Book of Magne

INF222 Crashcourse 2023

Sander Wiig

INF222 Crashcourse for v2023.

UNIVERSITY OF BERGEN
Faculty of Mathematics and Natural Sciences

Institute for Informatics
Bergen Language Design Laboratory
The University of Bergen
Bergen, Norway
May 14, 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[2].
- **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[1].
- for proofreading, and verifying what I have written.

Figure 1: `https://github.com/Swi005/Book-of-Magne/tree/v2023`

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.

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: Comparison between GPLs and DSLs

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. *TODO:Talk about meta languages, Object languages, and classify BTL, BIPL, and PIPL as DSL or GPL*

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 number 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:

```
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) have $8 + 2 + 1$ different values, just like in the Haskell example above.

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. The back end is the part of the compiler/interpreter that takes the intermediary representation and executes it(in the case of interpreters) or translates it back into code(compilers).

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]. Take for instance the following expression:

$$(1 + 2) * 13$$

The lexical analyzer would divide this into the following tokens: `["(", "1", "+", "2", ")", "*", "13"]`

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

This list of tokens is then sent to the **Syntax Analyzer**
The syntax analyzer takes the token list and builds a parse tree out of the tokens and the concrete syntax (fig.2.1).

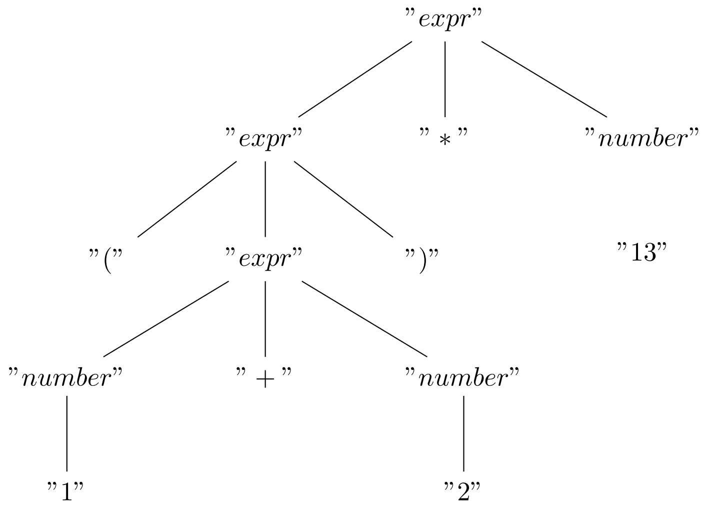


Figure 2.1: Parse tree

The parse tree is then converted into an **Abstract Syntax Tree** by the abstract syntax rules (fig.2.1).

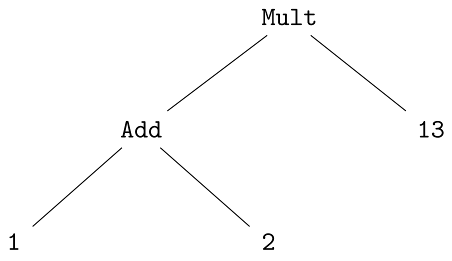


Figure 2.2: AST

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. Which 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 Fig. 2.3.

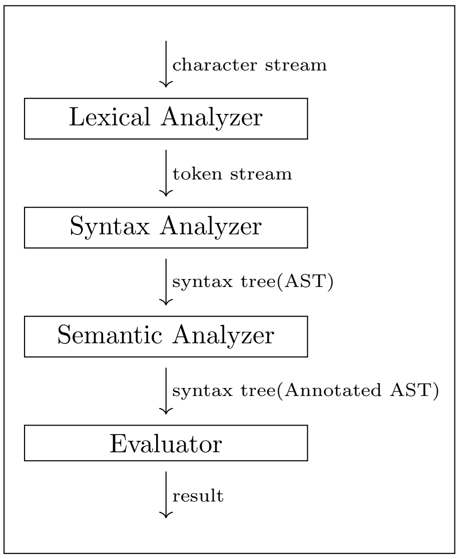


Figure 2.3: Phases of an interpreter

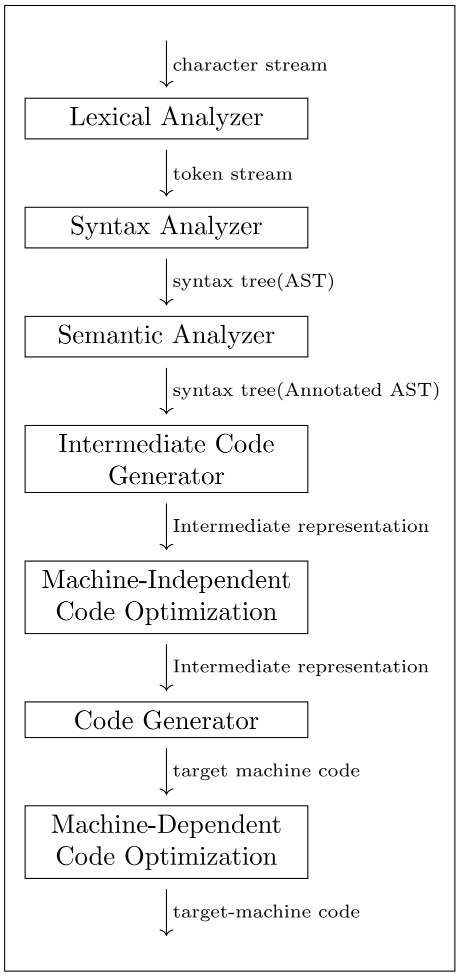

character stream
Lexical Analyzer
token stream
Syntax Analyzer
syntax tree(AST)
Semantic Analyzer
syntax tree(Annotated AST)
Intermediate Code Generator
Intermediate representation
Machine-Independent Code Optimization
Intermediate representation
Code Generator
target machine code
Machine-Dependent Code Optimization
target-machine code

Figure 2.4: Phases of a "normal" compiler

Figure 2.5: 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 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.

2.4 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.

