CS4212 - Compiler Design Study Guide

Sherisse Tan Jing Wen

December 2, 2024

Contents

Chapter 1

Introduction

Definition 1.0.1 – Compilers A Compiler is a program that translates from 1 Source Language into another Target Language.

Typically, this is a translation from a Higher-level language into a Lower-level language, or Intermediate Representation.

A typical Compiler Pipeline is as follows:

Source Code → Lexical Analysis → Parsing → Intermediate Representation/(s) → Target Code

1.1 Motivation for Compilers

Coding in Higher-level languages is much faster, and easier than coding in Lower-level languages. It is easier to read, and easier to reason with as well, although the extent of its readability depends on the particular Syntax of the language in question.

However, Lower-level languages allow us to provide better optimizations and interface more directly with the hardware of the machine.

Having compilers allow us to code in Higher-level languages, while still allowing for the optimizations and standardizations that Lower-level languages offer us.

1.2 Verifying the Correctness of Compilers

The motivation to verify the correctness of compilers can be summed up in 1 line: **Errors in Compilers are inherited by the Program**.

Typically, developers would reason about the behaviour of their program in terms of the Higher-level language they code in. Should the compiler be erroneous, the developer will not be able to identify if the errors in the behvaiour of the program is due to their code, or the compiler.

Specifically, this is where we can come up with the following specification:

$$\forall \text{P, input, interpret}_c(P, \text{input}) = \text{execute}_{\text{arm}}(\text{compile(P, input)})$$

that is, a Compiler is correct if the result of executed the compiled, translated program with input in *arm* assembly is equivalent to the behaviour when executed the interpreted source program with input in `c`.

However, this is not trivial to prove, as while as we can test specific test cases, we cannot guarantee that the above specification holds true for all cases.

Note:-

The space of all cases is infinite.

Hence, proving the correctness of compilers must often be done formally. *Le gasp*

Chapter 2

Programming Languages

2.1 Syntax

Definition 2.1.1 – Syntax The Syntax of a Programming Language refers to the sequence of character(s) which count as a legal Program in said language.

Syntax is also split into 2 types:

1. Concrete Syntax: All the possible accepted values / forms that is legal for the language
   Implements the Operational Semantics of the Programming Language in question, i.e., it defines the behaviour or meaning of the program.
2. Abstract Syntax: Hides the irrelevant parts of the Concrete Syntax
   I.e., comments, paranthesis, ;

2.2 Semantics

Definition 2.2.1 – Semantics The Semantics of a Programming Language refers to the meaning / behaviour of a legal program.

2.2.1 Operational Semantics

Operational Semantics are specified using just 2 inference rules, with judgements of the form `exp` $\Downarrow$ `val`, which can simply be read as "A program `exp` evaluates to the value `val`".

Note:-

This is also referred to as Call-by-Value Semantics, where function arguments are evaluated before subsitution.

Therefore, in the following example,

$$\frac{\texttt{exp\_1} \Downarrow n_1 \quad \texttt{exp\_2} \Downarrow n_2}{\texttt{exp\_1 + exp\_2} \Downarrow (\texttt{n\_1 [+] n\_2})}$$

We can interpret it as follows: If `exp_1` evaluates to a value `n_1` $\wedge$ `exp_2` evaluates to a value `n_2` $\rightarrow$ `exp_1 + exp_2` evaluates to `n_1 + n_2`

This will be important for Type Checking and Inference :3

2.3 Functional Programming Languages

Functional Programming Languages have the following properties:

- Functions can be passed as arguments
- Functions can be returned as values
- Functions nest: Inner functions can refer to variables bound in the Outer function

Examples of Functional Languages are: `OCaml, Scala, Haskell,` and more.

Chapter 3

X86Lite

X86Lite is a simplified version of Intel's widely used X86 Architecture. It contains only a subset of the instruction set available in the full X86 ISA, but it is enough to create simple, general-purpose programs.

Some key differences are:

1. No floating point values; Only Signed Integer values
2. No variations in bit-size for datatypes; Keep everything to 64-bits
3. Only about 20 instructions

3.1 Machine State

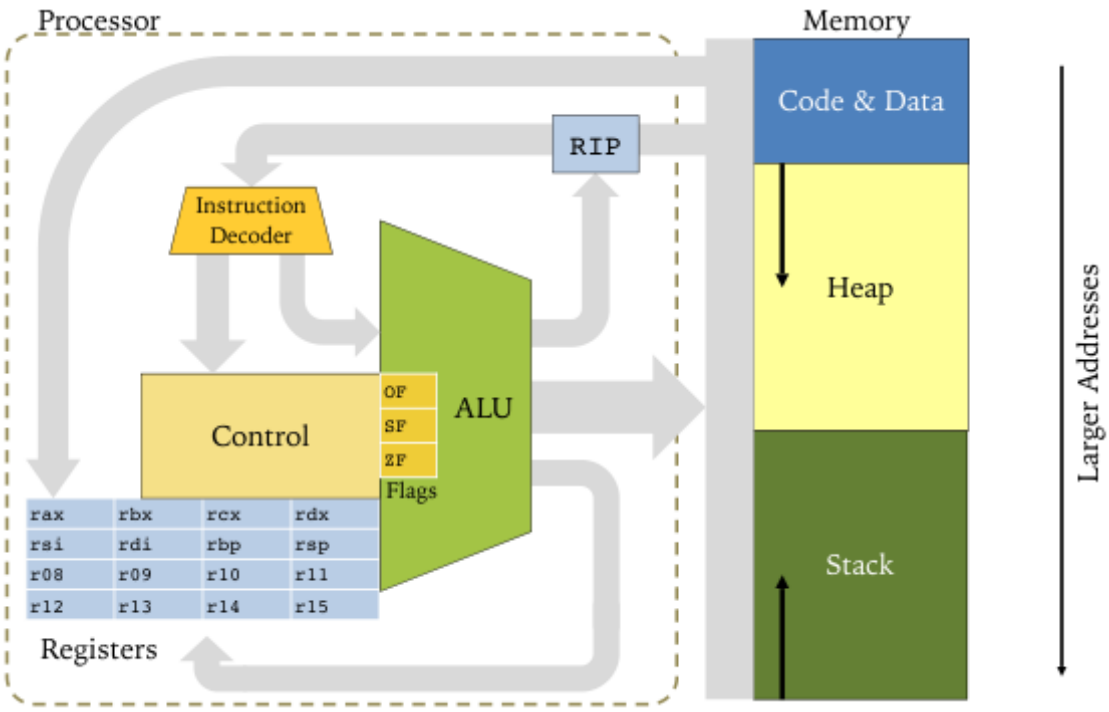

Figure 3.1: X86Lite Schematics

3.1.1 Registers

Registers are fast storage components that live in the CPU. Accessing data in Registers is very fast, however they are limited in size (64-bit) and number. As a result, we have to spill extra arguments / variables to the Stack in Memory.

However, since accessing the Memory is much slower, we also have to consider Efficient Register Allocation (this will be covered in the Optimization chapter).

The X86lite Machine State has $16 + 1 = 17$ 64-bit Registers:

1. `rax`: General Purpose Accumulator
2. `rbx`: Base register which points to data
3. `rcx`: Counter register for Strings and Loops
4. `rdx`: Data register for I/O
5. `rsi`: Pointer Register which points to the String Source register
6. `rdi`: Pointer Register which points to the String Destination register
7. `rbp`: Base Pointer which points to the (base of the) stack frame
8. `rsp`: Stack Pointer which points to the top of the stack
9. `r09 - r15`: General Purpose Registers
10. `rip`: Instruction Pointer which points to the current instruction

Note:-

When we say 'points to', we mean that the Register hold the address at which the value is stored.
I.e., `rip` points to the address the current instruction is stored at

3.1.2 Memory Model

The X86Lite Memory Model consists of 2^{64} bytes, where each data is a 64-bit (8-byte) quadword. Therefore:

- Legal Memory Addresses consist of 64-bit, quadword-aligned pointers
- All Memory Addresses we access must be evenly divisible by 8

3.2 Instruction Set

Note:-

The X86 Instruction Set described below is in AT&T Notation, which is the notation followed in CS4212.

An Instruction is composed of 3 different parts:

1. Instruction Operation: The Instruction we are trying to execute, i.e., `subq, notq`, etc.
2. Source Operand
3. Destination Operand

Operands are the values operated on by the Assembly Instruction, and come in 1 of a few types:

1. `Imm`: Immediate (Literal) Values
2. `Lbl`: Labels; representing machine addresses which have to be resolved from string labels to address values
3. `Reg`: Registers, i.e. `%rbx`
4. `Ind`: Indirect Machine Addresses; typically consists of a label / register address as well as an offset. The formula is as follows:

$$addr(ind) = Base + Disp.$$

Note:-

For Indirect Machine Addresses, there are 2 scenarios:
1. If used as a location, the result is the address: `addr(ind)`
2. If used as a value, the result is the value stored at the address, `Mem[addr(ind)]`

Control Flow

The Control Flow of the program is controlled by specific instructions: `J, Callq, Retq`. We perform these instructions to move to another block of code, i.e., when we reach an `if-else` branch, or when we call a function.

These instructions also alter the value of `Rip`. Specifically, the current `Rip` value is pushed onto the Stack and popped later to return to the current instruction, before `Rip` changes to point to the new instruction address.

