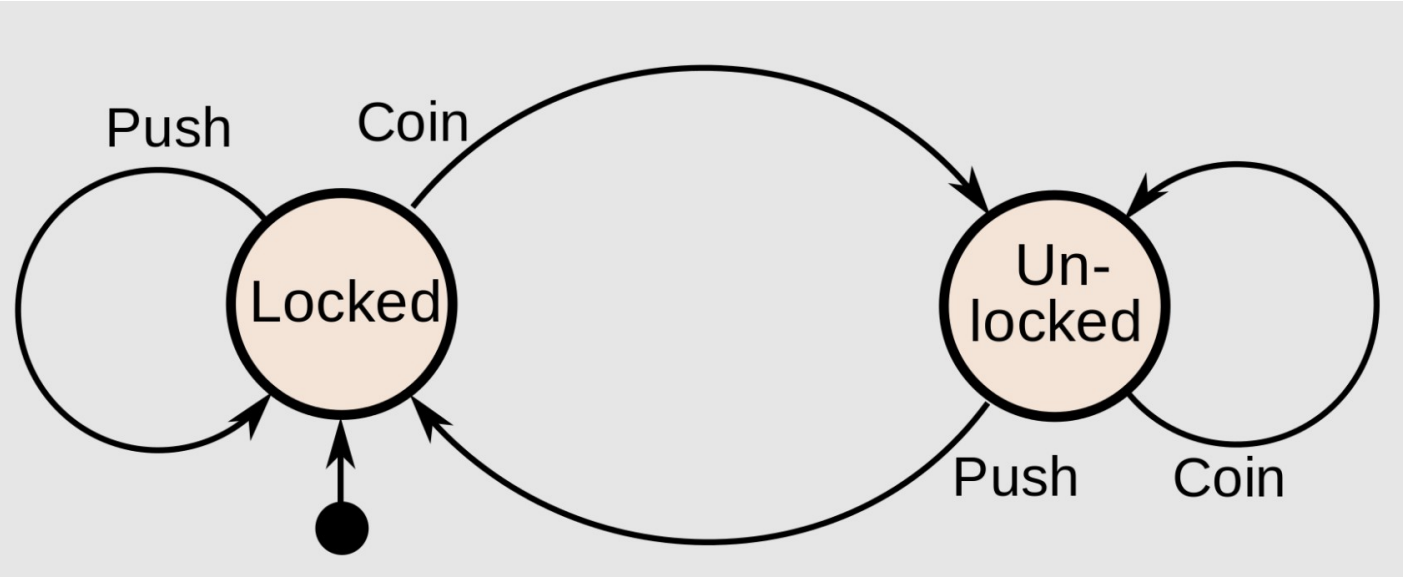
- A formal language consists of words whose letters are taken from an alphabet and are well-formed according to a specific set of rules.
- Examples
  - Alphabet $\Sigma = \{a, b\}$, Language = Σ^* - the set of all words over alphabet
  - Alphabet $\Sigma = \{a\}$, Language = $\{a\}^* = \{a^n\}$ - where n ranges over the natural numbers and "a^n" means "a" repeated n times (this is the set of words consisting only of the symbol "a");

Regular expressions

- Regular expressions are a family of descriptions that can be used to capture regular languages
- There’s a set of operations on regular expressions (concatenation, alternation, Kleene star: concatenation of zero or more strings from the set)
- There are multiple standards to describe the syntax of regular expressions

Matching regular expressions

- Regular expressions can be implemented as finite-state machines
- A finite-state machine is an abstract machine that can be in exactly one of a finite number of states



