FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Compiler frontend for a subset of C++ programming language
Student:	Daniel Král
Supervisor:	Ing. Tomáš Pecka
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2024/2025

Instructions

Get acquainted with the ANTLR parser generator [1] and LLVM compiler infrastructure [2], emphasizing the frontend of the compiler and LLVM IR (intermediate representation). Utilize these tools to develop a compiler frontend for a subset of the C++ language, covering at least basic types, pointers, arrays, functions, structs, classes, and standard control flow statements. Verify the functionality of your implementation by testing with a relevant set of sample codes. Document your frontend's source code, focusing on the abstract syntax tree and the process of generating LLVM IR.

[1] https://www.antlr.org/
[2] https://llvm.org/

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 8 February 2024 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Compiler frontend for a subset of C++ programming language

Daniel Král

Department of Theoretical Computer Science
Supervisor: Ing. Tomáš Pecka

May 15, 2024

Acknowledgements

I would like to thank my supervisor, Ing. Tomáš Pecka, for his guidance, his efforts to help me as much as possible during our consultations, and his patience with me when I was too stubborn to take good advice. I would also like to express gratitude to my family for their support during my studies at FIT CTU.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on May 15, 2024

Citation of this thesis

Král, Daniel. *Compiler frontend for a subset of C++ programming language.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Abstrakt

Překladače pro programovací jazyky jsou nezbytnou součástí vývoje moderního software. Tato práce se zabývá návrhem přední části překladače pro (skoro) podmnožinu jazyka C++ nazvanou C+−. Nejprve je specifikován rozsah C+−. Poté je popsáno využití ANTLR4 pro lexikální a syntaktickou analýzu a vytvoření abstraktního syntaktického stromu. Následuje sémantická analýza a generování mezikódu LLVM IR pomocí LLVM C++ API. Implementace překladače je otestována sadou ukázkových kódů.

Klíčová slova frontend překladače, LLVM IR, ANTLR, C++, sémantická analýza, generování kódu

Abstract

Compilers for programming languages are an essential part of modern software development. This thesis deals with the design of the frontend of a compiler for an (almost) subset of C++ called C+−. First, the scope of C+− is specified. Then the use of ANTLR4 for lexical and syntactic analysis and the creation of an abstract syntactic tree is described. This is followed by semantic analysis and generation of the LLVM IR intermediate code using LLVM C++ API. The implementation of the compiler is tested with a set of sample codes.

Keywords compiler frontend, LLVM IR, ANTLR, C++, semantic analysis, code generation

Contents

List of Figures

List of Listings

Introduction

Compilers are programs that are used by most programmers in the world every day. They allow us to write code in a high-level programming language without the need to worry about hardware architecture details. As such, they are an indispensable part of the modern software development process. They are complex pieces of software that are usually split into three stages – the frontend, the middle-end and the backend. The job of the front end is to parse the source code, verify that it is correct according to given language rules, and transform it into some intermediate representation. The middle end then takes this intermediate representation, and performs various optimizations and possibly other transformations. Finally, the back end generates code that can be run on a specific machine.

At FIT CTU, we have an undergraduate course Programming Languages and Compilers, BI(E,K)-PJP, which provides an introduction to compilers. The course lectures focus on the frontend stage of the compiler, and put a lot of attention to the first two parts of the front end – lexical analysis and syntactic analysis. The lectures do not, however, put much emphasis on semantic analysis and intermediate representation generation, which follow lexical and syntactic analysis in the frontend stage.

The goal of the thesis is to continue where we finished in BI(E,K)-PJP. We will use the ANTLR4 framework to handle lexical and syntactic analysis for us, and focus on semantic analysis and intermediate representation. For intermediate representation, we will use the LLVM Intermediate Representation (LLVM IR) from the LLVM framework. The compiled language will be a subset of the C++ programming language.

This thesis, especially the practical part, will be of use to BI(E,K)-PJP students. We show how the C++ API can be used to generate LLVM IR which is a part of the BI(E,K)-PJP semestral project.

Goals of the Thesis

The overarching goal is to write a compiler frontend that compiles a subset of C++ (and a few non-C++ features) to LLVM IR. This can be broken down into the following steps:

- Specify the implemented language.
- Use ANTLR4 to parse source code and create the abstract syntax tree (AST).
- Write code to perform semantic analysis on the AST.
- Write code to generate LLVM IR from the AST.
- Verify functionality of the compiler with test sample codes.

Chapter 1

Analysis and Theory

1.1 Compilers

Compilers are used every day by most programmers. They are complex pieces of software with multiple stages. Nowadays, they are generally split into three parts: the frontend, the middle-end and the backend [3].

The frontend itself is split into multiple stages. First, during lexical analysis, the source code is split into individual language tokens (such as keywords and identifiers). Then, during syntactic analysis, the frontend tries to compose the tokens into higher-level language structures, such as expressions and statements, based on the language's grammar. Around this stage, the abstract syntax tree (AST), which represents the semantics of the language, is often built. Next is the semantic analysis, during which the AST is analyzed for errors (such as type mismatches), and potentially modified. If the AST passes through semantic analysis, it is usually translated into an intermediate (or middle-end) representation, which is sent to the middle-end. For an analogy between the compiler frontend to the human language, please see Appendix A.

The middle-end representation is usually a low-level programming language. It is often expressed in the SSA (Single Static Assignment) form [4], because it allows for good optimizations [5]. The middle-end part of the compiler, sometimes also called the optimizer, takes this intermediate representation, and performs various target-independent optimizations, such as constant propagation, dead-code elimination or tail-call elimination [6].

Finally, the backend translates the middle-end representation into specific instruction sets such as `x86`, `IA-32` or `ARM`, optionally performing target-specific optimizations along the way.

An advantage of this 3-part architecture is that when a new language (or target architecture) is added into a compiler framework, only the frontend (or backend) has to be implemented, and the rest of the framework can be reused. This can be seen in Figure 1.2. For a specific example of this 3-part architecture, consider the GCC [7] architecture in Figure 1.1.

1.2 Existing C++ Compilers

Since the goal of this thesis is to write a compiler for subset of C++, we explore two commonly used C++ compilers: GCC and Clang/LLVM. A third widely

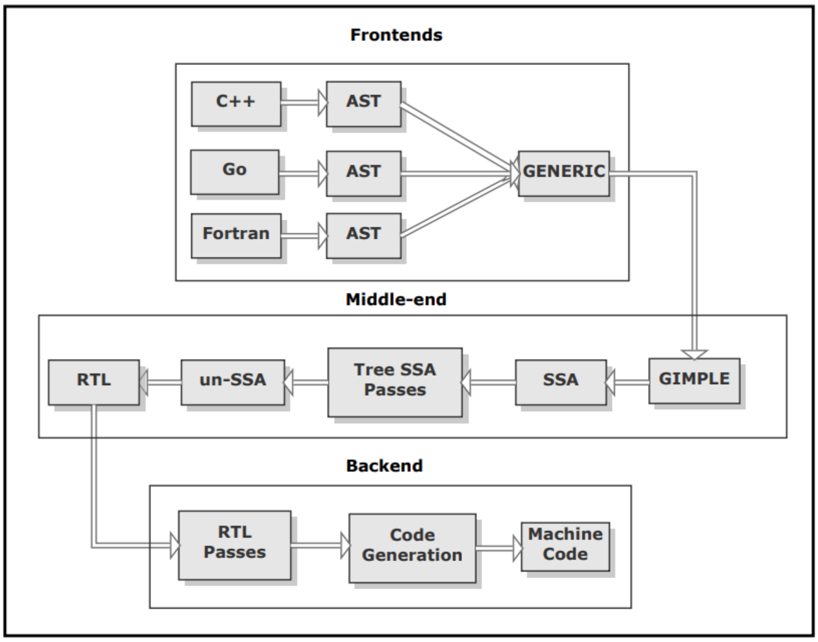

Figure 1.1: GCC Architecture [1]

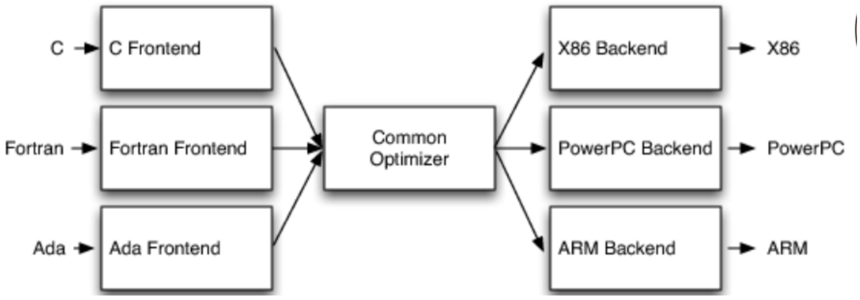

Figure 1.2: Compiler retargetability [2]

used compiler is MSVC [8], but since it's proprietary software, we were not able to find much information about its internals.

1.2.1 GCC

GCC, or the GNU Compiler Collection, is a set of open-source compiler tools. It has front ends for many languages including C, C++, Fortran and Ada [7]. Every GCC front end has it's own representation, which is then translated into Generic, *a language independent tree structure* [9]. Generic is then translated into GIMPLE, the GCC middle end representation [10]. GIMPLE has an SSA GIMPLE form, where most optimizations are performed. GIMPLE is then translated into RTL (Register Transfer Language), a low level intermediate representation [11]. Finally, the machine code is produced from the RTL.

1.2.2 Clang/LLVM

Clang is an open-source project under the LLVM Project umbrella [12]. It is a compiler frontend for languages in the C language family (such C, C++, Objective-C). It parses source code into an AST representation. The AST purposefully contains a lot of source code information so that aside from code generation, it can also be used for tasks such as static analysis and code refactoring [13], for example it is used by the clang-tidy linter [14]. For code generation, Clang translates the AST into LLVM Intermediate Representation (LLVM IR) [15]. Optimizations are performed on the LLVM IR, which is then translated into machine code using tools from the LLVM project [16].

1.3 LLVM Intermediate Representation (LLVM IR)

LLVM Intermediate Representation (LLVM IR) is a low-level programming language developed as a part of the LLVM Project. It aims to be target-independent, uses virtual registers and is written in SSA form, which means that a register can be assigned to only once. Many languages have frontends that compile to LLVM IR, such as C, C++, Swift , Julia and Rust [16]. LLVM IR is popular because many code-optimization tools have been written for it.

1.3.1 C++ API

A popular way to generate LLVM IR code is to use the LLVM C++ API. It provides functionality to build entire LLVM IR programs and perform various passes (such as optimization passes) over them.

The fundamental class is `llvm::Module`. It contains all information related to a translation unit. Another important class is `llvm::Context`. It holds information that can be shared between LLVM modules such as types. Next is the `llvm::IRBuilder`, which is the class through which the programmer creates specific instructions. These three classes are the base for LLVM IR generation with the C++ API. Now we introduce the most common classes that represent LLVM IR objects.

Value Probably the most used class during the process of LLVM IR generation is `llvm::Value`. As the name suggests, it represents a value. This can be a constant, global variable, local variable, function, and much more.

BasicBlock Another very important class is `llvm::BasicBlock`. Basic blocks group instructions together. If `llvm::Value` represents *what is computed*, `llvm::BasicBlock` represents *where it's computed*. Basic blocks are used to organize instructions inside functions.

Terminator Instructions Every basic block finishes with a terminator instruction. These are instructions that direct control-flow elsewhere. Among the most common are `ret` for returning from a function and `br` for branching.

```
grammar Expr;

prog:   (expr NEWLINE)* ;
expr:   term ((PLUS | MINUS) term)*;
term:   atom ((STAR | DIV ) atom)*;
atom:   INT;

NEWLINE : [\r\n]+ ;
INT     : [0-9]+ ;
PLUS    : '+';
MINUS   : '-';
STAR    : '*';
DIV     : '/';
```

Listing 1.1: Grammar Expr

GEP Instruction GEP stands for *Get Element Pointer* and it is an instruction that represents target-agnostic address computation. It is used to index arrays, pointers and to obtain pointers to struct fields. Among newcomers to LLVM IR, it is infamous because it tends to be hard to understand at first. For better understanding, we recommend the blogpost from LLVM [17].

PHI Instruction The `phi` instruction is a special type of `llvm::Value`. It's initialized with one of multiple values, depending on the basic block predecessor of the basic block where the `phi` instruction is located.

1.4 ANother Tool for Language Recognition 4 (ANTLR4)

ANTLR4 [18] (hereinafter referred to as "ANTLR") is a parser generator. From provided grammar, ANTLR can generate a parser for this grammar in target programming language (Java, C++, Python, Swift, and more [19]). ANTLR also facilitates visitation of the parse tree that the user can use to write their own passes over the parse tree.

The grammar is written in EBNF form [20] and can contain some extra elements. Consider grammar `Expr` in Listing 1.1 that parses simple expressions. The first line says that the grammar name is `Expr`. In the grammar, there are four rules – `prog, expr, term, atom` – and six terminals – `NEWLINE, INT, PLUS, MINUS, STAR, DIV`. The second line says that the rule `prog` can be expanded into any number of `expr` that is followed by a `NEWLINE`. `expr` is always expanded into `term` and possibly followed by any number of `PLUS` or `MINUS` and `term`.