```haskell
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.

```haskell
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.

```haskell
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).

```haskell
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.

```haskell
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.

```haskell
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
                             _->error "One of the args is not a Boolean"

typeCheck vars (BinOp And left right) =
                      case (typeCheck vars left, typeCheck vars right) of
                             (Boolean, Boolean)->Boolean
                             _->error "One of the args is not a Boolean"

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.

```
typeCheck vars (Choice test left right) =
                      case typeCheck vars test of
                             Boolean | l == r -> r
                                     | otherwise -> error "Args did not match"
                             _-> error "Test condition is not a Boolean"
                      where
                             l = typeCheck vars left
                             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(think 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.

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

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.
- **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.

Chapter 3

Store, State, and Storage

3.1 State

3.1.1 Store

When running a program we often want to remember intermediary values or have variables. For us to do this we need some way of storing values. To do this we create a **Store**. A store is very simply an array, usually containing either bytes or ValueTypes. The store is the program's memory(this is true in C). To access a value located at i in our store we simply access the value at array index i. Worth noting that the store is what we call the program heap and by tradition, it grows upwards. This means that new entries are stored at the highest available index in our store. This is because the stack grows upwards and gives us the best possible use of the memory.

3.1.2 Variables and Enviroment

We now have a place to store stuff, but how do we know where it is in the store? This is where **Enviroment**s comes into play.
An environment is a map between variables and their location in the store.
Elaborate more about variables as name bindings.

3.2 Scoping

TODO: Rewrite scoping section

Most variables are not visible to the entire program. In previous courses, you have in all likelihood encountered global and local variables when programming. Where global variables are variables that are visible to the entire program, and local variables are variables that are only visible to the method or function that they were declared in.

The term for where a variable is visible is called the **Scope** of a variable. Variable scoping is useful because it lets us keep variables in different parts of our programs separate. If you were to write a calculator it would be somewhat awkward if you could only use `x` and `y` once.

Scope can also apply to more than just variables, and all declarations usually have scope.

Most languages generally use one of three classes of scoping, and these can have wildly different effects on a program's semantics:

- **Runtime Scoping** - In Runtime Scoping the variable that is in scope is determined by the execution of the program and is the variable that was last seen. Runtime scoping makes no difference between declaration and usage. Variables are declared, initialized, and updated as the interpreter proceeds with the code. Depending on the code's branching structure, a variable may or may not be declared. This means that a variable's scope is often the entire program.
- **Static Scoping** - In Static Scoping the variable that is in scope is determined by the structure of the program. This means that the variable that is in scope is the variable that was declared in the closest enclosing scope. A variable's scope is often just the block where it was declared. Static scoping is easy to reason about and is the most common form of scoping used today.
- **Dynamic Scoping** - In dynamic scoping the scope of a variable is determined by the usage context, instead of its declaration. Dynamic scoping is often hard to reason about since a variable has to be reasoned about in every usage context. Dynamic scoping is therefore not used very often.

