Book of Magne

INF222 Crashcourse 2023

Sander Wiig

INF222 Crashcourse for v2023.
Some sections have been adapted from Anya's INF225 notes and course material.

UNIVERSITY OF BERGEN
Faculty of Mathematics and Natural Sciences

Institute for Informatics
University of Bergen
Norway
May 12, 2023

Contents

Preface

The goal of this script is to provide an overview of the course INF222 at the University of Bergen. It is by no means a comprehensive guide to the course, but rather a supplement to the lectures and exercises.
This was all written throughout a weekend and there is probably a whole lot wrong with the text and the code, so if you find any errors, please let me know by submitting a ticket or pull request on GitHub.
The Glossary is especially lacking. Due to the time constraints, I was not able to update all terms in the glossary, so if you find a term that is not defined, or seems wrong, please let me know.

Contributors

The following people have contributed in some way to this book:

- **Magne Haveraaen** for going over the course outline, and for teaching me what I know.
- **Anya Bagge** for her excellent INF225 notes, and for being a great teacher.
- **Jørn Lode** for his amazing figures that illustrate how states work.
- **Ralf Lämmel** for his book on Software Language Engineering, which much of this book is based on.

Chapter 1

What is a language

1.1 What is a programming language

A programming language

- is an artificial language(i.e made by us humans on purpose)
- used to tell machines what to do

More formally a programming language is a set of rules that converts some input, like strings, into instructions that the computer can follow. This is of course a very general description and it, therefore, follows that there are many different types of programming languages. IT therefore should come as little surprise that we group languages by features and properties. [2]

Types of languages

There are many ways of grouping languages. They can be grouped by Purpose, typing, paradigm, Generality vs. Specificity, and many more. For now, we're going to group them by paradigm, and Generality vs. Specificity.

Generality vs. Specificity

Languages are usually grouped into two categories when based on their specificity.

- DSL
- GPL

Domain Specific Languages are as the name suggests languages with a "specific" domain. DSLs usually have limited scope and use. Examples are JSON and SQL. A Domain-Specific Language is a programming language with a higher level of abstraction optimized for a specific class of problems. Optimized for a certain problem/domain. DSLs can be further subdivided into external DSL(separate programming languages), and internal DLS(language-like interface as a library.)

General Purpose Languages however are more general and can be used to solve many different problems in many different situations. These languages have a wide array of uses and are usually what we think of when we hear the words programming language. Examples of GPLs are Java and Haskell.

Characteristic	DSL	GPL
Domain	Small and well-defined domain	Generality, many use cases
Size	Small ASTs	Large ASTs, often user extensible
Lifespan	As long as their domain	years to decades
Extensibility	Usually not extensible by users	Provides mechanisms for extensibility

Figure 1.1: Some more comparisons between GPLs and DSLs[1]

Syntax and Semantics

All programming languages have two parts; the **Syntax**, and the **Semantics**.
Syntax is the study of *structure*, just as semantics is the study of *meaning*. Or in other words, the syntax tells us *how* to write legal programs, and the semantics tells us *what* those programs do.

1.2 Meta Programming

One of the harder things in the course is **Meta-Programming**. INF222 is usually the first time you've encountered meta-programming and it can be hard a hard concept to grasp. Meta-programming is programming *about* programming. More properly meta-programs treat other programs as data. When you see a data structure like **Basic Typed Language** or **Basic Imperative Programing Language** in Haskell it represents a program.

1.3 Sum of products

You may have encountered the term **Sum of Products**, lets's quickly run over why we use the terms *sum* and *product* to describe the data types, and show some examples in both Haskell and Java.