Note:-

There is no guarantee here that whenever we pop the Stack to retrieve the old `Rip` value, we pop the correct value. This is something that we, as compiler developers, should ensure when we perform the code translation.

Flags and Condition Codes

X86Lite Instructions also set Flags as a side-effect of some Instructions. These flags are:

1. `OF`: Overflow Flag; Set when the result is too big/small to fit into a single 64-bit register
2. `SF`: Sign Flag: Set to the sign of the result (0 = `+ve`; 1 = `-ve`)
3. `ZF`: Zero Flag: Set when the result is 0

Condition Code	Description
`eq`	*Equals:* This condition holds when `ZF` is set. (Intuitively, `SRC1 = SRC2` when `SRC2 - SRC1 = 0`.)
`neq`	*Not equals:* This condition holds when `ZF` is not set.
`lt`	*(Signed) less than:* This condition holds when `SF` does not equal `OF`, or when (`SF = 1` and `OF = 0`) or (`SF = 0` and `OF = 1`). The first case holds when the result of `SRC2 - SRC1` is negative and there has been no overflow. The second case holds when the result of `SRC2 - SRC1` is positive and there has been an overflow.
`le`	*(Signed) less than or equal:* This condition holds when (`SF` is not equal to `OF`) or `ZF` is set. This is equivalent to (`lt` or `eq`).
`gt`	*(Signed) greater than:* This condition holds when (not `le`) holds.
`ge`	*(Signed) greater than or equal:* This condition holds when (not `lt`) holds. Or, equivalently, if `SF` equals `OF`.

Table 3.1: X86Lite Condition Codes and Descriptions

Conditional Operations such as `J<CC>` and `Setb <CC>` thus evaluate the Condition Code provided based on the current flags and perform the appropriate actions.

Instruction Type	Instruction Syntax	Operation	Comments/Flags
Arithmetic	`subq SRC, DEST`	`DEST ← DEST - SRC`	Updates flags: CF, SF, ZF, OF
	`incq DEST`	`DEST ← DEST + 1`	Updates flags: SF, ZF, OF
	`decq DEST`	`DEST ← DEST - 1`	Updates flags: SF, ZF, OF
	`negq DEST`	`DEST ← -DEST`	Updates flags: CF, SF, ZF, OF
	`addq SRC, DEST`	`DEST ← DEST + SRC`	Updates flags: CF, SF, ZF, OF
	`imulq SRC, DEST`	`DEST ← DEST * SRC`	Multiplies; ignores flags
Logic	`xorq SRC, DEST`	`DEST ← DEST XOR SRC`	Clears OF, CF; Updates SF, ZF
	`notq DEST`	`DEST ← NOT DEST`	Bitwise negation; no flags affected
	`andq SRC, DEST`	`DEST ← DEST AND SRC`	Clears OF, CF; Updates SF, ZF
	`orq SRC, DEST`	`DEST ← DEST OR SRC`	Clears OF, CF; Updates SF, ZF
Bit-Manipulation	`shlq AMT, DEST`	`DEST ← DEST << AMT`	Logical left shift; Updates CF, SF, ZF, OF
	`salq AMT, DEST`	`DEST ← DEST << AMT`	Alias for `shlq`; same behavior
	`shrq AMT, DEST`	`DEST ← DEST >> AMT`	Logical right shift; Clears MSBs; Updates CF, SF, ZF
	`sarq AMT, DEST`	`DEST ← DEST >> AMT`	Arithmetic right shift; Preserves sign; Updates flags
Data-Movement	`leaq Ind, DEST`	`DEST ← addr(Ind)`	Loads address; ignores flags
	`movq SRC, DEST`	`DEST ← SRC`	Simple copy; ignores flags
	`pushq SRC`	`rsp ← rsp - 8; Mem[rsp] ← SRC`	Pushes value onto stack; ignores flags
	`popq DEST`	`DEST ← Mem[rsp]; rsp ← rsp + 8`	Pops value from stack; ignores flags
Control-Flow and Condition	`cmpq SRC1, SRC2`	`Set flags as if SRC2 - SRC1`	Updates flags: SF, ZF, OF, CF
	`jmp SRC`	`rip ← SRC`	Unconditional jump
	`callq SRC`	`pushq rip; rip ← SRC`	Procedure call; pushes return address
	`retq`	`rip ← Mem[rsp]; rsp ← rsp + 8`	Procedure return
	`jCC SRC`	`rip ← SRC if CC holds`	Conditional jump based on flags
	`setb CC, DEST`	`DEST[0] ← 1 if CC holds, else 0`	Sets byte based on condition

Table 3.2: X86Lite ISA Instructions Summary

3.3 System V AMD64 ABI Calling Convention

Caller Protocol

1. Push arguments 1 to 6 into the following registers: `Rdi, Rsi, Rdx, Rcx, R08, R09`, with excess arguments being pushed to the Stack.

   Note:-

   We will have to save the original values in these registers by pushing them onto the Stack first, from right to left, before we save the arguments.

2. Save the return address, i.e., the current `Rip`, by pushing it onto the Stack.
3. Set `Rip` to the new instruction address, i.e., the address of the first instruction in the function.

Callee Protocol

1. Save the old `Rbp` by pushing it onto the Stack.
2. Set `Rbp` to be the base of the Stack Frame, i.e., `movq %rsp, %rbp`
3. Allocate the required space for the Stack Frame on the Stack, i.e., `subq $128, %rsp`, where 128 bytes is the size of the Stack Frame.
4. Save all the other registers: `rbp, rbx, r12 - r15`

When returning,

1. Move the result into `rax`
2. Deallocate the Stack Space
3. Restore the original Base Pointer from the Stack, i.e., `popq %rbp`
4. Restore the original `rip` from the Stack, i.e., `retq`

Chapter 4

LLVMLite

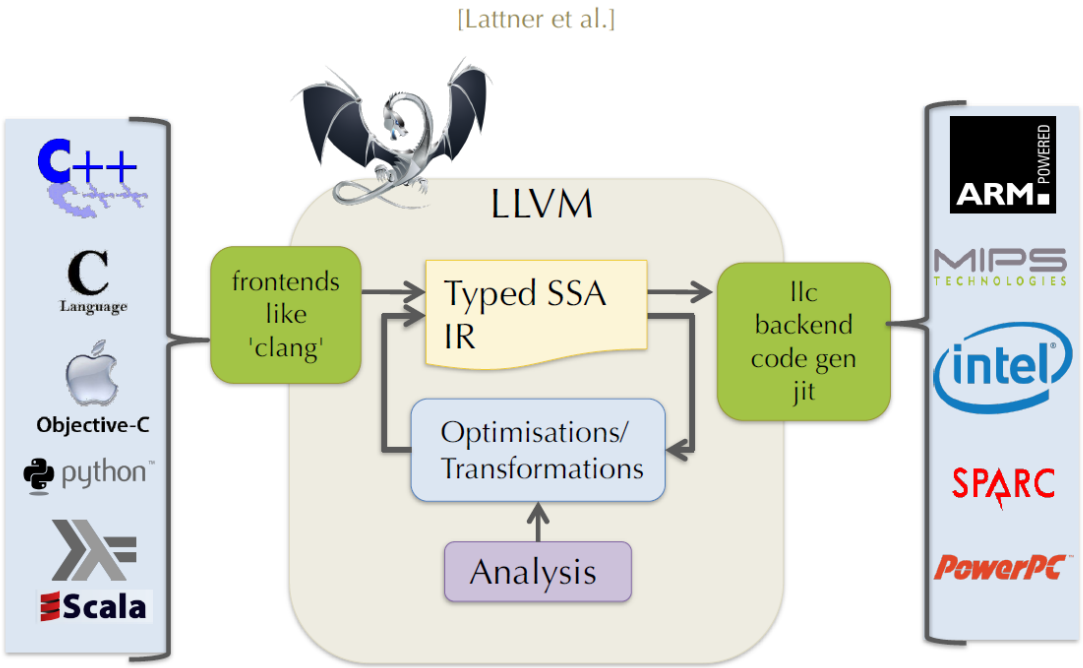

Figure 4.1: LLVM Infrastructure

LLVM is an Intermediate Representation that compiles down to various target architectures. It is used by a variety of different Higher-level Languages (as seen in the image above), and performs a variety of different optimizations when compiling down to the target architecture.

It has a `C`-like weakly typed system, supporting:

- A variety of different kinds of datatypes and structured data
- Top-level mutually recursive function definitions and function calls as primitives
- An infinite number of "Locals" to hold Intermediate results of computations
- An Abstract Memory Model that doesn't constrain the layout of data in Memory
- Dynamically allocated memory associated with a function invocation
- Statically and Dynamically allocated structured data
- Control Flow Graph representation of function bodies

4.1 Storage Models

LLVM also allows for additional Storage Components to make life easier:

- Local Variables (as `uid`)
- Global Declarations (as `gid`)
- Abstract Locations (References to Stack-Allocated Storage created by the `alloca` instruction)
- Heap-Allocated (Complex) Structures

4.1.1 Locals

Local Variables are defined by instructions of the form: `%uid = ...`, and **must** satisfy the Single Static Assignment (SSA) Invariant. They are intended to be abstract versions of normal Machine Registers.

Definition 4.1.1 – Single Static Assignment Invariant Each Variable can be assigned a value once and only once. That is, this enforces Immutability on Local Variables, and updates to an arbitrary Local Variable is done by creating a new Local Variable with the new value.