https://en.wikipedia.org/wiki/Finite-state_machine

Ambiguity resolution

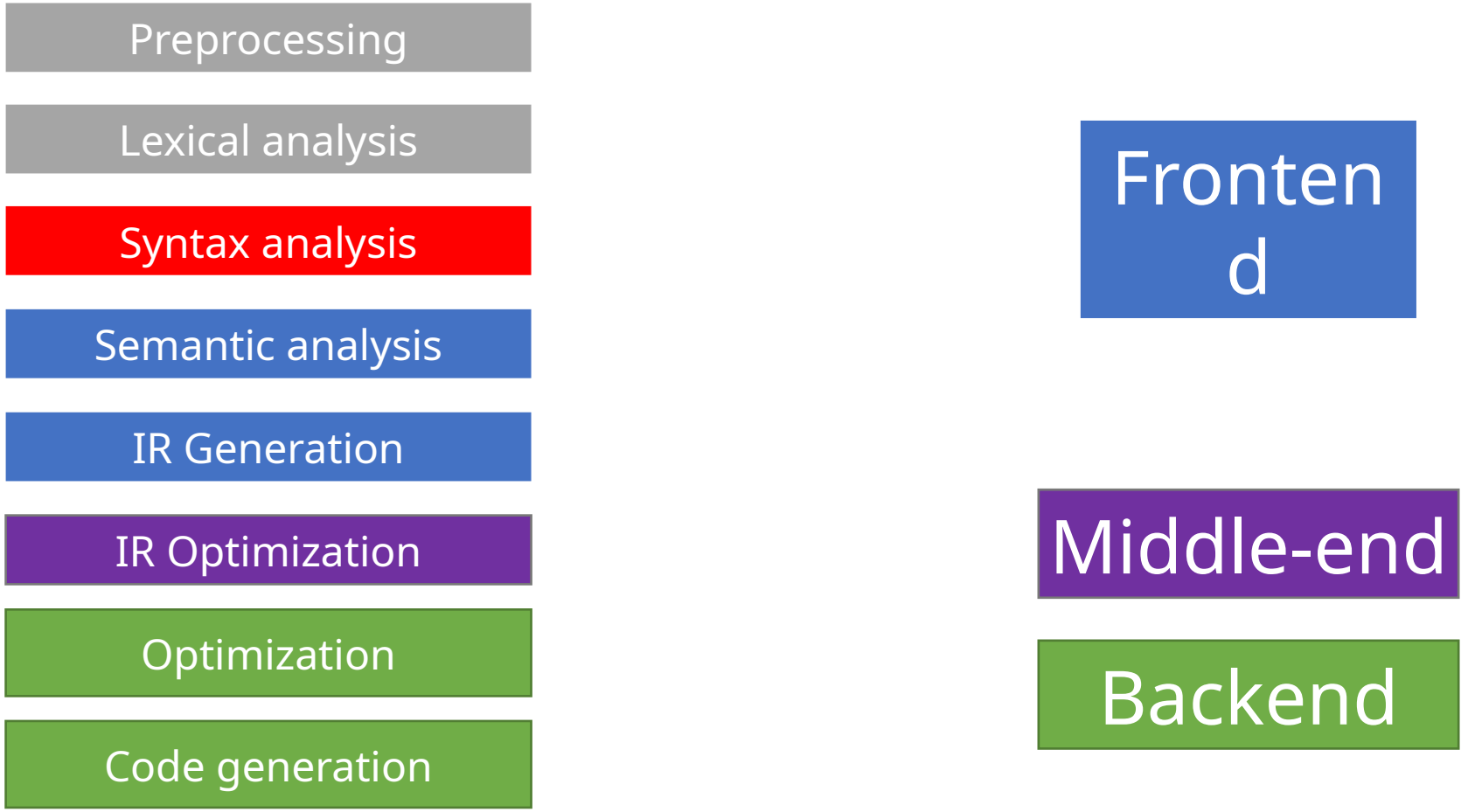
- Consider we have words do and double. How to assign tokens?
- Rules
  - Left-to-right scanning
  - Maximal munch – always match the longest occurrence

Lexical challenges

As seen by C++ compiler developers

```cpp
#include <vector>

typedef int A;

void foo() {
    // Am I a template or a right shift operator?
    std::vector<std::vector<int>> vec2;
}
```

Syntax analysis



Grammars

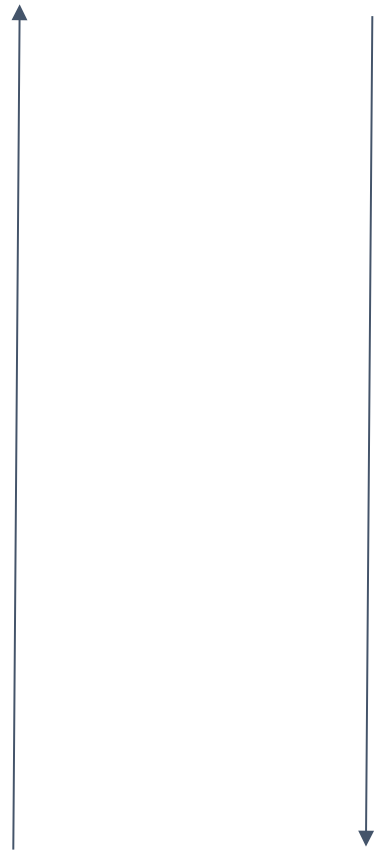
- A formal grammar is a set of rules that describe how to form strings from a language's alphabet that are valid according to the language syntax
- Example:

$$S \rightarrow aSb$$
$$S \rightarrow ab$$

Valid words: ab, aabb, aaabbb, …

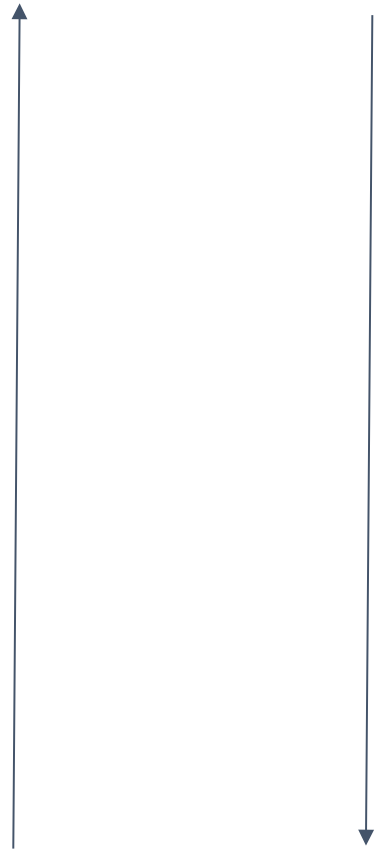
The Chomsky hierarchy

- Recursively enumerable (RE, Type 0)
  - Recursively enumerable subset in the set of all possible words over the alphabet of the language
- Context-sensitive grammars (CSG, Type 1)
  - Left-hand side (LHS) and right-hand side (RHS) may be surrounded by a context of terminal and nonterminal symbols
- Context-free grammars (CFG, Type 2)
  - LHS of each production rule consists only of a single nonterminal symbol
- Regular grammars (Type 3)
  - All production rules have at most 1 nonterminal symbol
  - The symbol is always either at the start or at the end of rule's RHS



The Chomsky hierarchy (examples)

- Expressiveness
- Recognition complexity



- Restrictions on Production Rules
- Ease of Parsing
- Determinism

- Recursively enumerable (RE, Type 0)
  - These grammars can describe any language that can be recognized by a Turing machine, including highly complex or undecidable problems. There are not many practical examples in programming due to their complexity and generality
- Context-sensitive grammars (CSG, Type 1)
  - Ones that can be represented by linear bound automata (some set of limitations applied on Turing machine)
- Context-free grammars (CFG, Type 2)
  - Programming languages
- Regular grammars (Type 3)
  - Regular expressions and finite automata can describe these languages

Our focus is set on highlighted types

Backus–Naur form

- BNF is a metasyntax notation for CFG, often used to describe programming languages
- All rules are written in the following format
  - `<symbol> ::= __expression__`
- Examples:
  - Python grammar is a mixture of EBNF and PEG (parsing expression grammar) https://docs.python.org/3/reference/grammar.html
  - YAML also has BNF description fractions in their spec: https://yaml.org/spec/1.2.2/
  - JSON has unofficial pure BNF description: https://github.com/JetBrains/Grammar-Kit/blob/master/testData/livePreview/Json.bnf
    , official spec also has only fractions of BNF https://www.json.org/json-en.html

BNF example: grammar for basic arithmetic expressions

```
<expression> ::= <term>
               | <expression> "+" <term>
               | <expression> "-" <term>
<term>       ::= <factor>
               | <term> "*" <factor>
               | <term> "/" <factor>
<factor>     ::= <number>
               | "(" <expression> ")"
<number>     ::= <digit>
               | <digit> <number>
<digit>      ::= "0" | "1" | "2" | "3" |
"4"
               | "5" | "6" | "7" | "8" |
"9"
```

Arithmetic expressions conforming given BNF grammar:

- 42
- 3+5
- 3+5*2
- (3+5)*2
- (1+2)*(3-4)/5

From concrete to abstract trees

- As a result of parsing, you will get a parse tree, which is also a concrete syntax tree
  - It contains all the small details about textual representation
- Abstract syntax tree omits those irrelevant details
  - Parenthesis, semicolons, commas…
- To get an abstract tree, you recursively traverse the parsing tree

Different types of parsing

- Top-down parsing
  - Beginning with the start symbol, try to guess the productions to apply to end up at the user's program
- Bottom-up parsing
  - Beginning with user's program, try to apply productions in reverse to convert the program back into the start symbol

LL(k) parsers

- Left-to-right, leftmost derivation
- Top-down parser for context-free languages
- k tokens of lookahead
- Most popular – LL(1) parsers
- Will not be covered in details by this course

