Programming Language Design Paradigms - 5CCS2PLD

Programming Languages **3**
- Programming Paradigms 4
- Definition of a programming language 4
- Computer Architecture 4
- Implementing a Programming Language 5
  - Compilation 5
  - Execution 5
  - Interpretation 6
  - Hybrid Implementation 6
- Syntax 6
- Grammar 7
  - Remarks 8
- Semantics 8
- Informal vs Formal Semantics 8
- Styles of Semantics 8
- Operational Semantics 9
- Transition Systems 10
- Abstract Machines 10

Imperative Languages **10**
- Variables 10
- Assignment 12
- Control Statements 12
- Structuring the Program 12
- Formal Description 12
  - Abstract Syntax of SIMP 13
- An Abstract Machine for SIMP 14
- Configurations 14
- Semantics of SIMP Programs 15
  - Transition Rules for Expressions 15
  - Transition Rules for Commands 16
  - Abstract Machine 17
  - Structural Approach to Operational Semantics 17

Induction **17**
- Principle of Mathematical Induction 17
  - Structural Induction 17
  - Proving properties for a list 18

Proving integer properties for SIMP 18
Proving properties for leads of trees 18
Inductive Definitions 18
Examples of Inductive Definitions 20
Principle of Rule Induction 20
Structural Operational Semantics (SOS) for SIMP 20
Reduction Semantics for SIMP (Small-step semantics) 21
Comparison with Abstract Machine 23
Big-Step Semantics 23
Transition System 23
Adding Variable Declarations to SIMP 25

Functional Programming 26
Haskell 26
Properties of Evaluation 27
Strategies 27
Functional Values 28
Syntax of Function Definitions 28
Function Application 29
Recursive Definitions 29
Evaluation of Recursive Functions 29
Function Definitions by Pattern Matching 30
Local Definitions 30
Evaluation of Local Definitions 30
Arithmetic Functions 31
First-Class functions 31
Curried Functions and Partial Applications 32
Types 33
Advantages of Static Typing 33
Types in Haskell 34
Polymorphism 34
Polymorphic Types 34
Example 34
Overloading aka ad-hoc polymorphism 35
Type Classes 35
Type Inference 36
Lists 36
Functions on Lists 37
User Defined types 37
Induction (proving properties of our programs) 38
Semantics 39
Syntax of SFUN 39

Variables 40
Programs in SFUN 41
Operations Semantics of SFUN 41
Examples call by value 42
Proof of unicity of normal forms 43
Call-by-name evaluation of SFUN 44
Types for SFUN 45
Typing SFUN programs 48
Example of typing SFUN Programs 48
Proving properties of SFUN programs 50

Logic Programming **50**
Domain of computation 50
Term 50
Literals 51
Clauses 51
Substitutions 52
Prolog Syntax 53
Clauses 53
Built-in predicates 53
Lists 55
Semantics 57
Unification 57
Unification Algorithm 57
Rules 58
More on the unification algorithm 58
The Principle of Resolution 60
SLD Resolution 61
Properties of SLD 61
Computing resolvents with SLD-resolution 62
SLD-resolution trees 63
SLD-resolution in Prolog 63
Failure Example 65
Back-tracking 65
Example of non-termination 66

Programming Languages

Programming languages: tools for writing software. They are a result of an evolution process that is likely to continue in the future.
We study them to:

- Choose the most suitable for each application
- Increase our ability to learn new languages

- Be able to design new languages, user-interfaces etc.

Languages are used in several phases in the software development process. A design method is a guideline for producing design like *functional* or *object-oriented*. Choosing a method which is not compatible with the language the effort increases.

Programming Paradigms

A language may enforce a style of programming called a **programming paradigm.**

- **Imperative Languages :** programs are decomposed into **computation steps,** reflecting the computer architecture. (C, Python and Java). It reflects how computers are built using processes.
- **Functional Languages:** they are based on the mathematical notion of a **function**. "A family of functions that are built together to produce a result". Some kind of expression that produces a result. (Lisp, Haskell, CaML, Scheme, Scala). Based on what should be computed, not how it should be computed.
  Note: C uses the word function a lot but **is not** functional.
- **Logic Languages:** programs describe a problem rather than defining an algorithmic implementation. You put your knowledge in a domain and then you ask questions. (Prolog)
- **Object-oriented Languages:** this is a feature more than a paradigm and could be put into any other. There are OO imperative languages (Java) or OO functional languages (CaML). It introduces the idea of objects, classes and inheritance.