This has the side-effect of enforcing Immutability.

4.1.2 Complex Structured Data

For complex data structures, we make liberal use of pointers in order to obtain access to all data (since they usually cannot fit into a single 64-bit block).

We can calculate the appropriate offset relative to the base pointer of the data, in order to access the data at the index that we want. However, to perform this check while still providing memory safety, the compiler has to know the size of the array / struct in question to compute the last legal memory address accessible.

Padding and Alignment

It is here that we also come to the topic of ensuring proper Padding and Alignment when dealing with data.

First, notice that different datatypes require less bits, and thus take up less space. It is sufficient to just consider Integers, of `I64`, and Booleans, of `I1`.

Therefore, to facilitate ease of data access, it is good practice (and indeed is universal practice currently) to ensure the proper padding and alignment to specific n-bit boundaries. This is also where we start hearing 32-bit vs 64-bit architectures.

Typically, data is stored such that, regardless of size, they always take up multiples of n-bits. This is referred to as 'Alignment' to the n-bit boundaries. Then, 'Padding' refers to the act of leaving empty spaces where necessary, to ensure that the proper Alignment is maintained.

Array Bounds Checking

First, convince yourself that the total space allocated for an Array of n variables is given by: (n * element size). Then, further convince yourself that accessing the i-th variable of an Array is given by: (base of array) + (i * element size).

The process of performing Array Bounds Checking is then given by:

1. Loading the Size of the Array
2. Comparing the Index to the Bound (Size of the Array)
3. If the Index is greater than the Bound, fail
4. Else, perform the Load of the actual element

Specifically, assume `%rax` holds the Base Pointer to `arr` and `%rcx` holds the Array Index `i`. Then, to access `arr[i]`, we do the following X86Lite instructions:

```
    movq -8(%rax) %rdx
    cmpq %rdx %rcx
    j L _ok
    callq _err_oob

  _ok:
    movq (%rax, %rcx, 8) dest
```

See that this entire operation is clearly more expensive as we have much more operations than if we just directly loaded the value. However, this can be improved by Hardware and Compiler Optimizations.

Note:-

In CS4212, arrays are stored as { `size, data` }, such that we can easily retrieve the size of the array.

4.2 Instruction Set

The full list of instructions is as follows:

Concrete Syntax	Operand → Result Types
`%L = BOP i64 OP1, OP2`	`i64 x i64 → i64`
`%L = alloca S`	`- → S*`
`%L = load S* OP`	`S* → S`
`store S OP1, S* OP2`	`S x S* → void`
`%L = icmp CND S OP1, OP2`	`S x S → i1`
`%L = call S1 OP1(S2 OP2, ..., SN OPN)`	`S1(S2, ..., SN)* x S2 x ... x SN → S1`
`call void OP1(S2 OP2, ..., SN OPN)`	`void(S2, ..., SN)* x S2 x ... x SN → void`
`%L = getelementptr T1* OP1, i32 OP2, ..., i32 OPN`	`T1* x i64 x ... x i64 → GEPTY(T1, OP1, ..., OPN)*`
`%L = bitcast T1* OP to T2*`	`T1* → T2*`

The above table describes the restrictions on the types that may appear as parameters of well-formed instructions and the constraints on the operands and result of the instruction for type-checking. We assume that named types have been replaced by their definitions.

For example, in the `call` instruction, each type parameter `S1`, ..., `SN` must be a simple type. When we type check a program containing this instruction, we must make sure that the operand `OP1` has exactly the function pointer type `S1(S2, ..., SN)*`, and that the remaining operands `OP2`, ..., `OPN` have types `S2`, ..., `SN`.

As can be seen from the table, there are some new instructions that LLVM provides, which the general X86Lite ISA doesn't provide natively (as a single instruction at least).

These instructions are:

- `alloca`: Allocates Stack Space and returns a reference to it.
  The returned reference is stored as a Locals, and the amount of space allocated is determined by the type of the data. The Stack Space can then be accessed using `load` and `store` instructions.
  This provides an Abstract version of Stack Slots.
- `bitcast`: A (potentially) unsafe cast for which misuse can cause bugs such as Segmentation Faults, Memory Corruption, etc.
- `gep`: getelementpointer (See the below section)

getelementpointer (GEP)

The `getelementptr` instruction has additional well-formedness requirements. Operands after the first must all be constants unless they are used to index into an array.

Note:-

LLVM actually requires the operands used to index into structs to be 32-bit integers. Rather than introducing 32-bit integers into our language, we will use our 64-bit constants and operands and assume the arguments of `getelementptr` always fall in the range $[0, \text{Int32.max\_int}]$.

The result type of a `getelementptr` instruction is described using the **GEPTY** function, which is defined in pseudocode as follows:

```
GEPTY : T -> operand list -> T
GEPTY ty operand::path' = GEPTY' ty path'

GEPTY' : T -> operand list -> T
GEPTY' ty                                  [] = typ
GEPTY' { ty_1, ..., ty_n } (Const m)::path' = GEPTY' ty_m path' when m <= n
GEPTY' [_ x ty]             operand::path' = GEPTY' ty path'
```

Notice that `GEPTY` is a partial function. When `GEPTY` is not defined, the corresponding instruction is malformed. This happens when, for example:

- The list of index operands provided is empty.
- An operand used to index a struct is not a constant.
- The type is not an aggregate, and the list of indices is not empty.

Also, note that a GEP instruction that indexes beyond the size of an array is well-formed. The length information on array tags is only present to help the compiler lay out data in memory and is not verified statically.

4.2.1 Types

Almost everything, if not everything, in LLVM is explicitly annotated with types. This includes functions, global data definitions, and instructions.

These types are divided into the following categories:

1. Simple: Appears on the stack and as arguments to functions
2. Aggregate: Only appears in global and heap-allocated data
3. Void: Only appears in the return type of instructions and functions that do not return a value
4. Named Types: Of the form `%IDENT = type T`; They define abbrieviations for types in the scope of the entire compilation unit

Note:-

`void` is essentially the ML unit type, but it has the additional restriction that it cannot appear as the type of an operand, so it is actually illegal to give it a name in the LLVM concrete syntax.

Note that unlike full LLVM, LLVM lite does not allow locals to hold structured data.

In the following table, we use `T` to range over simple and aggregate (non-void, non-function) types, `F` to range over function types, and `S` to range over simple types.

Concrete Syntax	Kind	Description
`void`	void	Indicates the instruction does not return a usable value.
`i1, i64`	simple	1-bit (boolean) and 64-bit integer values.
`T*`	simple	Pointer that can be dereferenced if its target is compatible with `T`.
`i8*`	simple	Pointer to the first character in a null-terminated array of bytes. Note: `i8*` is a valid type, but just `i8` is not. LLVMlite programs do not operate over byte-sized integer values.
`F*`	simple	Function pointer.
`S(S1, ..., SN)`	function	A function from `S1, ..., SN` to `S`.
`void(S1, ..., SN)`	function	A function from `S1, ..., SN` to `void`.
`{ T1, ..., TN }`	aggregate	Tuple of values of types `T1, ..., TN`.
`[ N x T ]`	aggregate	Exactly `N` values of type `T`.
`%NAME`	`*`	Abbreviation defined by a top-level named type definition.

Table 4.1: LLVMlite Type System

4.2.2 Global Definitions

Global data is defined by `@IDENT = global T G`, where `G` ranges over global initializers (described in the following table), and `T` is the associated type. The global identifier `@IDENT`, when used in the program, has type `T*`.

For example, the following program fragment has valid annotations:

```
@foo = global i64 42
@bar = global i64* @foo
@baz = global i64** @bar
```

Concrete Syntax	**Type**	**Description**
`null`	`T*`	The null pointer constant.
`[0-9]+`	`i64`	64-bit integer literal.
`@IDENT`	`T*`	Global identifier. The type is always a pointer of the type associated with the global definition.
`c"[A-z]*\00"`	`[N x i8]`	String literal. The size of the array `N` should be the length of the string in bytes, including the null terminator `\00`.
`[ T G1, ..., T GN ]`	`[N x T]`	Array literal.
`{ T1 G1, ..., TN GN }`	`{T1,...,TN}`	Struct literal.
`bitcast (T1* G1 to T2*)`	`T2*`	Bitcast.

Table 4.2: Global Initializers in LLVMlite

Chapter 5

Intermediate Representation

5.1 Motivation

With direct Source-to-Target Code Translation, there are a few issues with such a naive compiler:

- Little Optimizations can be performed to the resulting code
- Re-targeting the compiler to work for a different architecture, which uses a different ISA, is non-trivial

The introduction of Intermediate Representations allow us to:

1. Generate and Optimize Machine Independent Code
2. Abstract Machine Code, hiding details of the Target Architecture and its ISA

Intermediate Representations provide the Ideal stage for various checks and optimizations to occur. In fact, chaining Intermediate Representations can also allow for more and better optimizations than if we just had a single Intermediate Representation.

For example, `Rust` added a new Intermediate Representation (creatively named 'MIR' for Middle Intermediate Representation) which had several benefits, including but not limited to:

- Faster compilation and execution time
- More precise type-checking

Note:-

Adding a new Intermediate Representation stage can actually improve overall compilation and execution time, despite requiring another level of code translation!
See: MIR Rust Blog Post

One famous, and commonly used, Intermediate Representation is the `LLVM IR` which we have seen in the previous chapter!