To better illustrate the differences let's take a look at an example!

```
1   // A program to demonstrate static scoping.
2
3   int x = 10;
4
5   // Called by g()
6   int f()
7   {
8      return x;
9   }
10
11  // g() has its own variable
12  // named as x and calls f()
13  int g()
14  {
15     int x = 20;
16     return f();
17  }
18
19  int main()
20  {
21    print(g());
22    return 0;
23  }
```

Figure 3.1: Scoping Example

Runtime Scoping

If we execute the program with runtime-scoping the program will return 20. This is because the variable `x` is first declared in line 3, but is then overridden in `g()` on line 5. When we call `g()` on line 9, the variable `x` is set to 20, and this is then returned by `f()`.

Static Scoping

If we execute the program with static scoping the program will return 10. This is because the variable `x` is first declared on line 3, before being shadowed in `g()`, but because `x` is only 20 within `g()`, the `x` in `f()` is still the same as it was on line 3.

Dynamic Scoping

The result of executing the program with dynamic scoping is 20 because the variable `x` is set to 20 in `g()` and `g()` calls `f()`.

3.3 Data Structures in Memory

3.3.1 Arrays

An array is a type of data structure that holds a fixed number of elements of the same type. Arrays are usually stored in memory as a contiguous block of memory. Arrays are usually indexed by integers.

Let's look at an example.

```
        int foo[5] = {1, 2, 3, 4, 5};
```

Figure 3.2: Array of Ints with 5 elements.

In the code above we have an array consisting of 5 integers, addresses take up 1 byte of memory, and integers take up 4.

The array is stored in memory as a contiguous block of memory, and the variable `foo` is a pointer to the first element in the array(Figure 3.3). To access the elements in the array we can use the pointer `foo` and add an offset to it. The address for the $n'th$ element in an array is given by the following formula, where

$$\texttt{foo[n]} = \texttt{\&foo} + n * \texttt{sizeof(int)}$$

- `&foo` is the pointer to the array
- `n` is the index of the element we want to access
- `sizeof(int)` is the size of an integer in bytes

3.3.2 Matrixes and Multidimensional Arrays

Things get a little bit more complicated when dealing with multidimensional arrays, and the exact nature of how multidimensional arrays are stored in memory depends on the language.