Haskell

```haskell
data SomeType = A Bool Bool Bool
              | B Bool
              | C
```

Here the type SomeType has 3 constructors, A, B, and C, where A takes 3 parameters, B takes one, and C zero. The type of SomeType could be expressed algebraically as

$$\underbrace{(\text{Bool} \times \text{Bool} \times \text{Bool})}_{A} + \underbrace{\text{Bool}}_{B} + \underbrace{1}_{C}$$

The Bool type can take on 2 different values (False and True), so the constructor can construct $2 * 2 * 2 = 8$ different values, since there are 8 different combinations you can make from 3 booleans (e.g. A True False False is one example). The constructor B can produce 2 different values, and C can only produce one (not zero!). Thus, the total numbers of values of type SomeType is $8 + 2 + 1 = 11$, as the data type is the sum of the three products we've just described.
In short, a sum type denotes "one of" its constituent types (if your function takes SomeType as input, it will get either (A b1 b2 b3), (B b1) or C for some boolean values b1 . . .), while a product type denotes "all of" its constituent types (e.g. a value constructed with A will have all 3 booleans present.) Another way to express a product type in Haskell is with tuples, e.g.

```
1 type MyTriple = (Bool, Int, Char)
```

Java

In Java, we can model the same kind of data types using classes and inheritance. The instance variables of a class determine a "product" type, e.g.

```
1 class SomeClass {
2     boolean a;
3     boolean b;
4     boolean c;
5 }
```

Similar to the constructor for A in the previous section, there are $2 * 2 * 2 = 8$ different values an object of class SomeClass can have. Add another boolean, and you get 16 different values. Technically, a variable of type SomeClass can take one $8 + 1$ different values, since null is also a valid value for all object variables in Java, but sometimes we ignore this fact and tell the users of our functions to kindly not pass in null as an argument where we expect an actual object. Now, to get sum types, we might use class hierarchies in Java:

Now, an object of type SomeType can (ignoring null) take on $8 + 2 + 1$ different values, just like in the Haskell example above.

```
interface SomeType {}

class A implements SomeType {
    boolean a;
    boolean b;
    boolean c ;
}
class B implements SomeType {
    boolean a;
}
class C implements SomeType {
}
```

Chapter 2

Anatomy of an Interpreter

An **Interpreter** is a computer program that directly executes instructions written in a programming language. This differs from a **Compiler**s which translates a program from one language to another(usually to a lower level one ex. C or ASM).
We usually divide the compiler/interpreter process into two categories, front end, and back end. The front end is the part of the compiler/interpreter that takes the source code and converts it into some intermediate representation. In compilers, this is usually some form of byte code or 3-word code, that can then be translated into the target language. This is not necessary for an interpreter where the intermediary representation is often in the form of an annotated AST.

2.1 Phases of an interpreter

An interpreter is composed of several phases.
The front end consists of the lexical analyzer, the syntax analyzer, and the semantic analyzer.
The backend is the evaluator.
The **Lexical Analysis** takes the actual characters that the code is made up of and divides it up into its lexical tokens(by the tokenizer) using the concrete syntax of the program[1]. It then sends that token stream to the next part of the interpreter. The **Syntax Analyzer**
The syntax analyzer builds a parse tree out of the tokens and the concrete syntax. This is then converted into an AST by the abstract syntax rules. This AST is then handed over to the next part, the **Semantic Analyzer**. The AST is type-checked, checked for well-formedness, names are resolved, and types are inferred. The new AST, now with added information, is then given to the evaluator so that it can be evaluated and produce a result. See Figure 2. for a visual description of the steps.

[1]Not covered by this course, so you can safely ignore how this works.

