1 Introduction and Interpretation

1.1 Organization

[Slide 2] Module "Code Generation for Data Processing"

Learning Goals

- Getting from an intermediate code representation to machine code
- Designing and implementing IRs and machine code generators
- Apply for: JIT compilation, query compilation, ISA emulation

Prerequisites

- Computer Architecture, Assembly — ERA, GRA/ASP
- Databases, Relational Algebra — GDB
- Beneficial: Compiler Construction, Modern DBs

[Slide 3] Topic Overview

Introduction

- Introduction and Interpretation
- Compiler Front-end

Intermediate Representations

- IR Concepts and Design
- LLVM-IR
- Analyses and Optimizations

Compiler Back-end

- Instruction Selection
- Register Allocation
- Linker, Loader, Debuginfo

Applications

- JIT-compilation + Sandboxing
- Query Compilation
- Binary Translation

[Slide 4] Lecture Organization

- Lecturer: Dr. Alexis Engelke `engelke@in.tum.de`
- Time slot: Thu 10-14, 02.11.018
- Material: `https://db.in.tum.de/teaching/ws2425/codegen/`

Exam

- Written exam, 90 minutes, **no retake**, date TBD
- (Might change to oral on very low registration count)

[Slide 5] Exercises

- Regular homework, often with programming exercise
- Submission via POST request (see assignments)
  - Grading with $\{*, +, \sim, -\}$, feedback on best effort
- Exercise session modes:
  - Present and discuss homework solutions
  - Hands-on programming or analysis of systems (needs laptop)

Grade Bonus

- Requirement: $N - 2$ "sufficiently working" homework submissions **and** one presentations of homework in class (depends on submission count)
- Bonus: grades in $[1.3; 4.0]$ improved by $0.3/0.4$

[Slide 6] Why study compilers?

- Critical component of every system, functionality and performance
  - Compiler mostly *alone* responsible for using hardware well
- Brings together many aspects of CS:
  - Theory, algorithms, systems, architecture, software engineering, (ML)
- New developments/requirements pose new challenges
  - New architectures, environments, language concepts, ...
- High complexity!

[Slide 7] Compiler Lectures @ TUM

Compiler Construction IN2227, SS, THEO	Program Optimization IN2053, WS, THEO	Virtual Machines IN2040, SS, THEO
Front-end, parsing, semantic analyses, types	Analyses, transformations, abstract interpretation	Mapping programming paradigms to IR/bytecode

Programming Languages CIT3230000, WS	Code Generation CIT3230001, WS
Implementation of advanced language features	Back-end, machine code generation, JIT comp.

[Slide 8] Why study code generation?

- Frameworks (LLVM, . . .) exist and are comparably good, but often not good enough (performance, features)
  - Many systems with code gen. have their own back-end
  - E.g.: V8, WebKit FTL, .NET RyuJIT, GHC, Zig, QEMU, Umbra, . . .
- Machine code is not the only target: bytecode
  - Often used for code execution
  - E.g.: V8, Java, .NET MSIL, BEAM (Erlang), Python, MonetDB, eBPF, . . .
  - Allows for flexible design
  - But: efficient execution needs machine code generation

[Slide 9] Proebsting's Law

> "Compiler advances double computing power every *18* years."
>
> – Todd Proebsting, 1998[a]

[a] `http://proebsting.cs.arizona.edu/law.html`

- Still optimistic; depends on number of abstractions

The performance increases compilers can make on existing code are typically low. However, optimizing compilers gain more abilities in simplifying needlessly complex code, enabling the use of more abstractions and therefore higher level code. These abstractions are removed/optimized during compilation, enabling languages to promote these as *zero-cost abstractions*. They do, however, have a cost: compile times.

Also note that some of these "zero-cost" abstractions actually *do* have some runtime cost. For example, the mere possibility of C++ exceptions can cause less efficient machine code and might prevents optimizations due to the more complex control flow possibilities.

1.2 Overview

[Slide 10] Motivational Example: Brainfuck

- Turing-complete esoteric programming language, 8 operations
  - Input/output: `. ,`
  - Moving pointer over infinite array: `< >`
  - Increment/decrement: `+ -`
  - Jump to matching bracket if (not) zero: `[ ]`

```
++++++[->++++++<]>.
```

- Execution with pen/paper? ☹

[Slide 11] Program Execution

Programs

- High flexibility (possibly)
- Many abstractions (typically)
- Several paradigms

Hardware/ISA

- Low-level interface
- Few operations, imperative
- "Not easy" to write

[Slide 12] Motivational Example: Brainfuck – Interpretation

- Write an interpreter!

```
unsigned char state[10000];
unsigned ptr = 0, pc = 0;
while (prog[pc])
  switch (prog[pc++]) {
  case '.': putchar(state[ptr]); break;
  case ',': state[ptr] = getchar(); break;
  case '>': ptr++; break;
  case '<': ptr--; break;
  case '+': state[ptr]++; break;
  case '-': state[ptr]--; break;
  case '[': state[ptr] || (pc = matchParen(pc, prog)); break;
  case ']': state[ptr] && (pc = matchParen(pc, prog)); break;
  }
```

[Slide 13] Program Execution

Compiler

- Translate program to other lang.
- Might optimize/improve program

- C, C++, Rust → machine code
- Python, Java → bytecode

Interpreter

- Directly execute program
- Computes program result

- Shell scripts, Python bytecode, machine code (conceptually)

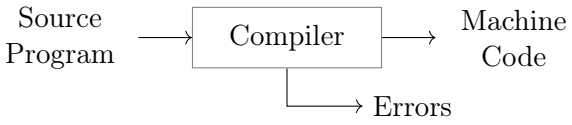
Multiple compilation steps can precede the "final interpretation"

1.3 High-Level Structure of Compilers

[Slide 14] Compilers

- Targets: machine code, bytecode, or other source language
- Typical goals: better language usability and performance
  - Make lang. usable at all, faster, use less resources, etc.
- Constraints: specs, resources (comp.-time, etc.), requirements (perf., etc.)
- Examples:
  - "Classic" compilers source → machine code
  - JIT compilation of JavaScript, WebAssembly, Java bytecode, ...
  - Database query compilation
  - ISA emulation/binary translation

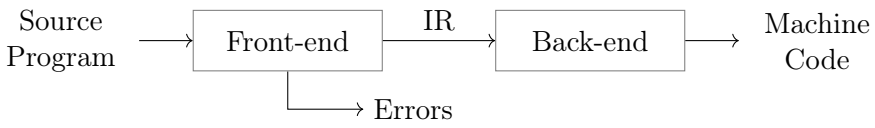
[Slide 15] Compiler Structure: Monolithic



- Inflexible architecture, hard to retarget

> Some languages like C are designed to be compilable in a single pass without building any intermediate representation of the code between source and assembly. Single-pass compilers exist, but often have very limited possibilities to transform the code. They might not even know basic code properties, e.g., the size of the stack frame, during compilation of a function.

[Slide 16] Compiler Structure: Two-phase architecture



Front-end

- Parses source code
- Detect syntax/semantical errors
- Emit *intermediate representation* ⇝ encode semantics/knowledge
- Typically: $\mathcal{O}(n)$ or $\mathcal{O}(n \log n)$

Back-end

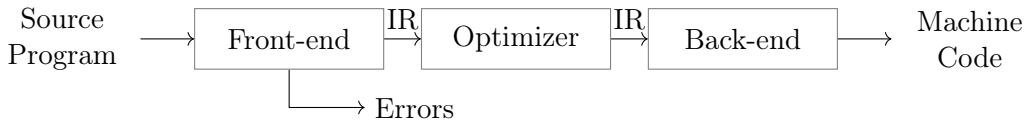
- Translate IR to target architecture
- Can assume valid IR (⇝ no errors)
- Possibly one back-end per arch.