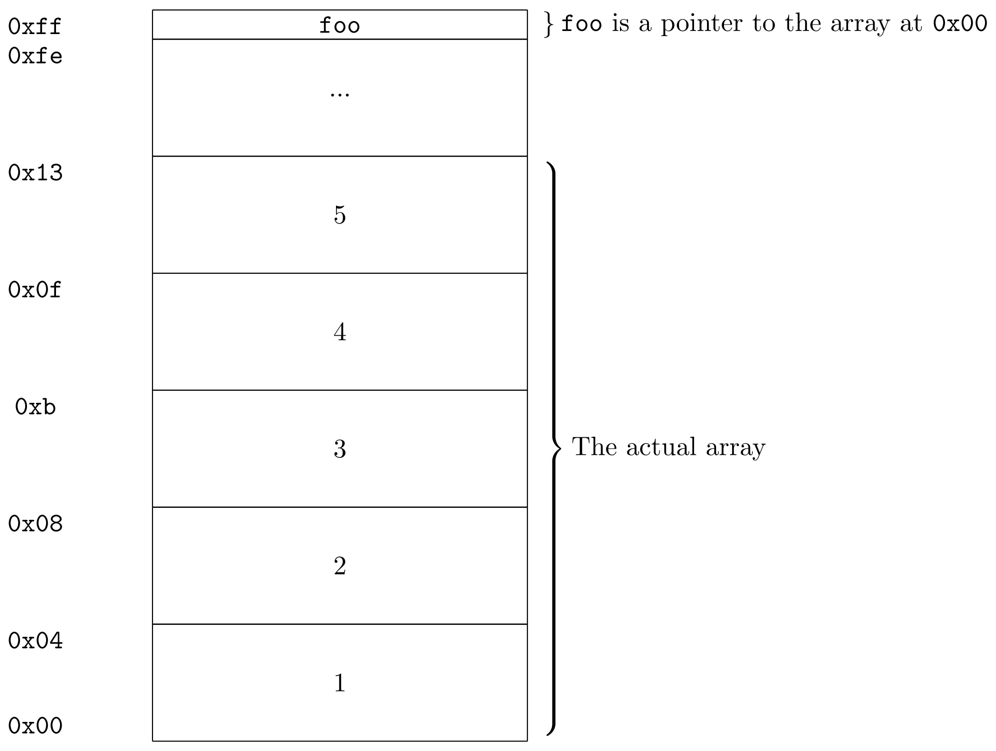


Figure 3.3: `foo` in memory

Java

C++

Fortran

3.3.3 Records, Structs

Records are a data structure that can hold multiple values of different types. Records are also known as structs, tuples, or classes depending on the language.

```
struct foo {
   int x; //Offset 0
   bool y; //Offset 4
   double z;//Offset 5
};
```

$$\begin{aligned}|\texttt{foo}| &= |\texttt{x}| + |\texttt{y}| + |\texttt{z}| \\ &= 4 + 1 + 8 = 13\end{aligned}$$

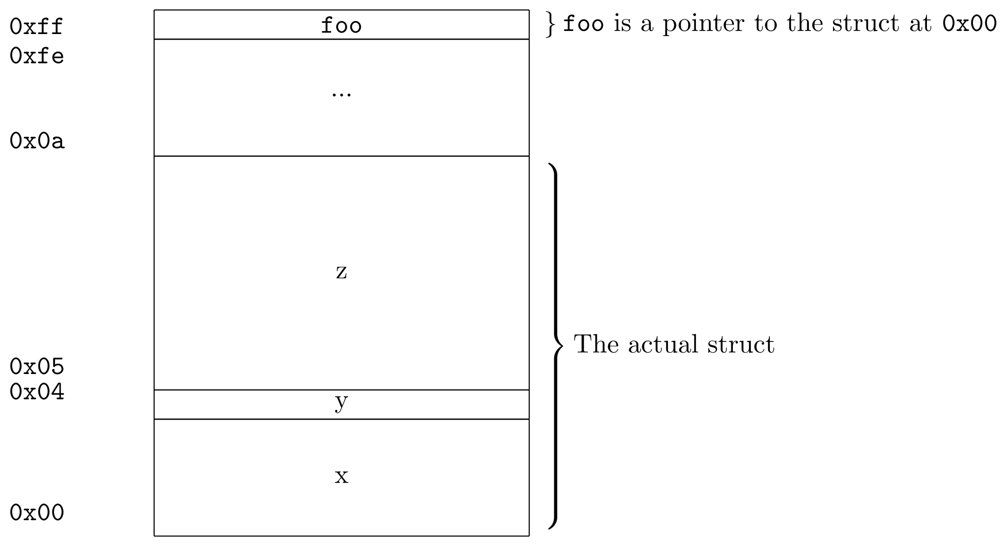


Figure 3.4: `foo` in memory

Arrays of Records

Speaking of procedures...

Chapter 4

Procedures

A procedure is a programming construct that abstracts away the implementation of an algorithm. Instead of writing 20 lines of code every time you want to run an algorithm we simply call a procedure with those 20 lines of code inside it. This increases the readability of our code and makes it easier to write it.

4.1 Procedure Declarations

In this course, we have a simplified procedure declaration inspired by PASCAL. Our procedure declarations have a list of parameters(variables that pass in/out of the procedure), another list of local variables(variables that are used within), and a single statement(the algorithm itself). If we were to translate this abstract syntax into something more concrete then a typical procedure might look something like this.

```
procedure p (upd a: integer , out b: integer ; obs c : boolean ) ;
var x: integer ;
    y: boolean ;
    z : boolean ;
begin
    if c then begin b := a ; x := b ; a := x end;
    y := not c ;
    c := y or c ;
end;
```

The abstract syntax would be something like this in Haskell;

```haskell
-- | Procedure declaration
data ProcedureDeclaration = Proc
                             String -- Name of the procedure
                             [Parameter] -- Parameter list
                             [VarDecl] -- Local variables
                             Stmt -- Statement part
                             deriving (Show, Eq, Read)

-- | Procedure parameters: mode and variable declaration
type Parameter = (Mode, VarDecl)

```

```haskell
-- | Parameter modes: observe, update, output
data Mode = Obs | Upd | Out
      deriving (Show, Eq, Read)

-- | Variable declaration: variable name and its typ
type VarDecl = (Var,Type)
```

4.1.1 Parameters & Local Variables

A **Local Variable** is a variable that only exits in a specific part of the program. We're going to talk about them in the context of procedures. These local variables are declared or defined and only used within that procedure. Since the procedure exists in a vacuum, the local variable is allowed to have the same name as variables in other parts of the program. Since the variable is stored at a different store location any change to the local variable wouldn't change the variable outside.