Syntax challenges

As seen by C++ compiler developers

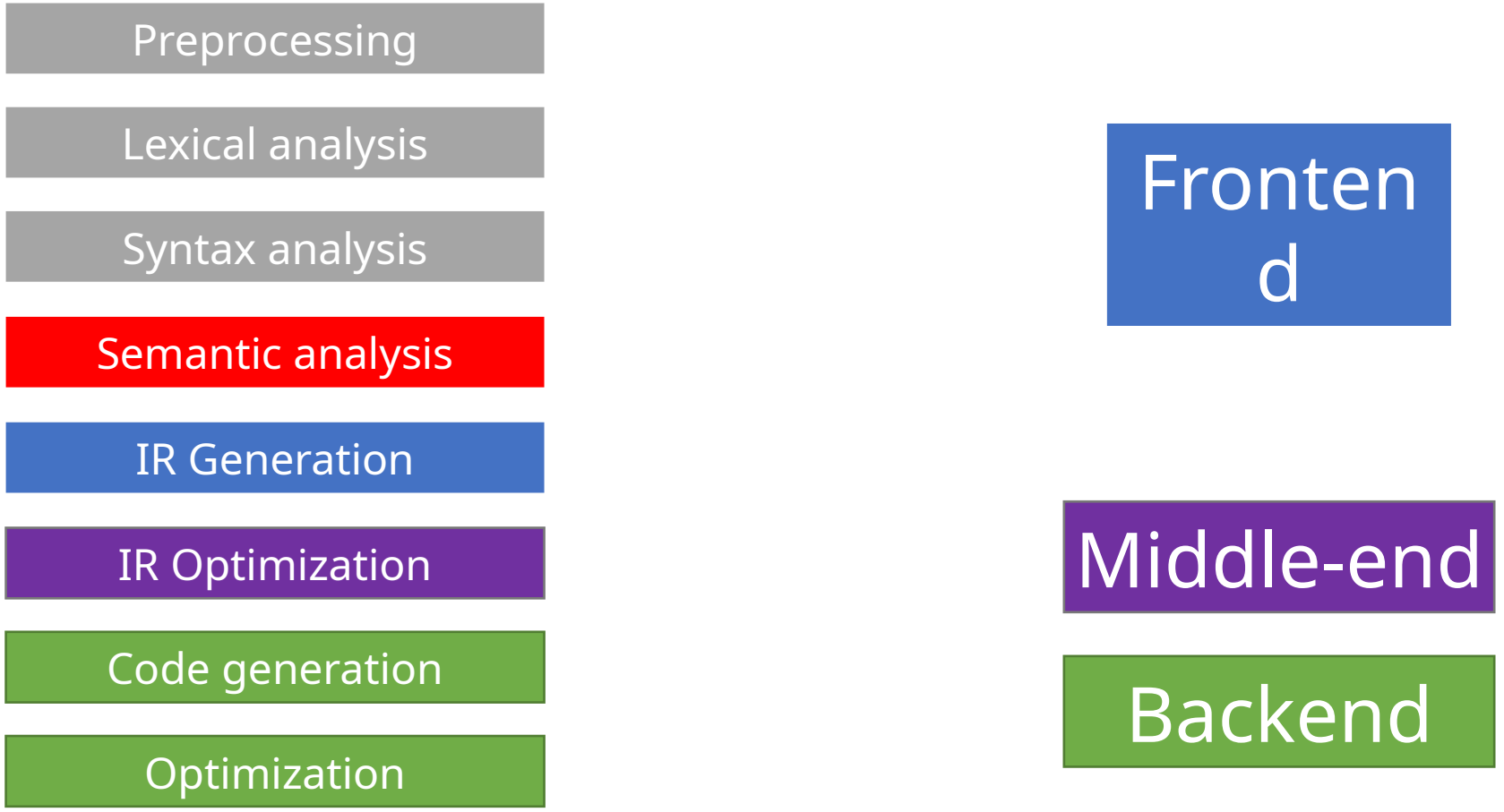
```cpp
#include <vector>

typedef int A;

void foo(int A) {
    // Am I a constructor call or a function declaration?
    std::vector<char> vec1(A);
}
```

A word of advice

- Do not re-invent the wheel!
- This problem has been solved before
- Try to use existing grammar description formats and use parser generators

Pic: https://www.ictworks.org/2015/07/10/stop-reinventing-the-flat-tire-with-custom-software-development/

Semantic analysis

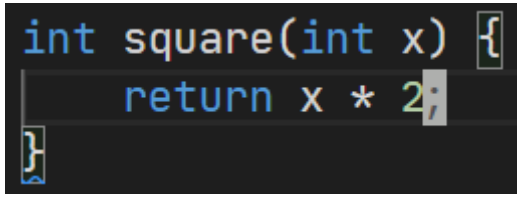


Goals of semantic analysis

- Ensure that the program has a well-defined meaning
  - Variables and symbols are defined before they are used
  - Expressions have the right types
  - …
- Gather useful information for later stages

Validity vs correctness

- Valid syntax does not mean your program is correct
- The following sample is only correct if x is 2



Symbol tables

- Basically, a hash map where key is symbol name and value is its graph node
- Except that it is scoped

```
int a = 10;

void foo(int a) {
    std::cout << a << "\n";
}
```

- Example implementation: https://llvm.org/doxygen/classllvm_1_1ScopedHashTable.html

OOP challenges

- Classes may have parents
- In that case we must look up symbol in base class symbol tables as well
- Actual implementations may be different for particular languages

Multiple inheritance

- Some languages allow multiple base classes
- In that case we must look up the symbol in each base class
- Again, rules depend on language design