- Contains $\mathcal{NP}$-complete problems

After parsing, all information is encoded in the IR, including references to source code constructs for debugging support. The input source code is (at least conceptually) no longer needed.

In practice, there are very rare cases where the back-end can also raise errors. This can happen, for example, when some very architecture-specific constraints might be hard to verify during parsing (e.g., inline assembly constraints in combination with available registers).

[Slide 17] Compiler Structure: Three-phase architecture



- Optimizer: analyze/transform/rewrite program inside IR

- Conceptual architecture: real compilers typically much more complex
  - Several IRs in front-end and back-end, optimizations on different IRs
  - Multiple front-ends for different languages
  - Multiple back-ends for different architectures

Example Clang/LLVM (will be covered in more detail later): Clang parses the input into an abstract syntax tree (IR 1), uses this for semantic analyses; then Clang transforms the code into LLVM-IR (IR 2), which is primarily used for optimization; then the LLVM back-end transforms the code further into LLVM's Machine IR (IR 3), executes some low-level optimizations and register allocation there; the assembly printer of the back-end then lowers the code further to LLVM's machine code representation (IR 4), before finally emitting machine code. Some optimizations inside this pipeline, e.g. vectorization, might even build further representation of the code.

Why are compilers using so many different code representations? Different transformations work best at different abstraction levels. Diagnosing unused variables, for example, requires information about the source code. Optimization of arithmetic computations is easier in a data-flow-focused representation, where no explicit variables exist. Low-level modifications, like folding operations into complex addressing modes of the ISA, need a code representation where ISA instructions are already present.

[Slide 18] Compiler Front-end

1. Tokenizer: recognize words, numbers, operators, etc. $\mathcal{R}e$
   - Example: `a+b*c` → `ID(a) PLUS ID(b) TIMES ID(c)`

2. Parser: build (abstract) syntax tree, check for syntax errors $\mathcal{CFG}$
   - Syntax Tree: describe grammatical structure of complete program Example: `expr("a", op("+"), expr("b", op("*"), expr("c"))`
   - Abstract Syntax Tree: only relevant information, more concise Example: `plus("a", times("b", "c"))`
3. Semantic Analysis: check types, variable existence, etc.
4. IR Generator: produce IR for next stage
   - This might be the AST itself

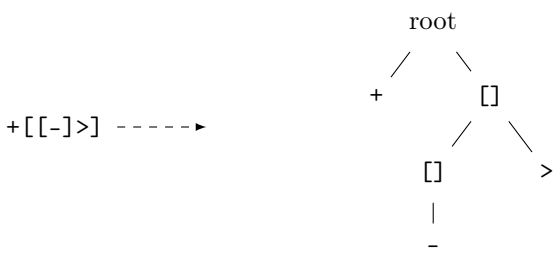
[Slide 19] Compiler Back-end

1. Instruction Selection: map IR operations to target instructions
   - Use target features: special insts., addressing modes, ...
   - Still using virtual/unlimited registers
2. Instruction Scheduling: optimize order for target arch.
   - Start memory/high-latency earlier, etc.
   - Requires knowledge about micro-architecture
3. Register Allocation: map values to fixed register set/stack
   - Use available registers effectively, minimize stack usage

1.4 Interpretation

[Slide 20] Motivational Example: Brainfuck – Front-end

- Need to skip comments
- Bracket searching is expensive/redundant
- Idea: "parse" program!
- Tokenizer: yield next operation, skipping comments
- Parser: find matching brackets, construct AST



[Slide 21] Motivational Example: Brainfuck – AST Interpretation

- AST can be interpreted recursively

```
struct node { char kind; unsigned cldCnt; struct node* cld; };
struct state { unsigned char* arr; size_t ptr; };
void donode(struct node* n, struct state* s) {
  switch (n->kind) {
  case '+': s->arr[s->ptr]++; break;
  // ...
  case '[': while (s->arr[s->ptr]) children(n, s); break;
  case 0: children(n, s); break; // root
  }
}
void children(struct node* n, struct state* s) {
  for (unsigned i = 0; i < n->cldCnt; i++) donode(n->cld + i, s);
}
```

[Slide 22] Motivational Example: Brainfuck – Optimization

- Inefficient sequences of +/-/</> can be combined
  - Trivially done when generating IR
- Fold patterns into more high-level operations

Look at some Brainfuck programs. Which patterns are beneficial to fold?

[Slide 23] Motivational Example: Brainfuck – Optimization

- Fold offset into operation
  - `right(2) add(1)` = `addoff(2, 1) right(2)`
  - Also possible with loops
- Analysis: does loop move pointer?
  - Loops that keep position intact allow more optimizations
  - Maybe distinguish "regular loops" from arbitrary loops?
- Get rid of all "effect-less" pointer movements
- Combine arithmetic operations, disambiguate addresses, etc.

[Slide 24] Motivational Example: Brainfuck – Bytecode

- Tree is nice, but rather inefficient ⇝ flat and compact bytecode
- Avoid pointer dereferences/indirections; keep code size small
- Maybe dispatch two instructions at once?
  - `switch (ops[pc] | ops[pc+1] << 8)`
- Superinstructions: combine common sequences to one instruction

> Dispatching multiple instructions at once can be problematic due to the explosion of cases that need to be implemented (often results in large jump tables and lots of code

with resulting cache misses and branch mispredictions). Often, it is advisable to not always switch over multiple neighbored instructions, but instead combine common sequences into superinstructions.

[Slide 25] Motivational Example: Brainfuck – Threaded Interpretation

- Simple `switch`–`case` dispatch has lots of branch misses
- Threaded interpretation: at end of a handler, jump to next op

```
struct op { char op; char data; };
struct state { unsigned char* arr; size_t ptr; };
void threadedInterp(struct op* ops, struct state* s) {
    static const void* table[] = { &&CASE_ADD, &&CASE_RIGHT, };
#define DISPATCH do { goto *table[(++pc)->op]; } while (0)

    struct op* pc = ops;
    DISPATCH;

CASE_ADD: s->arr[s->ptr] += pc->data; DISPATCH;
CASE_RIGHT: s->arr += pc->data; DISPATCH;
}
```

With threaded interpretation there is not a single indirect jump instruction inside the dispatcher, but one indirect jump instruction per operation. Each of these indirect jumps then occupies a different branch prediction slot in the CPU. If an operation of type X is typically followed by an operation of type Y, with threaded interpretation the CPU has a much better chance of correctly predicting the dispatch branch to the next operation, because the indirect jump at the end of operation X typically jumps to operation Y. Without threaded interpretation, there would be only a single indirect branch, which is much harder to predict.

Threaded interpretation is especially useful on older and less powerful CPUs. Recent CPUs (e.g., Intel since Skylake, AMD since Zen 3, Apple Silicon) store the history of indirect branches and use this for better prediction. On such processors, threaded interpretation might not improve performance (or gains might be lower).

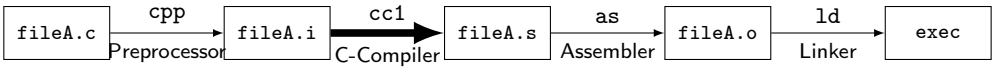
[Slide 26] Fast Interpretation

- Key technique to "avoid" compilation to machine code
- Preprocess program into efficiently executable bytecode
  - Easily identifiable opcode, homogeneous structure
  - Can be linear (fast to execute), but trees also work
  - Match bytecode ops with needed operations $\rightsquigarrow$ fewer instructions
- Perhaps optimize – if it's worth the benefit
  - Fold constants, combine instructions, ...
  - Consider superinstructions for common sequences
- For very cold code: avoid transformations at all

1.5 Context of Compilation

[Slide 27] Compiler: Surrounding – Compile-time

- Typical environment for a C/C++ compiler:



- Calling Convention: interface with other objects/libraries
- Build systems, dependencies, debuggers, etc.
- Compilation target machine (hardware, VM, etc.)