Priorities There are three rules – `expr, term, atom` – to express operator priorities. They ensure that `1+2*3` gets parsed as `1+(2*3)`. However, we can simplify the grammar as shown in Listing 1.2 to achieve the same result using ANTLR priorities.

```
grammar Expr;

prog:    (expr NEWLINE)* ;
expr:    expr (STAR | DIV) expr |
         expr (PLUS | MINUS) expr |
         INT;

NEWLINE : [\r\n]+ ;
INT     : [0-9]+ ;
PLUS    : '+';
MINUS   : '-';
STAR    : '*';
DIV     : '/';
```

Listing 1.2: Grammar Expr with priorities

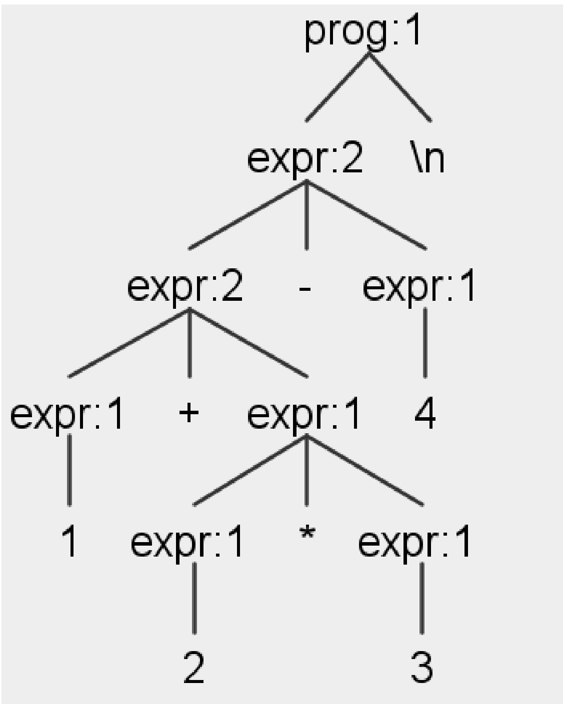

Figure 1.3: ANTLR parse tree of `1+2*3-4`

Now the rule `expr` has three alternatives and contains direct left-recursion. The order of the alternatives determines which is preferred if multiple of them are viable at given parsing step. We order alternatives from those with the highest precedence to those with the lowest. Consider a simple example, `1+2*3-4`, and the corresponding ANTLR parse tree in Figure 1.3. We can see the rule names and alternative numbers taken at each step, ending with terminal symbols as tree leaves.

```
grammar Expr;

@parser::members {
    bool inJapan() { /* ... */ }
    bool notFour() {
        getCurrentToken()->getText() != "4";
    }
}

prog:   (expr NEWLINE)* ;
expr:   expr (STAR | DIV) expr |
        expr (PLUS | MINUS) expr |
        {! inJapan() || notFour()}? INT;

NEWLINE : [\r\n]+ ;
INT     : [0-9]+ ;
PLUS    : '+';
MINUS   : '-';
STAR    : '*';
DIV     : '/';
```

Listing 1.3: Grammar Expr with actions

Actions Imagine we're using the grammar in elevator programs and want to forbid number 4 if the user is Japanese [1] . We can update the grammar to that in Listing 1.3. First we add use a new ANTLR construct `parser::members`. It contains two C++ functions called `inJapan` and `notFour`, which return whether the current location is Japan, and whether the next parsed token is four. Then, in `expr` we place `{! inJapan() || notFour()?}` before `INT`. The curly braces contain an ANTLR action. The action is a C++ expression that should be executed when the parser reaches this point. If the result of this expression is of boolean type, the curly braces can be followed by a question mark. Then, if the expression evaluates to true, the parser proceeds past the action. However, if it evaluates to false, the parser considers this rule alternative invalid and returns from the rule. So, our third `expr` alternative will only be parsed if the current location is not Japan, or if it is Japan and the number is not four.

Actions allows the programmer to conditionally allow or forbid some rules based on the context. One disadvantage is that they're written in the parser target language (C++ in our example), and thus the grammar loses its independence on target language when actions are added into it.

[1] Some East-Asian cultures have superstitions about the number four [21].

Chapter 2

Language Specification

This chapter explores all the features that are implemented by our compiler, and how (if) they differ from their counterparts in C++. We call the implemented language C+- to differentiate it from C++.

This chapter is not a formal language standard.

2.1 Type system

C++ is a statically typed language and so is C+-. There are 4 different type categories in C+-:

Simple types Represent basic types. This includes fundamental types (`int`, `char`, `bool`, `double`, `nullptr_t` and the incomplete type `void`), and user defined types (classes and structs). These types may be qualified as `const`, signifying that the underlying value cannot be changed after initialization.

C+- supports only signed integer types.

Pointer types Represent pointer to another type. This type can be any simple type, pointer type or array type. Pointer types may also be qualified as const, with the same meaning as simple types.

Unlike in C++, pointers to functions are not allowed in C+-.

Array types Represent array of fixed number of elements of some type. Element type can be any complete simple type, pointer type or array type.

Array types in C+- are always declared with known size (an integer literal).

Function types Represent a function signature. It is defined by the return type, list of parameter types, and whether the function takes variadic arguments. The return type can be `void`, any complete simple type or pointer type. A parameter type can be any complete simple type or pointer type.

Unlike in C++, parameter types cannot be declared of array type.

In the source code, the file `src/type/DerivedTypes.h` contains our implementation of these.

Similar types Two types are *similiar* if they are the same when stripped of all `const` qualifiers, and they are not function types [22]. For example, `const int` and `int` are similar. `const double * const **` and `double ** const *` are similar as well.

c-unqualified types The *c-unqualified* (or *const-unqualified*) version of type T is:

- T without `const` qualifier if T is a simple type or a pointer type
- T otherwise

For example, `int` is the c-unqualified version of `const int` and `int`. `double*` is the c-unqualified version of `double *const` and `double *`. However, the c-unqualified version of `const double *` is still `const double *`, because the `const` qualifier is removed only at the top level, the pointer in this case.

2.2 Expressions

Expressions are an indispensable part of C++ and C+-. They fundamentally allow to programmer to process manipulate data in the program. As such, they create a big part of most programming languages, and of C+-.

2.2.1 List of Expressions in C+-

C+- has these expressions:

- **literals:** integer (`for example 5`), character (`'c'`), boolean (`true`), float (`3.14`), string (`"ahoj"`), null pointer (`nullptr`), this expression (`this`).
- `a op b`: **binary expression** where `op` can is one of:

  ```
  +, -, *, /, %, &, |, ^, <<, >>, <, >, <=, >=, ==,
  !=, &&, ||
  ```

  with the same meaning as in C++.
- `a assign_op b` **assignment expression** where `assign_op` is one of:

  ```
  =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
  ```

  with the same meaning as in C++.
- `a, b` **comma expression**
- `f()` **call expression**
- `a.b` **and** `ap->b` **member of object expression**
- `s.m(), sp->m()` **object method call expression**
- `a[b]` **subscript expression**
- `a ? b : c` **conditional ternary expression**

- **a++, a-- post increment and decrement expressions**
- **unop a unary expression** where unop is one of:

  *, &, +, -, ~, !, ++, --, sizeof

  with the same meaning as in C++.
- **sizeof(type) sizeof type expression**

Please note that unlike in C++, the +, - and [] operators, when used with a pointer and integral operands, only work when the pointer is the left operand [2].

2.2.2 Value Categories

Every C++ expression has two independent properties, a type and a value category [24]. The type system has already been described, now we turn to value categories in C+-.

In C++, every expression belongs to exactly one of three value categories: lvalue, xvalue or prvalue [25]. C+- simplifies this system to two categories: lvalues and rvalues. lvalues in general represent values that have a location in memory, or as is often put, can be on the left-hand side of the assignment operator. On the contrary, rvalues in general represent values that do not have a location in memory, such as literals (except for string literals) or temporary results of expressions. Expressions that exist both in C++ and C+- work as follows:

- C++ prvalues are rvalues in C+-,
- C++ lvalues are lvalues in C+-,

and C++ xvalues are handled as follows in C+-:

- **a.m, member of object expression** with a as (p)rvalue is forbidden,
- **a[n] subscript expression** where a is an array rvalue is unachievable (array cannot be an rvalue),
- **a ? b : c ternary conditional expression** is lvalue if both b and c are lvalues (after implicit conversions take place), and rvalue if either b or c one are rvalues (after implicit conversions take place),
- **a, b comma expression** has the value type of b,
- **call expression** the result is always an rvalue.

A consequence of the first and last points is that the code in Listing 2.1, which is valid in C++, will produce an error in C+-.

[2] This means that the infamous C-(bug)feature 5[a] integer access by array [23] is invalid in C+-.

```
struct S {
    int a;
    S() {
        this->a = 42;
    }
};

void f() {
    S().a;  // c+- error: cannot access member of rvalue;
            // is ok in c++
}
```

Listing 2.1: Member access on result of call expression

2.2.3 Implicit Conversions

C++ is (in)famous for the complexity of the implicit conversions that are allowed by the standard [22].

C+- uses a simplified system, which is a subset of the C++ *standard conversion sequence*.

Obtaining rvalue When converting a value, the first step is to obtain an rvalue from the converted value. There are 4 options:

1. If the converted value is already an rvalue, this step is skipped,
2. if the converted value is an lvalue of simple or pointer type, lvalue-to-rvalue conversion takes place,
3. if the converted value is an lvalue of array type, it is converted to an rvalue of pointer to the first element of the array (array-to-pointer decay),
4. otherwise, the conversion is not well-formed.

Type conversions Once an rvalue is obtained, the following type conversions are allowed:

- `char`, `int`, `double`, `T*` to `bool`, where `T*` is any pointer type, with the meaning of *Does the value equal zero or nullptr?*,
- integral conversions (between `bool`, `char`, `int`),
- `double` to `int`,
- `int` to `double`,
- `T*` to `const T *`, const-strengthening of pointer element type,
- `T*` to `void*`, converting pointer of any type to void pointer,
- `nullptr_t` to `T*`, converting `nullptr` to any pointer type.

```
void f(char c);
void f(double d);
void g() {
    int i = 42;
    f(i);    // error: ambiguous call
}
```

Listing 2.2: Ambiguous call due to type conversion rules

The result of the type conversion is always an rvalue.

Only one implicit type conversion can be performed at a time. This means that an expression of type `char` cannot be implicitly converted to expression of type `double` (and vice versa), since the conversion would require two steps with `int` as the middleman type.

Finally, note that in C+-, integral conversions (e.g. `int` to `char`) are **not** of higher priority than other type conversions. An implication of this is that the code in Listing 2.2 will fail to compile in C+-, because the function call is ambiguous.

2.2.4 Explicit Conversions

C+- supports C-style casts, such as `(int*) ptr`. Every implicit type conversion can also be done explicitly, and a pointer value of any type can be explicitly cast to pointer of any other type.

2.3 Statements

Statements are the building blocks of functions. They are used for declarations inside function, control-flow, and more.

C+- has the following statements:

Declaration statement variable declaration(s) inside of a function.

Expression statement expression followed by a semicolon.

If (Else) statement conditional statement.

Compound statement sequence of statements inside curly braces, introduces new scope unless it's the body of a function or a For statement.

Do While statement with body and condition.

While statement with condition and body.

For statement with (possibly empty) initializer statement, optional condition, and optional expression to be executed after each iteration.

Break statement optionally with a positive integer, more on that below.

Continue statement optionally with a positive integer, more on that below.

Return statement with or without an expression.

```
if(int a = 5) {      // valid in C++, invalid in C+-
    // ...
}
```

Listing 2.3: Declaration as if-statement condition

```
int main() {
    int i = 0, res = 0;

    while(i < 5) {
        i += 1;
        res += 10;

        while(true) {
            break 2;
        }

        res += 42314; // this line will never be reached
    }
    return res; // return 10 for break 2, 50 for continue 2
}
```

Listing 2.4: Enhanced `break` and `continue`

If, While and For statements allow only for an expression in the condition. Declarations, such as in Listing 2.3, are not allowed.

2.3.1 Enhanced Break and Continue

In C+-, break and continue can be optionally followed by a positive integer [3]. This positive integer indicates from which loop should the program break or continue. `break 1` means breaking (continuing) from the inner most loop; the same as just `break` (`continue`) without any number. `break 2` means breaking (continuing) from the second inner most loop. `break 3` means breaking (continuing) from the third inner most loop, and so on. Consider the program in Listing 2.4. This program will return `10`. If `break 2` were substituted by `continue 2`, the program would return `50`.

2.4 Functions

Functions can be found in most programming languages. They are used to express a series of instructions that are repeated in the program. In programming, they are used heavily as a means of code re-use and to make code more readable by given functions apt names.

[3] This feature is why C+- is not a subset of C++.

```
int f(int a);
int f(const int b);
```

Listing 2.5: Multiple declarations of one function

Function signatures Functions are declared by their name, return type, list of parameters and whether they take variadic arguments (ellipsis at the end of argument list) or not. In C+-, function parameters must always be declared with a name, and can have default values. Even though functions can be declared with variadic arguments, these arguments are not accessible to the programmer in C+-. However, function declarations with variadic arguments can be used, called, and linked at link time (for example, `printf` can be linked from `libc` and `sqrt` can be linked from `libm`).

Multiple function declarations A function can be declared (but not defined) multiple times. Two declarations refer to the same function if:

- they are in the same scope,
- they have the same name,
- they have the same number of parameters and both or neither take variadic arguments,
- their return types match and all their c-unqualified parameter types match pair-wise.

For example, both declarations in Listing 2.5 refer to the same function. Default argument values can only be specified in one declaration (it doesn't matter which declaration, but they can be used only after the declaration where they had been specified).

Function overloading Functions can be overloaded in C+-. For two function declarations in the same scope with the same name to denote two different functions (overloads), they must meet at least one of the following:

- have a different number of parameters,
- have at least one parameter type differ (c-unqualified),
- or one takes variadic arguments, the other does not.

Listing 2.6 contains an example with four different overloads of function `f`.