Definition of a programming language

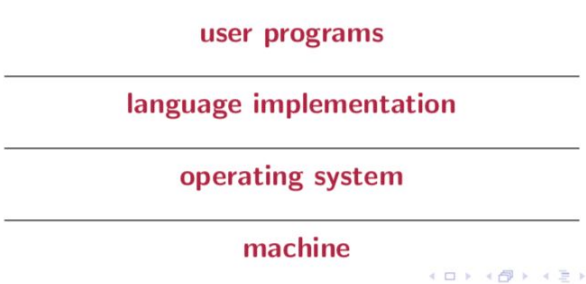
All programming languages have three main components:

1. **Syntax:** defines the rules and the words that are used to build expressions, commands and declarations to put together a program.
2. **Semantics:** give the meaning to programs, decides what to do when executed. Decides what happens when you write 'if'. In this case the semantics evaluate this to an condition expression.
3. **Implementation:** a software system that can read a program and execute it in a machine, plus a set of tools (compiler, editor, debugger, etc)

Computer Architecture

The three main components of a computer are:

- Memory
- Registers
- Processor, with a set of machine instructions for arithmetic and logic operations (machine language)



- The **operating system** gives a higher-level way to interact with the machine (processor).

- Language implementations (compilers) are built on top of these and make use of them. Remember that compilers normally ask you what operating system you are using.
- At a user program level you do not worry about the operating system or the language implementation (compiler). When you write a java program you don't program differently for OSX or for Windows.

Implementing a Programming Language

Languages can be implemented by:

- **Compiling** transforms programs/code into machine language. It optimizes this code to make a better implementation.
- **Interpreting** transforms programs/code into machine language. It doesn't optimize code as much but gives better error methods.
- **Hybrid Method** which combines compilation and interpretation (some hybrid methods can compile to a virtual machine and then combine it with an interpreter, like Java does).

Compilation

The task of the compiler is to read the source code and translate it into something that can be executed in your computer. Main phases of compilation:

1. **Lexical Analyzer (LEX):** identifies which characters form words (tokens) that represent key words, comments, whitespace. It outputs tokens which are interpreted by the syntax analyzer.
2. **Syntax Analyzer (Parsing):** checks that the syntax is correct (e.g. that we have closed an open bracket.) The output of a parser is a representation of the code in a data structure (tree) which is easy to use for the next stage. The tree outputted is called the Abstract Syntax Tree.
3. **Type and Semantics Analysis:** the tree outputted from the parser makes it easier by the type checker to check that the types are correct. It tries to optimize this tree to make it better. If we have a good compiler and have created a boolean condition which always results to true, it will make the condition redundant and it will only keep the part of the tree which is necessary albeit the one which is true. The outputted optimised tree will be used to generate code.
4. **Machine Code Generation**

Code ----> **LEX** -- tokens --> **Parser** ---- abstract syntax tree AST ----> **Type checker / optimiser** ---- optimized AST ----> Code Generator (for OS and processor).

Execution

The execution of a machine code occurs in a process called **fetch-execute cycle.** Each machine instruction to be executed is moved from memory (where programs reside) to the processor.

Fetch-execute cycle:

- Initialise program counter (in register of machine) - pointer to where we should start running code.
- Repeat forever:
    - Fetch the instruction pointed by the program counter
    - Increment the program counter
    - Decode and execute instruction

When all instructions are executed the control returns to the operating system (more complex when the system is running more than one process).

Interpretation

Tries to mimic the fetch-execute cycle with the program instructions. In this case there is no translation. The interpreter produces a virtual machine for the language.

- Since it has not compiled albeit it's not created an AST and optimized it, the running of the code will be slower.
- Since it runs code line by line, it will be easier to find an error. It's more difficult for a compiler to find error since when code is optimized it's more difficult to track where the error originated.
- Simpler languages are usually interpreted (Unix Shell Scripts, LISP)
- It's better to use interpreters when debugging or when learning the language.

Hybrid Implementation

Some languages are interpreted by compilation to an immediate language instead of machine language. This immediate language is then interpreted.

Instead of generating code for a processor, we invent a processor with different instructions which are sufficiently low level that look like the ones of the processor.

+ Portability. Programs can be executed in any machine that has an interpreter for the language. (Java Bytecode)

Some languages have both an interpreted and compiled version, the **interpreter** is used to develop and debug programs which are later compiled to increase execution speed.

Front end of the compiler: deals with the code.
Back end of the compiler: deals with the execution of the code.

Syntax

The syntax is concerned with the form of programs (how they are written).
It's given by:

- An *alphabet:* the set of characters can be used (concrete syntax).
- A *set of rules:* tell us how to form expressions, commands, declarations, etc. (abstract syntax)

We distinguish between:

- **Concrete syntax:** describes which chains of characters are well-formed programs (what programmers use, user manuals, symbols and words you can write).
- **Abstract syntax:** describes the syntax trees, to which a semantics is associated.

Grammar

To specify the syntax of a language we use grammars, given by:

- An alphabet $V = V_T \cup V_{NT}$.
- Rules
- Initial Symbol.