Parameter are the variables the procedure uses to communicate to the outside world. They are variables, but with one key difference; They also specify *how* it communicates with the outside world, the parameters specify if the variable is an output variable(write), observed only(read), or updatable(is this a word?)(read and write).
Another way of looking at parameters is that parameters are local variables that are initialized with the passed arguments at invocation time.

4.1.2 Performing a Procedure

When performing a function we can usually assume that the parameters have already been added to the environment and initialized and that except for those parameters we have a clean environment to work with. The performing of a procedure consists of the following steps;

1. The first is that we need to somehow remember the current environment or delete all the local variables when we're done. The best way is to get the current **Stackframe**.
2. We then add all the local variables.
3. We then execute the procedure statements.
4. We then reset the environment back to how it looked before using the saved stack frame from (1). This ensures that all local variables that shouldn't exist outside the procedure are removed.

4.2 Executing a Procedure Call

We now know how to actually perform a procedure, but for us to even be able to perform a procedure in the first place we need to prepare the environment and program for it. There are two main ways of making the program ready to perform a procedure. These are dependent on what type of **Argument** passing we are doing.

4.3 Arg Passing

Argument passing fall into one of two camps.

4.3.1 Copy Semantics

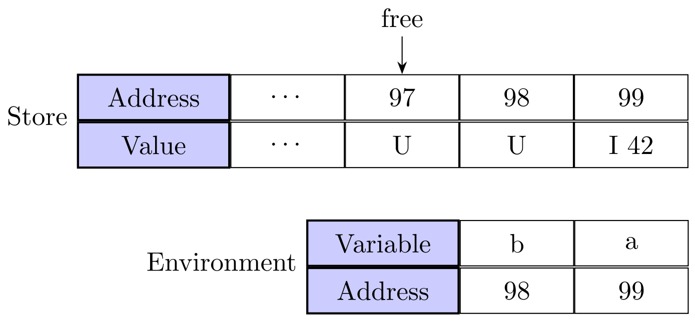
The first is **Copy Semantics**. In copy semantics, all the arguments are passed to the parameters by copying the value into them. With copy semantics, the procedure arguments are first evaluated and then bound to the parameters when they are added. The procedure is then performed, and the results are obtained. Those are then copied back to all the arguments that are upd or out. To further explain let's look at a trivial example to understand what happens in-store. Say we have a procedure that simply adds 5 to any number.

```
procedure add5(obs x: integer, out res: integer)
begin
   res = x + 5;
end;
```

Now let's look at the main procedure

```
procedure main()
   var a: integer;
   var b: integer;
begin
   a = 42;
   add5(a, b);
end;
```

Figure 4.1 shows us what the state looks like before we start executing the procedure call. The



Address	···	97	98	99
Value	···	U	U	I 42

Variable	b	a
Address	98	99

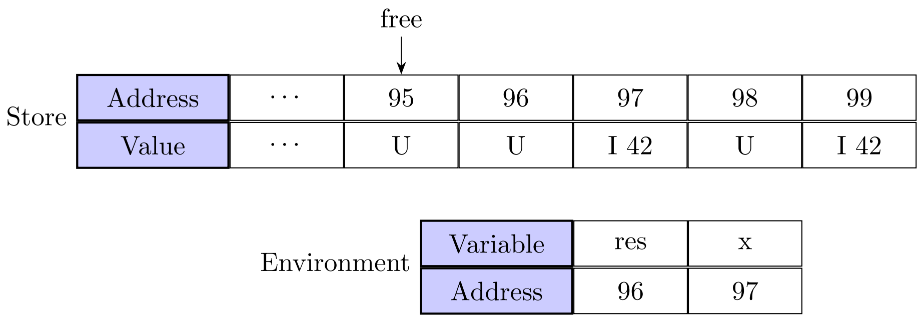
Figure 4.1: State before the procedure call

first thing we do when entering the procedure call is to save the stackframe so we can restore the environment to its proper place once we're done. We then clear the environment, ensuring that the only variable declared by the procedure is in scope. We then add all the parameters(in our case we add x and res) and copy the values of a and b into x and res respectively [1]. Figure 4.2 shows what the state looks like at this point.