However, two overloads cannot differ only in return type. Code in Listing 2.7 will lead to an error.

```
void f(int i);
void f(int i1, int i2);
void f(int i, ...);
void f(double d);
```

Listing 2.6: Function overloads

```
int f();
double f();   // error: differs from another overload only
              // in return type

int g();
const int g();  // error: differs from another overload
                // only in return type
```

Listing 2.7: Function overloads differ only in return type

Overload resolution If there are multiple overloads of one function name, a call expression to this name must undergo overload resolution. C+- simplifies the overload resolution rules of C++ [26]. The process of overload resolution in C+- has three steps:

1. Collect candidate functions.
2. Select viable functions from candidates functions.
3. Find the best matching function from viable functions.

First, the list of candidate functions is formed by name lookup in the current scope. Then viable functions are selected based on the m arguments passed in the call. A candidate function F with p parameters is viable if the following conditions are met:

1. the number of arguments is viable:
   - $m = p$,
   - $m < p$ and F has default argument values for parameters $m+1, m+2, ..., p$,
   - $m > p$ and F accepts variadic arguments.
2. every i-th argument for i in $1...min(p, m)$ can be implicitly converted to the type of i-th parameter

If the set of viable functions is empty, there's no matching function for the call.

Finally, we look for the best match from viable functions. First, for each viable overload function F, we construct a list with m elements where the i-th element describes how well the c-unqualified type of i-th argument matches c-unqualified type of i-th parameter. There are 4 possibilities, from the best to the worst:

1. no implicit conversion is needed (e.g. argument type `int` and parameter type `const int`),

2. a const qualifier must be added to the element type of pointer parameter (e.g. argument type `int*` and parameter type `const int*`),

3. implicit type conversion between non-similar types must take place (e.g. argument type `int` and parameter type `double`),

4. $i > p$, the argument is in the variadic arguments list.

For two viable overloads F1 and F2, we say that F1 is a better match than F2 if it matches every argument at least as well as F2, and matches at least one argument better than F2. If one overload is better than all the other overloads, it is chosen. Otherwise, the call is ambiguous.

Lastly, let us note that overload names are not required to be mangled in a specific way (unlike in C++). As a result, it may be impossible to link C+- binaries with other object files.

Builtin functions Finally, the following functions are automatically declared at the start of every C+- program:

- `printf int (const char *, ...)`
- `scanf int (const char *, ...)`
- `sprintf int (char *, const char *, ...)`
- `sscanf int (const char *, const char *, ...)`
- `malloc void *(int)`
- `free void (void *)`

This is so that C+- users don't have to declare these functions themselves whenever they want to use them (and since C+- doesn't support macros and header inclusion).

2.5 Classes

Classes are an important concept in programming. They allow the user to compose a type from other types, and give names to these types (fields) within the class type. They also provide a way to associate a set of functions (called methods) with its objects, which is very useful.

C+- has classes and structs. They allow for:

- access modification through the `public` and `private` keywords,
- non-static members (fields and methods),
- constructors.

The default access is `private` in classes and `public` in structs. Classes cannot be forward declared, and must be declared in the global scope. Class methods cannot be declared `const` and must be defined at declaration. There are no default constructors added by the compiler or the user.

An insignificant advantage of C+-, compared to C++, is that class definition does not have to be followed by a semicolon [4].

2.6 Declarations

Declarations allow the programmer to associate a name with some object. They are used by the programmer to express basic concepts, such as variables, functions, and classes, in the program.

2.6.1 List of Declarations in C+-

In C+-, the following declarations are possible within listed scopes:

- variable declaration, and possible initialization,
  - available in the global scope, class scopes (field declaration, without initializer), and local scopes
- function declaration (without definition),
  - available in the global scope,
- function definition
  - available in the global scope and class scopes (methods and constructors)
- class definition
  - available in the global scope
- empty declaration (extraneous semicolon)
  - available in the global scope

Multiple variables and/or functions can be declared at once with a shared *declaration specifier sequence*, just as in C++.

Note that unlike in C++, variables of class type are **not** default initialized by their zero argument constructor at declaration in C+-. They are in undefined state, the same as local uninitiliazed variables of fundamental types [27]. In example code in Listing 2.8, the value of `s1.a` is undefined, while value of `s2.a` is 42.

2.7 Grammar

The grammar of C+- can be found in the file `src/parser/grammar/CPM.g4`. We touch on some grammar rules in chapter 3.

[4] This is another place where we break from being a subset of C++.

```
struct S {
    int a;
    S() {
        this->a = 42;
    }
};
void f() {
    S s1;       // s1.a is undefined
    S s2 = S(); // s2.a is 42
//    S s3();     // this would be a function declaration,
                  // search for 'vexing-parse'
}
```

Listing 2.8: Class object default initialization

Chapter **3**

Parsing

We use ANTLR4 to parse source code and create the AST from it. In this chapter, we go into a little bit more detail about that. First, we start by describing our AST.

3.1 Abstract Syntax Tree

Abstract syntax tree (AST) is a data structure that can be found in most conventional compilers. It is a tree structure used to represent the semantics of given program. This chapter explores the implementation of AST in this thesis, which can be found in the `src/ast` directory (see Figure 3.1).

3.1.1 Polymorphism with `std::variant`

We decided to use `std::variant` from the C++ standard library to employ polymorphism in our AST. `std::variant` is a class template that represents type-safe union [28].

The following are some of the `std::variant` aliases which correspond to different AST nodes categories:

- `ast::Expr` represents an expression,
- `ast::Stmt` represents a statement,
- `ast::Declaration` represents a declaration.

Figure 3.1: Structure of the `src/ast` directory

```
src/ast/ ...................... directory with our implementaion's AST
├── base ................................. nodes used by all other nodes
├── class ..................................... nodes related to classes
├── decl ................................ nodes related to declarations
├── expr ................................. nodes related to expressions
├── func .................................... nodes related to functions
└── stmt .................................. nodes related to statements
```

3.1.2 Helper Classes

There are many (more than 50) classes in the `src/ast` directory which represent different types of nodes. In this section, we describe the few that are shared by all nodes.

class `ast::SourceInfo` In order to provide good error reporting to the user, we need to be able to map AST nodes back into the source file. For this, we have the class `SourceInfo`. It holds information about the line and column number where the code corresponding to the AST node starts.

class `ast::Node` There's some information that is shared by all the nodes, such as `ast::SourceInfo`. To share this among all nodes, there is the class `ast::Node`, which is the base class of all node types.

`ast::Node` is not polymorphic (doesn't have a virtual method or destructor) by design. We believe it would not be a good long-term design idea to intertwine two forms of dynamic polymorphism: `std::variant` and virtual methods. [5]

alias `ast::node_ptr` In AST design, child nodes of a node are usually represented by pointers. For this, we have the alias `node_ptr`.

`make_node` The `make_node` function is used to create a `node_ptr` of either given type, or of given type under given variant.

For example: `make_node<IntLiteral>(5)` will create an object of type `node_ptr<IntLiteral>`. However, `make_node<IntLiteral, Expr>(5)` will create an object of type `node_ptr<Expr>` which is a pointer to an `Expr` object that contains an `IntLiteral` object.

`change_node` Since `IntLiteral` does not inherit from `Expr`, an object of type `node_ptr<IntLiteral>` is not an object of type `node_ptr<Expr>`. This can sometimes cause issues when we hold an object of type `node_ptr<IntLiteral>`, but need to turn it into an object of type `node_ptr<Expr>`. For this, we have the `change_node` function. It consumes a `node_ptr` to a node type, and turns it into a `node_ptr` of a variant.

3.1.3 AST Dumping

Finally, for debugging and visualizations, we developed an AST visitor class called `AstDumper`. This class prints an AST to an output stream in a structured format that's similar to that that of `clang` [12]. Its outputs are used heavily in Chapter 4.

Our compiler can be used with the options `--ast-dump-raw` and `--ast-dump` to print the AST before semantic analysis and after semantic analysis, respectively.

[5] An exception to this is `ast::Decl`, which is a polymorphic class and base to `ast::FunctionDecl`. However, we only define the virtual destructor of `ast::Decl`, and we don't override any methods.

3.2 Grammar

The ANTLR grammar that describes our language can be found in the file `src/parser/grammar/CPM.g4`. It was created by starting with the C++ grammar from ANTLR GitHub [29] (distributed under the MIT license), and modifying it for our purposes.

The grammar is quite long and at times complicated, so we won't go rule-by-rule in this section. Rather, we explore a few interesting spots.

3.2.1 Declarators

Declarators are used to express that value of certain name has certain type. They are used in variable declarations, function definitions, parameter declarations, and more. In C+-, there are 3 fundamental parts of a declarator:

1. declaration specifier sequence, represented by rule `declSpecifierSeq`,
2. pointer declarator, represented by rule `pointerDeclarator`,
3. no pointer declarator, represented by rule `noPointerDeclarator`.

`declSpecifierSeq` specifies the underlying simple type, possible const qualifiers, and can be shared by multiple declarators. `pointerDeclarator` possible makes the underlying types pointer(s). `noPointerDeclarator` (see Listing 3.1) then takes the type that has been created by `pointerDeclarator` and has 4 options:

1. Finish the declarator with an identifier.
2. Add parameters and qualifiers, making the current type a function type, and nest into another no pointer declarator.
3. Add braces with potential expression inside, making the current type an array type, and nest into another `noPointerDeclarator`.
4. Nest into another `pointerDeclarator`.

For better understanding, consider the following example of declaration of multiple variables:

```
int i, *const p, **f(), arr[5], (*arr_ptr)[7];
```

5 values are declared:

1. `i` of type `int` (int),
2. `p` of type `int *const` (const pointer to int),
3. `f` of type `int** ()` (function that takes no arguments and returns pointer to pointer to int),
4. `arr` of type `int [5]` (array of 5 ints),
5. `arr_ptr` of type `int (*)[7]` (pointer to array of 7 ints).

Listing 3.2 describes how these declarators are parsed step-by-step.

```
noPointerDeclarator:
    // identifier
    declaratorID
    // function or array
    | noPointerDeclarator (
        // function
        parametersAndQualifiers
        // array
        | LeftBracket constantExpression? RightBracket
    )
    // nested pointer declarator
    | LeftParen pointerDeclarator RightParen;
```

Listing 3.1: noPointerDeclarator grammar rule

3.2.2 Binary Expressions

Different binary operators have different priorities [30]. In the official C++ grammar, this is achieved by having individual rules for operators of the same priority level, which contain the rule of the next higher priority operator. This is reflected in the ANTLR GitHub grammar [29], as show in Listing 3.3. This is what is needed to express all the binary operators found in C+- language. However, thanks to priorities in ANTLR4 grammars, we can simplify these 10 rules into one single rule as show in Listing 3.4. This makes the grammar shorter, makes the operator precedence clear, and simplifies visitation of the parse tree, as we need to implement only one visit method instead of ten.

3.2.3 Grammar Actions

Ideally, we would not have to put any ANTLR actions into our grammar since they make the grammar parser language-specific. However, C++ grammar is full of ambiguities and some can only be resolved by the context. Consider the code `i * j;`. Is this a declaration of variable `j` of type `i*`, or a multiplication of variables `i` and `j`? Without any knowledge of what code came before this line, this cannot be determined. In order to resolve this specific ambiguity, we add the following two actions (Listing 3.5 and Listing 3.6) to our grammar. The action in the rule `classHead` adds the name of a newly declared class into list of known types. The second action in rule `simpleTypeSpecifier` allows an identifier (that comes from rule `theTypeName`) to represent a type name only if it had been added to the list of known types. When the parser encounters the snippet `i * j;`, it first tries the route of variable declaration. When it gets to the first alternative in rule `simpleTypeSpecifier`, it proceeds if `i` has been added to the type list, and cuts off this branch of parsing if not. If this path was cut off, the parser then tries the multiplication path. [6]

[6] To see the difference, take a look at the valid sample in `tests/valid_inputs/i_times_j_amb.cpp`. Either run the `antlr4-parse` command with `-gui` to show the parse tree, or use our compiler with the `--ast-dump-raw` option to see the AST after parsing.

```
/* step 1, declaration specifier sequence */
int         // 'int' is the only declaration specifier,
            // continue with type 'int' for all declarators

/* step 2, pointer declarators, look for '*' from the left */
i               // no '*', continue with type 'int'
*const p        // '*const', continue with type 'int *const'
**f()           // '**', continue with type 'int **'
arr[5]          // no '*', continue with type 'int'
(*arr_ptr)[7]   // no '*', continue with type 'int'

/* step 3, no pointer declarators, look from the right */
i                   // identifier, return {i, int}
p                   // identifier, return {p, int *const}
f()                 // parameter list '()', continue into 'f'
                    // with type 'int** ()'
arr[5]              // array size '[5]', continue into 'arr'
                    // with type 'int[5]'
(*arr_ptr)[7]       // array size '[7]', continue into
                    // '(*arr_ptr)' with type 'int[7]'

/* step 4, nested no pointer declarators, look from the right */
f           // identifier, return '{f, int** ()}'
arr         // identifier, return '{arr, int[5]}'
(*arr_ptr)  // pointer declarator inside (),
            // continue into '*arr_ptr' with type 'int[7]'

/* step 5, pointer declarator, look for '*' from the left */
*arr_ptr    // '*', continue into 'arr_ptr'
            // with type 'int (*)[7]'

/* step 6, no pointer declarator, look from right */
arr_ptr     // identifier, return {arr_ptr, int (*)[7]}

/* all declarators have been parsed */
```

Listing 3.2: Declarator parsing example

```
multiplicativeExpression:
        pointerMemberExpression (
                (Star | Div | Mod) pointerMemberExpression
        )*;
additiveExpression:
        multiplicativeExpression (
                (Plus | Minus) multiplicativeExpression
        )*;
shiftExpression:
        additiveExpression (shiftOperator additiveExpression)*;
shiftOperator: Greater Greater | Less Less;
relationalExpression:
        shiftExpression (
                (Less | Greater | LessEqual | GreaterEqual)
        shiftExpression
        )*;
equalityExpression:
        relationalExpression (
                (Equal | NotEqual) relationalExpression
        )*;
andExpression: equalityExpression (And equalityExpression)*;
exclusiveOrExpression: andExpression (Caret andExpression)*;
inclusiveOrExpression:
        exclusiveOrExpression (Or exclusiveOrExpression)*;
logicalAndExpression:
        inclusiveOrExpression (AndAnd inclusiveOrExpression)*;
logicalOrExpression:
        logicalAndExpression (OrOr logicalAndExpression)*;
```

Listing 3.3: Binary operator precedence expressed by 10 rules

3.3 AST Building

Now let's turn to how our AST is built from the parse tree. Once ANTLR parses the input with our provided grammar, it returns the parse tree. Consider the *literal* rule in Listing 3.7 from our grammar, which handles literals. For the C++ parser, ANTLR automatically generates the class in Listing 3.8 to represent this rule. The methods that return `antlr4::tree::TerminalNode*` such as `IntegerLiteral` or `CharacterLiteral` correspond to terminals in the *literal* grammar rule. If a parsed literal is an integer literal, only the `IntegerLiteral` method will return a pointer to an object, while the other methods will return `nullptr`. We can use this to implement the `visitLiteral` method in our parse tree visitor, which takes a `LiteralContext*` and return an `ast::node_ptr<ast::Expr>` node that represents the literal in the AST. Here's the function's code of first three cases: Note that at the first line, we use the `src_info` function, which creates `ast::SourceInfo` from the rule context.