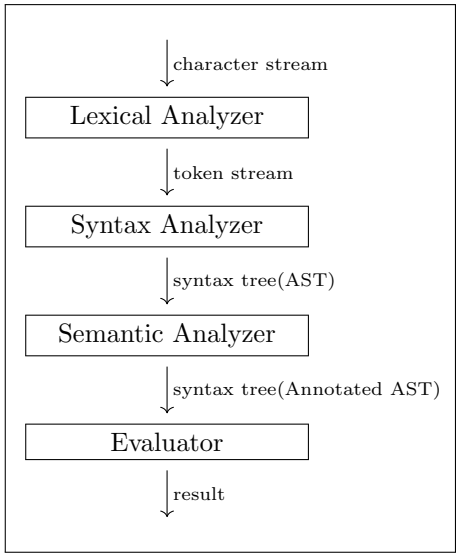

Figure 2.1: Phases of an interpreter

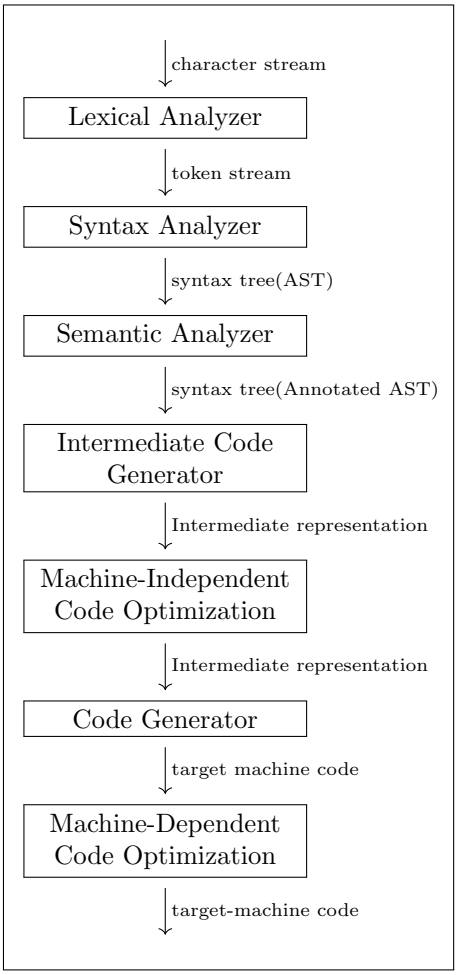

Figure 2.2: Phases of a "normal" compiler

Figure 2.3: Please appreciate the figures above, they were hard to make

2.2 ASTs & Static Analysis

2.3 ASTs

An **Abstract Syntax Tree** is the tree representation of the syntactic structure of a program; The abstract syntax describes the structure of the abstract syntax tree - it can be defined using a regular tree grammar, or an algebraic data type or term (in Rascal, ML, Prolog, ...), or an object-oriented inheritance hierarchy of node classes (Java, C++, ...), or as an S-expression (in Lisp languages). The abstract syntax tree can be used as an internal representation in a language processor, but it is not the only possible representation.
An abstract syntax can be generated by a grammar in the following way: For every non-terminal

type, there is a corresponding abstract syntax type. Each type has one constructor (or node type) corresponding to each production in the grammar, with one child for every symbol in the production that is not a literal token (e.g., punctuation, keywords, or spaces). If a constructor has only one child, of the same type, it can be removed (e.g., this would be the case for a parenthesis expression). You can do this process entirely based on the information contained in a parse tree. Translating a parse tree into a corresponding abstract syntax tree is called imploding the parse tree. Given an abstract syntax tree, it is possible to reconstruct a parse tree or program text, given the original grammar - though the resulting program may be slightly different in terms of spaces and punctuation. This is called unparsing or pretty-printing (particularly if the output is nicely formatted). Parsing, imploding, pretty-printing, and then reparsing may not yield the exact same parse tree as the original tree, but it should still implode to the same abstract syntax tree (otherwise there is a bug in your toolchain!).

Various phases in a language processor may change the abstract syntax tree, or use slightly different versions of the abstract syntax (e.g., after type checking, the nodes for variables include the type of the variable) - it is also possible to decorate or annotate the AST as processing proceeds. This adds extra information to the nodes in the AST, without impacting the structure of the abstract syntax. Important abstract syntax design considerations are;

- Simplicity. Generally, your compiler tools will do a lot of work on AST, and the fewer different cases you have to worry about, the better. For For example, if the processing of overloaded functions and operators is the same (which it is to some degree in C++), you may want to have only one AST node type to cover both. Having a lot of unnecessary nodes in the tree can be annoying as well, and may make processing slower.

- Good correspondence with the constructs of the language.

- Availability of information during processing. Some information that can be computed from the tree (such as type information) and might be encoded directly in the tree (at least at later stages) for easy processing.

- Being end-user friendly or familiar to most programmers isn't an important consideration - the abstract syntax may be radically different from the surface concrete syntax if that helps the compiler writer.