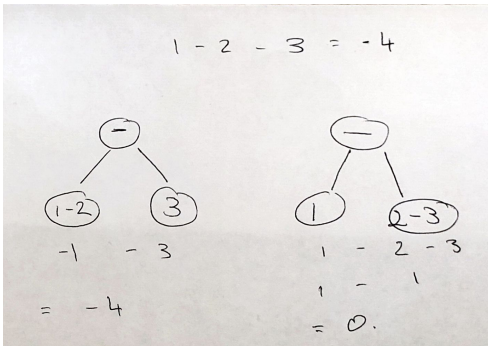
Alphabet tells us which characters we can use. The union of terminal and non-terminals.

- Terminal: symbols programmers can write (0,1,2, +, x).
- NonTerminal: used to explain how to write expressions (Exp, Op, Num).

Concrete syntax:

$$
\begin{array}{lll}
Exp & ::= & Num \mid Exp\ Op\ Exp \\
Op & ::= & + \mid - \mid * \mid div \\
Num & ::= & Digit \mid Digit\ Num \\
Digit & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

Problem:



This grammar is ambiguous, you can generate 1 - 2 - 3 but in different ways. We can generate it by:

This is because when you us two different compilers they might produce different results, which results in a disaster. This is why compilers avoid ambiguity by giving priorities and association. Or the use of brackets.

Abstract syntax:

$$
\begin{array}{lll}
e & ::= & n \mid op(e, e) \\
op & ::= & + \mid - \mid * \mid div
\end{array}
$$

This grammas defines trees, not strings.

***Question Example**: Show that 1 − 2 − 3 and 1 + 2 ∗ 3 are valid expressions in this language.*

Exp → ExpOpExp → NumOpExp → 1OpExp → 1 − Exp → 1 − ExpOpExp → 1 − NumOpExp→1 − 2OpExp→1 − 2 − Exp→1 − 2 − Num→1 − 2 − 3

- Exp is the initial symbol and it's non terminal (VNT)

- Terminal symbols are those which do not have definitions (1,2,3...)

Syntax identifiers: sequences of letters, numbers and quotes which can be used in the programming language.

Remarks

- Abstract syntax describes program representation
- Abstract syntax is not ambiguous
- We will always work with the abstract syntax of the language and study the semantics of the main constructs of programming languages.

Semantics

Semantics: define how a program behaves when its executed on a computer.

- **Static semantics (often referred to typing):** detect that programs are syntactically correct but will give errors in the execution.
    - E.g. 1 + ‘a’ will give an error because they are of different types.
- **Dynamic semantics (or just semantics):** specify the meanings of programs.

NB, *A static property is one that we can study without having to run the program.*

Informal vs Formal Semantics

Informal definitions are often in english, in manuals and are often imprecise and incomplete. Formal semantics are important for:

- **The implementation of the language:** behavior of each construct is specified. Abstraction of the execution process is independent of the machine.
- **Programmers:** provides tools or techniques to reason about programs and prove its properties (we can test a language’s properties).
- **Language designers:** allows to detect ambiguities in the constructs and suggests new constructs.

Note, formal descriptions can be complex, so only a part of the language is formally defined.

Styles of Semantics

Denotational Semantics: the meaning of expressions is given in an abstract, mathematical way. They describe the effect of each construct.

Axiomatic Semantics: use axioms* and deduction rules in a specific logic. Predicates or assertions are given before and after each construct, describing the constraints on program variables before and after the execution of the statement (preconditions, postcondition).

Operational Semantics: (The one we will use) the steps of computation that occur in our program. (What happens when we run a program from an initial state). The behaviour of the program during execution can be describe using a transition system (abstract machine).

***axioms:** a statement or proposition on which an abstractly defined structure is based.

Operational Semantics

Operational semantics is a transition system.

A *transition system* is specified by

- A set *Config* of *configurations* or *states*,
- A binary relation $\rightarrow \subseteq \mathit{Config} \times \mathit{Config}$, called *transition relation*.
  We use the notation $c \rightarrow c'$ (infix) to indicate that c, c' are related by $\rightarrow$.

Notation:
$c \rightarrow c'$ denotes a *transition* from c to c' (change of state).
$c \rightarrow^* c'$ is the reflexive transitive closure of $\rightarrow$. In other words,
$c \rightarrow^* c'$ holds iff there is a sequence of transitions:

$$c \rightarrow c_1 \rightarrow \ldots \rightarrow c_n = c'$$

where $n \geq 0$.

- Arrow means binary relation between configurations (from c, there's a transition to c')
- When there are multiple steps between two configurations we use a *.
- X represents the cartesian product
- * is similar to * in Regex, it means there are 0 or N number of steps between the change of state.

Transition Systems

We will distinguish an *initial* and *final* (also called *terminal*) subset of configurations, written I, T. For all $c \in T$ there is no c' such that $c \to c'$.
The idea is that a sequence of transitions from $i \in I$ to $t \in T$ represents a run of the program.