Oftentimes, these visit methods call other visit methods inside them, and they sometimes have to change the node type using the `ast::change_node`

```
binaryExpression:
    castExpression
    | binaryExpression (Star | Div | Mod) binaryExpression
    | binaryExpression (Plus | Minus) binaryExpression
    | binaryExpression (Greater Greater | Less Less)
        binaryExpression
    | binaryExpression (Less | Greater |
        LessEqual | GreaterEqual) binaryExpression
    | binaryExpression (Equal | NotEqual) binaryExpression
    | binaryExpression And binaryExpression
    | binaryExpression Caret binaryExpression
    | binaryExpression Or binaryExpression
    | binaryExpression AndAnd binaryExpression
    | binaryExpression OrOr binaryExpression
    ;
```

Listing 3.4: Single rule with ANTLR priorities to handle binary operator precedence

```
classHead:
    classKey classHeadName {add_ty($classHeadName.text);} ;
```

Listing 3.5: Grammar rule classHead

```
simpleTypeSpecifier:
    {ty_exists()}? theTypeName
    | Char
    | Bool
    | Int
    | Double
    | Void;
```

Listing 3.6: Grammar rule simpleTypeSpecifier

```
literal:
    IntegerLiteral
    | CharacterLiteral
    | FloatingLiteral
    | StringLiteral
    | BooleanLiteral
    | PointerLiteral;
```

Listing 3.7: `literal` grammar rule

```
class  LiteralContext : public antlr4::ParserRuleContext {
  public:
    LiteralContext(antlr4::ParserRuleContext *parent,
        size_t invokingState);
    virtual size_t getRuleIndex() const override;
    antlr4::tree::TerminalNode *IntegerLiteral();
    antlr4::tree::TerminalNode *CharacterLiteral();
    antlr4::tree::TerminalNode *FloatingLiteral();
    antlr4::tree::TerminalNode *StringLiteral();
    antlr4::tree::TerminalNode *BooleanLiteral();
    antlr4::tree::TerminalNode *PointerLiteral();
  };
```

Listing 3.8: `LiteralContext` class

```
declaration:
    simpleDeclaration
    | functionDefinition
    | classDefinition
    | emptyDeclaration;
```

Listing 3.9: `declaration` rule

function described before. An example of this is the `declaration` rule in Listing 3.9 and the corresponding `visitDeclaration` method in Listing 3.10.

This is how most visit rules are implemented: they visit child rules, obtain AST nodes from them, and combine these nodes into another node, or change the node type.

3.4 Error Reporting

If the input doesn't match the grammar, the parser will automatically print an error. Consider the code snippet and the error produced by ANTLR (and our compiler) in Listing 3.11.

However, if the input matches the grammar, but still contains an error (perhaps because our grammar is too loose in some places), our visitor will report the error along with printing the line which caused it. Consider the example with extraneous `const` specifier in Listing 3.12.

```
ast::node_ptr<ast::Declaration>
MyParserVisitor::visitDeclaration(
  ParserParser::DeclarationContext *ctx) {
    auto source_info = src_info(ctx);
    if(auto child = ctx->simpleDeclaration()) {
        auto sd = visitSimpleDeclaration(child);
        return change_node<SimpleDeclar, Declaration>(move(sd));
    }
    else if(ctx->functionDefinition()) {
        auto fd =
            visitFunctionDefinition(ctx->functionDefinition());
        return change_node<FuncDef, Declaration>(move(fd));
    }
    else if(ctx->classDefinition()) {
        auto cd = visitClassDefinition(ctx->classDefinition());
        return change_node<ClassDef, Declaration>(move(cd));
    }
    else if(ctx->emptyDeclaration())
        return make_node<EmptyDeclaration,
            Declaration>(move(source_info));
    report_unhandled_case("visitDeclaration", ctx);
}
```

Listing 3.10: `visitDeclaration` method

```
// declaration with unknown type
UnType t;

/* produced error: */
// line 2:0 no viable alternative at input 'UnType'
// error: invalid syntax
```

Listing 3.11: Input that doesn't match the grammar

```
const const int a;

/* produced error: */
// error: line 1: multiple const qualifiers
// const const int a;
```

Listing 3.12: Input that matches grammar, but still is syntactically invalid

Chapter **4**

Semantic Analysis

Code that passes syntactic analysis (conforms to the grammar) may still break the language rules in plenty of ways. This is where semantic analysis comes in. It takes in the AST, checks it for correctness and possibly updates it by adding new nodes or modifying old ones. In our project, this is done by the class `SemanticChecker`.

4.1 Program Correctness

First big responsibility of the `SemanticChecker` is to check that the program is correct according to the language rules. There are many ways a grammatically correct program can break the rules.

4.1.1 Scoping

The scoping rules of C+- are pretty simple. We have the global scope. Then, every class has its own scope. A function has its own scope with arguments and locally declared variables. Every for loop has its scope. And finally, a compound statement introduces a new scope unless it is the body of a function or a for loop.

In our semantic checker, the compiler keeps track of scopes by having a linked list, where every scope has a pointer to its parent scope (which is `nullptr` for the global scope). The compiler keeps a pointer to the current scope, and to the global scope. When a new scope is be introduced, the compiler creates it, sets the current scope as its parent, and saves the pointer to it into the current scope variable. When a scope finishes, current scope pointer back to its parent.

A scope itself than contains a map that maps names (identifiers) to lists of values, `std::map<std::string, std::vector<ScopeValue>>`. The reason we use a vector is due to function overloading – one identifier can refer to multiple functions.

4.1.2 Type Checking

There are many places where the compiler has to check if types match, or one type can be converted to another. For that, the compiler has the function `implicitly_convertible` which decides whether (and how) an rvalue of one type can be converted rvalue of another type. There's a lot of type checking in the compiler, the following are just some examples:

- `a = b` assignment expression
  - `a` type must not be const-qualified,
  - `a` cannot be an array or a function,
  - `b` must be implicitly convertible to the type of `a`.
- `a op b` binary expression
  - `op` must be defined for types of `a` and `b`, or they must be implicitly convertible such that `op` is defined [7].
- `a[b]` subscript expression
  - `a` must be of pointer type [8],
  - `b` must be of integral type.
- `a ? b : c` conditional ternary expression
  - `a` must be implicitly convertible to bool,
  - `b` and `c` must be implicitly convertible to a common type.
- `++ --` operators
  - the operand must be of integral or pointer type (C`+-` doesn't allow for operator overloading).
- `~` bit-not operator
  - the operand must be of integral type.
- `(type) expr` cast expression
  - `expr` must be explicitly convertible to `type`.

4.1.3 Value Category Checking

Another property of expressions, their value category, can also lead to incorrect programs. Consider code `5 = a + b`. The compiler has to catch that `5` is an rvalue, and that it cannot be on the left-side of assignment.

These are all the value category checks that the compiler performs:

- `a = b` assignment expression
  - `a` must be an lvalue.

[7] In the source code, this is handled in the `SemanticChecker::conversions_for_bin_op` method

[8] If `a` is of array type, it will first be decayed to pointer. More on that later in the chapter.

```
void f() {
    break; // error, not deep enough to call break
}

void g() {
    while(true)
        continue 2; // error, not deep enough to call continue 2
}
```

Listing 4.1: Incorrect break and continue

- `a ? b : c` conditional ternary expression
  - if `b` and `c` operands are lvalues and no implicit conversions are needed, the result is an lvalue,
  - otherwise the result is an rvalue.
- `++ --` operators
  - operand must be an lvalue.
- `&` address-of operator
  - the operand must be an lvalue.
- a.b member access
  - `a` cannot be an rvalue (see Chapter 2).

4.1.4 Statements

During semantic analysis, most statements are quite simple to handle. Usually, their children are simply visited. But these statements require additional processing:

Break and Continue Check that the `break` or `continue` level does not exceed current loop depth (both of the functions in Listing 4.1 should cause an error during compilation).

Return statement Check that the returned expression can be implicitly converted to the function return type; or if the return statement lacks the return expression, check that the function return type is `void`. Possible cases are shown in Listing 4.2.

For, While, DoWhile, If statements Check that the condition expression is implicitly convertible to bool. Also for the loop statements (For, While, DoWhile), increase current loop depth by one for the duration of the body.

Compound statement Create a new scope (unless specified not to according to Chapter 2, see example in Listing 4.3).

```
double f1() {
    return 42;  // ok, 'int' is implicitly convertible
                // to 'double'
}
double f2() {
    double *ptr;
    return ptr; // error, 'double*' is not convertible
                // to 'double'
}
void f3_help() {
    return;     // ok, return without expression
                // in 'void' function
}
void f3() {
    return f3_help();   // ok, both return type and
                        // returned expression are 'void'
}
void f4() {
    return f2();  // error, type of f2() is not 'void'
}
```

Listing 4.2: Return statement options

```
 void foo(int a) {
    int a;  // error: redeclaration of 'a'

    for(int i = 0; i < 5; i++) {
        int i;  // error: redeclaration of 'i'
        // ...
    }
}
```

Listing 4.3: Special cases of redeclaration inside compound statement

4.1.5 Functions

Functions require a lot of logic from the compiler. They have two stages – declaration and definition. They can be re-declared, and their parameter types can differ in their const-qualification between different declarations. They can have default arguments. They can be overloaded, and function calls have to go through overload resolution. Let's take a look at how the compiler handles some of these issues.

Function name resolution When a function (declaration or definition) is first encountered, the function must be declared in the scope. First, the compiler has to check that the name doesn't already refer to a different kind of object, such as a variable. Then, if the scope already contains function(s) under this name, the new function must be compared with each of these functions.

```
void f(double d = nullptr); // error: object of type
                            // 'nullptr_t'
                            // cannot be converted to type
                            // 'double'
void g(int i = 5, double d);    // error: gap in default
                                // arguments
```

Listing 4.4: Invalid default arguments

```
int f(int i);
// int i1 = f(); // error: no matching function for call of 'f'
int f(int i = 5);   // re-declare function with default arg
int i2 = f();       // ok, 'f' is called as 'f(5)'
```

Listing 4.5: Late definition of default argument

```
int f(int i, double d = 3.14);
int f(int i, double d = 3.14);  // error: default argument
                                // redefinition
```

Listing 4.6: Default argument redefinition

Possible cases are:

1. the new function is a re-declaration of an old function,
2. the new function is a different overload than all the other overloads,
3. the new function differs from an old function only in return type (which is invalid), and an error must be raised.

Default arguments After name resolution and function declaration, the compiler handles default arguments. The provided values must be convertible to the parameter types. Also, once default argument value is defined for parameter number i, all parameters $i+1$, $i+2$, …, must also have default argument values. Both declarations in Listing 4.4 should not compile. If the function passes those checks, the compiler has to save the default argument values as it goes through the program. This is illustrated in Listing 4.5.

Finally, the compiler must check that default arguments are only defined once, even if the shared default argument values are the same (see Listing 4.6).

Overload resolution When a function is called, the compiler does overload resolution to determine whether there's exactly one function in current scope that matches the call the best. This is done according to the rules in Chapter 2.

```
struct S {
    int foo() {
        return this->m;
    }

    int m;
};
```

Listing 4.7: Class member referenced before declaration

4.1.6 Classes

Moving onto classes, they are the only place in our language where objects can be referred to before they are declared. Consider the code in Listing 4.7. In member function `foo`, we are referring to member field `m`, which has not been declared at the time of `foo`'s definition. This is viable code in C++ and in C+-. This means that the compiler cannot process a class definition with just one linear pass. The semantic analyzer has to do (at least) two passes over the class definition.

In the first pass, it collects all the members and their types, and creates the class symbol table from this. Function default arguments or bodies are not processed, as they might refer to members that have not been declared yet.

In the second pass, function default arguments and bodies are processed using the symbol table created in the first pass.

When processing default arguments of methods, we must check that they do not refer to non-static class members or the implicit `this` argument [31]. However, they can contain class constructors [9].

Finally, whenever a class member is accessed outside the class, the compiler checks that it is not private.

4.2 AST Structure Modifications

Second class of responsibilities of the semantic checker is to add new nodes such that the AST truly reflects the written program, as the parser is limited in its capabilities.