[Slide 28] Compiler: Surrounding – Run-time

- OS interface (I/O, ...)
- Memory management (allocation, GC, ...)
- Parallelization, threads, ...
- VM for execution of virtual assembly (JVM, ...)
- Run-time type checking
- Error handling: exception unwinding, assertions, ...
- Reflection, RTTI

[Slide 29] Motivational Example: Brainfuck – Runtime Environment

- Needs I/O for `.` and `,`
- Error handling: unmatched brackets
- Memory management: infinitely sized array

How to efficiently emulate an infinitely sized array?

[Slide 30] Compilation point: AoT vs. JIT

Ahead-of-Time (AoT)

- All code has to be compiled
- No dynamic optimizations
- Compilation-time secondary concern

Just-in-Time (JIT)

- Compilation-time is critical
- Code can be compiled on-demand
  - Incremental optimization, too
- Handle cold code fast
- Dynamic specializations possible
- Allows for `eval()`

Various hybrid combinations possible

[Slide 31] Introduction and Interpretation – Summary

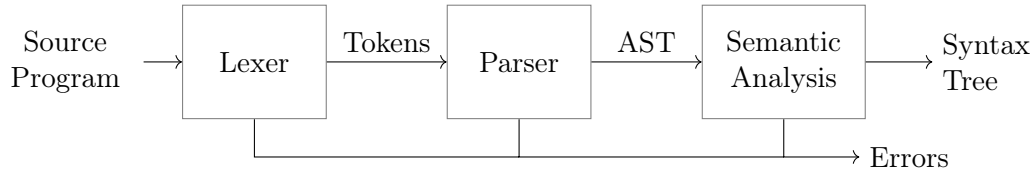
- Compilation vs. interpretation and combinations
- Compilers are key to usable/performant languages
- Target language typically machine code or bytecode
- Three-phase architecture widely used
- Interpretation techniques: bytecode, threaded interpretation, ...
- JIT compilation imposes different constraints

[Slide 32] Introduction and Interpretation – Questions

- What is typically compiled and what is interpreted? Why?
  - PostScript, C, JavaScript, HTML, SQL
- What are typical types of output languages of compilers?
- How does a compiler IR differ from the source input?
- What is the impact of the language paradigm on optimizations?
- What are important factors for an efficient interpreter?
- What are key differences between AoT and JIT compilation?

2 Compiler Front-end

[Slide 34] Compiler Front-end



- Typical architecture: separate lexer, parser, and context analysis
  - Allows for more efficient lexical analysis
  - Smaller components, easier to understand, etc.
- Some languages: preprocessor and macro expansion

2.1 Lexing

[Slide 35] Lexer

- Convert stream of chars to stream of words (*tokens*)
- Detect/classify identifiers, numbers, operators, ...
- Strip whitespace, comments, etc.

`a+b*c` → `ID(a) PLUS ID(b) TIMES ID(c)`

- Typically representable as regular expressions

[Slide 36] Typical Token Kinds

- Punctuators `( ) [ ] { } ; = + += | ||`
- Identifiers `abc123 main`
- Keywords `void int __asm__`
- Numeric constants `123 0xab1 5.7e3 0x1.8p1 09.1f`
- Char constants `'a' u'œ'`
- String literals `"abc\x12\n"`
- Internal `EOF COMMENT UNKNOWN INDENT DEDENT`
  - Comments might be useful for annotations, e.g. `// fallthrough`

> Indentation-based languages like Python need separate tokens for indent/dedent, the indentation level is tracked in the lexer. Parsing numbers may need special care to correctly handle all possible cases of integer and floating-point numbers.

[Slide 37] Lexer Implementation

```
struct Token { enum Kind { IDENT, EOF, PLUS, PLUSEQ, /*...*/ };
  std::string_view v; Kind kind; };
Token next(std::string_view v) {
  if (v.empty()) return Token{v, Token::EOF};
  if (v.starts_with("+=")) return Token{"+="sv, Token::PLUSEQ};
  if (v.starts_with("+")) return Token{"+"sv, Token::PLUS};
  switch (v[0]) {
  case ' ', '\n', '\t': return next(v.substr(1)); // skip whitespace
  case 'a' ... 'z', 'A' ... 'Z', '_': {
    Token t = // ... parse identifer, e.g. using regex
    if (auto kind = isKeyword(t.v)) return Token{*kind, t.v};
    return t;
  }
  case '0' ... '9': // ... parse number
  default: return Token{v.substr(0, 1), Token::ERROR};
  }
}
```

> This is just a minimal and non-optimized implementation to illustrate the concept. Performance-focused implementations do not use explicit regular expressions but write the state machine into code.
>
> The struct `Token` has room for improvement. First, a `string_view` is unnecessarily large with 16 bytes, most tokens are smaller than 2^{16} bytes. Some tracking of the source locations is advisable for attaching diagnostics to their origin inside the code, for example by storing a file ID and the byte offset into the file. By tracking the byte offsets of line breaks, the line number can be reconstructed in $\mathcal{O}(\log n)$ from the byte offset.
>
> Another optimization strategy is string interning, where identifiers are converted into unique integers (or pointers) during parsing. During later phases, comparing interned strings is much more efficient, as it is just an integer/pointer comparison. Another benefit is that the entire input file does not need to be kept in memory during parsing.

[Slide 38] Lexing C??=

```
main() <%
  // yay, this is C99??/
  puts("hi world!");
  puts("what's up??!");
%>
```

Output: `what's up|`

- Trigraphs for systems with more limited encodings/char sets
- Digraphs to provide a more readable alternative...

> Besides digraphs, trigraphs, and the preprocessor, C has another weird property: identifier names can be split by \, which concatenates two lines. It is necessary to construct the "real" identifier first. To simplify memory management in such cases, a bump pointer allocator (allocate large chunks of memory from the OS, then simply bump the end pointer for every allocation) can be useful to store such constructed names.

[Slide 39] Lexer Implementation

- Essentially a DFA (for most languages)
  - Set of regexes → NFA → DFA
- Respect whitespace/separators for operators, e.g. `+` and `+=`
- Automatic tools (e.g., flex) exist; most compilers do their own
- Keywords typically parsed as identifiers first
  - Check identifier if it is a keyword; can use perfect hashing
- Other practical problems
  - UTF-8 homoglyphs; trigraphs; pre-processing directives

> A tool to generate perfect hash tables from a set of keywords is gperf. Example, compile with `gperf -L C++ -C -E -t <input>`:
>
> ```
> struct keyword {char* name; int val; }
> %%
> int, 1
> char, 2
> void, 3
> if, 4
> else, 5
> while, 6
> return, 7
> ```

2.2 Parsing

[Slide 40] Parsing

- Convert stream of tokens into (abstract) syntax tree
- Most programming languages are context-sensitive
  - Variable declarations, argument count, type match, etc. ⇝ separated into semantic analysis

  Syntactically valid: `void foo = doesntExist / "abc";`
- Grammar usually specified as CFG

[Slide 41] Context-Free Grammar (CFG)

- Terminals: basic symbols/tokens
- Non-terminals: syntactic variables

- Start symbol: non-terminal defining language
- Productions: non-terminal → series of (non-)terminals

$$
\begin{aligned}
\mathit{stmt} &\rightarrow \mathit{whileStmt} \mid \mathit{breakStmt} \mid \mathit{exprStmt} \\
\mathit{whileStmt} &\rightarrow \textbf{while}\ (\ \mathit{expr}\ )\ \mathit{stmt} \\
\mathit{breakStmt} &\rightarrow \textbf{break}\ ; \\
\mathit{exprStmt} &\rightarrow \mathit{expr}\ ; \\
\mathit{expr} &\rightarrow \mathit{expr} + \mathit{expr} \mid \mathit{expr}\ *\ \mathit{expr} \mid \mathit{expr} = \mathit{expr} \mid (\ \mathit{expr}\ ) \mid \textbf{number}
\end{aligned}
$$