5.2 Different Stages of Intermediate Representations

Typically, there are 3 broad stages of Intermediate Representations, although each broad stage can have multiple Intermediate Representations in itself..

High-Level Intermediate Representations In general, Higher-level Intermediate Representations are the closest to the Source Language in terms of both Semantics and Syntax. They typically:

- Preserve constructs, although they narrow most, if not all of them
- Perform higher-level Optimizations and checks, i.e., Constant Folding, Function Inlining, etc.

Mid-Level Intermediate Representations

Mid-Level Intermediate Representations are the bridge between the High-Level Intermediate Representation and the Low-Level Intermediate Representation. They typically have a Syntax closer to that of the Abstract Syntax, while having some elements of the Target Language.

For example, they

- may have unstructured jumps, abstract registers or memory locations
- may rename variables, i.e., to SSA Format

Low-Level Intermediate Representations

Finally, Low-Level Intermediate Representations are typically the closest to the Target Language, both in terms of Syntax and Semantics.

They usually:

- Introduce extra psuedo-instructions and machine dependent assembly code
- Allow for lower-level optimizations depending on the Target Architecture

5.3 Creating Intermediate Representations

When creating Intermediate Representations, we consider several main aspects:

1. **Modularity**
   The Intermediate Representation needs to be an easy translation target from the Source Language to the Target Language
2. **Narrow Interface**
   The Intermediate Representation should have a narrower interface than the Source Language, that is, it should have a smaller number of constructs and be less complex in general

However, it is important to note that there is no 1-size-fits-all approach for Intermediate Representations. By its very nature, Intermediate Representations are typically specific to the overall Source and Target Language of the compiler, which makes it difficult to generalise for all cases due to the differing Syntax and Semantics of both languages in question.

Chapter 6

Lexical Analysis

Lexical Analysis is the process of converting a stream of characters into tokens, which represent indivisible "chunks" of text, typically delimited by whitespace, but not always (language dependent).

The topic of Lexing can be summed up into a single question: **"How do we match tokens?"**

6.1 Motivation

Lexing entire programs into tokens by hand is difficult and tedious. It requires us to:

- Precisely define tokens
- Match multiple tokens simulataneously

It is also difficult to compose / interleave tokenizer code, and it is hard to maintain and extend the Lexer.

6.2 Lexing

A principled solution to Lexing revolves around using Regular Expressions (See the below subsection).

Generally, for each token, we also attach an 'Action' to it. Then, when we match a token, we run the 'Action' attached to it, which denotes how to convert that string of characters matched into the Token we want.

However, we can also encounter scenarios whereby we can find multiple matches for a single string. For example, `ifx` would possibly match to a String `ifx`, or the Keyword `if`, followed by the Identifier `x`. Therefore, to break such ties, we usually enforce the **'Longest Match'**, as well as provide **Priorities** to certain tokens, i.e. `Keyword` $>$ `Identifiers`.

6.2.1 Lexer Generators

Lexer Generators are programs that implement Lexical Analysis to perform lexing.

Generally, they work as follows:

1. Take each Regular Expression R_i and its associated action A_i.
2. Compute the Non-Deterministic Finite Automata (NFA) formed by $(R_1 \mid R_2 \mid \cdots \mid R_n)$.
3. Compute the Deterministic Finite Automata (DFA) for the NFA computed above in Step 2.
    - There may be multiple accept states, which may thus result in different action(s) performed.
4. Compute the minimal equivalent of the DFA.
    - There is a standard algorithm for this by Myhill & Nerode.
5. Produce the Transition Table for this DFA.
6. Check Longest Match and/or Token Priorities to get the token for the current 'chunk' of text.

To support better error reporting, Lexer Generators typically store metadata such as:

- Current line number,
- Character position, and
- Other relevant context.

There are a variety of (open-source) Lexer Generators available.

6.2.2 Regular Expressions

Regular Expressions precisely describe sets of strings that must match with what is described by the Regular Expression in question.

A Regular Expression has one of the following basic forms:

Regular Expression, R	**Description**
ϵ	Empty String
'a'	Character
$R_1 \mid R_2$	Alternatives; Either R_1 or R_2
$R_1 R_2$	Concatenation; R_1 followed by R_2
R^*	Kleene Star; ≥ 0 repetitions of R

The following extensions to the above basic form are also typically built into Regular Expression Parsers:

Regular Expression, R	**Description**
"foo"	Equivalent to 'f', 'o', 'o'
R^+	Equivalent to $RR^*; \geq 1$ repetitions of R
$R?$	Equivalent to $(\epsilon \mid R); \geq 0$ occurrences of R
['a' - 'z']	One of any character from 'a' to 'z'
[^ '0' - '9']	Any character except any from 0 to 9
R as x	Name the string matched by R as x

6.2.3 Automata

The underlying logic for how Regular Expressions work, are to use Automatas (DFAs and NFAs), whereby each character is an input. We say that the String can be matched to an arbitrary token $\iff$ the String can reach the Accepting State of the Token's DFA / NFA representation.

Deterministic Finite Automata (DFA)

A DFA is deterministic, meaning there is only one possible next state for each state and input combination.

The advantages of DFAs are as follows:

- Easier to reason about as the automaton's action for each input is fully determined.
- Well-suited for table-based implementations.

Non-Deterministic Finite Automata (NFA)

In contrast, an NFA may have multiple possible next states for a single input. Thus, at each step, the automaton can have ≥ 1 next states. An NFA accepts if there exists a path to an accepting state.

NFAs are nice in that they allow for more complex automata, since we have more flexibility in creating the automata and have less restrictions on what counts as a a valid NFA. However, it is less obvious how to implement NFAs efficiently.

6.3 Alternatives

Alternatives to the usual Lexing process is to use Brzozowski's Derivatives.

Chapter 7

Syntactic Analysis

Syntactic Analysis, also known as Parsing, takes in a stream of tokens (generally from the Lexing stage) and outputs an (Abstract) Syntax Tree, provided the source code in question respects the language's grammar.

7.1 Motivation

While lexing involves constructing Deterministic Finite Automata (DFA) or Non-Deterministic Finite Automata (NFA), they have a significant limitation: **they cannot "count"**.

What this means is that since DFAs and NFAs are linear in nature, they are unable to traverse up and down recursive structures, a property which is required to parse nested expressions.

For example, consider the problem of determining whether a String has balanced parentheses. DFAs and NFAs are insufficient for such a task as we cannot count the number of left brackets we have encountered thus far, and therefore, cannot determine if the String is balanced.

7.2 Parsing

The general strategy for parsing is as follows:

1. Parse the token stream to traverse the concrete syntax of the language (Result $\rightarrow$ Parse Tree)
2. While traversing, build a tree representing the abstract syntax (Result $\rightarrow$ Abstract Syntax Tree)

However, before we discuss the exact details of how to construct the Parse and Abstract Syntax Trees, we should first familiarise ourselves with Context-Free Grammers (CFGs).

7.2.1 Context-Free Grammars (CFGs)

CFGs provide a concise way to specify programming languages. A CFG accepts a string if and only if there is a derivation from the start symbol to that string.

A CFG consists of:

- A set of terminals (symbols that cannot be expanded further).
- A set of non-terminals (symbols that can be expanded further).
- A designated non-terminal called the *Start Symbol*.
- A set of productions of the form:

$$LHS \mapsto RHS$$

  where:
  - LHS is a non-terminal.
  - RHS is a string of terminals and non-terminals.

Example: Balanced Paranthesis Language

The following CFG defines the balanced parentheses language:

$$S \mapsto (S)S \quad | \quad \epsilon$$

- Terminals: ϵ, '(', ')'
- Non-Terminal: S
- Productions: 2

Using |, multiple productions for the same LHS can be written on a single line.

Considerations in CFGs

When defining CFGs, consider the following:

1. **No loops:** Each production must have at least one terminal (i.e., it is *productive*).
2. **No vacuous CFGs:** Each non-terminal must eventually rewrite to terminals only.
3. Consider the **associativity** and **precedence** of operators.

Associativity and Precedence A grammar can be:

- **Left-Associative (Left-Recursive):**
$$S \mapsto S + E \quad | \quad E$$
- **Right-Associative (Right-Recursive):**
$$S \mapsto E + S \quad | \quad E$$

The difference lies in the parse tree's nesting, which grows to the left or right. Ambiguity in grammar often affects associativity and precedence, and can usually be resolved by restructuring productions to limit recursion to one side.

7.2.2 Parse Trees

Parse Trees typically represent the concrete syntax of the input code. They are derived based on the Context-Free Grammar (CFG) of the Language in question.

- **Leaves:** Terminals of the grammar.
- **Internal Nodes:** Non-terminals of the grammar.

Note:-

Performing an *In-Order Traversal* of the Parse Tree yields the input sequence of tokens.

The Parse Tree can then be converted into an Abstract Syntax Tree (AST), which represents the abstract syntax by removing unnecessary information.

7.2.3 Derivation Orders

When applying grammar productions, there are standard derivation orders:

1. **Leftmost derivation:** Apply productions to the leftmost non-terminal first.
2. **Rightmost derivation:** Apply productions to the rightmost non-terminal first.

Typically, regardless of the strategy, the resulting Parse Tree is the same.