A transition system is **deterministic** if

$$\text{for all } c, c_1, c_2 : c \to c_1 \text{ and } c \to c_2 \text{ implies } c_1 = c_2$$

Deterministic: (The one we will use most) when a system is at a particular state there is only ONE other possible state, not several. ***To remember,*** *it's determined where to go next.*

E.g. $\text{for all } c, c_1, c_2 : c \to c_1 \text{ and } c \to c_2 \text{ implies } c_1 = c_2$

Abstract Machines

Abstract machine: a transition system that specifies an interpreter for a programing language. Useful for implementing a programming language since they describe the execution of each com1man/d.

Imperative Languages

Informal Description: one of the main components of the computer is memory, storing instructions and data. In imperative languages, variables represent memory cells.

Variables

Properties:

- **Name**: usually a string of characters with some constraints.
- **Type**: a range of values allowed for the variable and operations available.
  - \+ Having types helps detecting programming errors at an early stage before the machine code is created (errors are detected by the type and semantics analysis) making the debugging process faster and easier.
  - \+ Types help in the design process (simple form of specification)
  - \+ A program that passes the type controls isn't guaranteed to be correct, but in a strongly typed language, it's free of runtime type errors.
  - \- If the program doesn't have a type inference system then writing programs is more difficult since we have to explicitly specify the type.
- **Address:** memory address to which the variable is associated (also called l-value)

  - NB, in some languages we can associate more than one name with the same address.
- **Value:** the contents of the memory associated with the variable (called r-value).
- **Lifetime:** *the time between allocation and deallocation. Aka* time during which the variable is allocated in a specific memory location. Some variables may be available but not for the whole duration of the program.
  Classification of variables according to lifetime:
  - **Static variables:** variables which are available during the whole duration of the program. Bound to a memory location before program execution begins and until it ends.
    + Efficiency, there is no allocation, deallocation time during program execution thus we can address them directly (no runtime overhead).
    - Space cannot be reused during runtimes (recursive programs need dynamically allocated memory).

    NB, don't confuse with static variables in java (in the Java Context, these are the ones which are allocated to memory at the start).
  - **Stack dynamic variables:** allocated when the declaration of the variable is processed at run time (e.g. local variables of recursive calls).
  - **Explicit heap-dynamic variables:** stored in the heap, created by the code (they are called explicit because we explicitly wrote in the code that the variable was created). They are nameless memory cells that are allocated and deallocated by program instructions, using pointers (e.g using new in Java, delete in C++).
  - **Implicit heap-dynamic variables:** variables are bound to storage only when they are assigned values (e.g. strings and arrays in JavaScript).
    + Flexibility
    - Difficult error detection (you have less control over what occurs with the variable).
- **Scope:** the range of instruction in which the variable is visible (can be used). It has to do with where the variable is declared.
  - **Local variable:** variables declared in the block of program that is executed
  - **Global variable**: variables that are not local (normally available within the whole program).

  Variables can use static or dynamic scoping rules. ***Static*** means we can know what happens to the variable without running the code (by looking at it) but if it's ***dynamic*** you have to think of how the program is ran.

E.g. Static Scope in Pascal:

```
program main;
var x :  real;
procedure P1;
var x :  integer;                    local variable
begin
...  x ...
end;
procedure P2;
begin
...  x ...                          global variable
end;
begin main
...  x ...                          global variable
end.
```

Static Scope: can be determined before the execution of the program (at compile time).

A static scoped variable will search in the current block (local block) then go to the parent block until reaching the main. If it's not in the main it will give an error.

Dynamic Scope: the scope of the variable depends on the calling sequence of subprograms instead of their declaration thus it can only be determined at runtime.

+ When the parameters that need to be passed are the variables defined in the caller, then there is no need of parameter-passing, those variables are implicitly visible in the called subprogram.

In the example before, with static scope rules in P2 we will use var x: real since there is no declaration within P2. With dynamic scope the ref to x in P2 may refer to P1 or in main depending on who called P2

Assignment

Variable-name = expression
The value of the expression is stored in the variable until a new value is given. *In typed languages the expression and the variable must have compatible types*. (Note that in some languages assignment can be written i++ or i--).

NOTE: The = sign should not be confused with equivalence, sometimes := can be used for assignment.

Control Statements

Programs in imperative languages are sequences of instructions. The next will be called unless a control statement is used.
Main control statements in imperative languages are:

- **Selection constructs (if-then-else)** - selecting different paths
- **Looping constructs (while)** - allow you to repeat an execution of a block of commands.
- **Branching instructions**

Structuring the Program

Block: a group of statements delimited by keywords or separators (e.g. a method, brackets). A block may contain commands, declarations and scope.
Subprogram: a generic name for a named block. These can be invoked and executed after these. A subprogram may have parameters called *formal parameters* that change the output of the subprograms.

Formal Description

An abstract machine is a transition system, which specifies an interpreter for the programming language.