IR Generation

Preprocessing

Lexical analysis

Syntax analysis

Semantic analysis

IR Generation

IR Optimization

Optimization

Code generation

Fronten
d

Middle-end

Backend

IR generation

- Consumes AST
- Produces LLVM IR
  - more on that on lecture 04

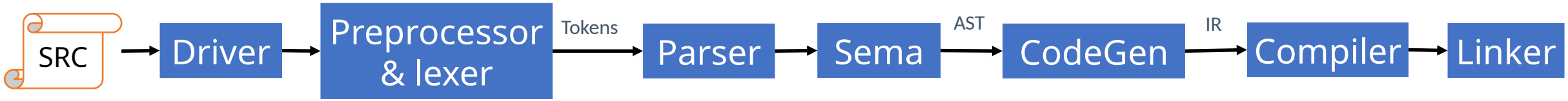
```
int sum(int a, int b) {
    return a + b;
}
```

```
`-FunctionDecl 0xc26d8c8 <<source>:3:1, line:5:1> line:3:5 sum 'int (int, int)'
  |-ParmVarDecl 0xc26d760 <col:9, col:13> col:13 used a 'int'
  |-ParmVarDecl 0xc26d7e0 <col:16, col:20> col:20 used b 'int'
  `-CompoundStmt 0xc26da68 <col:23, line:5:1>
    `-ReturnStmt 0xc26da58 <line:4:5, col:16>
      `-BinaryOperator 0xc26da38 <col:12, col:16> 'int' '+'
        |-ImplicitCastExpr 0xc26da08 <col:12> 'int' <LValueToRValue>
        | `-DeclRefExpr 0xc26d9c8 <col:12> 'int' lvalue ParmVar 0xc26d760 'a' 'int'
        `-ImplicitCastExpr 0xc26da20 <col:16> 'int' <LValueToRValue>
          `-DeclRefExpr 0xc26d9e8 <col:16> 'int' lvalue ParmVar 0xc26d7e0 'b' 'int'
```

```
define dso_local noundef i32 @sum(int, int)(i32 noundef %0, i32 noundef %1) #0 !dbg !10 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, ptr %3, align 4
  call void @llvm.dbg.declare(metadata ptr %3, metadata !16, metadata !DIExpression()), !dbg !17
  store i32 %1, ptr %4, align 4
  call void @llvm.dbg.declare(metadata ptr %4, metadata !18, metadata !DIExpression()), !dbg !19
  %5 = load i32, ptr %3, align 4, !dbg !20
  %6 = load i32, ptr %4, align 4, !dbg !21
  %7 = add nsw i32 %5, %6, !dbg !22
  ret i32 %7, !dbg !23
}

declare void @llvm.dbg.declare(metadata, metadata, metadata) #1

attributes #0 = { mustprogress noinline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-width"="0" "no-trapping-math"="
attributes #1 = { nocallback nofree nosync nounwind speculatable willreturn memory(none) }
```

Overview of Clang

Simplified compiler flow



Clang driver

- Clang supports multiple compatibility modes: with GCC and MSVC
- The part that parses options and issues compile commands is called driver
- Output can be seen with -### flag

```
root@32b489cda6d9:/tmp# clang++ -### sample.cpp
clang version 15.0.0 (https://github.com/intel/llvm 551c07bb5d25b5740327e94cd3d135f6aff7b81e)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /opt/sycl/bin
 "/opt/sycl/bin/clang-15" "-cc1" "-triple" "x86_64-unknown-linux-gnu" "-emit-obj" "-mrelax-all" "--mrelax-relocations" "-disable-free" "-clear-ast-before-backend" "-main-file-name" "sample.cpp" "-mrelocation-model" "static" "-mframe-pointer=all" "-fmath-errno" "-ffp-contract=on" "-fno-rounding-math" "-mconstructor-aliases" "-funwind-tables=2" "-target-cpu" "x86-64" "-tune-cpu" "generic" "-mllvm" "-treat-scalable-fixed-error-as-warning" "-debugger-tuning=gdb" "-fcoverage-compilation-dir=/tmp" "-resource-dir" "/opt/sycl/lib/clang/15.0.0" "-internal-isystem" "/usr/lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9" "-internal-isystem" "/usr/lib/gcc/x86_64-linux-gnu/9/../../../../include/x86_64-linux-gnu/c++/9" "-internal-isystem" "/usr/lib/gcc/x86_64-linux-gnu/9/../../../../include/c++/9/backward" "-internal-isystem" "/opt/sycl/lib/clang/15.0.0/include" "-internal-isystem" "/usr/local/include" "-internal-isystem" "/usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-linux-gnu/include" "-internal-externc-isystem" "/usr/include/x86_64-linux-gnu" "-internal-externc-isystem" "/include" "-internal-externc-isystem" "/usr/include" "-fdeprecated-macro" "-fdebug-compilation-dir=/tmp" "-ferror-limit" "19" "-fgnuc-version=4.2.1" "-fcxx-exceptions" "-fexceptions" "-fcolor-diagnostics" "-faddrsig" "-D__GCC_HAVE_DWARF2_CFI_ASM=1" "-o" "/tmp/sample-282fcc.o" "-x" "c++" "sample.cpp"
 "/usr/bin/ld" "-z" "relro" "--hash-style=gnu" "--eh-frame-hdr" "-m" "elf_x86_64" "-dynamic-linker" "/lib64/ld-linux-x86-64.so.2" "-o" "a.out" "/lib/x86_64-linux-gnu/crt1.o" "/lib/x86_64-linux-gnu/crti.o" "/usr/lib/gcc/x86_64-linux-gnu/9/crtbegin.o" "-L/usr/lib/gcc/x86_64-linux-gnu/9" "-L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib64" "-L/lib/x86_64-linux-gnu" "-L/lib/../lib64" "-L/usr/lib/x86_64-linux-gnu" "-L/usr/lib/../lib64" "-L/opt/sycl/bin/../lib" "-L/lib" "-L/usr/lib" "/tmp/sample-282fcc.o" "-lstdc++" "-lm" "-lgcc_s" "-lgcc" "-lc" "-lgcc_s" "-lgcc" "/usr/lib/gcc/x86_64-linux-gnu/9/crtend.o" "/lib/x86_64-linux-gnu/crtn.o"
```

