INF222 Crashcourse

Procedures - v2023

Sander Wiig

Bergen Language Design Laboratory
Institute for Informatics
University of Bergen

May 15, 2023

UNIVERSITY OF BERGEN
Faculty of Mathematics and Natural Scien

Script and Presentation

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

What is a Programming Language?

A Programming language is ...

- used to tell a computer what to do

What is a Programming Language?

A Programming language is ...

- used to tell a computer what to do
- usually artificial

Grouping Languages by domain

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

- DSLs, small, targeted at specific problems. Internal/embedded vs external

Grouping Languages by domain

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

- DSLs, small, targeted at specific problems. Internal/embedded vs external
- GPLs, large, many uses.

Grouping Languages by domain

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 extendible	Extendable

Figure: Comparison between GPLs and DSLs

Syntax and Semantics

Definition

All languages consist of two parts

- **Syntax** - Defines shape

Syntax is defined by a grammar.
Grammar is not covered in this course :)

Syntax and Semantics

Definition

All languages consist of two parts

- **Syntax** - Defines shape
- **Semantics** - Defines meaning

Syntax is defined by a grammar.

Grammar is not covered in this course :)

Meta Programming

A metaprogram is a program that works on *other* programs.
Compilers and Interpreters are examples of metaprograms

Definition

- **Object Language** - Langage that gets compiled/interpreter
- **Meta Language** - Language used to implement the compiler/interpreter

Sum of Products

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

Examples

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

`Bool` is either `True` or `False`.

The total number of values of type SomeType is $8 + 2 + 1 = 11$.

Sum of Products

```
1   interface SomeType {}
2   class A implements SomeType {
3       boolean a;
4       boolean b;
5       boolean c ;
6   }
7   class B implements SomeType {
8       boolean a;
9   }
10  class C implements SomeType {
11  }
```

Figure: Sum of Products in Java

Questions?

Compiler vs. Interpreters

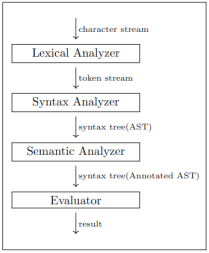


Figure 2.3: Phases of an interpreter

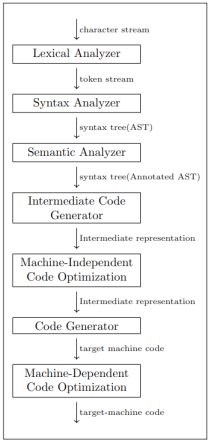


Figure 2.4: Phases of a "normal" compiler

NB! IMPORTANT

Compilers vs Interpreters!

We often end up using the term Compiler to talk about both Compilers and Interpreters, this is because they are very similar.
This course deals exclusively with interpreters so unless stated otherwise assume that we are talking about interpreters.

Expr vs AST

- **Expr**: Expressions are terms that can be evaluated to a value, e.g. `1+2*3`
- **Stmt**: Statements are terms that are executed and result in a change of state. e.g. `var a = 1+2*3`

Lexical Analysis

Lexical Analysis breaks up strings into tokens. Also called a tokenizer.

Example

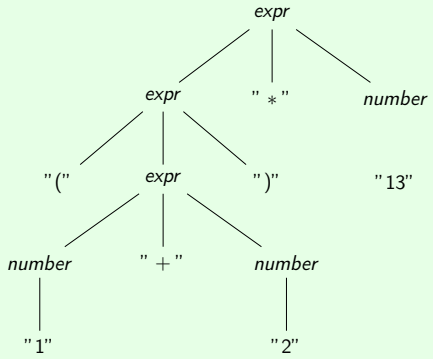
$$(1 + 2) * 13$$

This gets tokenized into

```
["(","1","+",2,")","*","13"]
```

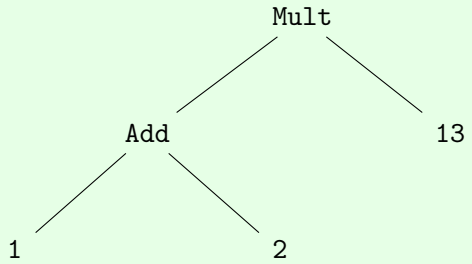
Syntax Analyser

Example



AST

Example



Semantic Analyisis

Semantic analysis lets us find out if the program is wellformed at find bugs at compile time, instead of at runtime. It also annotates the AST with certain information that's necessary for execution.

Wellformedness

For a program to be wellformed, we need to check for the following:

- **Type correctness**: The types of expressions in the program are correct.
- **Scope correctness**: The variables used in the program are declared.
- **Flow correctness**: The program is not stuck in an infinite loop.
- and more...

The first point is checked by a **type checker**.