[Slide 42] Hand-written Parsing – First Try

- One function per non-terminal
- Check expected structure
- Return AST node
- Need look-ahead!

```
NodePtr parseBreakStmt() {
  consume(Token::BREAK);
  consume(Token::SEMICOLON);
  return newNode(Node::BreakStmt);
}
NodePtr parseWhileStmt() {
  consume(Token::WHILE);
  consume(Token::LPAREN);
  NodePtr expr = parseExpr();
  consume(Token::RPAREN);
  NodePtr body = parseStmt();
  return newNode(Node::WhileStmt,
    {expr, body});
}
NodePtr parseStmt() {
  // whoops!
}
```

[Slide 43] Hand-written Parsing – Second Try

- Need look-ahead to distinguish production rules
- Consequences for grammar:
  - No left-recursion
  - First n terminals must allow distinguishing rules
  - $LL(n)$ grammar; n typically 1

  ⇒ Not all CFGs (easily) parseable (but most programming langs. are)
- Now... expressions

```
NodePtr parseBreakStmt() { /*...*/ }
NodePtr parseWhileStmt() { /*...*/ }

NodePtr parseStmt() {
  Token t = peekToken();
  if (t.kind == Token::BREAK)
    return parseBreakStmt();
```

```
  if (t.kind == Token::WHILE)
    return parseWhileStmt();
  // ...
  NodePtr expr = parseExpr();
  consume(Token::SEMICOLON);
  return newNode(Node::ExprStmt,
    {expr});
}
```

[Slide 44] Ambiguity

$$expr \rightarrow expr + expr \mid expr \ \mathbf{*} \ expr \mid expr = expr \mid (\ expr\ ) \mid \mathbf{number}$$

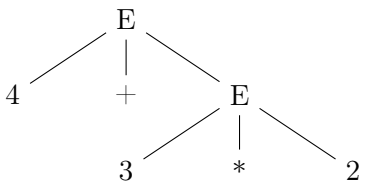
Input: $4 + 3 * 2$

> The grammar, as specified, is ambiguous, there are two possible ways to parse the input.

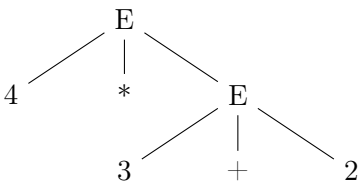
[Slide 45] Ambiguity – Rewrite Grammar?

$$primary \rightarrow (\ expr\ ) \mid \mathbf{number}$$
$$expr \rightarrow primary + expr \mid primary \ \mathbf{*} \ expr \mid primary = expr \mid primary$$

Input: $4 + 3 * 2$



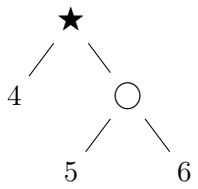
Input: $4 * 3 + 2$



> The grammar is no longer ambiguous, but the result might not be expected, conventionally, multiplication has a stronger binding than addition.

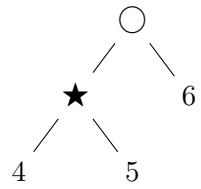
[Slide 46] Ambiguity – Precedence

Input: $4 \bigstar 5 \bigcirc 6$



If $prec(◯) > prec(★)$ or equal prec. and ★ is right-assoc.

> Examples:
> - $4 + 5 \cdot 6$ ($prec(\cdot) > prec(+)$)
> - $a = b = c$ ($=$ is right-assoc.)
> $b = c$ should be executed first



If $prec(◯) < prec(★)$ or equal prec. and ★ is left-assoc.

> Examples:
> - $4+5 < 6$ ($prec(<) < prec(+)$)
> - $a + b - c$ ($+$ is left-assoc.)
> $a + b$ should be executed first

[Slide 47] Hand-written Parsing – Expression Parsing

- Start with basic expr.:
- Number, variable, etc.
- Parenthesized expr.
  - Parse full expression
  - Next token must be `)`
- Unary expr: followed by expr. with higher prec.
  - `-` < unary `-` < `[]`/`->`

```
NodePtr parseExpr(unsigned minPrec=0);
NodePtr parsePrimaryExpr() {
  switch (Token t = next(); t.kind) {
  case Token::IDENT:
    return makeNode(Node::IDENT, t.v);
  case Token::NUMBER: // ...
  case Token::MINUS:
    // Only exprs with high precedence
    return makeNode(Node::UMINUS,
      {parseExpr(UNARY_PREC)});
  case Token::LPAREN: // ...
  // ...
  }
}
```

[Slide 48] Hand-written Parsing – Expression Parsing

- Only allow ops. with higher prec. on the right child
  - Right-assoc.: allow same
- Lower prec.: return + insert higher up in the tree

```
OpDesc OPS[] = { // {prec, rassoc}
  [Token::MUL] = {12, false},
  [Token::ADD] = {11, false},
```

```
  [Token::EQ] = {2, true},
  [Token::QUEST] = {3, true}, // ?:
}
NodePtr parseExpr(unsigned minPrec=1) {
  auto lhs = parsePrimaryExpr();
  while (auto op = OPS[next().kind];
         op.prec >= minPrec) {
    // ... handle (, [, ?: ...
    auto newPrec = op.rassoc ?
      op.prec : op.prec + 1;
    auto rhs = parseExpr(newPrec);
    lhs = makeNode(op.nodeKind,
      {lhs, rhs});
  }
  return lhs;
}
```

Example for input: `a = 3 * 2 + 1;`

	Rec. Depth 1	Rec. Depth 2	Rec. Depth 3
`minPrec`	1		
`lhs`	`a`		
`op` (prec/assoc)	= (2/r)		
`minPrec`	1	2	
`lhs`	`a`	`3`	
`op` (prec/assoc)	= (2/r)	* (12/l)	
`minPrec`	1	2	13
`lhs`	`a`	`3`	`2`
`op` (prec/assoc)	= (2/r)	* (12/l)	+ (11/l)
`minPrec`	1	2	
`lhs`	`a`	`3*2`	
`op` (prec/assoc)	= (2/r)	+ (11/l)	
`minPrec`	1	2	12
`lhs`	`a`	`3*2`	`1`
`op` (prec/assoc)	= (2/r)	+ (11/l)	; (0/–)
`minPrec`	1	2	
`lhs`	`a`	`(3*2)+1`	
`op` (prec/assoc)	= (2/r)	; (0/–)	
`minPrec`	1		
`lhs`	`a=((3*2)+1)`		
`op` (prec/assoc)	; (0/–)		

[Slide 49] Top-down vs. Bottom-up Parsing

Top-down Parsing

- Start with top rule
- Every step: choose expansion
- LL(1) parser
  - Left-to-right, Leftmost Derivation
- "Easily" writable by hand
- Error handling rather simple
- Covers many prog. languages

Bottom-up Parsing

- Start with text
- Reduce to non-terminal
- LR(1) parser
  - Left-to-right, Rightmost Derivation
  - Strict super-set of LL(1)
- Often: uses parser generator
- Error handling more complex
- Covers nearly all prog. languages

[Slide 50] Parser Generators

- Writing parsers by hand can be large effort
- Parser generators can simplify parser writing a lot
  - Yacc/Bison, PLY, ANTLR, ...
- Automatic generation of parser/parsing tables from CFG
  - Finds ambiguities in the grammar
  - Lexer often written by hand
- Used heavily in practice, unless error handling is important

[Slide 51] Bison Example – part 1

```
%define api.pure full
%define api.value.type {ASTNode*}
%param { Lexer* lexer }
%code{
static int yylex(ASTNode** lvalp, Lexer* lexer);
}
%token NUMBER
%token WHILE "while"
%token BREAK "break"

// precedence and associativity
%right '='
%left '+'
%left '*'
```

[Slide 52] Bison Example – part 2

```
%%
stmt : WHILE '(' expr ')' stmt { $$ = mkNode(WHILE, $1, $2); }
     | BREAK ';'               { $$ = mkNode(BREAK, NULL, NULL); }
     | expr ';'                { $$ = $1; }
     ;
expr : expr '+' expr           { $$ = mkNode('+', $1, $2); }
     | expr '*' expr           { $$ = mkNode('*', $1, $2); }
     | expr '=' expr           { $$ = mkNode('=', $1, $2); }
     | '(' expr ')'            { $$ = $1; }
     | NUMBER
     ;
%%
static int yylex(ASTNode** lvalp, Lexer* lexer) {
    /* return next token, or YYEOF/... */ }
```

> Compile with `bison -dg input.ypp`, it will emit a C++ header, the implementation file, and also a graph showing the state machine of he parser.