7.2.4 LL(1) Grammer

With normal Grammer, we may not be able to parse them 'top-down' with only a single Lookahead symbol. To convince yourself, consider the following example:

$$S \mapsto E + S \mid E$$
$$E \mapsto \text{number} \mid (S)$$

Then, with only a single lookahead symbol, we cannot accurately determine what the proper production to apply is in the following scenario: $(((1))) + 42$, whether it follows the Production of $S \mapsto E + S$ or $S \mapsto E$.

Furthermore, just increasing the number of lookahead symbols we have doesn't help as we then have another issue of lowered performance **and** this doesn't even resolve our original issue since adding another () into the String would result in the same ambiguity.

To convert an arbitrary grammer into LL(1) Grammer, we have to

1. "Left-factor" the grammer
2. Eliminate Left Recursion

Therefore, the same problematic grammer from before, can be converted into the following:

$$\begin{array}{l} S \mapsto E + S \mid E \\ E \mapsto \text{number} \mid (S) \end{array} \quad \rightarrow \quad \begin{array}{l} S \mapsto SE' \\ S' \mapsto \epsilon \\ S' \mapsto +S \\ E \mapsto \text{number} \mid (S) \end{array}$$

7.2.5 LR Grammer

By their nature, LR Grammers are more expressive than LL Grammers, as it:

- Can handle left, and right, recursive grammers (and thus, virtually all Programming Languages)
- Is easier to express Programming Language syntax (no left factoring is required)

7.3 Parsing Algorithms

7.3.1 LL(1) Parsing

LL(1) Parsing stands for:

- **L**eft-to-right Scanning
- **L**eft-most Derivation
- **1** lookahead symbol

This means that in LL(1) Parsing, we scan tokens from left-to-right, only looking ahead to the next token, and apply productions on the leftmost non-terminal token first.

Note:-

In LL(1) Parsing, we only use LL(1) Grammers.

LL(1) Parsing is a Top-down, Predictive Parsing Algorithm, great for simple hand-written implementation with fine-tuned error control. It is driven by a predictive parsing table of non-terminal $*$ input-token $\rightarrow$ production.

However, LL(1) Parsing has the following problems:

- Grammer must be LL(1), and cannot be left recursive
- We can extend to LL(k), this just makes the Parse Table constructed bigger
- There are some CFGs that cannot be transformed to LL(k)

These are problems that we will resolve in LR(1) Parsing later..

Algorithm

The Algorithm to construct the Parse Table is as follows:

1. Consider a given production: $A \mapsto \gamma$
2. Construct the set of all input tokens that may appear first in strings that can be derived from γ, and thus, add the production $\mapsto \gamma$ to the (A, token)-th entry for each such token
3. If γ can derive ϵ (the empty string), then we construct the set of all input tokens that may follow the nonterminal A in the grammer. We thus add the production $\mapsto \epsilon$ to the (A, token)-th entry for each such token

Note:-

If we can have multiple productions for a given entry, the grammer is not LL(1)

Example to construct Parse Table

Given the following grammer:

$$
\begin{aligned}
T &\mapsto S\$ \\
S &\mapsto ES' \\
S' &\mapsto \epsilon \\
S' &\mapsto +S \\
E &\mapsto \text{number} \mid (S)
\end{aligned}
$$

Note:-

As general convention, T is the Start Symbol, and \$ is the End-of-file token

Therefore, we can construct the `First` and `Follow` sets as follows:

$$
\begin{aligned}
&\text{First}(T) = \text{First}(S) \\
&\text{First}(S) = \text{First}(E) \qquad\qquad \text{Follow}(S') = \text{Follow}(S) \\
&\text{First}(S') = \{+\} \qquad\qquad \text{Follow}(S) = \{\$, ')'\} \cup \text{Follow}(S') \\
&\text{First}(E) = \{\text{number}, '('\}
\end{aligned}
$$

The parse table based on the above 'First' and 'Follow' sets is thus as follows:

	number	**(**	**)**	**+**	**$**
T	$\mapsto S\$$	$\mapsto S\$$			
S	$\mapsto ES'$	$\mapsto ES'$			
S'			$\mapsto \epsilon$	$\mapsto +S$	$\mapsto \epsilon$
E	$\mapsto$ number	$\mapsto (S)$			

Converting Parse Table to Code

Once we have the Parse Table, we can then convert it to code using the following algorithm:

1. Define n mutually recursive functions, 1 for each non-terminal A (Let this be `parse_A`)
   - Assuming the stream of tokens is globally available, the type of `parse_A` is `unit -> ast` if `A` is not an auxillary non-terminal
   - Parse functions for the auxillary non-terminals take extra ast's as inputs, 1 for each non-terminal in the "factored" prefix
2. Each function "peeks" at the lookahead token and then follows the production rule in the corresponding entry as follows:
   (a) Consume terminal tokens from the token stream
   (b) Call `parse_A` to create the sub-tree for each non-terminal `A`
   (c) If the rule ends in an auxillary non-terminal, call it with the appropriate ast's
   (d) Otherwise, this function builds the ast tree itself and returns it

7.3.2 LR Parsing

LL(1) Parsing stands for:

- **L**eft-to-right Scanning
- **R**ight-most Derivation
- **1** lookahead symbol

This means that in LR(1) Parsing, we scan tokens from left-to-right, only looking ahead to the next token, and apply productions on the rightmost non-terminal token first.

In contrast to LL(1) Parsing, LR(1) Parsing works bottom-up instead of top-down, using "Shift-Reduce" Parsing instead to construct the right-most derivation of a program in the grammer. It also has better error detection and recovery, and thus, is used by many Parser Generators.

Shift/Reduce Parsing

Shift/Reduce Parsing keeps track of a Stack of tokens (for which the production has been undetermined), and the remaining token stream. Then, for each token in the token stream, there are two possible actions:

1. **Shift**: Shift the current token to the top of the Stack
2. **Reduce**: Reduce tokens γ in the Stack into a production by replacing them, and the current non-terminal X s.t. $X \mapsto \gamma$ is a production

Then, for each token in the token stream, if we can reduce this token, along with the tokens currently in the Stack, into a production, we reduce. Otherwise, we shift this token onto the Stack and repeat for the next token.

However, notice that this then creates some ambiguity. We can have two types of conflicts:

1. **Shift/Reduce Conflict**: Should the Parser shift the current token, or reduce it into a new production?
2. **Reduce/Reduce Conflict**: There are two valid productions that can be applied, which should the Parser apply?

Typically, having either of the above two conflicts indicates errors with the Grammer (i.e., Ambiguities in Associativity / Precedence), although having a greater lookahead symbol would resolve this issue in certain scenarios as well.

Chapter 8

(Untyped) Lambda Calculus

Before we talk about Semantic Analysis, and specifically, Type Checking in the next Chapter, we first discuss Lambda Calculus, and specifically Untyped Lambda Calculus.

8.1 Lambda Calculus

Lambda Calculus is a minimal programming language which only has variables, functions and function application (for Untyped Lambda Calculus). Despite this minimal syntax, Untyped Lambda Calculus is still Turing Complete, and is used as the Foundation for a lot of research in Programming Languages.

Definition 8.1.1 – Formal Definition of λ-terms The definitions are as follows:

1. **Variable**: $x \in V \rightarrow x \in \Lambda$
2. **Application**: $M, N \in V \rightarrow (MN) \in \Lambda$
3. **Abstraction**: $x \in V \land M \in V \rightarrow (\lambda x \cdot M) \in \Lambda$

where V is the Finite Set of Variables, and Λ is the Set of Lambda Terms

When we talk about each λ-term, we can also categorise the variables used into 2 types:

1. **Bound**: Variables defined inside the function
2. **Free**: Variables defined outside the function and thus, is not local

Specifically, understanding whether a variable is Bound or Free, can help indicate what is the Scope of the variable.

Definition 8.1.2 – Formal Definition of the Set of Free Variables for a λ-term We can also define the same for the Set of Free Variables (FV) for an arbitrary λ-term:

1. **Variable**: $FV(x) = \{x\}$
2. **Application**: $FV(MN) = FV(M) \cup FV(N)$
3. **Abstraction**: $FV(\lambda x \cdot M) = FV(M) \setminus \{x\}$

- A λ-term with no Free Variables is considered "closed"
- A λ-term with at least 1 Free Variable is considered "open"

With Lambda Calculus, we can also perform Subsitution, which helps us to interpret and simplify complex Function Applications. Specifically, we can subsitute a (closed) value v for some variable x in an expression e by: Replacing all **free** occurrences of x in e by v.