We have now prepared the procedure so that it can be performed. Figure 4.3 shows how the state looks after its been performed.

Now comes the fun part, *cleaning*! Just performing the function isn't enough, if we were to just drop it here we would have an environment that is drastically different than then when we started. We could just reset the stackframe to what it was when we started, but then b wouldn't get its new value. What we need to do is copy the values of the out/upd params back to their

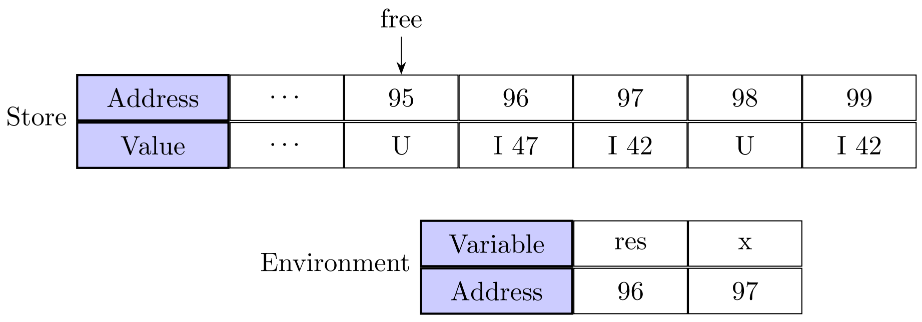
[1]since res is out it isn't initialized



Address	···	95	96	97	98	99
Value	···	U	U	I 42	U	I 42

Variable	res	x
Address	96	97

Figure 4.2: State when entering add5 (copy semantics).

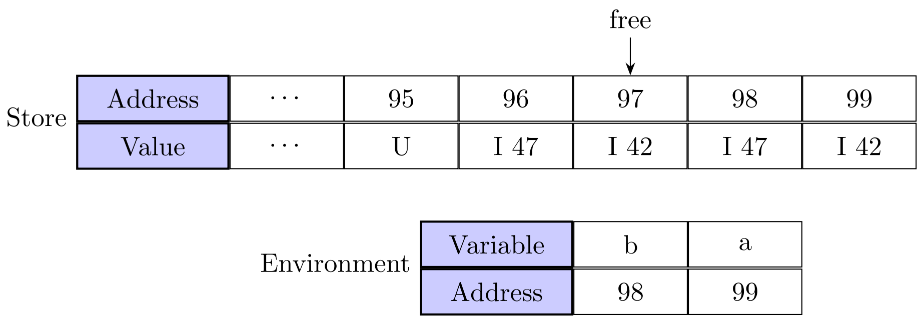


Address	···	95	96	97	98	99
Value	···	U	I 47	I 42	U	I 42

Variable	res	x
Address	96	97

Figure 4.3: State after res = x +5 in add5 (copy semantics).

argument variables. Then we can reset the stackframe making our state look like fig. 4.1. As you can see the free pointer now points to where x used to point and b now has the same value as a res. That means that the next time we add something to the environment it will overwrite those values in the store.



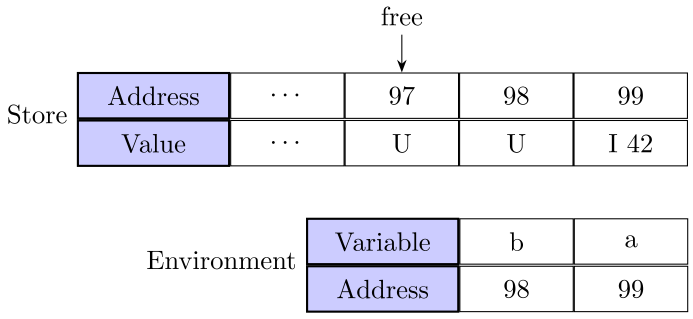
Address	···	95	96	97	98	99
Value	···	U	I 47	I 42	I 47	I 42

Variable	b	a
Address	98	99

Figure 4.4: State after add5(a, b) in main (copy semantics).

4.3.2 Reference Semantics

Now, what if `add5` could instead write *directly* to b in `main` instead of having to allocate its own copy which is later copied? This is the main advantage of **Reference Semantics**. Lets reuse the example from before. As you can see the from fig. 4.5 the state before we enter the procedure call is the same as during copy semantics.