[Slide 53] Parsing in Practice

- Some use parser generators, e.g. Python some use hand-written parsers, e.g. GCC, Clang, Swift, Go
- Optimization of grammar for performance
  - Rewrite rules to reduce states, etc.
- Useful error-handling: complex!
  - Try skipping to next separator, e.g. `;` or `,`
- Programming languages are not always context-*free*
  - C: `foo* bar;`
  - May need to break separation between lexer and parser

> In fact, many compilers[a] use hand-written parsers, because they allow for better error messages a more graceful handling of syntax errors, leading to more reported errors during a single (failing) ompilation.
>
> [a] `https://notes.eatonphil.com/parser-generators-vs-handwritten-parsers-survey-2021.html`

[Slide 54] Parsing C++

- C++ is not context-free (inherited from C): `T * a;`
- C++ is ambiguous: `Type (a), b;`
  - Can be a declaration or a comma expression
- C++ templates are Turing-complete[1]
- C++ *parsing* is hence *undecidable*[2]

[1] TL Veldhuizen. *C++ templates are Turing complete*. 2003. URL: `http://port70.net/~nsz/c/c%2B%2B/turing.pdf`.

[2] J Haberman. *Parsing C++ is literally undecidable*. 2013. URL: `https://blog.reverberate.org/2013/08/parsing-c-is-literally-undecidable.html`.

  - Template instantiation combined with C `T * a` ambiguity

2.3 Semantic Analysis

[Slide 55] Semantic Analysis

- Syntactical correctness $\not\Rightarrow$ correct program `void foo = doesntExist / ++"abc";`
- Needs context-sensitive analysis:
  - Variable existence, storage, accessibility, ...
  - Function existence, arguments, ...
  - Operator type compatibility
  - Attribute allowance
- Additional type complexity: inference, polymorphism, ...

[Slide 56] Semantic Analysis: Scope Checking with AST Walking

- Idea: walk through AST (in DFS-order) and validate on the way
- Keep track of scope with declared variables
  - Might need to keep track of defined types separately
- For identifiers: check existence and get type
- For expressions: check types and derive result type
- For assignment: check lvalue-ness of left side
- *Might* be possible during AST creation
- Needs care with built-ins and other special constructs

> There are two ways of implementing a scoped hash table:
>
> - Chain of hash maps: $Scope = (Map[Name \to Type]\ names, Scope\ parent)$. This is, however, very slow for deeply nested scopes, as all hash maps of the parent scopes must be queried. Hash map lookups are fairly expensive.
> - Hash map of lists: $Map[Name \to List[Tuple[Depth, Type]]]$. For every identifier, the type at a given scope nesting depth is stored. Invalidation can be implemented with an epoch counter for every depth. The downside is that this hash map can grow very large, as entries are never removed.

[Slide 57] Semantic Analysis and Post-Parsing Transformations

- Check for error-prone code patterns
  - Completeness of `switch`, out-of-range constants, unused variables, ...
- Check method calls, parameter types
- Duplicate code for templates
- Make implicit value conversions explicit
- Handle attributes: visibility, warnings, etc.
- Mangle names, split functions (OpenMP), ABI-specific setup, ...

- Last step: generate IR code

2.4 Miscellaneous

[Slide 58] Parsing Performance

Is parsing/front-end performance important?

- Not necessarily: normal compilers
  - Some languages (e.g., Rust) need unbounded time *for parsing*
- Somewhat: JIT compilers
  - Start-up time is generally noticable
- Somewhat more: Developer tools
  - Imagine: waiting for seconds just for updated syntax highlighting
  - Often uses tricks like incremental updates to parse tree

[Slide 59] Data Types

- Important part of programming languages
- Might have large variety and compatibility
  - Numbers, Strings, Arrays, Compound Types (struct/union), Enum, Templates, Functions, Pointers, ...
  - Class hierarchy, Interfaces, Abstract Classes, ...
  - Integer/float compatibility, promotion, ...
- Might have implicit conversions

[Slide 60] Data Types: Implementing Classes

- Simple `class`/`struct`: trivial, just bunch of fields
  - Methods take (pointer to) `this` as implicit parameter
- Single inheritance: also trivial – extend struct at end
- Virtual methods: store vtable in object representation
  - vtable = table of function pointers for virtual methods
  - Each sub-class has their own vtable
- Multiple inheritance is much more involved
- Dynamic casts: needs run-time type information (RTTI)

[Slide 61] Recommended Lectures

AD IN2227 "Compiler Constructions" covers parsing/analysis in depth

AD CIT3230000 "Programming Languages" covers dispatching/mixins/...

[Slide 62] Compiler Front-end – Summary

- Lexer splits input into tokens
  - Essentially Regex-Matching + Keywords; rather simple
- Parser constructs (abstract) syntax tree from tokens
  - Top-down vs. bottom-up parsing
  - Typical: top-down for control flow; bottom-up for expressions
  - Respect precedence and associativity for operators
- Semantic analysis ensures meaningful program
- Some data structures are complex to implement
- Some programming languages are more difficult to parse

[Slide 63] Compiler Front-end – Questions

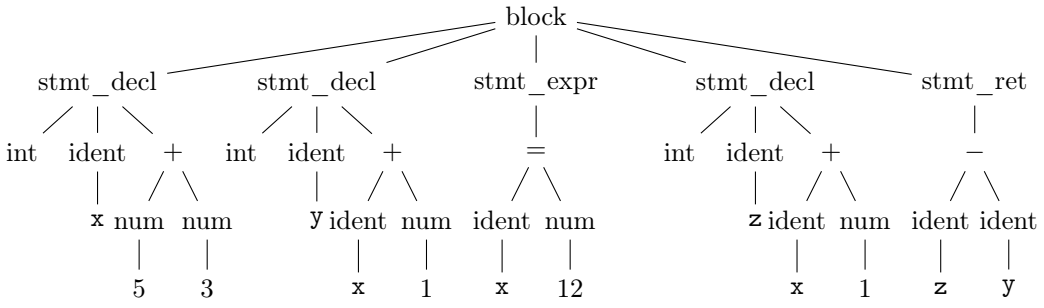
- What are typical components of a compiler front-end?
- What output does the lexer produce?
- How does a parser disambiguate rules?
- What is the typical way to handle operator precedence?
- Why are not all programming languages describable using CFGs?
- How to implement classes with virtual functions?

3 Intermediate Representations

[Slide 65] Intermediate Representations: Motivation

- • So far: program parsed into AST
- \+ Great for language-related checks
- \+ Easy to correlate with original source code (e.g., errors)
- – Hard for analyses/optimizations due to high complexity
  - – variable names, control flow constructs, etc.
  - – Data and control flow implicit
- – Highly language-specific

[Slide 66] Intermediate Representations: Motivation



Question: how to optimize? Is `x+1` redundant? ⇝ hard to tell ☹

> In this representation, it is very easy to see that the two `+1` operations have different operands on the left side and are therefore not trivially redundant.