Type Checking

Example

AST

```
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)
```

Type Checking

Example

Types

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

Type Checking

Example

```
1  typeCheck ::VarDecl->Expr->ExprType
2  typeCheck _ (I _) = Integer
3  typeCheck _ (B _) = Boolean
4
5  typeCheck vars (Var name) =
6                 case lookup name vars of
7                         Just tp -> tp
8                         Nothing -> error $ "No variable could be found named
                              "++name
```

Type Checking

Example

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

Type Checking

Example

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

Type Checking

Example

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

Type Checking

Example

```
1 typeCheck vars (BinOp Or left right) =
2                       case (typeCheck vars left, typeCheck vars right) of
3                              (Boolean, Boolean)->Boolean
4                              _->error "One of the args is not a Boolean"
```

Type Checking

Example

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

Type Checking

Example

```
1 typeCheck vars (BinOp Eq left right) = Boolean
```

Type Checking

Example

```
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
```

Questions?

State

The state is the place where the program stores its variables and data.

State

The state is divided up into two parts:

- **Enviroment**: The place where the program stores its variables. dictionary, where keys are variable names and values, are pointers to locations in the store. The environment also contains the Free Pointer, which points to the next free location in memory.
- **Store**: The place where the program stores its data, aka Memory. The store is usually an array of values. Most often the values are bytes and types often take up multiple slots. Semantic analysis is used to determine the size of the types and to figure out addresses.

Scoping

Scoping is the process of determining where a variable is visible to the program. Scoping is usually divided into three different classes:

- **Runtime Scoping**
- **Static Scoping**
- **Dynamic Scoping**

Scoping

Example

```
1  int x = 10;
2
3  int f()
4  {
5    return x;
6  }
7
8  int g()
9  {
10   int x = 20;
11   return f();
12 }
13
14 int main()
15 {
16   print(g());
17   return 0;
18 }
```

Arrays

Arrays

Arrays are a collection of elements of the same type. They're most often stored as a contiguous block of memory.
Formula for accessing an element in an array:

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

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

Arrays

Example

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

Multidimensional Arrays

Multidimensional arrays are arrays of arrays.

Multidimensional Arrays

There are two main ways of storing multidimensional arrays:

- **Array of Pointers**: Each element in the array is a pointer to another array somewhere else in memory.
- **Contiguous Block**: The entire multidimensional array is stored as a contiguous block of memory. Where each row is stored sequentially.

Multidimensional Arrays

Forumla for Contiguous Block

$$\texttt{bar[i][j]} = \texttt{\&bar} + i * \texttt{length(row)} + j * \texttt{sizeof(T)}$$

where

- `&bar` is the pointer to the array
- `i` is the row index
- `j` is the column index
- `sizeof(T)` is the size of type T in bytes
- `length(row)` is the length of the row

Multidimensional Arrays

Example

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

Records

Records are a collection of elements of different types. Each element is called a field.

Records

Records are stored as a contiguous block of memory. With each field stored consecutively.
To access a field in a record we need to know the offset of the field in the record.

Records

Records

Formula for accessing a field in a record:

$$\texttt{foo.bar} = \texttt{\&foo} + \texttt{offset(bar)}$$

where

- `&foo` is the pointer to the record
- `bar` is the field that we want to access
- `offset(bar)` is the offset of the field `bar` in the record

Records

Example

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

Arrays of Records

Records

Formula for accessing a field of the nth element in an array of records:

$$\texttt{foo[n].bar} = \texttt{\&foo} + n * \texttt{sizeof(T)} + \texttt{offset(bar)} \tag{1}$$

where

- `&foo` is the pointer to the array
- `n` is the index of the element we want to access
- `sizeof(T)` is the size of type `T` in bytes
- `bar` is the field that we want to access
- `offset(bar)` is the offset of the field `bar` in the record

Questions?

What is a Procedure

- Procedures are "programs within programs"
- Procedures have their own environment

Functions vs. Procedures

$$\text{Functions} \neq \text{Procedures} \tag{2}$$

Functions are expressions, procedures are statements.
Very often same implementation.

Anatomy of a Procedure

```
1
2 procedure <procedure_name> (<params>)
3     <procedure code>
```

Params

- **OBS** - "read only"
- **UPD** - "read/write"
- **OUT** - "write only"

Declaration vs. Calling

```
1  program Proc_Example
2  begin
3      procedure swap (upd x: integer, upd y:integer)
4      begin
5          var tmp : integer; //
6          tmp := x;
7          x := y;
8          y := tmp;
9      end
10
11     procedure main ()
12     begin
13         call tmp(4,5);
14     end
15 end
```

Figure: Swap Procedure

Parameter Semantics

Two types of parameter semantics:

- Reference semantics
- Copy semantics

Reference Semantics

- Parameters become aliased to arguments
- Points to same memory address
- Unsafe, but sometimes useful