2.3.1 Sum of products

You may have encountered the term **Sum of Products**, lets's quickly run over why we use the terms *sum* and *product* to describe the data types, and show some examples in both Haskell and Java.

Haskell

```
1 data SomeType = A Bool Bool Bool
2               | B Bool
3               | C
```

Here the type SomeType has 3 constructors, A, B, and C, where A takes 3 parameters, B takes one, and C zero. The type of SomeType could be expressed algebraically as

$$\underbrace{(\text{Bool} \times \text{Bool} \times \text{Bool})}_{A} + \underbrace{\text{Bool}}_{B} + \underbrace{1}_{C}$$

The Bool type can take on 2 different values (False and True), so the constructor can construct $2 * 2 * 2 = 8$ different values, since there are 8 different combinations you can make from 3 booleans (e.g. A True False False is one example). The constructor B can produce 2 different values, and C can only produce one (not zero!). Thus, the total numbers of values of type SomeType is $8 + 2 + 1 = 11$, as the data type is the sum of the three products we've just described.
In short, a sum type denotes "one of" its constituent types (if your function takes SomeType as input, it will get either (A b1 b2 b3), (B b1) or C for some boolean values b1 . . .), while a product type denotes "all of" its constituent types (e.g. a value constructed with A will have all 3 booleans present.) Another way to express a product type in Haskell is with tuples, e.g.

```haskell
type MyTriple = (Bool, Int , Char)
```

Java

In Java, we can model the same kind of data types using classes and inheritance. The instance variables of a class determine a "product" type, e.g.

```java
class SomeClass {
    boolean a;
    boolean b;
    boolean c;
}
```

Similar to the constructor for A in the previous section, there are $2 * 2 * 2 = 8$ different values an object of class SomeClass can have. Add another boolean, and you get 16 different values. Technically, a variable of type SomeClass can take one $8 + 1$ different values, since null is also a valid value for all object variables in Java, but sometimes we ignore this fact and tell the users of our functions to kindly not pass in null as an argument where we expect an actual object. Now, to get sum types, we might use class hierarchies in Java:

```java
interface SomeType {}

class A implements SomeType {
    boolean a;
    boolean b;
    boolean c ;
}
class B implements SomeType {
    boolean a;
}
```

```
class C implements SomeType {
}
```

Now, an object of type SomeType can (ignoring null) take on $8 + 2 + 1$ different values, just like in the Haskell example above.

2.4 Static Analysis

2.4.1 Type Checking

Programming language typing always falls into one or two categories; static, and dynamic typing. Static typing is when type checking happens at compile-time, while dynamic typing happens during runtime. We will focus on static typing. Statically typed languages typically have these properties.

- Variables and data structures must be declared before use.
- Variables and data structure fields can only hold values of the declared type.
- Operations(i.e functions, procedures, methods) and types must be declared.
- Declaring the exact types of variables and operations isn't always needed. Some languages use type inference (Discussed in 3.2.3).

What is a type checker and how does it work? A type checker is a meta-program that checks that verifies that the type of some construct(lists, expressions, etc.) matches what's expected of it. For example, a type checker will check that the Plus **Expression** takes two Integers. This lets the type checker discover and report certain errors before the program runs. To do this a type checker needs to know;

- How the language should look.
- The language types.
- Rules for assigning types to the constructs.

Let's do a practical example

Here's the abstract syntax for our Simple Typed Language or STL for short.

```
type VarDecl = [(String, ExprType)]

data Expr =
       Var String
       | I Int
       | B Bool
       | BinOp Op Expr Expr
       | UnOp Op Expr
       | Choice Expr Expr Expr deriving (Show, Eq)
data Op = Plus | Mult | Or | And | Not | Eq deriving (Show, Eq)
```

Before we can continue we need to define the types that are allowed. We decide to use two types, Integers, and Booleans.

```
data ExprType = Integer | Boolean deriving (Show, Eq)
```

Now to the meat of the exercise, the type checker itself. The usual way to do this in Haskell(and most other languages I've experienced) is to define a series of recursive functions, one for each expression/operand.

```
typeCheck ::VarDecl->Expr->ExprType
```