[Slide 67] Intermediate Representations: Motivation

$$
\begin{array}{llllll}
x_1 & \leftarrow & 5 & + & 3 \\
y_1 & \leftarrow & x_1 & + & 1 \\
x_2 & \leftarrow & 12 & & \\
z_1 & \leftarrow & x_2 & + & 1 \\
tmp_1 & \leftarrow & z_1 & - & y_1 \\
\text{return} & & tmp_1 & &
\end{array}
$$

Question: how to optimize? Is `x+1` redundant? ⇝ No! ☺

[Slide 68] Intermediate Representations

- Definitive program representation inside compiler
  - During compilation, only the (current) IR is considered

> In practice, there are, of course, exceptions to the general rule; sometimes an IR contains references to a previous/higher-level IR. An example is LLVM's low-level Machine IR, which only represents single functions and therefore references to global variables use the higher-level LLVM IR.

- Goal: simplify analyses/transformations
  - *Technically*, single-step compilation is possible for, e.g., C ... but optimizations are hard without proper IRs
- Compilers *design* IRs to support frequent operations
  - IR design can vary strongly between compilers
- Typically based on **graphs** or **linear instructions** (or both)

[Slide 69] Compiler Design: Effect of Languages – Imperative

- Step-by-step execution of program modification of state
- Close to hardware execution model
- Direct influence of result

- Tracking of state is complex
- Dynamic typing: more complexity
- Limits optimization possibilities

```
void addvec(int* a, const int* b) {
  for (unsigned i = 0; i < 4; i++)
    a[i] += b[i]; // vectorizable?
}
func:
  mov [rdi], rsi
  mov [rdi+8], rdx
  mov [rdi], 0 // redundant?
  ret
```

> Tracking state, especially when memory is involved, is one of the main challenges during optimization. In the first example, the loop is not easily vectorizable, because `a` and `b` could point to the same underlying array (e.g., with `addvec(buf + 1, buf)`).

[Slide 70] Compiler Design: Effect of Languages – Declarative

- Describes execution target
- Compiler has to derive good mapping to imperative hardware

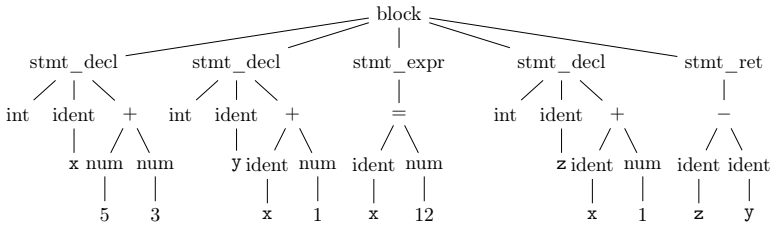
- Allows for more optimizations
- Mapping to hardware non-trivial

  - Might need more stages
  - Preserve semantic info for opt!
- Programmer has less "control"

```
select s.name
from studenten s
where exists (select 1
              from hoeren h
              where h.matrno=s.matrno)
let rec fac = function
    | 0 | 1 -> 1
    | n -> n * fac (n - 1)
```

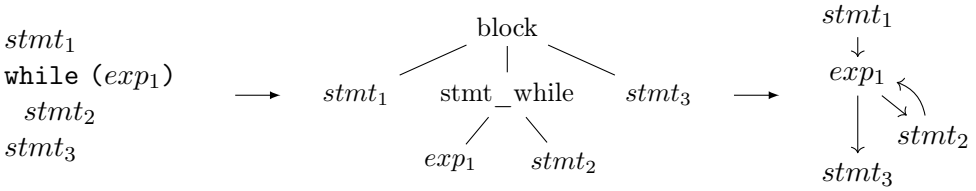
[Slide 71] Graph IRs: Abstract Syntax Tree (AST)

- Code representation close to the source
- Representation of types, constants, etc. might differ
- Storage might be problematic for large inputs



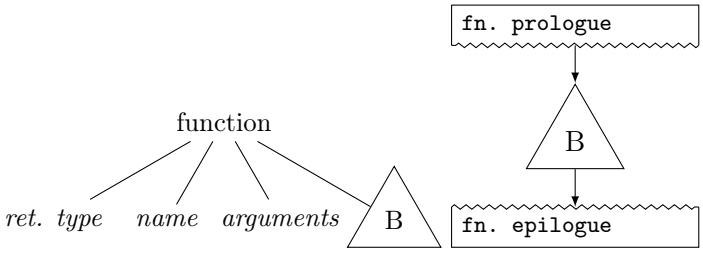
[Slide 72] Graph IRs: Control Flow Graph (CFG)

- Motivation: model control flow between different code sections
- Graph nodes represent **basic blocks**
  - Basic block: sequence of branch-free code (modulo exceptions)
  - Typically represented using a linear IR

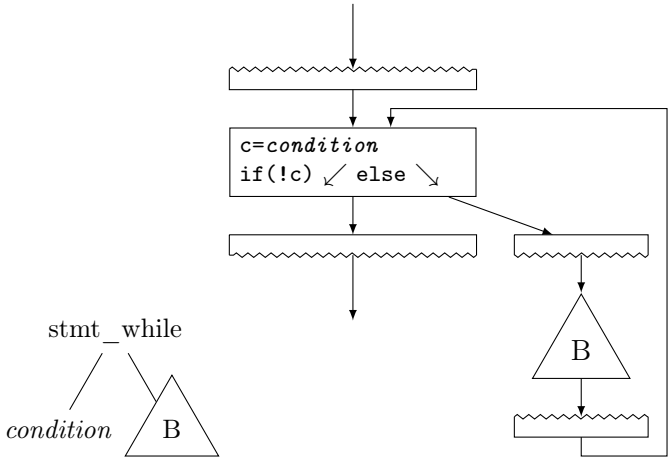


[Slide 73] Build CFG from AST – Function

- Idea: Keep track of current insert block while walking through AST



[Slide 74] Build CFG from AST – While Loop

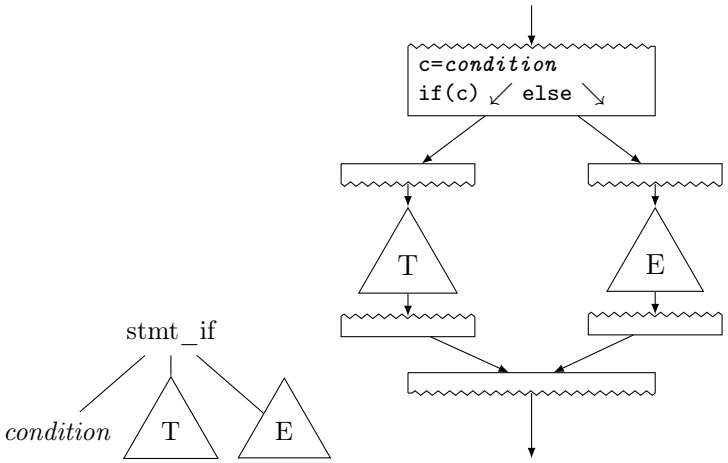


Written in pseudo-code:

```
IRValue generateCFG(ASTNode* node, BasicBlock*& insPos) {
  switch (node->kind()) {
  case ASTNode::Function:
    insPos = generatePrologue(node);
    generateCFG(node->child(0), insPos);
    generateEpilogue(insPos);
    return nullptr;
  case ASTNode::Block:
    for (ASTNode* child : node->children())
      generateCFG(child, insPos);
    return nullptr;
  case ASTNode::While: {
    BasicBlock* cond = newBlock();
    BasicBlock* body = newBlock();
    BasicBlock* end = newBlock();
    branchTo(insPos, cond);
    insPos = cond;
    IRValue brcond = generateCFG(node->child(0), insPos);
    // NB: generateCFG can modify insPos
    branchToCond(insPos, brcond, body, end);
    insPos = body;
    generateCFG(node->child(1), insPos);
    branchTo(insPos, cond);
    insPos = end;
    return nullptr;
```

```
  }
  // ...
  }
}
```