https://godbolt.org/z/rWhqnzhT7

Preprocessor

GCC:

```
gcc -E in.cpp
```

GNU C compiler preprocessor as a standalone binary. You can use it directly to preprocess C files.

```
cpp in.cpp in.i
```

Clang:

```
clang -E in.cpp
```

https://godbolt.org/z/5nzzbE15T

Lexer

- Starts right after preprocessor
- Flags
  - -c – compile only
  - -Xclang -dump-tokens – bypass driver and pass -dump-tokens to FE

```
int square(int x) {
  return x * x;
}
```

```
root@32b489cda6d9:/tmp# clang++ -c -Xclang -dump-tokens sample.cpp
int 'int'        [StartOfLine]  Loc=<sample.cpp:1:1>
identifier 'square'      [LeadingSpace] Loc=<sample.cpp:1:5>
l_paren '('              Loc=<sample.cpp:1:11>
int 'int'                Loc=<sample.cpp:1:12>
identifier 'x'   [LeadingSpace] Loc=<sample.cpp:1:16>
r_paren ')'              Loc=<sample.cpp:1:17>
l_brace '{'      [LeadingSpace] Loc=<sample.cpp:1:19>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<sample.cpp:2:3>
identifier 'x'   [LeadingSpace] Loc=<sample.cpp:2:10>
star '*'         [LeadingSpace] Loc=<sample.cpp:2:12>
identifier 'x'   [LeadingSpace] Loc=<sample.cpp:2:14>
semi ';'                 Loc=<sample.cpp:2:15>
r_brace '}'      [StartOfLine]  Loc=<sample.cpp:3:1>
eof ''          Loc=<sample.cpp:3:2>
```

https://godbolt.org/z/7bPxao3Pz

Lexer internals

- Token kinds defined in [clang/include/clang/Basic/TokenKinds.def]

```
KEYWORD(float                       , KEYALL)
KEYWORD(for                         , KEYALL)
KEYWORD(goto                        , KEYALL)
KEYWORD(if                          , KEYALL)
KEYWORD(inline                      , KEYC99|KEYCXX|KEYGNU)
KEYWORD(int                         , KEYALL)
KEYWORD(_ExtInt                     , KEYALL)
```

- Consumed by [clang/include/clang/Parse/Parser.h]