Definition 8.1.3 – Formal Definition for Subsitution in λ-terms The definitions are as follows:

1. **Variable**:
   - $x\ \{N/x\} = x$
   - $y\ \{N/x\} = y \iff x \neq y$
2. **Application**: $(P\ Q)\ \{N/x\} = (P\ \{N/x\})(Q\ \{N/x\})$
3. **Abstraction**: $(\lambda y \cdot P)\ \{N/x\} = (\lambda y \cdot (P\ \{N/x\}) \iff y \notin FV(N)$

where P and Q are two arbitrary Expressions

(Untyped) Lambda Calculus Examples

1. Translations between Code / Math and Lambda Calculus

Math / Code	**Lambda Calculus**
$f(x) = x^2 + 1$	$\lambda x \cdot x^2 + 1$
$x \mapsto x^2 + 1$	
	$(\lambda x \cdot x^2 + 1)5$
$f(5) = x^2 + 1$	$(\lambda x \cdot x^2 + 1)\{5/x\}$
$f(x) = x$	$\lambda x \cdot x$
$f(x, y) = x$	$\lambda x \cdot (\lambda y \cdot x)$
$f(m, g, z) = g(m\ g)x$	$\lambda m \cdot (\lambda g \cdot (\lambda x \cdot (g\ ((m\ g)\ x))))$

2. Determining Free Variables

Lambda Calculus	**Determining FV**
$\lambda x \cdot xy$	$FV(\lambda x \cdot xy) = FV(xy) n \{x\}$
	$= \{x, y\} n \{x\}$
	$= \{y\}$
$x(\lambda x \cdot xy)$	$FV(x_1(\lambda x_2 \cdot x_2 y)) = FV(x_1) \cup FV(\lambda x \cdot xy)$
	$= \{x_1\} \cup \{y\}$
	$= \{x_1, y\}$

3. Subsituting Variables

   1. $(\lambda x \cdot (\lambda y \cdot x + y))\ \ 2\ \ 3 \equiv [\lambda y \cdot x + y]\ \{2/x\}\ \ 3$
      $\equiv [\lambda y \cdot 2 + y]\ \ 3$
      $\equiv 2 + 3$

   2. $[\lambda x \cdot (\lambda y \cdot x + y)]\ \{1/x\} \equiv \lambda y \cdot 1 + y$

   3. $(\lambda y_1 \cdot y_1 x)\ \{(\lambda z \cdot zy)/x\} \equiv \lambda y_1 \cdot y_1\ (\lambda z \cdot zy)$

In the following subsections, we will introduce some differing properties of programs, as well as how we can interpret why they occur and what impact they have on the semantics of programs by using Untyped Lambda Calculus.

8.1.1 Variable Capture

Notice that with Subsitution, there is the possibility where if we naively try to "subsitute" an **open** term, a Bound Variable may capture Free Variable(s). What this refers to, is that an originally Free Variable may, once a subsitution occurs, become Bound.

As an example, consider the following:

Variable Capture Occurs	**Variable Capture does not Occur**
$\lambda x \cdot [(\lambda y \cdot \lambda x \cdot (x\ y))\ (\lambda z \cdot x)]$	$\lambda x_1 \cdot [(\lambda y \cdot \lambda x_2 \cdot (x_2\ y))\ (\lambda z \cdot x)]$
$\equiv \lambda x \cdot [(\lambda x \cdot xy)\ \{(\lambda z \cdot x)/y\}]$	$\equiv \lambda x_1 \cdot [(\lambda x_2 \cdot x_2 y)\ \{(\lambda z \cdot x)/y\}]$
$\equiv \lambda x \cdot [\lambda x \cdot x\ (\lambda z \cdot x)]$	$\equiv \lambda x_1 \cdot [\lambda x_2 \cdot x_2\ (\lambda z \cdot x_2)]$

In the course of Programming, we almost always reuse variable names. An easy example is in loops, where we always use `i` as the index for the loop.

However, with the reuse of variable names, we can encounter the above situation where the same variable names are used in both the outer and inner functions, and thus scopes, which results in confusion and deviations from the expected behaviour.

This problem occurs more frequently when Dynamic Binding and Dynamic Scoping is used, although this can still occur when Static Binding and Static Scoping is used.

If this is part of the Programming Langauge's semantics, it is usually treated as a "feature" which may be helpful in certain scenarios. Specifically, this provides greater flexibility to the Programmer, but it does require more careful coding in order to ensure expected behaviours.

An easy solution is referred to as **"Hygiene Subsitution"**, where we simply rename variables to avoid ambiguity. This can be seen in the above example where Variable Capture does not occur.

8.1.2 Alpha Equivalence

From the above section on Variable Capture, and the solution of Hygiene Subsitution, we can easily see that the names of Bound Variables do not matter to the semantics of the program. In contrast, the names of Free Variables do matter.

Easily, see that in the following example: $\lambda x \cdot yx$

- If we change it to $\lambda z \cdot yz$, the semantics remain the same
- However, if we change it to $\lambda x \cdot zx$, the semantics now differ

Intuitively, we can see that y and z in the second of the above example can refer to different (evaluated) values, which would change the semantics of the program.

Definition 8.1.4 – Alpha Equivalent Two terms that differ only by consistent renaming of Bound Variables

Note:-

Students who cheat by "renaming" variables are trying to exploit Alpha Equivalence

8.1.3 Variable Shadowing

Variable Shadowing occurs when a variable declared in an inner function, `v_1`, has the same name as another variable declared in an outer function, `v_2`.

Then, `v_1` shadows `v_2` as the value obtained when the variable is lookup-ed in the current environment in the inner function will be that of `v_1`.

Chapter 9

Semantic Analysis

The Semantic Analysis Phase does the following:

- Resolve symbol occurrences to declarations / binders; that is, check that all variables are properly bound in the scope they occur
- Type-check AST

It mainly manipulates a *Symbol Table*, which contains a mapping from symbols to information about those symbols (i.e., its type, location in the source text, etc.). This Symbol Table is then used to help translation into Intermediate Representations.

Note:-

Semantic Analysis may not be a separate phase, e.g. it may be incorporated into IR translation

Additionally, Semantic Analysis may also decorate the AST, i.e., attach type information to expressions, or replace symbols with references to their symbol table entry.

9.1 Type-Checking with Simply Typed Lambda Calculus

Definition 9.1.1 – Type There are two main schools of thought when we consider Types, and what exactly they are:

- **Intrinsic View** (Church-style): A type is syntactically part of a program
  - A program that cannot be typed is not a program at all
  - Types do not have an inherent meaning, they just help define the syntax of a program
- **Extrinsic View** (Curry-style): A type is a property of a program
  - For any program, and every type, either the program has that type or not
  - A program can have multiple types, or it may have no types

We like to perform **Type-Checking** in order to catch certain errors at compile-time, which would eliminate an entire class of mistakes (type-errors) that would otherwise lead to run-time errors. However, to perform Type Checking, we also need to perform Type Inference to derive Type Information from the source code.

Additionally, obtaining Type Information can also assist us in Code Generation if there are no errors. For example, executing `1 + 2` is different semantically as compared to executing `"hello" + "world"`.

Definition 9.1.2 – Type Systems A Type System consists of a System of Judgements and Inference Rules where:

- **Judgement** (Extrinsic view): A claim, which may or may not be valid
- **Inference Rule**: Rules used to derive valid judgements from other valid judgements

For example, the following is a simple Type System for the addition of integers, which can simply be interpreted as: "If e_1 and e_2 have type `int`, then so does $e_1 + e_2$".

$$\frac{\vdash e_1 : int \qquad \vdash e_2 : int}{\vdash e_1 + e_2 : int}$$

As we can see from the example above, an Inference Rule consists of a list of *premises*, $J_1, \ldots, J_n$, one conclusion, J, and optionally a Side-Condition:

$$\frac{J_1 \qquad \ldots \qquad J_n}{J}\,(\texttt{SIDE-CONDITION})$$

Then, to interpret each Inference Rule:

- **Reading Top-Down**: If J_1, ..., J_n are valid, and the side-condition holds, then J is also valid
- **Reading Bottom-Up**: To prove J is valid, it is sufficient to prove J_1, ..., J_n are valid (without proving Side-Condition)

Next, we have to look at Type Judgements. Type Judgements take the form of `G ⊢ e : t`, which is interpreted as: "Under the Type Environment `G`, the expression `e` has type `t`".

It is helpful to look at the Type Environment `G` as simply a lookup table, which maps Variables to Types. Specifically, `G` is just a set of bindings of the form: `x_1 : t_1, x_2: t_2, \dots, x_n : t_n`. For example, `x : int, b : bool`.

However, we can also have slightly more complicated mappings such as: `x : int, b : bool ⊢ if (b) 3 else x : int`. This says that under the Type Environment `G`, where `x : int, b: bool`, the expression `e = if (b) 3 else x` has type `t`. Thus, we can also infer the following:

- `b` must be a `bool`, i.e., `x : int, b: bool ⊢ b: bool`
- `3` must be a `int`, i.e., `x : int, b: bool ⊢ 3: int`
- `x` must be a `int`, i.e., `x : int, b: bool ⊢ x: int`

Theorem 9.1.1 – Type Safety (General Languages) If $\vdash P : t$ is a Well-Typed program, then either:

1. The program terminates in a well-defined way, OR
2. The program continues computing forever

Type Safety thus rules out undefined behaviour such as:

- Abusing "unsafe" casts
- Treating non-code values as code
- Breaking the Type Abstractions of the language

Note that by the definition above, a Well-defined termination could include:

- Halting with a return value
- Raising an exception

Typically when performing Type Checking, we build the Type System Bottom-Up, and say that the Program is Type-Safe $\iff$ the Derivation Tree of the Program exists, and can be formed.