4.2.1 Implicit Conversions

Consider the program in Listing 4.8. The builtin operator + takes two rvalues, so `a` must first be implicitly converted to an rvalue. However, this is not reflected in the AST before semantic analysis, because the parser doesn't differentiate between lvalue and rvalue expressions (see Listing 4.9). So the semantic checker adds a node for this conversion to the AST (see Listing 4.10).

[9] In C++, a default argument can also contain references to static class methods or fields that have not been declared yet. However, C+- doesn't allow for static members, so we are only concerned with constructors.

```
int a = 37;
int b = a + 5;  // implicit lvalue-to-rvalue for 'a'
```

Listing 4.8: Example with lvalue-to-rvalue conversion

```
-InitDeclarator <line:2:5>
 |-Decl <line:2:5> b 'int'
  -BinaryExpr <line:2:9> '+'
   |-IdExpr <line:2:9> a
    -IntLiteral <line:2:13> 5
```

Listing 4.9: AST of `b` before semantic analysis

```
-InitDeclarator <line:2:5>
 |-Decl <line:2:5> b 'int'
  -BinaryExpr <line:2:9> '+'
   |-LValToRValExpr <line:2:9>
   |  -IdExpr <line:2:9> a, declared on line 1
    -IntLiteral <line:2:13> 5
```

Listing 4.10: AST of `b` after semantic analysis

The same has to be done with the implicit array-to-pointer decay conversion. In C+-, when an lvalue of array type is used in an expression context, it is automatically decayed to pointer to the first element unless:

- the array is the operand of the `&` address-of operator,
- the array is the operand of `sizeof` operator [10].

Finally there are implicit type conversions. These can be found in many places, such as converting condition expression to `bool`, one operand of binary expression to another, or converting function call argument to type of function parameter. Since the converted value must always be an rvalue, implicit type conversions are often preceded by lvalue-to-rvalue or array-to-pointer-decay conversions. This is illustrated in the example in Listing 4.11 and the corresponding AST in Listing 4.12.

4.2.2 Other Structure Modifications

Now let's turn to some of the cases that don't fall under the category of implicit conversions.

[10] This is based on C array to pointer conversion, since they cover what is implemented in C+- rules [32].

```
void f(void *ptr, int size);
int main() {
    int arr[5];
    char c;
    f(  arr, // int[5] -> int*, int* -> void*
        c    // lvalue-to-rvalue, char->int
        );
}
```

Listing 4.11: Implicit conversions combined

```
-CallExpr <line:5:5> 'void (ptr to void, int)', ...
 |-IdExpr <line:5:5> f, declared on line 1
 |-ImplicitTypeCastExpr <line:5:5> 'ptr to void'
 |  -ArrToPtrExpr <line:0:0>
 |    -IdExpr <line:5:9> arr, declared on line 3
  -ImplicitTypeCastExpr <line:5:5> 'int'
    -LValToRValExpr <line:5:5>
      -IdExpr <line:6:9> c, declared on line 4
```

Listing 4.12: AST of implicit conversions combined

```
int main() {
    int arr[5];
    sizeof arr;
}
```

Listing 4.13: `sizeof arr` example

`sizeof` an expression the unary expression `sizeof` with an expression as its operand is turned into a `sizeof(type)` expression. The reason is that the type of the operand is known at compile type, and so our compiler can replace the `ast::UnaryExpr` node with a `ast::SizeofTypeExpr` node. Consider the example in Listing 4.13, the AST before semantic analysis (Listing 4.14) and the AST after semantic analysis (Listing 4.15).

The advantage of this is a simpler AST. However, this can be a disadvantage for refactoring tools that would use the AST to rename variables [13].

```
-ExprStmt <line:3:5>
  -UnaryExpr <line:3:5> 'sizeof'
    -IdExpr <line:3:12> arr
```

Listing 4.14: AST of `sizeof arr` before semantic analysis

```
-ExprStmt <line:3:5>
  -SizeofType <line:3:5> '[5 x int]'
```

Listing 4.15: AST of `sizeof arr` after semantic analysis

```
 struct S {
    int a;
    int foo() {
        return a + 5;   // implicit 'this->a'
    }
};
```

Listing 4.16: Implicit class member access

```
-BinaryExpr <line:4:16> '+'
 |-IdExpr <line:4:16> a
   -IntLiteral <line:4:20> 5
```

Listing 4.17: AST of implicit member access before semantic analysis

Implicit member access inside class methods During parsing, any time an identifier is encountered in the context of an expression, such as `a` in `a + 5`, an `ast::IdExpr` node is created. This node signifies the use of a value from the local scope. However, `a` doesn't always have to be from the local scope; it can refer to a non-static class member. Consider the code in Listing 4.16. If we look at the AST before semantic analysis (Listing 4.17), the `a` in `a+5` is an `ast::IdExpr`, even though it refers to a non-static class member. This is fixed during semantic analysis (see Listing 4.18).

Adding default arguments to functions calls When the programmer calls a function with less arguments than is the number of parameters, default arguments have to be used. During semantic analysis, the compiler recognizes this and adds these default arguments to the call argument list.

```
-BinaryExpr <line:4:16> '+'
 |-LValToRValExpr <line:4:16>
 |  -MemberAccess <line:4:16> ->a
 |    -ImplicitThisExpr <line:4:16>
  -IntLiteral <line:4:20> 5
```

Listing 4.18: AST of implicit member access after semantic analysis

```
double global_d;
int foo(int i, double d = global_d, void *ptr = nullptr);
// ...
int a = foo(42);
```

Listing 4.19: Default arguments used in a call

```
-CallExpr <line:4:9> 'int (int, double, ptr to void)', ...
  |-IdExpr <line:4:9> foo, declared on line 2
  |-IntLiteral <line:4:13> 42
  |-DefaultArgExpr <line:4:9>
  |  -LValToRValExpr <line:2:5>
  |    -IdExpr <line:2:27> global_d, declared on line 1
   -DefaultArgExpr <line:4:9>
     -ImplicitTypeCastExpr <line:2:5> 'ptr to void'
       -NullptrLiteral <line:2:49>
```

Listing 4.20: AST of call expression with default arguments

Consider the example in Listing 4.19 and the corresponding AST of the call expression in Listing 4.20.

4.3 AST Node Modifications

Sometimes, all that's needed is to modify the nodes that already exist. We discuss these cases and their usefulness in the following paragraphs.

Multiple function declarations C+- allows for multiple declarations of one function. To the programmer, this brings some advantages such as being able to call a function in a translation unit, while only knowing the function signature, and not the function definition. To the compiler engineer, this brings some headache. The compiler has to known whether the function that it's currently processing refers to a function that has already been declared, or not. This matters when we're generating the intermediate representation (in our case LLVM IR), where a function should be created only once. For this reason, when the compiler encounters a function declaration that matches a previous declaration in the same scope, it saves the pointer to the first declaration of this function. This is later used by the compiler during LLVM IR generation.

Name resolution During semantic analysis, the compiler is doing scoping and name resolution. As a part of that, when an identifier is used in an expression, the compiler has to determine what value this name refers to in the current scope (possibly by performing overload resolution for functions). Once it resolves this, it saves a pointer to the declarator of that value in the AST. This allows AST passes that follow semantic analysis not to track symbol tables. For an example, see Listing 4.21 and the corresponding AST in Listing 4.22.

```
int f(int i);
int f(void *p);
int main() {
    int i;
    f(i) + f(nullptr);
}
```

Listing 4.21: Name resolution example

```
-ExprStmt <line:5:5>
  -BinaryExpr <line:5:5> '+'
   |-CallExpr <line:5:5> 'int (int)', ... line: 1
   | |-IdExpr <line:5:5> f, declared on line 1
   |  -LValToRValExpr <line:5:5>
   |    -IdExpr <line:5:7> i, declared on line 4
   -CallExpr <line:5:12> 'int (ptr to void)', ... line: 2
    |-IdExpr <line:5:12> f, declared on line 2
     -ImplicitTypeCastExpr <line:5:12> 'ptr to void'
       -NullptrLiteral <line:5:14>
```

Listing 4.22: AST of `f(i) + f(nullptr);` in Listing 4.21

```
void f() {
    int i = 0;
    i += 3.14;  // value from i must be converted to 'double'
}
```

Listing 4.23: Plus-equals operator with type conversion of left-hand side operand

Assignment left-hand side type When dealing with compound assignment (any assignment operator except for the simple `=`), we face a unique challenge: when we're doing the compound operation before the assignment itself (e.g. the `+` of `+=` operator), we might have to convert the left-hand side value to another type. Consider the code in Listing 4.23 and its equivalent with simple assignment operator in Listing 4.24.

As shown previously in this chapter, the `i` in `i + 3.14` would first undergo lvalue-to-rvalue conversion, and then `int` to `double` type conversion. Finally, the result of the addition would undergo `double` to `int` conversion before being stored back to `i`. We have to replicate this behavior with `i += 3.14`, but we cannot add an lvalue-to-rvalue conversion node to `i`, because as was noted before, the left-hand side of assignment must be an lvalue. So to allow for this, the compiler saves the type to which the left-hand side must be converted in the AST. The AST passes that do code generation after semantic analysis can then use this information to do the following:

```c
void f() {
    int i = 0;
    i = i + 3.14;    // value from i must be converted
                     // to 'double'
}
```

Listing 4.24: Equivalent of Listing 4.23 with simple assign operator

```c
void f() {
        int *p;
        // error, pointer + double is invalid
        p + 3.14;
}
```

Listing 4.25: Example with an error found during semantic analysis

```
line 4:2: error: invalid operands for binary op +:
                    'ptr to int' and 'double'
        p + 3.14;
```

Listing 4.26: Error message of error during semantic analysis

1. load value from lhs (obtain lhs rvalue),
2. convert this lhs rvalue to the type saved in the AST,
3. perform the operation (e.g. +) with converted lhs rvalue and rhs,
4. convert the result back to original lhs type,
5. store the converted result in the lhs.

In section 5.4.4, we show how this is done during the LLVM IR generation.

4.4 Error Reporting

If `SemanticChecker` encounters a semantic error, it reports it with a message of what went wrong, and string of the line. Consider example in Listing 4.25 and the corresponding error message from our compiler in Listing 4.26.

Errors are signaled by an exception from `SemanticChecker`, and so there's no error recovery – the compiler catches only one error at a time.

Chapter **5**

LLVM IR Generation

Now that the AST has been checked for correctness, and has been modified to include all the necessary information such as implicit conversions, we can visit the AST and use the LLVM C++ API to create LLVM IR of our program. In this chapter, we describe some of the interesting details of this process. We don't describe all the basics of the C++ API, as that would be a lot of text, and has been done elsewhere [33, 34]. Readers interested in how specific AST nodes are processed within our compiler are encouraged to look into the source code. The AST pass that generates LLVM IR is handled in the class `LLBuilder`.

Finally, note that we are using LLVM 14 along with some features that are not present in the newer LLVM versions (such as typed pointers).

5.1 Scoping

`LLBuilder` doesn't keep track of scopes, it has is a map that maps declarators to values: `std::map<const ast::Declarator *, llvm::Value *> vals`. In this map, the compiler saves `llvm::AllocaInst*` created for local variables and `llvm::GlobalVariable*` for global variables (the common base type of these two is `llvm::Value`) when they are declared. Later, when a variable is used in an expression context, the compiler uses the `ast::IdExpr::var` member, that was saved during semantic analysis, to find the value in the `vals` map (see Listing 5.1).

```
llvm::Value *LLBuilder::operator()(const ast::IdExpr &node) {
    check(node.var.has_value());
    const ast::Decl *ast_decl = node.var.value();
    return vals.at(ast_decl);
}
```

Listing 5.1: `ast::IdExpr` visit method in `LLBuilder`

```
int a = 40;
int b = 2;
int c = a + b;
```

Listing 5.2: Global variables initialization

5.1.1 Global Variables and Constructors

Global variables have a special place in C++ in that if their declaration includes initialization, they are to be initialized before the `main` function is executed. How do we achieve this behavior in LLVM IR? We could simply pretend that they are executed before `main` by putting the initializer instructions at the beginning of `main`. But what if the compiled program doesn't contain the `main` function, and is intended to be later linked with another program? How do we make sure that the global variables will still be initialized correctly? Fortunately, LLVM IR was developed with C and C++ in mind, and it provides exactly what we need, which is the global variable `@llvm.global_ctors` [35]. Simplified, this is a global variable of array type, where the programmer can store pointers to functions that they want executed before the `main` function. It has two interesting properties:

1. the linkage of this variable is `appending`,

2. the array elements are structs that contain an integer and two pointers.

First, the `appending` linkage specifies that this variable can be declared in multiple (LLVM) modules, and that if those modules are linked together, these lists (arrays) will be concatenated [15]. In the array element type, the struct with one integer and two pointers, the integer specifies the priority of given element (function). The lesser it is, the sooner the function will be executed. For example, if the array `@llvm.global_ctors` contained two struct objects with priorities 100 and 1, respectively, the function from the second struct would be executed before the first. The first pointer in the struct is a pointer to the function that should be executed. Finally, the other pointer allows the programmer to additionally specify when the function should (not) be executed. It is often left as `null` pointer.

Coming back to our `LLBuilder` and the LLVM IR generation, when the compiler encounters a global variable with an initializer, it creates the variable `@llvm.global_ctors`, creates a function where all global initializations will take place, and puts a pointer to this function into the array. Consider the example of program with three global variables in Listing 5.2. Our compiler will produce LLVM IR in Listing 5.3. We can observe how all three variables are initialized in the `run_global_ctors.cpp` function [11] , and that this function is included in the initializer of the `@llvm.global_ctors` variable.