Running a procedure with reference semantics

1. Get stackframe

Running a procedure with reference semantics

1. Get stackframe
2. Wipe environment

Running a procedure with reference semantics

1. Get stackframe
2. Wipe environment
3. Add parameters to the environment with same address as arg

Running a procedure with reference semantics

1. Get stackframe
2. Wipe environment
3. Add parameters to the environment with same address as arg
4. run the procedure code

Running a procedure with reference semantics

1. Get stackframe
2. Wipe environment
3. Add parameters to the environment with same address as arg
4. run the procedure code
5. restore the environment

Copy Semantics

- Parameters are declared as variables and initialized with args' value
- Safer
- More intuitive behavior
- More complicated to implement

Running a procedure with copy semantics

1. Get stackframe

Running a procedure with copy semantics

1. Get stackframe
2. Get values of args

Running a procedure with copy semantics

1. Get stackframe
2. Get values of args
3. Wipe environment

Running a procedure with copy semantics

1. Get stackframe
2. Get values of args
3. Wipe environment
4. Add parameters to environment

Running a procedure with copy semantics

1. Get stackframe
2. Get values of args
3. Wipe environment
4. Add parameters to environment
5. init those parameters with the arg values

Running a procedure with copy semantics

1. Get stackframe
2. Get values of args
3. Wipe environment
4. Add parameters to environment
5. init those parameters with the arg values
6. run the procedure code

Running a procedure with copy semantics

1. Get stackframe
2. Get values of args
3. Wipe environment
4. Add parameters to environment
5. init those parameters with the arg values
6. run the procedure code
7. get the values of the parameters

Running a procedure with copy semantics

1. Get stackframe
2. Get values of args
3. Wipe environment
4. Add parameters to environment
5. init those parameters with the arg values
6. run the procedure code
7. get the values of the parameters
8. restore the environment

Running a procedure with copy semantics

1. Get stackframe
2. Get values of args
3. Wipe environment
4. Add parameters to environment
5. init those parameters with the arg values
6. run the procedure code
7. get the values of the parameters
8. restore the environment
9. copy the parameter values back to the args

Swap example v2

```
1  procedure GroupSwap (upd x: integer, upd y :integer)
2  begin
3      y := x + y;
4      x := y - x;
5      y := y - x;
6  end;
7
8  procedure SelfSwap();
9  begin
10     var a = 5;
11     call GroupSwap (a, a);
12 end;
```

Reference semantics

```
1  procedure GroupSwap (upd x: integer, upd y :integer)
2  begin
3      y := x + y;
4      x := y - x;
5      y := y - x;
6  end;
7
8  procedure SelfSwap();
9  begin
10     var a = 5;
11     call GroupSwap (a, a); //x =5, y = 5
12 end;
```

Reference semantics

```
1  procedure GroupSwap (upd x: integer, upd y :integer)
2  begin
3      y := x + y;// y = x + y = 5+5 => y = 10
4      x := y - x;
5      y := y - x;
6  end;
7
8  procedure SelfSwap();
9  begin
10     var a = 5;
11     call GroupSwap (a, a); //x =5, y = 5
12 end;
```

Reference semantics

```
1  procedure GroupSwap (upd x: integer, upd y :integer)
2  begin
3      y := x + y;// y = x + y = 5+5 => y = 10
4      x := y - x;// x = y - x = 10 - 5 => x = 5
5      y := y - x;
6  end;
7
8  procedure SelfSwap();
9  begin
10     var a = 5;
11     call GroupSwap (a, a); //x =5, y = 5
12 end;
```

Reference semantics

```
1  procedure GroupSwap (upd x: integer, upd y :integer)
2  begin
3      y := x + y;// y = x + y = 5+5 => y = 10
4      x := y - x;// x = y - x = 10 - 5 => x = 5
5      y := y - x;// y = y - x = 10 - 5 => y = 5
6  end;
7
8  procedure SelfSwap();
9  begin
10     var a = 5;
11     call GroupSwap (a, a); //x =5, y = 5
12 end;
```

Copy semantics

```
1  procedure GroupSwap (upd x: integer, upd y :integer)
2  begin
3      y := x + y;
4      x := y - x;
5      y := y - x;
6  end;
7
8  procedure SelfSwap();
9  begin
10     var a = 5;
11     call GroupSwap (a, a); //x =a, y = a
12 end;
```

Copy semantics

```
1  procedure GroupSwap (upd x: integer, upd y :integer)
2  begin
3      y := x + y;// a = a + a = 5+5 => a = 10
4      x := y - x;
5      y := y - x;
6  end;
7
8  procedure SelfSwap();
9  begin
10     var a = 5;
11     call GroupSwap (a, a); //x =a, y = a
12 end;
```