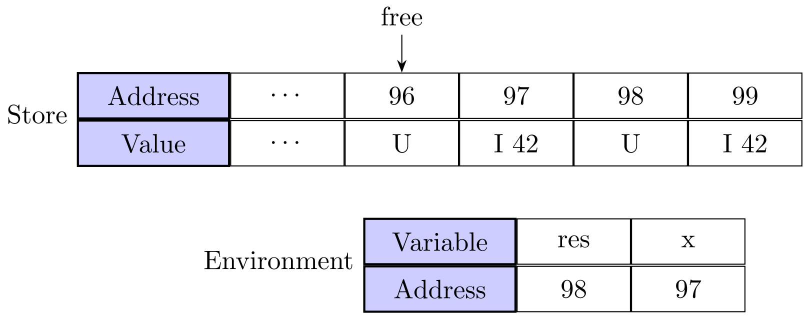


Store				
Address	⋯	97	98	99
Value	⋯	U	U	I 42

Environment		
Variable	b	a
Address	98	99

Figure 4.5: State before the procedure call

It is now we encounter our first change. Like before we get the stackframe before clearing the environment and adding the parameters, but here comes the change. instead of allocating space and copying the value of the args to the upd/out params we instead set their address to the address of the corresponding argument. We now get a state that looks like fig. 4.6. Since x is an obs parameter it is still allocated and copied to as normal.

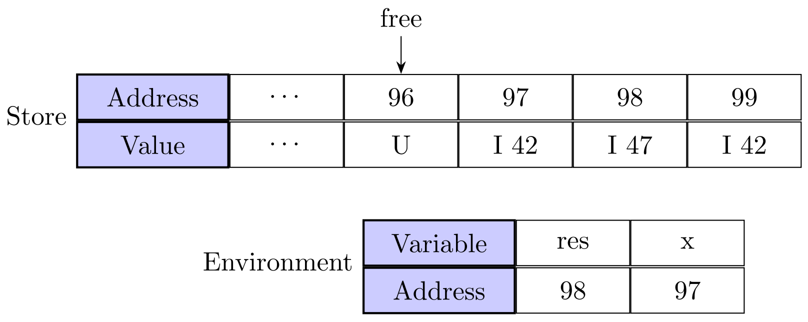


Store					
Address	⋯	96	97	98	99
Value	⋯	U	I 42	U	I 42

Environment		
Variable	res	x
Address	98	97

Figure 4.6: State when entering add5 (reference out).

Since res now points to the same place as b, any changes to ref will also be reflected by b. This makes cleanup much easier! infact all we have to do after performing the procedure is to reset the enviroment back to where it was and call it a day makng the state look like fig. 4.7.



free

Store

Address	· · ·	96	97	98	99
Value	· · ·	U	I 42	I 47	I 42

Environment

Variable	res	x
Address	98	97

Figure 4.7: State after res = x+5 in add5 (reference out).

Chapter 5

Signatures

So far we've defined all our operations as part of our AST. By introducting signatures into our compilers we can abstract away the implementation of our operations from the AST. This allows us to change the implementation of our operations without changing the AST. Before we can do this we need to talk a bit about the theory behind signatures and algebras

5.1 A little bit of Theory

5.1.1 Signatures

Signatures are a way to define a set of operations and their types.

Formaly a signature[1] is defined as $I = \langle S, F \rangle$ where

- S is a set of sorts(aka typenames), and
- F is a set of function declarations $f : s_1, ..., s_n \to s$, for $s1, \ldots, s_n, s \in S$

Signatures alone do not define any semantics, they are just a way to define which operations and types exist, but not what they do. To define the semantics we need to define what we call an algebra.

Here's an example of a signature.

$$\begin{aligned} Nat = \langle & \{N\}, \\ & \{zero : N, \\ & \ succ : N \to N\}\rangle \end{aligned}$$

5.1.2 Algebras

An algebra is a way to define the semantics of a signature. An algebra assigns every sort to a domain and every function to a function on the domains. An algebra A for a signature $I = \rangle S, F\langle$ defines

- a set $[\![s]\!]_A$ for every sort $s \in S$, and
- a total function $[\![f]\!]_A : [\![s_1]\!]_A \times ... \times [\![s_k]\!]_A \to [\![s]\!]_A$ for every $(f : s_1, ..., s_k \to s) \in F$

[1] also called an *Interface*

It's possible to have multiple algebras for a signature. Here are a couple of algebras for the Nat signature.

$$\begin{aligned}
[\![N]\!]_{A_1} &= \mathcal{N} \\
[\![zero]\!]_{A_1} &= 0 \\
[\![succ]\!]_{A_1} &= \lambda n \mapsto n + 1 \\
\\
[\![N]\!]_{A_2} &= \{1, 2, 7\} \\
[\![zero]\!]_{A_2} &= 2 \\
[\![succ]\!]_{A_2} &= \lambda n \mapsto 7
\end{aligned}$$

5.2 Implementing Algebraic Specifications

Integrating signatures and algebras into our interpreter means we can use Algebraic specification theory to formalize and reason about it. It also lets us change the implementation of our operations without changing the AST and is one step towards user-defined types, and generic programming.