[11] The dot . in the `run_global_ctors.cpp` function name is intentional as to avoid name-collisions with user defined functions.

```
@global_a = global i32 0
@llvm.global_ctors = appending constant [1 x { i32, void ()*, i8* }]
    [{ i32, void ()*, i8* }
    { i32 65535, void ()* @run_global_ctors.cpp, i8* null }]
@global_b = global i32 0
@global_c = global i32 0

define internal void @run_global_ctors.cpp()
    section ".text.startup" {
entry:
  store i32 40, i32* @global_a, align 4
  store i32 2, i32* @global_b, align 4
  %0 = load i32, i32* @global_a, align 4
  %1 = load i32, i32* @global_b, align 4
  %2 = add i32 %0, %1
  store i32 %2, i32* @global_c, align 4
  ret void
}
```

Listing 5.3: Global variables initialization in LLVM IR

5.2 Functions

Functions are the building blocks of LLVM IR. LLVM IR functions are quite similar to C++ functions – they can be declared and/or defined, their signature has the same structure as C++ functions, they have the `ret` instruction to mirror the `return` statement. For the most part, all the compiler has to do to is to visit individual statements within the function and generate code for them. However, there are a few things that require special attention.

5.2.1 Function Arguments

For one, arguments in C++ are by default mutable (unless they are const-qualified), while they are not in LLVM IR. Consider the code in Listing 5.4. This is viable code in C++, which rewrites whatever was in argument `a` with value 5. However, consider the line-by-line equivalent of this code in Listing 5.5. If we try to compile this with `clang`, we will get the error that's at the bottom of Listing 5.5, which says the that operand must be a pointer. The fundamental difference between C++ and LLVM IR here is that in C++, a function argument is an is mutable, while in LLVM IR, it is not mutable. To bridge this gap, our compiler creates an `alloca` instruction for each argument and stores the initial value there, before generating code for the body. Then, the arguments can be treated in LLVM IR as any other variables. If we return back to the last example, the LLVM IR our compiler generates can be found in Listing 5.6.

```c
void foo(int a) {
    a = 5;
}
```

Listing 5.4: Function with mutated argument

```llvm
define void @foo(i32 %a) {
entry:
  store i32 5, i32 %a, align 4
  br label %return

return:                           ; preds = %entry
  ret void
}

; Error caused at compilation:
; test.ll:6:16: error: store operand must be a pointer
;   store i32 5, i32 %a, align 4
;               ^
; 1 error generated.
```

Listing 5.5: Naive LLVM IR representation of code in Listing 5.4

```llvm
define void @foo(i32 %a) {
entry:
  %a.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4   ; store initial value
  store i32 5, i32* %a.addr, align 4
  br label %return

return:                           ; preds = %entry
  ret void
}
```

Listing 5.6: Our LLVM IR representation of code in Listing 5.4

```
int f() {
    return 42;
}
```

Listing 5.7: Function with return instruction

```
define i32 @f() {
entry:
  %ret_val = alloca i32, align 4
  store i32 42, i32* %ret_val, align 4
  br label %return

return:                          ; preds = %entry
  %0 = load i32, i32* %ret_val, align 4
  ret i32 %0
}
```

Listing 5.8: Function with return instruction in LLVM IR

5.2.2 Single Return Instruction

In LLVM IR, a function can have multiple return instructions in different basic blocks. However, it's common practice to have a single return instruction per function (`clang 17`, `julia 1.10.0` and `rustc 1.78.0` do this in their LLVM IR). Our compiler does this as well. When it processes a function definition (`ast::FuncDef`), it

1. creates a basic block called `return`,
2. and an alloca with type of the function return type is created and called `ret_val` (if the function return type is not void).

What happens in the basic block `return` is simple: for void functions, it calls `ret void`. For not void functions, it creates a load from the `ret_val` and calls `ret` with this load.

When a return statement is encountered in the function body, the returned expression result is saved in `ret_val`, and the program branches to `return` basic block. For an example, consider Listing 5.7 and Listing 5.8.

5.2.3 Implicit Return Instructions

In C++, functions that return void or are `main` have an implicit (`void` or `0`) return, which means that if there's no return statement at the end of the function, one is implicitly added. In LLVM IR, however, there are no such exceptions. So how do we achieve this behavior? First, for the LLVM `main` function, we store `0` into the `ret_val` variable before entering the function body. Then, for both the `void` functions and the `main` function, the compiler checks if there's no terminator instruction at the end of the body, and if not, adds a branch instruction to the `return` basic block (see Listing 5.9).

```
// generate function body
codegen(*node.body);
// generate implicit return for main and void functions
if(! builder.GetInsertBlock()->getTerminator())
    if(node.declarator->id == "main" || ret_ty->isVoidTy())
        builder.CreateBr(return_bb);

/* code from LLBuilder::operator()(const ast::FuncDef &node) */
```

Listing 5.9: Generation of implicit return in `LLBuilder`

5.3 Statements

Statements get much more interesting during code generation than they were during semantic analysis.

5.3.1 For, While, Do While, If

First, these statements: `ast::ForStmt`, `ast::WhileStmt`, `ast::DoWhileStmt`, `ast::IfStmt` have a common skeleton during LLVM IR generation. They all share three features:

1. They have a condition expression that directs control flow.
2. They have a body to which control flow should redirect if the condition evaluates to true.
3. They have a place where control flow should be redirected if the condition evaluates to false.

And then they have their individual rules. We'll illustrate the process of LLVM IR generation on what we deem the most complex of these four statements – the `for` statement.

ForStmt Aside from the three stages mentioned above, a `for` statement possibly has an initializer statement and an extra expression that should be evaluated after each loop iteration. The condition is optional as well. The process of executing a `for` statement goes like this:

1. Execute the initializer statement.
2. Check if the condition is true; if so, go to step 3, otherwise go to step 5 [12].
3. Execute the body.
4. Execute the post-iteration expression and return to step 2.
5. Continue with code that follows the `for` statement.

[12] If the `for` statement doesn't have a condition (e.g. `for(int i = 0; ; i++)`, it always branches from step 2 to step 3, as if the condition was `true`.

```
int fact(const int n) {
    int res = 1;
    for(int i = 1; i <= n; ++i)
        res = i * res;
    return res;
}
```

Listing 5.10: Function calculating factorial

Each of these steps has its own basic basic block in LLVM IR. Consider the example in Listing 5.10, which contains a function that uses a `for` statement to calculate the factorial of a non-negative integer, and the corresponding LLVM IR generated by our compiler in Listing 5.11. In the `entry` basic block, we see the LLVM IR for code before the for loop. Then we branch to `for.pre_3` basic block, which represents the initializer statement `int i = 1`. From then on, the LLVM IR follows the steps described above.

Note that there are some special cases which have to be handled – specifically when the statement body contains a terminator instruction (in C+-, this can come from a `break`, `continue` or a `return`). These are handled with the `llvm::BasicBlock::getTerminator()` and `llvm::pred_empty()` functions.

BreakStmt and ContinueStmt Related to loops are `break` and `continue` statements. They are the same in that cause a premature exit from a loop body. However, they differ in the destination of the exit. In the case of `break`, the entire loop ought to be stopped. In the case of `continue`, the program should skip the rest of the loop body and continue.

To enable this, our compiler keeps two vectors of basic blocks, one for the `break`, the other for `continue`. In the compiler source code, they are named `break_bbs` and `continue_bbs`. Now, whenever a loop is processed, the correct basic blocks are saved in these the vectors before entering the loop body. For an example, consider Listing 5.12, which is a snippet from the visit method of `ast::ForStmt`. First, basic blocks for individual stages of the `for` statement are created. Then, we add the `end` basic block to `break_bbs` and the `post_iter` basic block to `continue_bbs`.

With this setup, the LLVM IR generation `break` statement is simple, as shown in Listing 5.13. Remember that C+- enables `break` and `continue` from multiple loops (see Section 2.3.1), which is why the vector index is calculated the way it is.

5.3.2 Unreachable Code Elimination

C++ allows the programmer to write unreachable (dead) code. Consider the example in Listing 5.14.

This is valid C++ code, even though the declaration statement `int a = -1` is unreachable. Generating this dead code in LLVM IR would not only be waste of bytes, but also lead to ill-formed LLVM IR. The dead code would be placed after a terminator instruction (in our case, a `ret` corresponding to the `return 42`), putting the terminator instruction in the middle of the basic block, which

```
define i32 @fact(i32 %n) {
entry:
  %ret_val = alloca i32, align 4
  %n.addr = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  %res = alloca i32, align 4
  store i32 1, i32* %res, align 4
  br label %for.pre_3

for.pre_3:                        ; preds = %entry
  %i = alloca i32, align 4
  store i32 1, i32* %i, align 4
  br label %for.cond_3

for.cond_3:                       ; preds = %for.post_3, %for.pre_3
  %0 = load i32, i32* %i, align 4
  %1 = load i32, i32* %n.addr, align 4
  %2 = icmp sle i32 %0, %1
  br i1 %2, label %for.body_3, label %for.end_3

for.body_3:                       ; preds = %for.cond_3
  %3 = load i32, i32* %i, align 4
  %4 = load i32, i32* %res, align 4
  %5 = mul i32 %3, %4
  store i32 %5, i32* %res, align 4
  br label %for.post_3

for.post_3:                       ; preds = %for.body_3
  %6 = load i32, i32* %i, align 4
  %7 = add i32 %6, 1
  store i32 %7, i32* %i, align 4
  br label %for.cond_3

for.end_3:                        ; preds = %for.cond_3
  %8 = load i32, i32* %res, align 4
  store i32 %8, i32* %ret_val, align 4
  br label %return

return:                           ; preds = %for.end_3
  %9 = load i32, i32* %ret_val, align 4
  ret i32 %9
}
```

Listing 5.11: Function calculating factorial in LLVM IR

```
/* code from LLBuilder::operator()(const ast::ForStmt &node)*/

llvm::BasicBlock * preloop = /* ... */,
            * cond = /* ... */,
            * body = /* ... */,
            * post_iter = /* ... */,
            * end = /* ... */;
// add bbs for break and continue
break_bbs.push_back(end);
continue_bbs.push_back(post_iter);
// generate preloop
// ...
```

Listing 5.12: Adding basic blocks to `break_bbs` and `continue_bbs`

```
void LLBuilder::operator()(const ast::BreakStmt &node) {
    // break_level=1 will result in break_bbs.back()
    llvm::BasicBlock *bb = break_bbs.at(break_bbs.size() -
                                        node.break_level);
    builder.CreateBr(bb);
}
```

Listing 5.13: `LLBuilder` visit method of `ast::BreakStmt`

```
int foo() {
    return 42;
    int a = -1;
}
```

Listing 5.14: Unreachable code

is wrong [15]. To avoid generating dead code (statements), our compiler checks if the current basic block contains a terminator instruction, and if so, it doesn't generate the LLVM IR (see Listing 5.15).

5.4 Expressions

Producing LLVM IR for expression AST nodes is usually quite straight-forward. In this section, we explore some exceptions to this rule.

5.4.1 Representation of `void*` and `nullptr`

Most C++ types have a logical counterpart in LLVM IR. Simple type `int` can be represented as `i32` (assuming a four-byte integer). Pointer type `int*` can be

```
void LLBuilder::operator()(const ast::Stmt &node) {
    // don't generate ir for dead code
    if(builder.GetInsertBlock()->getTerminator())
        return;
    return std::visit(*this, node);
}
```

Listing 5.15: Skipping unreachble code during LLVM IR generation

represented as i32* [13]. Array type int[5] can be represented as [5 x i32]. Function type void (int, char) can be represented as (void) (i32, i8). One exception to this intuitive type translation from C++ to LLVM IR is the pointer to void type void*. Even though LLVM IR has the type void as a possible function return type, it does not allow for native pointer to void [37]. Thus our compiler must use a different LLVM IR type to represent our C++ void* type. We use LLVM IR i8* (which is suggested by clang error if we try to compile LLVM IR with void*).

We face a similar issue with nullptr, which doesn't have a single representation in LLVM IR with typed pointers. So we use the LLVM IR null value of i8*. This makes the implicit conversion from nullptr_t to T* easy, since the LLVM pointer to pointer conversion can be used (more on that later in a moment).

5.4.2 Binary Expressions

After implicit conversions take place, a binary expression can fall into one of four categories based on the operands' types:

1. integral and integral,
2. double and double,
3. pointer and pointer,
4. pointer and integer.

Categories 1-3 can be intuitively mapped onto llvm::IRBuilder functions. Listing 5.16 contains a few examples from the LLBuilder::create_binary_op function that does this in our compiler. For integers, we can see that some operations specify that the use of the signed (S) version of the operation such as SDiv.

Note that the create_binary_op function is not only used when visiting ast::BinaryExpr. It is used by other expressions that can be (at least partly) decomposed into a binary operation. a[b] is equivalent to a+b, c++ requires a c+1 to happen at some point, etc.