Creating a small simple definition of a programming language called SIMP

Abstract Syntax of SIMP

Programs

$$P ::= C \mid E \mid B$$

P: Program; C: command; E: expression; B: boolean expression
:= represents assignment, *l* is a reference in memory, !*l* is a value in memory.

Commands

$$C ::= skip \mid l := E \mid C; C \mid if\ B\ then\ C\ else\ C \mid while\ B\ do\ C$$

Integer Expressions

$$E ::= !l \mid n \mid E\ op\ E \qquad op ::= + \mid - \mid * \mid /$$

where

- $n \in Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ (integers)
- $l \in L = \{l_0, l_1, \ldots\}$ (locations or variables). The expression $!l$ denotes the value stored in l.

Boolean Expressions

$$B ::= True \mid False \mid E\ bop\ E \mid \neg B \mid B \wedge B$$

$$bop ::= > \mid < \mid =$$

Commands create trees:
If B then C else C creates a tree with 3 sub-trees. One for the boolean condition, one for the true condition and one for the false condition.
Exclamation mark tells us that we have to read for the address of the variable and read its content (it's a dereference)

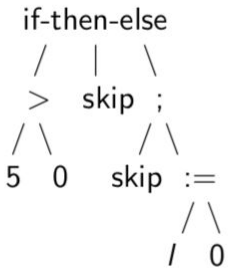
Swapping the content of variables looks like this:

z := !x ;
x := !y ;
y := !z ;

The grammars specify the **abstract** syntax of SIMP, therefore rules specify labelled trees rather than strings. The leaf nodes of the tree are labelled by elements of $Z \cup \{True, False\} \cup L \cup \{skip\}$ while the non-leaf nodes are labelled by operators and commands.

Example:

The abstract syntax tree



is written in a textual notation as:

$$\text{if } 5 > 0 \text{ then skip else (skip; } l := 0)$$

using brackets where there is ambiguity.

An Abstract Machine for SIMP

We define an abstract machine with:

1. A control stack **c** - this is where we put the commands.
2. An auxiliary stack **r** (also called results stack) - a little bit of memory (register) where we can leave intermediate results.
3. **Memory** or **store**, modelled by a partial function m, dom(m) denotes the locations where m is defined.

To tell the machine we want the value at l we will:

Notation: $m[l \mapsto n]$ is the function that maps each $l' \neq l$ to the value $m(l')$, and l to the value n.
More precisely:
$m[l \mapsto n](l) = n$
$m[l \mapsto n](l') = m(l') \quad \text{if } l \neq l'$

The abstract machine is defined by a set of **configurations** and a set of **transition rules**.

Configurations

A configuration is a triple: <Control (Stack), Results (Stack), Memory (Array)>
When you start the machine you start it with the program you want to run on the Control Stack.

$$c ::= nil \mid i \cdot c$$
$$i ::= P \mid op \mid \neg \mid \wedge \mid bop \mid := \mid if \mid while$$
$$r ::= nil \mid P \cdot r \mid I \cdot r$$

Definition of the control and results stack:
Where P, bop and op are the non-terminals used in the rules defining SIMP programs

The model of execution will define the transitions between the initial and final configuration.

Initial Configurations: $\langle C \cdot nil, nil, m \rangle$
Final Configurations: $\langle nil, nil, m \rangle$

NB, that nil means that the Stack is empty. NB2, C dot nil means that there is a C in the top of the stack followed by nil.

Semantics of SIMP Programs

For a sequence of transitions

$$(C \cdot nil, nil, m) \longrightarrow^* (nil, nil, m')$$

A program C executed in the state m terminates successfully producing the state m'

If there is a sequence of transitions

$$(E \cdot c, r, m) \longrightarrow^* (c, v \cdot r, m')$$

$$(B \cdot c, r, m) \longrightarrow^* (c, v \cdot r, m')$$

then we say that **the value of the expression E (resp. B) in the state m is v.**

Transition Rules for Expressions

//Download chapter 4 book or try to implement yourself.