Parser

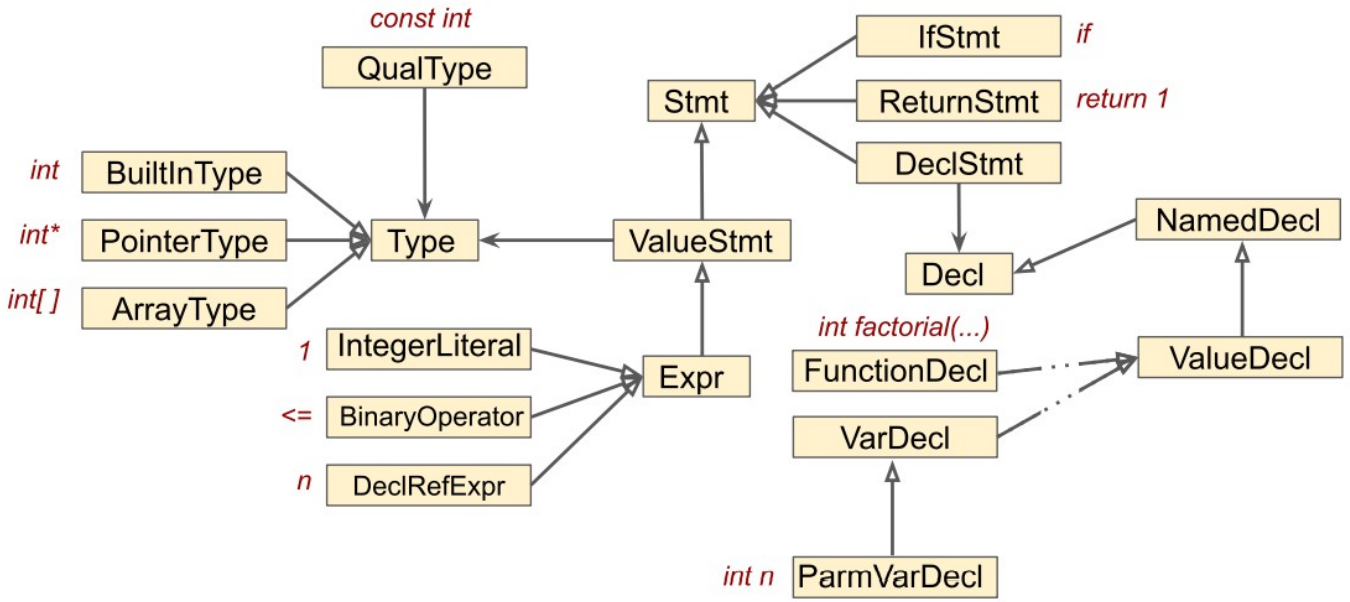
- Handwritten recursive-descent parser
- Tentative parsing by looking at the tokens ahead
  - the parser can “peek” ahead and choose the best path without committing prematurely
- Try to recover from errors as much as possible
  - Prevents cascading errors by synchronizing at known recovery points
- Suggest FIX-IT hints
  - Analyzes error contexts to suggest potential fixes (e.g., missing semicolons, unmatched parentheses)
  - Aims to improve developer productivity by providing actionable guidance

Sema

- Tightly coupled with parser
- Verifies AST before it is sent out to other clients
- Biggest client of diagnostics subsystem
  - Most warnings and errors come out of here
- Sema verifies that the syntax that passes the parsing stage also makes sense according to the rules of the language
  - This involves checking for type errors, ensuring that variables are declared before use, enforcing scope rules, and more.

AST

AST Nodes



See full diagram: `https://clang.llvm.org/doxygen/inherits.html`

arm

https://llvm.org/devmtg/2019-10/slides/ClangTutorial-Stulova-vanHaastregt.pdf

https://godbolt.org/z/6Wq38MbPj

Diagnostics

- Defined in Diagnostic*Kind.td

```
let CategoryName = "Parse Issue" in {

def err_expected : Error<"expected %0">;
def err_expected_either : Error<"expected %0 or %1">;
def err_expected_after : Error<"expected %1 after %0">;

def err_param_redefinition : Error<"redefinition of parameter %0">;
def warn_method_param_redefinition : Warning<"redefinition of method parameter %0">;
def warn_method_param_declaration : Warning<"redeclaration of method parameter %0">,
  InGroup<DuplicateArgDecl>, DefaultIgnore;
```

- Diagnostic engine tries to render output in human-readable format
  - Not always successful, especially in heavily templated code

TableGen language

- The LLVM TableGen language is a domain-specific language used within the LLVM project framework.
- Key points:
  - Data Description Tool
  - Code Generation
  - Extensible with writing additional code generators
  - Language is widely used in LLVM frontend and backend, MLIR and other components
- Language reference: https://llvm.org/docs/TableGen/ProgRef.html

TableGen example

Here is the example:

- code snippet that could be used to add new attribute for clang

```
[[clang::my_attr]]
void foo() {

}

__attribute__((my_attr
))
void foo() {

}
```

```
// In File: clang/include/clang/Basic/Attr.td

include "clang/Basic/AttrDocs.td"

def MyAttr : InheritableAttr {
  let Documentation = [Undocumented];
  // This specifies the attribute is for
functions.
  let Subjects = SubjectList<[Function],
ErrorDiag>;
  // The spelling of the attribute in the source
code.
  let Spellings = [CXX11<"clang", "my_attr">];
  // The attribute does not take any arguments.
  let Args = [];
}
```