Example

Given the following Type System, with 5 Inference Rules:

$$\frac{}{G \vdash i : int}\,(\texttt{int}) \qquad \frac{x : T \in G}{G \vdash x : T}\,(\texttt{var}) \qquad \frac{G \vdash e_i : int \qquad G \vdash e_2 : int}{G \vdash e_1 + e_2 : int}\,(\texttt{add})$$

$$\frac{G, x : T \vdash e : S}{G \vdash \text{fun}(x : T) \to e : T \to S}\,(\texttt{fun}) \qquad \frac{G \vdash e_1 : T \to S \qquad G \vdash e_2 : T}{G \vdash e_1 e_2 : S}\,(\texttt{app})$$

The Derivation Tree for $\vdash (\text{fun } (x : int) \to x + 3)\ 5 : int$ can thus be constructed as such:

$$\dfrac{\dfrac{\dfrac{x : int \in x : int}{x : int \vdash x : int}\texttt{(var)} \qquad \dfrac{}{x : int \vdash 3 : int}\texttt{(int)}}{x : int \vdash x + 3 : int}\texttt{(add)}}{\dfrac{\vdash (\text{fun}(x : int) \to x + 3) : int \to int}{\vdash (\text{fun}(x : int) \to x + 3)5 : int}\texttt{(app)}}\texttt{(fun)} \qquad \dfrac{}{\vdash 5 : int}\texttt{(int)}$$

However, the same Type System cannot construct a **Valid** Derivation Tree for the following program:

$\vdash$ (fun $(x : int) \to x\ 3)\ 5 : T$

$$\dfrac{\dfrac{\dfrac{x : int \to T \notin x : int}{x : int \vdash x : int \to T}\texttt{(var)} \qquad \dfrac{}{x : int \vdash 3 : int}\texttt{(int)}}{x : int \vdash x\ 3 : T}\texttt{(app)}}{\dfrac{\vdash (\text{fun}(x : int) \to x\ 3) : int \to T}{\vdash (\text{fun}(x : int) \to x\ 3)5 : T}\texttt{(app)}}\texttt{(fun)} \qquad \dfrac{}{\vdash 5 : int}\texttt{(int)}$$

Notice that at the top-most level, we get that $x : int \to T \notin x : int$ and therefore, the program is ill-typed. In fact, the above program will raise an exception along the lines of "Expected x to be of $int \to T$, but got int".

9.1.1 Well-Formed Types

In languages with type definitions, we also need additional rules to define well-formed types. Judgements take the form $H \vdash t$, where

- H is a set of type names
- t is a type

We can interpret $H \vdash t$ as: "Assuming H lists well-formed types, t is a well-formed type".

Therefore,trfrac combining this with the Type Judgements, we can intepret Judgements of the form $H; G; rt \vdash s$ as "With Type Definitions H, assuming Type Environment G, s is a valid statement within the context of a function that returns a value of type rt".

9.1.2 Subtyping Relations

Theorem 9.1.2 – Soundness of Subtyping Relations We say that a Subtyping relation $T_1 <: T_2$ is sound if it approximates the underlying semantic subset relation

For example, If $T_1 <: T_2$ implies $[T_1] \subseteq [T_2]$, then $T_1 <: T_2$ is sound.

Especially with Object-Oriented Programming (OOP), although this is applicable in non-OOP programs as well, Subtyping relations appear frequently throughout programs. This does not even have to be the explicit subtyping relations defined by programmers, but can even be something like: Positive Integers $\subseteq$ Integers.

Given any two types T_1 and T_2, we can calculate their Least Upper Bound (LUB) according to the subtyping hierarchy. For example, `LUB(True, False) = Bool` and `LUB(Int, Bool) = Any`.

We can also use this to define Inference Rules for `If-Else` Statements like so:

$$\dfrac{G \vdash e_1 : bool \qquad E \vdash e_2 : T_1 \qquad G \vdash e_3 : T_2}{G \vdash \text{if } (e_1)\ e_2\ \text{ else } e_3 : \texttt{LUB}(T_1, T_2)}\texttt{If-Bool}$$

Here, notice that in functions, we have to convert a supertype S_1 to T_1, and T_2 to S_2. Thus, the argument type is contravariantand the output type is covariant.

Additionally, we can also perform Subtyping for Function Types as follows:

$$\dfrac{S_1 <; T_1 \qquad T_2 : S_2}{(T_1 \to T_2) <: (S_1 \to S_2)}\texttt{Function-Subtyping}$$

Finally, Subtyping judgements also allow us to uniformly integrate them into the type system through the following Subsumption Rule:

$$\frac{G \vdash e : T \qquad T <: S}{G \vdash e : S}\texttt{Subsumption}$$

Thus, we can treat any value of type T as a value of type S whenever $T <: S$

Note:-

Adding a Subsumption Rule will make searching for Type Derivations more difficult since this rule can be applied elsewhere since trivially, $T <: T$.
However, careful engineering of the typing system can incorporate the Subsumption Rule into a more Deterministic Algorithm (See: Oat Type System)

9.2 Scope-Checking with Free Variables

The System to Scope Check (Untyped) Lambda Calculus is as follows:

1. The variable x is free
$$\frac{x \in G}{G \vdash x}$$
2. G contains the free variables of e_1 and e_2
$$\frac{G \vdash e_1 \qquad G \vdash e_2}{G \vdash e_1\ e_2}$$
3. x is available in the function body
$$\frac{G \cup \{x\} \vdash e}{G \vdash \text{fun} x \to e}$$

Notice the System's similarity with the Formal Definition (copied from above):

Definition 9.2.1 – Formal Definition of the Set of Free Variables for a λ-term We can also define the same for the Set of Free Variables (FV) for an arbitrary λ-term:

1. **Variable**: $FV(x) = \{x\}$
2. **Application**: $FV(MN) = FV(M) \cup FV(N)$
3. **Abstraction**: $FV(\lambda \cdot M) = FV(M) \setminus \{x\}$

Example: To prove that $(\lambda x \cdot x)(\lambda y \cdot y)$ has no free variables $\iff \emptyset \vdash (\lambda x \cdot x)(\lambda y \cdot y)$

$$\frac{\dfrac{\dfrac{x \in \{x\}}{\{x\} \vdash x}}{\emptyset \vdash \lambda x \cdot x} \qquad \dfrac{\dfrac{y \in \{y\}}{\{y\} \vdash y}}{\emptyset \vdash \lambda y \cdot y}}{\emptyset \vdash (\lambda x \cdot x)(\lambda y \cdot y)}$$

$\therefore$ Since we can construct a Valid Derivation Tree for the above, the statement is true!

Note:-

Type Checking == Proving Properties of Programs!

The equivalent `OCaml` code for Scope-Checking can thus be constructed as follows:

```
let rec scope_check (g: VarSet.t) (e: exp) : unit =
  begin match e with
    | Var x -> if VarSet.member x g then () else failwith (x ^ " not in scope")
    | App(e1, e2) -> ignore (scope_check g e1); scope_check g e2
    | Fun(x, e) -> scope_check (VarSet.union g (VarSet.singleton x)) e
  end
```

Chapter 10

Optimizations

To write fast programs in practice:

1. Pick the right algorithms and data structures
   - These have a much bigger impact on performance than compiler optimizations
   - Reduce the number of operations required
   - Reduce memory accesses
   - Minimise indirection
2. **Then**, turn on compiler optimizations
3. Profile to determine program hot spots
4. Evaluate whether the algorithm / data structure design works
5. Tweak the source code until the optimizer does the "right thing" to the machine code

10.1 The Zoo of Optimizations

1. Optimizing Array Bounds Checks in Loops
   In a `for` loop, if the bounds on the index is known, we can perform the Array Bounds Check once and only once.
2. Loop Unrolling
   Rewrite a loop as follows:

   ```
   for (int i = 0; i < 100; i++) {
     s = s + a[i];
   }
   ```

   into:

   ```
   for (int i = 0; i < 99; i=i+2) {
     s = s + a[i];
     s = s + a[i + 1];
   }
   ```

   This allows us to reduce the number of checks done, and the number of `Jmp` instructions that have to be made
3. Constant Folding
   If operands are known at compile time, we can perform the operation statically, i.e.:

Before Constant Folding		**After Constant Folding**
`int x = (2 + 3 * y)`	$\rightarrow$	`int x = 5 * y`
`b & false`		`false`

4. Constant Propagation
   If the value is known to be a constant, we can replace the use of the variable by the constant.
5. Copy Propogation
   If one variable is assigned to another, replace the uses of the assigned variable with the copied variable.
6. Dead Code Elimination
   If a side-effect free statement can never be observed, it is safe to eliminate the statement

7. Function Inlining
   Replace a function call with the body of the function itself, with arguments rewritten to be local variables. This is mainly profitable to:
   - Eliminate the Stack Manipulation, Jump, etc.
   - Enable further optimizations
8. Common Subexpression Elimination
   Fold redundant computations together, i.e.,

```
[a + i*4] = [a + i*4] + 1   →   t = a + i*4
                                [t] = [t] + 1
```

9. and more ...

Note:-

Some optimization locations can be created by Code Translations and/or Earlier Optimizations

10.2 Code Analysis for Safety

Blindly optimizing code at all locations will usually result in unsafe optimizations, that is, optimizations that alter the expected behaviour of the program.

For example, consider the following code:

```
unit f(int[] a, int[] b, int[] c) {
  int j = ...; int i = ...; int k = ...;
  b[j] = a[i] + 1;
  c[k] = a[i];
  return;
}
```

optimized using Common Subexpression Elimination into:

```
unit f(int[] a, int[] b, int[] c) {
  int j = ...; int i = ...; int k = ...;
  t = a[i];
  b[j] = t + 1;
  c[k] = t;
  return;
}
```

This may appear safe at first glance, however, consider the scenario where `a` and `b` are aliases of each other and `i == j`. Then, the expected behaviour of the program changes, since `c[k]` will have a different expected value.