I have found it easiest, to begin with, the cases that form the basic "building blocks" of a language, the literals. It is easy to know the type of Int Literals(Integer obviously), and Bool Literals(Boolean).

```
typeCheck _ (I _) = Integer
typeCheck _ (B _) = Boolean
```

After these "base" cases we add a case for Vars, this is slightly more complicated since the type is dependent on the list of var declarations, so we need to check the VarDecl list.

```
typeCheck vars (Var name) =
               case lookup name vars of
                      Just tp -> tp
                      Nothing -> error $ "No variable could be found named "++name
```

We use a pretty clever Haskell expression called "case". This lets us pattern match the result of the function call in lookup. I strongly recommend that you all learn how to use these since they are extremely useful and I'll be using them liberally during this example.
We now move on to the ops.

```
typeCheck vars (UnOp Not expr) =
                      case typeCheck vars expr of
                             Boolean-> Boolean
                             _-> error "Argument not boolean"

typeCheck vars (BinOp Plus left right) =
                      case (typeCheck vars left, typeCheck vars right) of
                             (Integer, Integer)->Integer
                             _->error "One of the args is not an Integer"

typeCheck vars (BinOp Mult left right) =
                      case (typeCheck vars left, typeCheck vars right) of
                             (Integer, Integer)->Integer
                             _->error "One of the args is not an Integer"

typeCheck vars (BinOp Or left right) =
                      case (typeCheck vars left, typeCheck vars right) of
                             (Boolean, Boolean)->Boolean
```

```
19                              _->error "One of the args is not a Boolean"
20
21 typeCheck vars (BinOp And left right) =
22                         case (typeCheck vars left, typeCheck vars right) of
23                                (Boolean, Boolean)->Boolean
24                                _->error "One of the args is not a Boolean"
25
26 typeCheck vars (BinOp Eq left right) = Boolean
```

These all more-or-less follow the same pattern [2]. They all check that the arguments are of the correct type and return the operand type if so, if not they raise an error. Eq is the odd one out since it returns a boolean no matter what since it checks if two expressions are the same.
The last case is the most complex. Choice tests a boolean condition and returns one of the two expressions depending on the value. The problem is that we don't know which branch will return. We have therefore decided both branches need to be of the same type.

```
1 typeCheck vars (Choice test left right) =
2                         case typeCheck vars test of
3                                Boolean | l == r -> r
4                                        | otherwise -> error "Args did not match"
5                                _-> error "Test condition is not a Boolean"
6                         where
7                                l = typeCheck vars left
8                                r = typeCheck vars right
```

Here we begin by checking that the test evals to a boolean type, if so we check that both branches have the same type and return that type. Note that even though an evaluator would only evaluate one of the two branches, we still type-check them both.

2.4.2 Wellformedness

For a program to be **Wellformed**it needs to satisfy all the constraints(kinda like rules) on it. This means that the program follows all the rules for it like;

- The program conforms to the AST.
- It is typed correctly
- All procedure calls/declarations are wellformed.
- and much more.

To check if a program is wellformed we usually implement a so-called constraint checker. These are pretty much just unit tests. The recipe for a constrain checker is as follows;

- **Negative test cases** - Designate one negative test case for each constraint that should be checked. Ideally, each such test case should violate only one constraint.
- **Reporting** - Choose an approach to "reporting". The result of constraint violation may be communicated either through a boolean value, as a list of errors, or by throwing an exception.

[2] Maya made me use fancier words, apparently "pretty similar" isn't good enough.

- **Modularity** - Implement each constraint in a separate function, thereby allowing modularity and testing.
- **Testing** - The constraint violations must be correctly detected for the negative test cases. The positive test case must pass.

Bibliography

[1] Ralf Lämmel. *Software Language Engineering*. Springer International Publishing, 2018.

[2] Anya Bagge w/Ralf Lämmel Vadim Zaytsev. Inf225 notes. Lecture Notes for INF225, University of Bergen, Norway, 2016.