$$
\begin{array}{rcl}
\langle n \cdot c, r, m\rangle & \rightarrow & \langle c, n \cdot r, m\rangle \\
\langle b \cdot c, r, m\rangle & \rightarrow & \langle c, b \cdot r, m\rangle \\
\langle \neg B \cdot c, r, m\rangle & \rightarrow & \langle B \cdot \neg \cdot c, r, m\rangle \\
\langle (B_1 \ \wedge \ B_2) \cdot c, r, m\rangle & \rightarrow & \langle B_1 \cdot B_2 \cdot \wedge \cdot c, r, m\rangle \\
\langle \neg \cdot c, b \cdot r, m\rangle & \rightarrow & \langle c, b' \cdot r, m\rangle \quad \text{if } b' = \mathit{not}\ b \\
\langle \wedge \cdot c, b_2 \cdot b_1 \cdot r, m\rangle & \rightarrow & \langle c, b \cdot r, m\rangle \ \text{ if } b_1 \ \mathit{and}\ b_2 = b \\
\langle (E_1 \ \mathit{op}\ E_2) \cdot c, r, m\rangle & \rightarrow & \langle E_1 \cdot E_2 \cdot \mathit{op} \cdot c, r, m\rangle \\
\langle (E_1 \ \mathit{bop}\ E_2) \cdot c, r, m\rangle & \rightarrow & \langle E_1 \cdot E_2 \cdot \mathit{bop} \cdot c, r, m\rangle \\
\langle \mathit{op} \cdot c, n_2 \cdot n_1 \cdot r, m\rangle & \rightarrow & \langle c, n \cdot r, m\rangle \quad \text{if } n_1 \ \mathit{op}\ n_2 = n \\
\langle \mathit{bop} \cdot c, n_2 \cdot n_1 \cdot r, m\rangle & \rightarrow & \langle c, b \cdot r, m\rangle \ \text{ if } n_1 \ \mathit{bop}\ n_2 = b \\
\langle !l \cdot c, r, m\rangle & \rightarrow & \langle c, n \cdot r, m\rangle \quad \text{if } m(l) = n
\end{array}
$$

c : control stack
r : result stack
m : memory
Line 1: we load n (which is a number token) we get the value which is at the control stack and moves it to re results stack.

Line 2: moves a boolean value from the control stack to the result stack.

Line 3: Not and then push B transitions to B negation of the control stack, when we have a negation on top of the control stack we negate the negation and negate the result stack (as seen in line 5)

Transition Rules for Commands

$$
\begin{array}{rcl}
\langle \mathit{skip} \cdot c, r, m\rangle & \rightarrow & \langle c, r, m\rangle \\
\langle (l := E) \cdot c, r, m\rangle & \rightarrow & \langle E \cdot := \cdot c, l \cdot r, m\rangle \\
\langle := \cdot c, n \cdot l \cdot r, m\rangle & \rightarrow & \langle c, r, m[l \mapsto n]\rangle \\
\langle (C_1; C_2) \cdot c, r, m\rangle & \rightarrow & \langle C_1 \cdot C_2 \cdot c, r, m\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle (\mathit{if}\ B\ \mathit{then}\ C_1\ \mathit{else}\ C_2) \cdot c, r, m\rangle & \rightarrow & \langle B \cdot \mathit{if} \cdot c, C_1 \cdot C_2 \cdot r, m\rangle \\
\langle \mathit{if} \cdot c, \mathit{True} \cdot C_1 \cdot C_2 \cdot r, m\rangle & \rightarrow & \langle C_1 \cdot c, r, m\rangle \\
\langle \mathit{if} \cdot c, \mathit{False} \cdot C_1 \cdot C_2 \cdot r, m\rangle & \rightarrow & \langle C_2 \cdot c, r, m\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle (\mathit{while}\ B\ \mathit{do}\ C) \cdot c, r, m\rangle & \rightarrow & \langle B \cdot \mathit{while} \cdot c, B \cdot C \cdot r, m\rangle \\
\langle \mathit{while} \cdot c, \mathit{True} \cdot B \cdot C \cdot r, m\rangle & \rightarrow & \langle C \cdot (\mathit{while}\ B\ \mathit{do}\ C) \cdot c, r, m\rangle \\
\langle \mathit{while} \cdot c, \mathit{False} \cdot B \cdot C \cdot r, m\rangle & \rightarrow & \langle c, r, m\rangle
\end{array}
$$

Abstract Machine

+ explains the execution of the commands step by step, which is useful if we have to implement the language.
+ It is not very intuitive. Many transitions are doing just phrase analysis, only a few really perform computations.

END of PLD3

Structural Approach to Operational Semantics

The transitions for a compound statement should be defined in terms of the transitions for its constituent sub-statements, in other words the definition should be **inductive**

Induction

A method to write definitions of things. You have to prove the base case, then you provide a way to prove for the one after the base case. If we prove for n, we prove for all cases.

Principle of Mathematical Induction

For any property P(n) of natural numbers

to prove $\forall n \in \mathcal{N}.P(n)$ it is sufficient to show:

- **Base Case:** $P(0)$
- **Induction Step:** $\forall n \in \mathcal{N}.P(n) \Rightarrow P(n+1)$

Example:

We prove that for all natural numbers n:

$$\sum_{i=1}^{n}(2i-1) = n^2$$

Base: $0 = 0^2$

Induction Step: Assume $\sum_{i=1}^{n}(2i-1) = n^2$.

$$\sum_{i=1}^{n+1}(2i-1) = \sum_{i=1}^{n}(2i-1) + 2(n+1) - 1 = n^2 + 2n + 1 = (n+1)^2$$

Structural Induction

We have to adapt induction to data structures (ie lists)

Proving properties for a list

We make an empty list nil and a non-empty list by cons(h,l)

- Cons is a constructor of a list, what we use t o add to the list.
- where h is the head and i the tail. cons(h,l) puts l as the head of the list.