Performing Code Analysis thus helps us to perform safer optimizations. The main Code Analysis we will discuss is **Dataflow Analysis**, which helps us to perform:

- Liveness Analysis (Register Allocation)
- Reaching Definitions Analysis (Constant Propogation)
- Available Expressions Analysis (Common Subexpression Elimination & Copy Propagation)
- Alias Analysis

Note:-

At the end of the day, Compiler Optimizations are good-to-haves.
That is, if given a choice between the Correctness of the Compiler and having an arbitrary Optimization, the Correctness of the Compiler should **always** win.

Definition 10.2.1 – Dataflow Analysis A fundamental technique in static program analysis, used to compute information about the possible states a program can reach. It allows us to analyze and reason about program properties.

10.2.1 Liveness Analysis

Liveness Analysis involves computing Live Variables between each statement, and is used for Register Allocation (See Section 10.3).

Specifically, we can allocate two variables v_1 and v_2 to the same register $\iff$ they are not live at the same time.

Definition 10.2.2 – Live Variable A variable `v` is live at a program point if `v` is defined before the program point, and used after it.

Formally, a variable `v` is live on edge `e` if:

- There is a node `n` in the Control Flow Graph s.t. `use[n]` contains `v`, AND
- There is a directed path from `e` to `n` s.t. for every statement `s'` on the path, `def[s']` does not contain `v`

The formula to compute the $in[n]$ and $out[n]$ is as follows:

- $out[n] := \cup_{n' \in succ[n]} in[n']$
- $in[n] := gen[n] \cup (out[n] - kill[n])$

Liveness Analysis is a Backward Analysis, which starts from the Empty Set. Here, note that the Empty Set would be the most informative in terms of the analysis, as it says that nothing is live, and thus, we can just perform register allocation indiscriminately.

Then, throughout the course of the Analysis, we will add ($\cup$) elements to the Set *monotically*, and thus, reach a fixed point which would represent the liveness of the variables at each statement.

10.2.2 Reaching Definitions Analysis

Reaching Definitions involves computing where each definition statement in the code reaches, and is used to help determine when it is safe to perform Constant Propagation on a particular Constant/Variable.

Specifically, we say that it is safe to perform Constant Propagation on a Constant/Variable if it has not been redefined, and thus, still holds the same value.

Note:-

This does not take into account aliasing scenarios which would make this unsafe. However, we can perform Alias Analysis to check this.

The formula to compute the $in[n]$ and $out[n]$ is as follows:

- $in[n] := \cup_{n' \in pred[n]} out[n']$
- $out[n] := gen[n] \cup (in[n] - kill[n])$

Reaching Definitions Analysis is a Forward Analysis, which starts from the Empty Set. Here, note that the Empty Set would be the most informative in terms of the analysis, as it says that the arbitrary variable `x` is not redefined.

Then, throughout the course of the Analysis, we will add ($\cup$) elements to the Set *monotically*, and thus, reach a fixed point which would represent the reaching variable definitions in the program.

10.2.3 Available Expressions Analysis

Available Expressions is very similar to Reaching Definitions. Available Expressions additionally allows us to check that the Variable still has the same value, rather than just managing to reach a certain statement.

In fact, Available Expressions is used along with Reaching Definitions to perform Copy Propagation. Besides that, Available Expressions is also used to perform Common Subexpression Elimination.

The formula to compute the $in[n]$ and $out[n]$ is as follows:

- $in[n] := \cap_{n' \in pred[n]} out[n']$
- $out[n] := gen[n] \cup (in[n] - kill[n])$

Available Expressions Analysis is a Forward Analysis, which starts from the Power Set. Here, note that the Power Set would be the most informative in terms of the analysis, as it says that all expressions are available.

Then, throughout the course of the Analysis, we will remove ($\cap$) elements from the Set *monotically*, and thus, reach a fixed point which would represent the available expressions in the program.

10.2.4 General Dataflow Analysis

Considering each of the 3 Dataflow Analysis above, we can notice the following:

1. Each analysis has a domain over which they solve constraints
   - **Liveness**: Sets of variables
   - **Reaching Definitions & Available Subexpressions**: Sets of nodes
2. Each analysis has a notion of $gen[n]$ and $kill[n]$, used to explain how information propogates across a node
3. When aggregating information from the in/out flow,
   - Union expresses a property that holds for some path
   - Intersection expresses a property that holds for all path
4. When propagating information,
   - Forward: $in[n]$ is defined in terms of predecessor nodes $out[n]$
   - Backward: $out[n]$ is defined in terms of successor nodes $in[n]$

Thus, each of the different Dataflow Analysis are very similar. In fact, we can construct a General Dataflow Analysis Framework. Before we get into that however, the main items to know of:

- **Domain of Dataflow Values** ($\mathcal{L}$)
- **Top** ($T \in (L)$): Represents having the "maximum" amount of information, the set which represents the optimal position in terms of the Analysis. It is often used as the starting point
- **Meet ($\sqcap$) / Join ($\sqcup$) Operation** : Combines information from multiple paths.
- **Transfer Function (F)**: Defines how information is transformed at each node.

Forward Analysis: Propagates information in the direction of program execution:

$$\text{Out}[n] = F_n(\text{In}[n]), \quad \text{In}[n] = \sqcap_{p \in \text{pred}(n)} \text{Out}[p].$$

Backward Analysis: Propagates information against the direction of program execution:

$$\text{In}[n] = F_n(\text{Out}[n]), \quad \text{Out}[n] = \sqcap_{s \in \text{succ}(n)} \text{In}[s].$$

Type of Analysis	Top Element ($\top$)	Meet/Join	Information Ordering	Direction
Liveness	$\emptyset$ (no live variables)	$\cup$ (union)	Subset ($\subseteq$)	Backward
Reaching Definitions	$\emptyset$ (no reaching definitions)	$\cup$ (union)	Subset ($\subseteq$)	Forward
Available Subexpressions	Set of all expressions (all are available)	$\cap$ (intersection)	Superset ($\supseteq$)	Forward

Table 10.1: Comparison of Dataflow Analyses Properties

Worklist Algorithm

The worklist algorithm iteratively computes dataflow values until a fixed point is reached:

1. Initialize In[n] and Out[n] for all nodes $n \in N$.
2. Add all nodes to the worklist.
3. While the worklist is not empty:
   (a) Remove a node n from the worklist.
   (b) Recompute Out[n] and In[n].
   (c) If changes occur, add affected nodes to the worklist.

10.3 Register Allocation

10.3.1 Linear Scan (Greedy)

Linear-Scan Register Allocation works by greedily allocating a register to variables as and when they become live, then allowing another variable to use the register when the original variable is no longer live.

Advantages:

- Very efficient to allocate (linear time, not including Liveness Analysis)
- Produces acceptable code in many instances
- Allocation step works in a single pass, hence, code can be generated while iterating

- Often used in Just-in-Time (JIT) Compilers

However, Register Allocation is itself not a greedy problem, and thus, this method is not optimal in all cases.

A quick example would be to consider branching, whereby variables in the branch not taken could take up register slots despite not being required.

10.3.2 Graph Coloring

Graph Coloring is a simple problem (introduced in basic Discrete Mathematics for Graphs), however despite that, it is actually NP-Hard.

Note:-

Graph Coloring is polynomial on certain graphs

Additionally, Graph Coloring presents the perfect problem to model our Register Allocation. Specifically, we can construct an Inteference Graph s.t.

- Nodes are variables
- There is an edge between two nodes n and m if n is live at the same time as m

Thus, the problem of Register Allocation is simplified to determining how to color our graph with k = number of available register.

Of course, there is always the scenario where the graph is uncolorable with k colors. In that case, we can simply "spill" a variable onto the stack, which when translated to the Graph Coloring Problem, simply refers to removing a node from the graph. This is also referred to as "Simplifying" the graph.

Typically, we want to simplify the graph, we find a node with degree $< k$, then recursively try to color the remaining subgraph. There are many ways of deciding what node to spill, including but not limited to:

- Pick one that isn't used very frequently
- Pick one that isn't used in a (deeply nested) loop (similar to the above)
- Pick one that has high interference

We can also coalesce nodes to reduce the overall complexity of the graph. For example, if we perform `bitcast` or any other move-related nodes, we can combine the two nodes together.

To do so safely, there are two main algorithms:

1. Brigg's Algorithm: It is safe to coalesce $x\&y$ if the resulting node will have fewer than k neighbors
2. George's Algorithm: It is safe to coalesce $x\&y$ if for every neighbor t of x, either t already interferes with y or t has degree $< k$

Specifically, the full algorithm is as follows:

1. Build Interference Graph, precoloring nodes as necessary
2. Reduce the graph, building the stack of nodes to color
   (a) Simplify the graph as much as possible without removing nodes that are move-related. The remaining nodes should be of high-degree, or be move-related
   (b) Coalesce move-related nodes using Brigg's or George's strategy
   (c) Since coalescing can reveal more nodes that can be simplified, repeat 2a and 2b until no node can be simplified or coalesced
   (d) If no nodes can be coalesced, remove move-related edges and keep trying
3. If there are non-precolored nodes left, mark one for spilling, remove it from the graph and repeat Step 2
4. When only precolored nodes remain, start coloring by popping simplified nodes off the top of the stack