[13]Note that since LLVM 17, typed pointers such as i32* are no longer supported [36]

```
if(lhs_ty->isIntegerTy() && rhs_ty->isIntegerTy()) {
    // llvm requires binary operands to be of the same type
    check(lhs_ty == rhs_ty);
    switch(op) {
        case ast::Plus: return builder.CreateAdd(lhs, rhs);
        case ast::Minus: return builder.CreateSub(lhs, rhs);
        case ast::Star: return builder.CreateMul(lhs, rhs);
        case ast::Div: return builder.CreateSDiv(lhs, rhs);
        case ast::Mod: return builder.CreateSRem(lhs, rhs);
// ...
// doubles
else if(lhs_ty->isDoubleTy() && rhs_ty->isDoubleTy()) {
    check(lhs_ty == rhs_ty);
    switch(op) {
        case ast::Plus: return builder.CreateFAdd(lhs, rhs);
        case ast::Minus: return builder.CreateFSub(lhs, rhs);
        case ast::Star: return builder.CreateFMul(lhs, rhs);
// ...
// pointer indexing
else if(lhs_ty->isPointerTy() && rhs_ty->isIntegerTy()) {
    llvm::Type *elem_type = lhs_ty->getPointerElementType();
    llvm::Value *index = nullptr;
    switch(op) {
        case ast::Plus:
            index = rhs;
            break;
        case ast::Minus:
            // make it 'lhs + (-rhs)'
            index = create_unary_minus(rhs);
            break;
        default:
            check(false, "unimplemented operator for"
                         "pointer and integer");
    }
    return builder.CreateGEP(elem_type, lhs, index);
}
// pointer comparison and difference
else if(lhs_ty->isPointerTy() && rhs_ty->isPointerTy()) {
    check(lhs_ty == rhs_ty);
    switch(op) {
        // use unsigned comparison for pointers
        case ast::Greater: return builder.CreateICmpUGT(lhs,
                                                         rhs);
        case ast::Less: return builder.CreateICmpULT(lhs, rhs);
// ...
```

Listing 5.16: Generating code for binary expressions with `llvm::IRBuilder`

```
// to bool conversions
else if(dest_ty == types.at("bool")) {
    llvm::Value *zero = llvm::Constant::getNullValue(val_ty);
    llvm::Value *res = create_binary_op(val, zero,
                                        ast::NotEqual);
    res->setName("tobool");
    return res;
}
```

Listing 5.17: Implementation of *to bool* conversion in `LLBuilder`

5.4.3 Type Conversions

Type conversions, whether implicit or explicit, can occur in many places in the AST. Here we take a look at how they're done in LLVM IR. They are handled in the `LLBuilder::convert` function.

To `bool` conversion The conversion *to bool* is special. That's because it's not really a type conversion, but rather a binary operation expressing inequality of the converted value to zero or `nullptr`. And as such, it is implemented in `LLBuilder` (see Listing 5.17).

Integer conversions For integer conversions, LLVM IR provides builtin functions. Integer extension (widening) can be achieved with `sext` and `zext` instructions, which do signed and zero extension respectively. Integer truncation (narrowing) can be achieved with the `trunc` instruction. The `llvm::IRBuilder` provides convenient functions `CreateSExtOrTrunc` and `CreateZExtOrTrunc`, which create either extension or truncation based on the converted value type and the destination type. We use the `CreateSExtOrTrunc` function, since C+- integers are signed. One special case that has to be handled is `bool` to `char` or `int` extension. Because `LLBuilder` uses the `i1` type to represent `bool`, we must use the `zext` instruction in order to have `true` converted to `1`. If `sext` was used, `0b1` (`i1`, value: true) would become `0b11111111` (`i8`, value: -128). We need it to become `0b00000001` (`i8`, value: 1), so we use `zext`.

`double` to `int` and `int` to `double` For conversions between signed integer and floating types, LLVM IR has the `sitofp` (signed integer to floating point) and `fptosi` (floating point to signed integer) instructions.

Pointer to pointer conversion For pointer to pointer conversion, we use the LLVM IR `bitcast` instruction.

Example with all conversions Example code in Listing 5.18 and the corresponding LLVM IR in Listing 5.19 show how these conversions look in the LLVM IR.

```c
void f() {
    int i, *p;
    double d;
    char c;
    (bool)      p;    // ptr to bool
    (bool)      i;    // int to bool
    (int)       c;    // integer extension
    (char)      i;    // integer truncation
    (double)    i;    // si to fp
    (int)       d;    // fp to si
    (void*)     p;    // pointer to pointer
}
```

Listing 5.18: Possible type conversions

```llvm
define void @f() {
entry:
  %i = alloca i32, align 4
  %p = alloca i32*, align 8
  %d = alloca double, align 8
  %c = alloca i8, align 1
  %0 = load i32*, i32** %p, align 8
  %tobool = icmp ne i32* %0, null   ; ptr->bool
  %1 = load i32, i32* %i, align 4
  %tobool1 = icmp ne i32 %1, 0      ; int->bool
  %2 = load i8, i8* %c, align 1
  %int_conv = sext i8 %2 to i32     ; char->int
  %3 = load i32, i32* %i, align 4
  %int_conv2 = trunc i32 %3 to i8   ; int->char
  %4 = load i32, i32* %i, align 4
  %sitofp = sitofp i32 %4 to double ; int->double
  %5 = load double, double* %d, align 8
  %fptosi = fptosi double %5 to i32 ; double->int
  %6 = load i32*, i32** %p, align 8
  %ptrcast = bitcast i32* %6 to i8* ; ptr->ptr
  br label %return

return:                     ; preds = %entry
  ret void
}
```

Listing 5.19: Possible type conversions in LLVM IR

```
void f() {
    int i = 0;
    i += 3.14;  // value from i must be converted to 'double'
}
```

Listing 5.20: Compound assignment with type conversion of left-hand side operand

```
define void @f() {
entry:
  %i = alloca i32, align 4
  store i32 0, i32* %i, align 4
  %0 = load i32, i32* %i, align 4
  %sitofp = sitofp i32 %0 to double
  %1 = fadd double %sitofp, 3.140000e+00
  %fptosi = fptosi double %1 to i32
  store i32 %fptosi, i32* %i, align 4
  br label %return

return:                          ; preds = %entry
  ret void
}
```

Listing 5.21: LLVM IR of compound assignment with type conversion of left-hand side operand

```
int main() {
    int i1 = 1, i2 = 21;
    bool b = true;
    return b ? 2*i2 : -i1;
}
```

Listing 5.22: Ternary conditional operator

5.4.4 Compound Assignment

Earlier in Section 4.2.2, we said that compound assignment requires special attention during code generation. The reason being that the left-hand side (assignment destination) is an lvalue, while it is also needed as an rvalue for the compute operation. `LLBuilder` uses the algorithm outlined in Chapter 4. For an example, consider the C++ code in Listing 5.20 and the LLVM IR equivalent generated by our compiler in Listing 5.21.

5.4.5 Short-Circuit Evaluation

Another feature that's in C++ (and in most programming languages) is short-circuit evaluation [38]. This means that for the following three operators:

1. `a && b` logical and operator, b will only be evaluated if a is evaluated to true,

2. `a || b` logical or operator, b will only be evaluated if a is evaluated to false,

3. `a ? b : c` ternary conditional operator, if `a` is evaluated to true, only `b` is evaluated; otherwise only `c` is evaluated.

This has several advantages over always evaluating all operands:

- Avoid unnecessary resource consumption, e.g.:

  ```
  nine_out_of_ten_times_true() || very_time_expensive()
  ```

  will not call `very_time_expensive` nine out of ten times (unless the function names lie, of course).

- Allow the programmer to write code that's safe with short-circuit evaluation, but could cause undefined behavior (e.g. segfault) without it:

  ```
  ptr ? ptr->value : -1
  ```

In order to achieve short-circuit evaluation, we use the `phi` instruction. Consider the `a ? b : c` conditional ternary operator as a showcase. The compiler creates three basic blocks `then`, `else` and `end`. The condition `a` is evaluated and based on the result, the program branches either to `then` (true) or `else` (false). Then, `b` is generated starting [14] in `then` and `c` is generated starting in `else`. After generating both `b` and `c`, the compiler unconditionally branches to `end`. Finally, inside basic block `end`, the `phi` instruction is used to obtain either `b` or `c` based on the predecessor basic block. For an example, consider code in Listing 5.22 and the corresponding LLVM IR in Listing 5.23.

5.4.6 `sizeof` Type

To get portable, target independent `sizeof` type, the GEP instruction can be used. The compiler pretends that there is an element of given type at the `null` address, calculates the offset to the element of index 1 from that, and converts this to an integer [39].

5.5 Classes

Finally, classes. During LLVM IR generation, they are split into two parts – class variables (fields) and class methods.

[14] We say *starting*, because code generation of `b` can create some basic blocks and finish in a different block than *then*. For an example, see the LLVM IR generated for program `tests/valid_inputs/ternary_op_contains_log_ops.cpp`

```
; variable initializations
; ...
; here the ternary operator starts:
  %0 = load i1, i1* %b, align 1
  br i1 %0, label %ter_then.4, label %ter_else.4

ter_then.4:                    ; preds = %entry
  %1 = load i32, i32* %i2, align 4
  %2 = mul i32 2, %1
  br label %ter_end.4

ter_else.4:                    ; preds = %entry
  %3 = load i32, i32* %i1, align 4
  %4 = sub i32 0, %3
  br label %ter_end.4

ter_end.4:                     ; preds = %ter_else.4, %ter_then.4
  %5 = phi i32 [ %2, %ter_then.4 ], [ %4, %ter_else.4 ]
  store i32 %5, i32* %ret_val, align 4
  br label %return
```

Listing 5.23: Ternary conditional operator expressed in LLVM IR with the `phi` instruction

```
struct Customer {
    int age;
    double balance;
    char name[50];
};
```

Listing 5.24: Example of a `struct`

```
%Customer = type { i32, double, [50 x i8] }
```

Listing 5.25: LLVM IR representation of Customer `struct` from Listing 5.24.

5.5.1 Structs in LLVM IR

LLVM IR has the keyword `type`, which allows the programmer to create data types consisting of other types, potentially of other composite types (structs). In this sense, the LLVM IR struct is just like a C struct. A difference between C and LLVM IR structs, however, is that members of LLVM IR structs are not named, but identified by their index in the struct type list. Consider the `struct` in Listing 5.24 and the LLVM IR equivalent in Listing 5.25.

```
struct S {
    int i;
    double d;
};

void f() {
    S s;
    s.i = 42;
    s.d = 3.14;
}
```

Listing 5.26: Struct member access

```
%S = type { i32, double }

define void @f() {
entry:
  %s = alloca %S, align 8
  %S.i = getelementptr %S, %S* %s, i32 0, i32 0
  store i32 42, i32* %S.i, align 4
  %S.d = getelementptr %S, %S* %s, i32 0, i32 1
  store double 3.140000e+00, double* %S.d, align 8
  br label %return

return:                 ; preds = %entry
  ret void
}
```

Listing 5.27: Struct member access in LLVM IR

5.5.2 MemberAccess with the GEP Instruction

How do we access class members in LLVM IR? With the GEP instruction. During visitation of `ast::ClassDef`, which represents a class definition, `LLBuilder` saves the order of member fields. Then, when member access is encountered during code generation, `LLBuilder` maps the C++ member name onto the index in LLVM IR representation. Consider the example in Listing 5.26 and the LLVM IR generated by our compiler in Listing 5.27.

5.5.3 Class Methods

LLVM IR doesn't enable the programmer to create class methods, it only allows us to create functions. Thus, we represent class methods (constructors included) simply as LLVM IR functions. During code generation, class methods don't require any special treatment compared to normal functions. However, there's one small optimization that can be. The **this** expression is always a prvalue value in C++ [40], and so an rvalue in C+-.

```
struct S {
    int i;
    int foo() {
        return 2*this->i;
    }
};
int foo(S *const this_) {
    return 2*this_->i;
}
```

Listing 5.28: Class method and global function equivalent

```
%S = type { i32 }

define i32 @"S::foo"(%S* %this) {
entry:
  %ret_val = alloca i32, align 4
  %S.i = getelementptr %S, %S* %this, i32 0, i32 0
  %0 = load i32, i32* %S.i, align 4
  %1 = mul i32 2, %0
  store i32 %1, i32* %ret_val, align 4
  br label %return

return:                    ; preds = %entry
  %2 = load i32, i32* %ret_val, align 4
  ret i32 %2
}

define i32 @foo(%S* %this_) {
entry:
  %ret_val = alloca i32, align 4
  %this_.addr = alloca %S*, align 8              ; extra inst
  store %S* %this_, %S** %this_.addr, align 8    ; extra inst
  %0 = load %S*, %S** %this_.addr, align 8       ; extra inst
  %S.i = getelementptr %S, %S* %0, i32 0, i32 0
  %1 = load i32, i32* %S.i, align 4
  %2 = mul i32 2, %1
  store i32 %2, i32* %ret_val, align 4
  br label %return

return:                    ; preds = %entry
  %3 = load i32, i32* %ret_val, align 4
  ret i32 %3
}
```

Listing 5.29: Class method and global function equivalent in LLVM IR

This means that for the implicit `this` method argument, the compiler doesn't have to generate the initial `alloca` and `store` instructions mentioned earlier in this chapter, because it can never be mutated. And so our compiler doesn't generate them. This is illustrated in Listing 5.28, which contains a class method `S::foo`, and a global function `::foo`, which are structurally equivalent. LLVM IR for `S::foo` is shorter than for `::foo` (see Listing 5.29), because the `alloca`, `store`, `load` instructions were not needed.

Chapter 6

Testing

In the last chapter of this thesis, we describe how we verify the functionality of our compiler. All files related to testing are in the `tests` directory, which has the following structure:

```
tests
├── invalid_inputs ................ directory with invalid C+- samples
│   ├── parsing .... directory with samples that should fail during parsing
│   └── sema ...... directory with samples that should fail during semantic
│       analysis
├── valid_inputs ..................... directory with valid C+- samples
├── astdump.cpp .. program to test that a valid sample produces expected
│   AST dump
├── parsing-invalid.cpp ..... program to test a sample that should fail
│   during parsing
├── README.md description of the directories valid_inputs and invalid_inputs
├── run.cpp program to test that a valid sample produces expected output
│   when run
└── sc-invalid.cpp .... program to test a sample that should fail during
    semantic analysis
```

6.1 Valid Samples

The directory `tests/valid_inputs` contains programs that fall within the C+- specification, and so should pass through our compiler and produce valid LLVM IR. We test that this LLVM IR, when compiled to an executable, produces the correct return code and output. For a `sample` test, there can be the following files:

1. `sample.cpp`: the program itself, this file must always exist.