[Slide 75] Build CFG from AST – If Condition



[Slide 76] Build CFG from AST: Switch

Linear search

```
t ← exp
if t == 3: goto B3
if t == 4: goto B4
if t == 7: goto B7
if t == 9: goto B9
goto BD
```

+ Trivial
– Slow, lot of code

Binary search

```
t ← exp
if t == 7: goto B7
elif t > 7:
  if t == 9: goto B9
else:
  if t == 3: goto B3
  if t == 4: goto B4
goto BD
```

+ Good: sparse values
– Even more code

Jump table

```
t ← exp
if 0 ≤ t < 10:
  goto table[t]
goto BD

table = {
  BD, BD, BD, B3,
  B4, BD, ... }
```

+ Fastest
– Table can be large, needs ind. jump

[Slide 77] Build CFG from AST: Break, Continue, Goto

- `break`/`continue`: trivial
  - Keep track of target block, insert branch
- `goto`: also trivial
  - Split block at target label, if needed
  - But: may lead to irreducible control flow graph

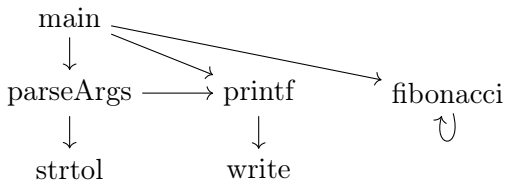
[Slide 78] CFG: Formal Definition

- **Flow graph:** $G = (N, E, s)$ with a digraph (N, E) and entry $s \in N$

  - Each node is a basic block, s is the entry block
  - $(n_1, n_2) \in E$ iff n_2 might be executed immediately after n_1
  - All $n \in N$ shall be reachable from s (unreachable nodes can be discarded)
  - Nodes without successors are end points

[Slide 79] Graph IRs: Call Graph

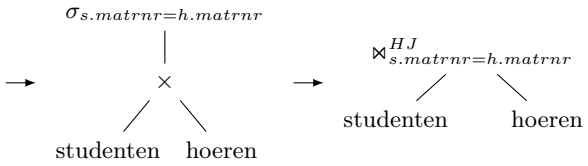
- Graph showing (possible) call relations between functions
- Useful for interprocedural optimizations
  - Function ordering
  - Stack depth estimation
  - ...



[Slide 80] Graph IRs: Relational Algebra

- Higher-level representation of query plans
  - Explicit data flow
- Allow for optimization and selection actual implementations
  - Elimination of common sub-trees
  - Joins: ordering, implementation, etc.

```
SELECT s.name, h.vorlnr
FROM studenten s, hoeren h
WHERE s.matrnr = h.matrnr
```



[Slide 81] Linear IRs: Stack Machines

- Operands stored on a stack
- Operations pop arguments from top and push result
- Typically accompanied with variable storage
- Generating IR from AST: trivial
- Often used for bytecode, e.g. Java, Python

\+ Compact code, easy to generate and implement
− Performance, hard to analyze

```
push 5
push 3
```

```
add
pop x
push x
push 1
add
pop y
push 12
pop x
push x
push 1
add
pop z
```

[Slide 82] Linear IRs: Register Machines

- Operands stored in registers
- Operations read and write registers
- Typically: infinite number of registers
- Typically: three-address form
  - *dst = src1 op src2*
- Generating IR from AST: trivial
- E.g., GIMPLE, eBPF, Assembly

$$
\begin{array}{llllll}
x & \leftarrow & 5 & + & 3 \\
y & \leftarrow & x & + & 1 \\
x & \leftarrow & 12 & & \\
z & \leftarrow & x & + & 1 \\
tmp_1 & \leftarrow & z & - & y \\
\text{return} & & tmp_1 & &
\end{array}
$$

[Slide 83] Example: High GIMPLE

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}
int fac (int n)
gimple_bind < // <-- still has lexical scopes
  int D.1950;
  int res;

  gimple_assign <integer_cst, res, 1, NULL, NULL>
  gimple_goto <<D.1947>>
  gimple_label <<D.1948>>
  gimple_assign <mult_expr, _1, n, n, NULL>
  gimple_assign <mult_expr, res, res, _1, NULL>
  gimple_assign <plus_expr, n, n, -1, NULL>
```

```
  gimple_label <<D.1947>>
  gimple_cond <ne_expr, n, 0, <D.1948>, <D.1946>>
  gimple_label <<D.1946>>
  gimple_assign <var_decl, D.1950, res, NULL, NULL>
  gimple_return <D.1950>
>
```

```
$ gcc -fdump-tree-gimple-raw -c foo.c
```

[Slide 84] Example: Low GIMPLE

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}
```

```
int fac (int n)
{
  int res;
  int D.1950;

  gimple_assign <integer_cst, res, 1, NULL, NULL>
  gimple_goto <<D.1947>>
  gimple_label <<D.1948>>
  gimple_assign <mult_expr, _1, n, n, NULL>
  gimple_assign <mult_expr, res, res, _1, NULL>
  gimple_assign <plus_expr, n, n, -1, NULL>
  gimple_label <<D.1947>>
  gimple_cond <ne_expr, n, 0, <D.1948>, <D.1946>>
  gimple_label <<D.1946>>
  gimple_assign <var_decl, D.1950, res, NULL, NULL>
  gimple_goto <<D.1951>>
  gimple_label <<D.1951>>
  gimple_return <D.1950>
}
```

```
$ gcc -fdump-tree-lower-raw -c foo.c
```

[Slide 85] Example: Low GIMPLE with CFG

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}
```

```
int fac (int n) {
  int res;
  int D.1950;
  <bb 2> :
  gimple_assign <integer_cst, res, 1, NULL, NULL>
  goto <bb 4>; [INV]
  <bb 3> :
  gimple_assign <mult_expr, _1, n, n, NULL>
  gimple_assign <mult_expr, res, res, _1, NULL>
```