To prove that a property P holds for every list, it's sufficient to prove that:

- **Base Case:** $P(nil)$,
- **Induction Step:** $P(l) \Rightarrow P(cons(h, l))$ for all h, l.

If the program is true for P(l) can we prove that it's true for all l. If we can do so we can conclude that our model works for any list.

Proving integer properties for SIMP

Proving a property P holds for all integer expressions in SIMP:

1. Base case:
    - Prove P(n) for all integers.
    - Prove(!*l*) for all locations of l (variable values).

2. Inductive Step:
    - For all integer expressions E, E' and operators op:
      Prove that P(E) and P(E') implies P(E op E')

Proving properties for leads of trees

With finite labels trees in order to prove P it is sufficient to show:

- **Base Case:** $P(l)$ for all leaf nodes l
- **Induction Step:**
  for each tree constructor c (with $n \geq 1$ arguments):
  $\forall t_1, \ldots, t_n . P(t_1) \wedge \ldots \wedge P(t_n) \Rightarrow P(c(t_1, \ldots, t_n))$

In the case of trees, if we prove that the property holds for every subtree then we prove that the property will work for every tree.

Inductive Definitions

NB, in natural numbers you can have a base case which is 0 but in a program you can have certain configurations put in place by default.

We can also use induction to define subsets of a given set T. Example: to check that a syntax tree is correct of 3 + 1, we split the logic into: I can build 1 (axiom), I can build 3 (axiom) and I can build 3 + 1. We can do this using a derivation tree.

We will write inductive definitions using axioms (which represent the base case) and rules (repetitions of the induction step).

Definition:

An axiom is an element of T.
A rule is a pair (H, c) where

- H is a non-empty subset of T, called the **hypotheses** of the rule
- c is an element of T, called the **conclusion** of the rule.

The subset I of T inductively defined by a collection of axioms A and rules R consists of those $t \in T$ such that

- $t \in A$, or
- there are $t_1, \ldots, t_n \in I$ and a rule (H, c) such that $H = \{t_1, \ldots, t_n\}$ and $t = c$.

^ explanation we have a set of axioms and rules from which we have created a subset *I* of *T*. We call each element of *I, t.* Thus *t* is an element of T such that:

- t is an element of the axioms set (it's an axiom)
- There are elements t1,t2,t…,tn which are part of of the inductively defined set *I* ***and*** a rule (H,c) such that H = {t1,t2,...,tn} and t = c.

To show that an element t of T is in *I* it's sufficient to **show that t is an axiom, or that there is a proof.**

$$\begin{array}{ccccc} & & t & & \\ & / & \ldots & \backslash & \\ & t_1 & & t_n & \\ / & \backslash & & / & \backslash \\ t_{11} \ldots t_{1m_1} & & & t_{n1} \ldots t_{nm_n} & \\ \vdots & & & \vdots & \end{array}$$

where the leaves are axioms and for each non-leaf node t_i there is a rule $(\{t_{i1}, \ldots, t_{im_i}\}, t_i)$.
This kind of proof is usually written:

$$\dfrac{\dfrac{\vdots}{t_{11}} \quad \ldots \quad \dfrac{\vdots}{t_{1m_1}}}{t_1} \qquad \ldots \qquad \dfrac{\vdots}{t_n}$$
$$\dfrac{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}{t}$$

Examples of Inductive Definitions

1. **Natural Numbers:**
   Axiom:

   $$0$$

   Rule:

   $$\frac{n}{n+1}$$

   H
   ---
   c
2. **Evaluation relation for integer expressions in SIMP:**
   Notation:
   $(E, m) \Downarrow n$ means that E evaluates to n in the state m.
   Axioms:
   $(n, m) \Downarrow n$ for all integer numbers n
   $(!l, m) \Downarrow n$ if $l \in dom(m)$ and $m(l) = n$
   Rule:

   $$\frac{(E_1, m) \Downarrow n_1 \quad (E_2, m) \Downarrow n_2}{(E_1 \; op \; E_2, m) \Downarrow n} \text{ if } n = n_1 \; op \; n_2$$

   H
   ---
   c

1. If n is in the rules. Then n + 1 is also.

These rules give us the same information as the transitions but we didn't have to stipulate other info like what the stacks are and that we have to push and pop stuff.

Principle of Rule Induction

Let *I* be a set defined by induction with axioms and rules (A,R).
To show that P(i) holds for all *i* in *I*, it's sufficient to prove:

- *Base Case:* $\forall a \in A. P(a)$
- *Induction Step:*
  $\forall(\{h_1, \ldots h_n\}, c) \in R.\ P(h_1) \wedge \ldots \wedge P(h_n) \Rightarrow P(c).$

- TBC

Structural Operational Semantics (SOS) for SIMP

SOS of SIMP is an alternative to the abstract machine. There are two styles.
Small-step semantics: defined using *reduction* relation.