2. `sample.ret`: this file contains the return code that should be returned from the `main` function of the program. If this file doesn't exist, the expected return value is `0`.

3. `sample.in`: the input that should be passed to the program on `stdin`. If this file doesn't exist, no input is passed to the program.

4. sample.output: the output on stdout that's expected from the program. If this file doesn't exist, the program should output nothing.

5. sample.ast: the expected AST dump of the sample after semantic analysis.

Our samples in tests/valid_inputs are mostly simple programs that test a single functionality, such as a binary operator, an implicit conversion, or function overloading. Many of them test a special case of some feature.

6.2 Invalid Samples

Our tests also include samples that are incorrect according to the language specification, and should fail during compilation.

First, there are samples that should fail during the parsing stage of the compilation. This either means that they don't match the grammar, or they do, but are still wrong syntactically (such as the earlier mentioned example with multiple const qualifiers in Listing 3.12).

Second, there are samples that should be parsed, but fail during semantic analysis. These are also mostly short snippets of code containing a error. These include declaring an array of void elements, call of a private constructor, assigning into an rvalue, and much more.

6.3 Test Programs

Along with these samples, we provide four C++ programs, which work on individual samples.

- run.cpp tests that our compiler compiles a sample into LLVM IR without an error, then uses clang to compile the LLVM IR into an executable, and runs the executable to see if it produces correct output (using the sample.in, sample.ret and sample.output files described above).

- astdump.cpp tests that a valid sample is AST-dumpable before semantic analysis and produces expected AST after semantic analysis.

- parsing-invalid.cpp tests that the compilation of a sample fails during parsing.

- sc-invalid.cpp tests that the compilation of a sample fails during semantic analysis.

6.4 CTest

Our CMakeLists.txt provides a convenient way to run all the tests. After configuring the project with cmake and building it, ctest can be run from the build directory to test our compiler with the above-mentioned samples. run.cpp and astdump.cpp are run on valid samples, sc-invalid.cpp and parsing-invalid.cpp are run on invalid samples.

6.5 Results

Our implementation passes all tests on samples from the `valid_inputs` (more than 150 samples) and `invalid_inputs` (more than 130 samples) directories.

We also compiled our implementation by both `gcc` and `clang` (with optimizations `-O2` and `-fsanitize`) to remove any possible errors, warnings and memory leaks from our implementation. Some warnings were produced by the ANTLR runtime library, which was used as a third party tool in this thesis.

Testing helped us uncover many edge cases of individual features. For example, thanks to testing, we discovered that during LLVM IR generation, the `zext` instruction must be used to for the `bool` (`i1` in our LLVM IR) promotion to `int`. We learnt from this and added the `valid_inputs/true_is_one.cpp` sample.

Furthermore, we were able to confidently do big refactors of the project, and be sure there were no regressions made. Originally, the implementation had semantic analysis and LLVM IR generation merged into one AST pass. When we separated it into two passes, the tests gave us confidence that no functionality was lost during this internal architecture transition.

Conclusion

The goal of this thesis was to implement a working compiler frontend for a subset of the C++ programming language, using ANTLR and LLVM IR.

In Chapter 1, we introduced two existing C++ compilers, GCC and Clang, and then LLVM Intermediate Representation (LLVM IR) and the ANTLR parser generator.

In Chapter 2, we specified all the functionality included in our subset of C++, which we called C+-. This included types, expressions, statements, function, classes and declarations.

In Chapter 3, we took described our implementation of the AST, looked at the grammar of the language and how our compiler uses ANTLR to build the AST from user's input source code.

In Chapter 4, we concerned ourselves with semantic analysis. This included checking for program correctness, modifying the AST to be more information complete, and error reporting.

In Chapter 5, we turned to the process of LLVM IR generation from the AST. We showed how LLVM IR can be used to express high-level programming concepts, such as loops, short-circuit evaluation, and class methods.

In Chapter 6, we describe our testing methods. We use more than 150 valid code samples to test that the described functionality works. We also add more than 130 invalid code samples to test that the compiler catches certain errors, and does so at the correct stage of the compilation.

Future work might be aimed at extending capabilities of the compiler to cover more of the C++ programming language, such as class destructors, class inheritance, virtual methods, templates or exceptions. The current implementation could also be improved by adding consistent name mangling to function overloads in LLVM IR, or by refactoring the AST so that it could store more information from semantic analysis (such as the types of expressions).

Bibliography

[1] Wicht, B. Cache-Friendly Profile Guided Optimization. [phd thesis], [Accessed 2024-05-12]. Available from: https://www.researchgate.net/publication/307545338_Cache-Friendly_Profile_Guided_Optimizationl

[2] Lattner, C. The Golden Age of Compiler in an Era of Hardware/Software co-design. [online], [Accessed 2024-05-12]. Available from: https://canvas.eee.uci.edu/courses/43849/files/17973444/download

[3] Nystrom, R. A Map of the Territory. [online], [Accessed 2024-04-22]. Available from: https://craftinginterpreters.com/a-map-of-the-territory.html

[4] GCC. Single Static Assignment. [online], [Accessed 2024-04-22]. Available from: https://gcc.gnu.org/onlinedocs/gccint/SSA.html

[5] Steven Bosscher, D. N. GCC gets a new Optimizer Framework. [online], [Accessed 2024-05-12]. Available from: https://lwn.net/Articles/84888/

[6] LLVM. LLVM's Analysis and Transform Passes. [online], [Accessed 2024-05-12]. Available from: https://llvm.org/docs/Passes.html

[7] GCC. GCC, the GNU Compiler Collection. [online], [Accessed 2024-04-22]. Available from: https://gcc.gnu.org/

[8] Microsoft. Compiling a C/C++ project. [online], [Accessed 2024-05-12]. Available from: https://learn.microsoft.com/en-us/cpp/build/reference/compiling-a-c-cpp-program?view=msvc-170

[9] GCC. Generic. [online], [Accessed 2024-04-22]. Available from: https://gcc.gnu.org/wiki/GENERIC

[10] GCC. GIMPLE language. [online], [Accessed 2024-04-22]. Available from: https://gcc.gnu.org/wiki/GIMPLE

[11] GCC. Register Transfer Language. [online], [Accessed 2024-04-22]. Available from: https://gcc.gnu.org/wiki/RTL

[12] LLVM. Clang: a C language family frontend for LLVM. [online], [Accessed 2024-04-22]. Available from: https://clang.llvm.org

[13] Apple. Clang vs Other Open Source Compilers. [online], [Accessed 2024-04-22]. Available from: https://opensource.apple.com/source/clang/clang-23/clang/tools/clang/www/comparison.html

[14] LLVM. Clang-tidy. [online], [Accessed 2024-04-22]. Available from: https://clang.llvm.org/extra/clang-tidy/

[15] LLVM. LLVM Language Reference Manual. [online], [Accessed 2024-05-08]. Available from: https://llvm.org/docs/LangRef.html

[16] LLVM. The LLVM Compiler Infrastructure. [online], [Accessed 2024-04-22]. Available from: https://llvm.org

[17] LLVM. The Often Misunderstood GEP Instruction. [online], [Accessed 2024-04-22]. Available from: https://llvm.org/docs/GetElementPtr.html

[18] ANTLR. ANTLR. [online], [Accessed 2024-04-22]. Available from: https://www.antlr.org/

[19] Terrence Parr, S. H. ANTLR v4. [online], [Accessed 2024-05-11]. Available from: https://github.com/antlr/antlr4

[20] Pattis, R. E. EBNF: A Notation to Describe Syntax. [online], [Accessed 2024-05-03]. Available from: https://ics.uci.edu/~pattis/misc/ebnf2.pdf

[21] Koichi. Growing up with an irrational fear of the number four. [online], [Accessed 2024-04-22]. Available from: https://www.tofugu.com/japan/number-four-superstition/

[22] cppreference.com. Implicit conversions. [online], [Accessed 2024-04-25]. Available from: https://en.cppreference.com/w/cpp/language/implicit_conversion

[23] NullUserException. Accessing arrays by index[array] in C and C++. [online], [Accessed 2024-05-14]. Available from: https://stackoverflow.com/q/5073350

[24] cppreference.com. Expressions. [online], [Accessed 2024-04-25]. Available from: https://en.cppreference.com/w/cpp/language/expressions

[25] cppreference.com. Value categories. [online], [Accessed 2024-04-25]. Available from: https://en.cppreference.com/w/cpp/language/value_category

[26] cppreference.com. Overload resolution. [online], [Accessed 2024-04-25]. Available from: https://en.cppreference.com/w/cpp/language/overload_resolution

[27] cppreference.com. Fundamental types. [online], [Accessed 2024-05-12]. Available from: https://en.cppreference.com/w/cpp/language/types

[28] cppreference.com. std::variant. [online], [Accessed 2024-04-25]. Available from: https://en.cppreference.com/w/cpp/utility/variant

[29] Various, I. J. S. . C++14 Grammar. [online], [Accessed 2024-04-24]. Available from: https://github.com/antlr/grammars-v4/tree/master/cpp

[30] cppreference.com. C++ Operator Precedence. [online], [Accessed 2024-05-05]. Available from: https://en.cppreference.com/w/cpp/language/operator_precedence

[31] cppreference.com. Default arguments. [online], [Accessed 2024-05-06]. Available from: https://en.cppreference.com/w/cpp/language/default_arguments

[32] cppreference.com. Array declaration. [online], [Accessed 2024-05-07]. Available from: https://en.cppreference.com/w/c/language/array

[33] LLVM. Kaleidoscope: Implementing a Language with LLVM. [online], [Accessed 2024-05-08]. Available from: https://llvm.org/docs/tutorial/

[34] Rathi, M. A Complete Guide to LLVM for Programming Language Creators. [online], [Accessed 2024-05-08]. Available from: https://mukulrathi.com/create-your-own-programming-language/llvm-ir-cpp-api-tutorial/

[35] LLVM. The 'llvm.global_ctors' Global Variable. [online], [Accessed 2024-04-22]. Available from: https://llvm.org/docs/LangRef.html#the-llvm-global-ctors-global-variable

[36] LLVM. Opaque Pointers. [online], [Accessed 2024-05-10]. Available from: https://llvm.org/docs/OpaquePointers.html

[37] Lattner, C. Eliminating the 'void' Type. [online], [Accessed 2024-05-09]. Available from: https://www.nondot.org/sabre/LLVMNotes/EliminatingVoid.txt

[38] cppreference.com. Logical operators. [online], [Accessed 2024-05-11]. Available from: https://en.cppreference.com/w/cpp/language/operator_logical

[39] Lattner, C. Implementing Portable sizeof, offsetof and Variable Sized Structures in LLVM. [online], [Accessed 2024-04-23]. Available from: https://nondot.org/sabre/LLVMNotes/SizeOf-OffsetOf-VariableSizedStructs.txt

[40] cppreference.com. The this pointer. [online], [Accessed 2024-05-09]. Available from: https://en.cppreference.com/w/cpp/language/this

Appendix **A**

Human Language Analogy to the Compiler Frontend

If we were to draw an analogy between the compiler frontend and natural human language, the job of lexical analysis is to split a sentence (or text) into individuals words and punctuation, and report if an unknown word (say `xalobopa`) was used. The job of syntactic analysis is to determine whether given sentences are correct grammatically. For example, consider the following English "sentence": *"The and if object very called."* We can see that the even though the individual words are viable English words, the sentence as a whole is gibberish, and is not grammatically correct. In the language of compilers, this "sentence" would pass through the lexical analysis (individual words exist), but not through the syntactic analysis (the sentence structure doesn't make sense). Now, a sentence might be grammatically correct, but still not make sense. Consider the following example: *"A grain of sand was riding a bicycle."* This sentence is grammatically correct, but obviously doesn't make sense – grains of sand don't ride bicycles! Consider another example: *"Anna spent the whole week in London. On Monday, she decided to grab breakfast near the Eiffel Tower."* This sure is a good phrase if you wish to infuriate both the English and the French, but other than that, it doesn't make sense – Anna could not have grabbed breakfast near the Eiffel Tower (France), if she was in London (England). We could show many more examples to underline the point: just because a sentence (or text) is grammatically correct, does not imply that it is *semantically* correct. The equivalent is true in programming languages, which is why we need semantic analysis.

Appendix B

Acronyms

ANTLR ANother Tool for Language Recognition

API Application Programming Interface

AST Abstract Syntax Tree

EBNF Extended Backus-Naur Form

LLVM IR LLVM Intermediate Representation

GCC GNU Compiler Collection

GEP Get Element Pointer

RTL Register Transfer Language

SSA Single Static Assignment

Appendix C

Contents of Attached Files

```
impl ................................directory with implementation files
├── README.md ..............................description of the project
├── license.txt ................................license of the project
├── CMakeLists.txt ...............CMakeLists for building and testing
├── examples ....................directory with example C+- programs
├── .gitlab-ci.yml ..................................GitLab CI file
├── src ......................directory with implementation source code
│   ├── ast ....................directory with representation of the AST
│   ├── ast_dumper .........directory with AST visitor for AST dumping
│   ├── ll_builder ...directory with AST visitor for LLVM IR generation
│   ├── parser ...............................directory with the parser
│   ├── semantic_checker directory with AST visitor for semantic analysis
│   ├── type .........................directory with type representation
│   ├── utils ............................directory with shared classes
│   └── visitor_template ...........directory with AST visitor template
├── tests ...............directory with test samples and tester programs
└── ci ............................directory with scripts for GitLab CI
```

Figure C.1: Structure of `impl.zip`

```
thesis ...............directory with files related to the text of the thesis
└── src ................................directory with LaTeX source files
```

Figure C.2: Structure of `thesis.zip`