```
  gimple_assign <plus_expr, n, n, -1, NULL>
  <bb 4> :
  gimple_cond <ne_expr, n, 0, NULL, NULL>
    goto <bb 3>; [INV]
  else
    goto <bb 5>; [INV]
  <bb 5> :
  gimple_assign <var_decl, D.1950, res, NULL, NULL>
  <bb 6> :
gimple_label <<L3>>
  gimple_return <D.1950>
}
$ gcc -fdump-tree-cfg-raw -c foo.c
```

[Slide 86] Linear IRs: Register Machines

- Problem: no clear def–use information
  - Is $x + 1$ the same?
  - Hard to track actual values!
- **How to optimize?**

$\Rightarrow$ Disallow mutations of variables

$$
\begin{array}{llllll}
x & \leftarrow & 5 & + & 3 \\
y & \leftarrow & x & + & 1 \\
x & \leftarrow & 12 & & \\
z & \leftarrow & x & + & 1 \\
tmp_1 & \leftarrow & z & - & y \\
\text{return} & & tmp_1 & &
\end{array}
$$

[Slide 87] Single Static Assignment: Introduction

- Idea: disallow mutations of variables, value set in declaration
- Instead: create new variable for updated value
- SSA form: every computed value has a unique definition
  - Equivalent formulation: each name describes result of one operation

$$
\begin{array}{llllll}
x & \leftarrow & 5 & + & 3 \\
y & \leftarrow & x & + & 1 \\
x & \leftarrow & 12 & & \\
z & \leftarrow & x & + & 1 \\
tmp_1 & \leftarrow & z & - & y \\
\text{return} & & tmp_1 & &
\end{array}
\quad \longrightarrow \quad
\begin{array}{llllll}
v_1 & \leftarrow & 5 & + & 3 \\
v_2 & \leftarrow & v_1 & + & 1 \\
v_3 & \leftarrow & 12 & & \\
v_4 & \leftarrow & v_3 & + & 1 \\
v_5 & \leftarrow & v_4 & - & v_2 \\
\text{return} & & v_5 & &
\end{array}
$$

[Slide 88] Single Static Assignment: Control Flow

- How to handle diverging values in control flow?
- Solution: Φ-nodes to merge values depending on predecessor

– Value depends on edge used to enter the block
– All Φ-nodes of a block execute concurrently (ordering irrelevant)

$$
\begin{array}{rll}
\mathit{entry}: & x \leftarrow & \ldots \\
 & \text{if } (x > 2) \text{ goto } \mathit{cont} & \\
\mathit{then}: & x \leftarrow & x * 2 \\
\mathit{cont}: & \text{return} & x
\end{array}
$$

$$\rightarrow$$

$$
\begin{array}{rll}
\mathit{entry}: & v_1 \leftarrow & \ldots \\
 & \text{if } (v_1 > 2) \text{ goto } \mathit{cont} & \\
\mathit{then}: & v_2 \leftarrow & v_1 * 2 \\
\mathit{cont}: & v_3 \leftarrow & \Phi(\mathit{entry}: v_1, \mathit{then}: v_2) \\
 & \text{return} & v_3
\end{array}
$$

[Slide 89] Example: GIMPLE in SSA form

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}
int fac (int n) { int res, D.1950, _1, _6;
  <bb 2> :
  gimple_assign <integer_cst, res_4, 1, NULL, NULL>
  goto <bb 4>; [INV]
  <bb 3> :
  gimple_assign <mult_expr, _1, n_2, n_2, NULL>
  gimple_assign <mult_expr, res_8, res_3, _1, NULL>
  gimple_assign <plus_expr, n_9, n_2, -1, NULL>
  <bb 4> :
  # gimple_phi <n_2, n_5(D)(2), n_9(3)>
  # gimple_phi <res_3, res_4(2), res_8(3)>
  gimple_cond <ne_expr, n_2, 0, NULL, NULL>
    goto <bb 3>; [INV]
  else
    goto <bb 5>; [INV]
  <bb 5> :
  gimple_assign <ssa_name, _6, res_3, NULL, NULL>
  <bb 6> :
gimple_label <<L3>>
  gimple_return <_6>
}
$ gcc -fdump-tree-ssa-raw -c foo.c
```

[Slide 90] SSA Construction – Local Value Numbering

- Simple case: inside block – keep mapping of variable to value

Code

$$
\begin{array}{llllll}
x & \leftarrow & 5 & + & 3 \\
y & \leftarrow & x & + & 1 \\
x & \leftarrow & 12 & & \\
z & \leftarrow & x & + & 1 \\
tmp_1 & \leftarrow & z & - & y \\
\text{return} & & tmp_1 & &
\end{array}
$$

SSA IR

$$
\begin{array}{lll}
v_1 & \leftarrow & \texttt{add } 5, 3 \\
v_2 & \leftarrow & \texttt{add } v_1, 1 \\
v_3 & \leftarrow & \texttt{const } 12 \\
v_4 & \leftarrow & \texttt{add } v_3, 1 \\
v_5 & \leftarrow & \texttt{sub } v_4, v_2 \\
 & & \texttt{ret } v_5
\end{array}
$$

Variable Mapping

$$
\begin{array}{lll}
x & \rightarrow & v_3 \\
y & \rightarrow & v_2 \\
z & \rightarrow & v_4 \\
tmp_1 & \rightarrow & v_5
\end{array}
$$

[Slide 91] SSA Construction – Across Blocks

- SSA construction with control flow is non-trivial
- Key problem: find value for variable in predecessor
- Naive approach: Φ-nodes for all variables everywhere
  - Create empty Φ-nodes for variables, populate variable mapping
  - Fill blocks (as on last slide)
  - Fill Φ-nodes with last value of variable in predecessor
- Why is this a bad idea? ⇒ *don't do this!*
  - *Extremely inefficient, code size explosion, many dead* Φ

[Slide 92] SSA Construction – Across Blocks ("simple"[1])

- Key problem: find value in predecessor
- Idea: *seal* block once all direct predecessors are known
  - For acyclic constructs: trivial
  - For loops: seal header once loop block is generated
- Current block not sealed: add Φ-node, fill on sealing
- Single predecessor: recursively query that
- Multiple preds.: add Φ-node, fill now

[1] M Braun et al. "Simple and efficient construction of static single assignment form". In: *CC*. 2013, pp. 102–122. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-37051-9_6.pdf.

> Confer the (very readable) paper for a more formal specification of the algorithm. The removal of trivial and redundant Φ-nodes is not strictly required.

[Slide 93] SSA Construction – Example

```
int foo(int n) {
  int res = 1;
  while (n) {
    res *= n * n;
    n -= 1;
  }
  return res;
}
```

```
          func foo(v1)
  entry:  sealed; varmap: n→ v1, res→ v2
          v2 ← 1
 header:  sealed; varmap: n→ φ1, res→ φ2
          φ1 ← φ(entry: v1, body: v6)
          φ2 ← φ(entry: v2, body: v5)
          v3 ← equal φ1, 0
          br v3, cont, body
   body:  sealed; varmap: n→v6, res→ v5
          v4 ← mul φ1, φ1
          v5 ← mul φ2, v4
          v6 ← sub φ1, 1
          br header
   cont:  sealed; varmap: res→ φ2
          ret φ2
```

[Slide 94] SSA Construction – Pruned/Minimal Form

- Resulting SSA is *pruned* – all ϕ are used
- But not *minimal* – ϕ nodes might have single, unique value
- When filling ϕ, check that multiple real values exist
  - Otherwise: replace ϕ with the single value
  - On replacement, update all ϕ using this value, they might be trivial now, too
- Sufficient? Not for irreducible CFG
  - Needs more complex algorithms[2] or different construction method[3]

AD IN2053 "Program Optimization" covers this more formally

[2] M Braun et al. "Simple and efficient construction of static single assignment form". In: *CC*. 2013, pp. 102–122. URL: https://link.springer.com/content/pdf/10.1007/978-3-642-37051-9_6.pdf.

[3] R Cytron et al. "Efficiently computing static single assignment form and the control dependence graph". In: *TOPLAS* 13.4 (1991), pp. 451–490. URL: https://dl.acm.org/doi/pdf/10.1145/115372.115320.

[Slide 95] SSA: Implementation

- Value is often just a pointer to instruction
- ϕ nodes placed at beginning of block
  - They execute "concurrently" and on the edges, after all
- Variable number of operands required for ϕ nodes
- Storage format for instructions and basic blocks
  - Consecutive in memory: hard to modify/traverse
  - Array of pointers: $\mathcal{O}(n)$ for a single insertion...
  - Linked List: easy to insert, but pointer overhead

Is SSA a graph IR?

Only if instructions have no side effects,consider `load`, `store`, `call`, ...
These *can* be solved using explicit dependencies as SSA values, e.g. for memory

[Slide 97] Intermediate Representations – Summary

- An IR is an internal representation of a program
- Main goal: simplify analyses and transformations
- IRs typically based on graphs or linear instructions
- Graph IRs: AST, Control Flow Graph, Relational Algebra
- Linear IRs: stack machines, register machines, SSA
- Single Static Assignment makes data flow explicit
- SSA is extremely popular, although non-trivial to construct

[Slide 98] Intermediate Representations – Questions

- Who designs an IR? What are design criteria?
- Why is an AST not suited for program optimization?
- How to convert an AST to another IR?
- What are the benefits/drawbacks of stack/register machines?
- What benefits does SSA offer over a normal register machine?
- How do ϕ-instructions differ from normal instructions?