Copy semantics

```
1  procedure GroupSwap (upd x: integer, upd y :integer)
2  begin
3      y := x + y;// a = a + a = 5+5 => a = 10
4      x := y - x;// a = a - a = 10 - 10 => a = 0
5      y := y - x;
6  end;
7
8  procedure SelfSwap();
9  begin
10     var a = 5;
11     call GroupSwap (a, a); //x =a, y = a
12 end;
```

Copy semantics

```
1  procedure GroupSwap (upd x: integer, upd y :integer)
2  begin
3      y := x + y;// a = a + a = 5+5 => a = 10
4      x := y - x;// a = a - a = 10 - 10 => a = 0
5      y := y - x;// a = a - a = 0 - 0 => a = 0
6  end;
7
8  procedure SelfSwap();
9  begin
10     var a = 5;
11     call GroupSwap (a, a); //x =a, y = a
12 end;
```

Questions?

Signatures

Signatures let us abstract away the implementation of our operations.
Signatures also has many other benefits:

- Makes it easier to reason about our code
- Makes it easier to reuse code

Signatures

Signatures are a way to define a set of operations.

Signature

Formaly a signature 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 $f : s_1, ..., s_n \rightarrow s$, for $s1, \ldots, s_n, s \in S$

!IMPORTANT!

Signatures alone do not implement any semantics, they are just a way to define which operations and types exist, but not what they do.

Algebras

Algebras lets us define the semantics of a signature.

Algebras

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 $(f : s_1, ..., s_k \to s) \in F$

Fun Fact!

It's possible to have multiple algebras for a signature.

Implementing Signatures

Example

Simple AST with vars, constants, and a few expressions

```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)
```

Implementing Signatures

Example

Evaluator for the AST

```
1  eval :: Env -> Expr -> Int
2  eval _ (Lit n) = n
3  eval env (Add e1 e2) = eval env e1 + eval env e2
4  eval env (Sub e1 e2) = eval env e1 - eval env e2
5  eval env (Mult e1 e2) = eval env e1 * eval env e2
6  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
7  eval env (LEQ e1 e2) = if eval env e1 <= eval env e2 then 1 else 0
8  eval env (Var string) = case lookup string env of
9                        Just n -> n
10                       Nothing -> error $ "Variable " ++ string ++ " not found in environment"
```

Implementing Signatures

Example

We modify the AST to only contain constants, variables, and a function call

```
1 type Env valueDomain = [(String, valueDomain)]
2 data Expr valueDomain
3            = Lit valueDomain
4            | Var String
5            | FunCall String [Expr valueDomain]
```

Implementing Signatures

Example

We also edit eval to reflect the changes to the AST, Note the addition of funmod which corresponds to the algebra for our signature.

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

Implementing Signatures

Example

We can now create our signature

```
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")
           ])
```

Implementing Signatures

Example

We create our value domain

```
1 data VD = Bool Bool | Int Int
```

Implementing Signatures

Example

And then we implement the algebra for the signature.

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

Abstract Data Types

Implementing Signatures lets us implement ADTs into our langauge.

Abstract Data Types

Abstract Data Types lets users define a data structure by its operations.
In other words ADT define a signature for a data type. Interfaces in Java are one example of ADTs in a language.

ADTs in Java

Example

Stack ADT in Java

```
1  public interface IStack<T> {
2
3      public void push(T item);
4
5      public T pop();
6
7      public T peek();
8
9  }
```

ADTs in Java

Example

Implementation of Stack

```
1  class Stack<T> implements IStack<T> {
2
3      private List<T> stack;
4      public Stack() {
5          stack = new ArrayList<T>();
6      }
7      public void push(T item) {
8          stack.add(item);
9      }
10
11     public T pop() {
12         return stack.remove(stack.size()-1);
13     }
14
15     public T peek() {
16         return stack.get(stack.size()-1);
17     }
18
19 }
```

ADTs in Java

Example

Since IStack doesn't define implementation we can choose whatever semantic. In this case Stacks and Queues only differ in the behviour of pop.

```
1  public class Queue<T> implements IStack<T>{
2      private List<T> queue;
3      public Queue() {
4          queue = new ArrayList<T>();
5      }
6
7      public void push(T item) {
8          queue.add(item);
9      }
10
11     public T pop() {
12         return queue.remove(0);
13     }
14
15     public T peek() {
16         return queue.get(0);
17     }
18 }
```

Questions?

Assertions

Assertions are statements that check if some predicate holds. Assertions help make sure that a program is always in a valid state. If an assertion fails, a program crash is triggered.

Assertion Types

There is usually a couple of categories of assertions.

- **Pre-Condition**: assertions that must hold before a function or procedure is called. Ensures that call is made with valid args.
- **Post Conditions**: assertions that must hold after a procedure/function. e.g. return type, valid result, etc.
- **Invariants**: assertions that must always hold, e.g some var is always above a certain value.

Lykke til på eksamen!

Takk for meg :)