The AST in Figure. 5.2 can be rewritten to make use of signatures.

```haskell
type Env = [(String, Int)]

data Expr
    = Lit Int
    | Add Expr Expr
    | Sub Expr Expr
    | Mult Expr Expr
    | Div Expr Expr
    | LEQ Expr Expr
    | Var String
    deriving (Show, Eq)

eval :: Env -> Expr -> Int
eval _ (Lit n) = n
eval env (Add e1 e2) = eval env e1 + eval env e2
eval env (Sub e1 e2) = eval env e1 - eval env e2
eval env (Mult e1 e2) = eval env e1 * eval env e2
eval env (Div e1 e2) = if eval env e2 /= 0 then div (eval env e1) (eval env e2) else
    error $ "Division by zero: " ++ show e1 ++ " / " ++ show e2
eval env (LEQ e1 e2) = if eval env e1 <= eval env e2 then 1 else 0
eval env (Var string) = case lookup string env of
                        Just n -> n
                        Nothing -> error $ "Variable " ++ string ++ " not found in
                            environment"
```

Figure 5.1:

We do this by creating a new AST without any operations apart from Literals, Variables, and Function calls. And instead of returning a `Int` we make it so that the AST can work for any `valuedomain`(FIgure 5.2). We also change the evaluator to reflect this change. One major

```haskell
type Env valueDomain = [(String, valueDomain)]
data Expr valueDomain
          = Lit valueDomain
          | Var String
          | FunCall String [Expr valueDomain]
```

Figure 5.2:

change(apart from the lack of operations) is that we now need to pass the algebra(`funmod`) to the evaluator.

```haskell
eval :: (String -> [valueDomain] -> valueDomain) -> Env valueDomain ->Expr valueDomain
    -> valueDomain
eval fmod env (Lit n) = n
eval fmod env (Var string) = case lookup string env of
                      Just n -> n
                      Nothing -> error $ "Variable " ++ string ++ " not found in
                          environment"
eval fmod env (FunCall f args) = fmod f (map (eval fmod env) args)
```

Figure 5.3:

Not that we've adapted the AST for signatures it's time to implement a signature for the operations we removed.

```haskell
intrinsics :: Signature
intrinsics = (["Int", "Bool"],[
              ("add", ["Int", "Int"], "Int"),
              ("sub", ["Int", "Int"], "Int"),
              ("mult", ["Int", "Int"], "Int"),
              ("div", ["Int", "Int"], "Int"),
              ("leq", ["Int", "Int"], "Bool")
          ])
```

Figure 5.4:

You may have noticed one major change from the old AST. The old AST only worked on `Int`s, but we've added a second sort `Bool` to the signature. Since Haskell only lets us use one type for the valuedomain we need to create a new type that can hold both `Int`s and `Bool`s.

We then define our algebra for the functions in the signature.

Now, there are some downsides to using signatures. For one we no longer get to use Haskell's built-in type system to check that we're using the right types. That means that we need to implement and adapt our type checker to work with signatures.

```haskell
data VD = Bool Bool | Int Int
```

Figure 5.5:

```haskell
intrinsicSemantics :: String -> [VD] -> VD
intrinsicSemantics "add" [Int a, Int b] = Int (a + b)
intrinsicSemantics "sub" [Int a, Int b] = Int (a - b)
intrinsicSemantics "mult" [Int a, Int b] = Int (a * b)
intrinsicSemantics "div" [Int a, Int 0] = error $ "Division by zero: " ++ show a ++ " /
    0"
intrinsicSemantics "div" [Int a, Int b] = Int (div a b)
intrinsicSemantics "leq" [Int a, Int b] = Bool (a <= b)
```

Figure 5.6:

5.2.1 ADT's

Abstract Data Types are

5.3 Generic Programming

5.3.1 Concepts

Chapter 6

MDCS

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.