Lab assignment #0

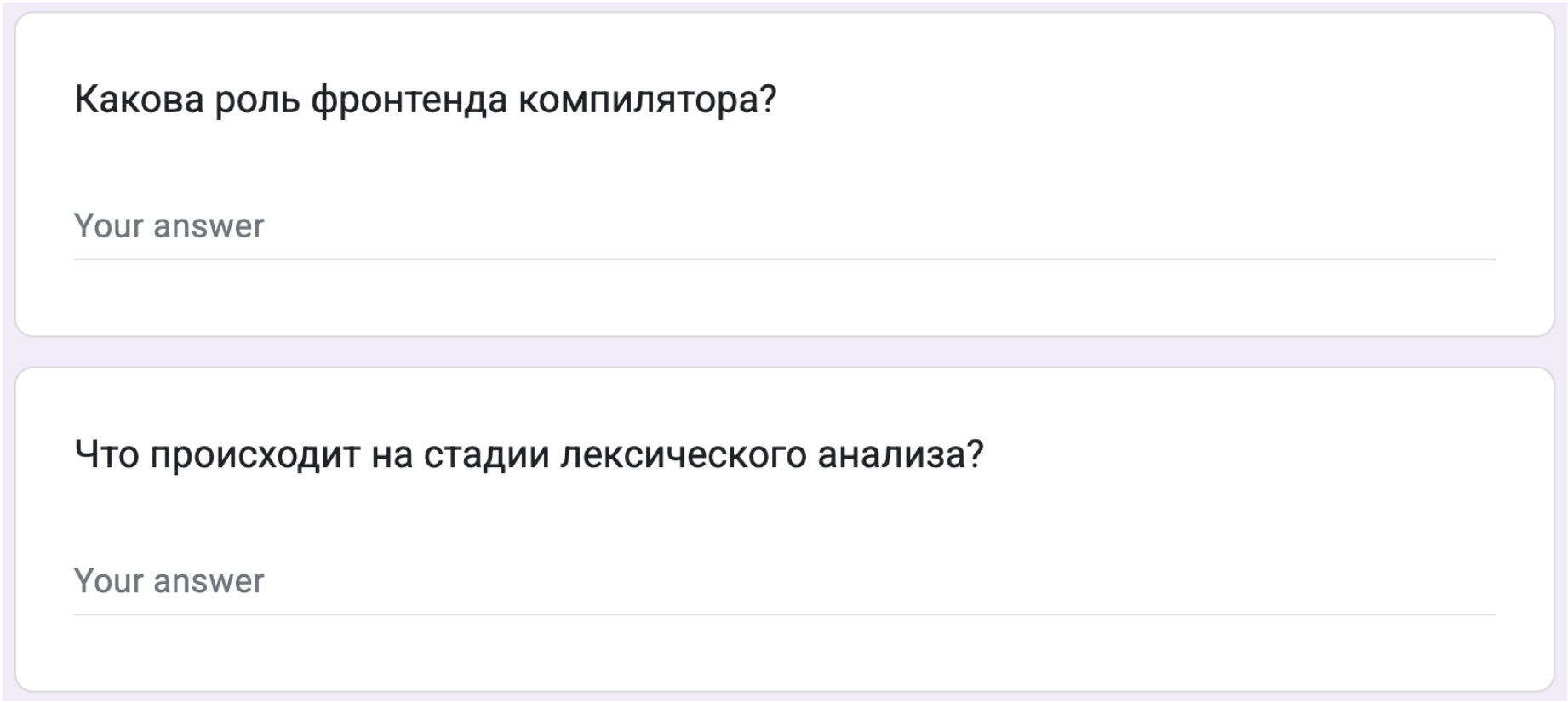
- Try to build LLVM and Clang from source code and run tests
  - Source code: https://github.com/NN-complr-tech/compiler-course-2025
  - Follow build instructions: https://llvm.org/docs/CMake.html
  - Make sure to run check-clang and check-llvm targets, e.g.
    - cmake --build . --target check-clang
- Better done on Linux or macOS. Windows is also OK.
- Goal: learn how to work with basic LLVM infrastructure
- Build may take several hours
- Build usually requires 2GB of RAM per 1 thread. One can reduce number of threads with *-j n* flag, where n is a number of threads

Next time…

- Look closer at practical usages of clang AST
  - Working with AST
  - Clang plugins
  - Clang-tidy and static analysis

Test

https://forms.gle/sA5VYne7p9UyHdbH9



Backup: me@gooddoog.ru

Extra materials

- An overview of Clang, LLVM Dev 2019 - https://www.youtube.com/watch?v=5kkMpJpIGYU
- What is C++, Chandler Carruth, Titus Winters - https://www.youtube.com/watch?v=LJh5QCV4wDg
- Гладкий А. В. Формальные грамматики и языки (RU)
- Хопкрофт Дж., Мотвани Р., Ульман Дж. Введение в теорию автоматов, языков и вычислений (RU)
- Hacking on clang - https://clang.llvm.org/hacking.html
- Lessons in TableGen - https://www.youtube.com/watch?v=45gmF77JFBY
- The Clang AST - a tutorial - https://www.youtube.com/watch?v=VqCkCDFLSsc