- *Small steps that a machine can perform.*
- This is **deterministic:** meaning the result (sometimes called evaluation sequence) for each expression or program is unique. The sequence of events may be finite or infinite but if the result exists it will be unique (proven by induction)

Big-step semantics: defined using an *evaluation* relation.

Reduction Semantics for SIMP (Small-step semantics)

We define a transition system with configurations

$$\langle P, s\rangle$$

where P is a SIMP program and s is a store (memory) represented by a partial function from locations to integers.

Notation: $s[l \mapsto n]$ denotes the function s' that coincides with s except that it associates to l the value n. More precisely:
$s[l \mapsto n](l) = n$
$s[l \mapsto n](l') = s(l')$ if $l \neq l'$
The transition relation is inductively defined by the axioms and rules:

Small-Step Semantics For Expressions:

$$\frac{}{\langle !l, s\rangle \to \langle n, s\rangle,\ \text{if } s(l) = n}\ (\text{var}) \qquad \frac{}{\langle n_1\ op\ n_2, s\rangle \to \langle n, s\rangle,\ \text{if } n = (n_1\ op\ n_2)}\ (\text{op})$$

$$\frac{}{\langle n_1\ bop\ n_2, s\rangle \to \langle b, s\rangle,\ \text{if } b = (n_1\ bop\ n_2)}\ (\text{bop})$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 op E_2, s\rangle \to \langle E_1' op E_2, s'\rangle}\ (\text{op}_\text{L}) \qquad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle E_1 op E_2, s\rangle \to \langle E_1 op E_2', s'\rangle}\ (\text{op}_\text{R})$$

$$\frac{\langle E_1, s\rangle \to \langle E_1', s'\rangle}{\langle E_1 bop E_2, s\rangle \to \langle E_1' bop E_2, s'\rangle}\ (\text{bop}_\text{L}) \qquad \frac{\langle E_2, s\rangle \to \langle E_2', s'\rangle}{\langle E_1 bop E_2, s\rangle \to \langle E_1 bop E_2', s'\rangle}\ (\text{bop}_\text{R})$$

$$\frac{}{\langle b_1 \wedge b_2, s\rangle \to \langle b, s\rangle,\ \text{if } b = (b_1\ and\ b_2)}\ (\text{and})$$

$$\frac{}{\langle \neg b, s\rangle \to \langle b', s\rangle,\ \text{if } b' = not\ b}\ (\text{not}) \qquad \frac{\langle B_1, s\rangle \to \langle B_1', s'\rangle}{\langle \neg B_1, s\rangle \to \langle \neg B_1', s'\rangle}\ (\text{notArg})$$

$$\frac{\langle B_1, s\rangle \to \langle B_1', s'\rangle}{\langle B_1 \wedge B_2, s\rangle \to \langle B_1' \wedge B_2, s'\rangle}\ (\text{and}_\text{L}) \qquad \frac{\langle B_2, s\rangle \to \langle B_2', s'\rangle}{\langle B_1 \wedge B_2, s\rangle \to \langle B_1 \wedge B_2', s'\rangle}\ (\text{and}_\text{R})$$

Small-Step Semantics For Commands:

$$\frac{\langle E, s\rangle \to \langle E', s'\rangle}{\langle I := E, s\rangle \to \langle I := E', s'\rangle}\ (:=_R) \qquad \frac{}{\langle I := n, s\rangle \to \langle skip, s[I \mapsto n]\rangle}\ (:=)$$

$$\frac{\langle C_1, s\rangle \to \langle C_1', s'\rangle}{\langle C_1; C_2, s\rangle \to \langle C_1'; C_2, s'\rangle}\ (\text{seq}) \qquad \frac{}{\langle skip; C, s\rangle \to \langle C, s\rangle}\ (\text{skip})$$

$$\frac{\langle B, s\rangle \to \langle B', s'\rangle}{\langle \mathit{if}\ B\ \mathit{then}\ C_1\ \mathit{else}\ C_2, s\rangle \to \langle \mathit{if}\ B'\ \mathit{then}\ C_1\ \mathit{else}\ C_2, s'\rangle}\ (\text{if})$$

$$\frac{}{\langle \mathit{if}\ \mathit{True}\ \mathit{then}\ C_1\ \mathit{else}\ C_2, s\rangle \to \langle C_1, s\rangle}\ (\text{if}_T)$$

$$\frac{}{\langle \mathit{if}\ \mathit{False}\ \mathit{then}\ C_1\ \mathit{else}\ C_2, s\rangle \to \langle C_2, s\rangle}\ (\text{if}_F)$$

$$\frac{}{\langle \mathit{while}\ B\ \mathit{do}\ C, s\rangle \to \langle \mathit{if}\ B\ \mathit{then}\ (C; \mathit{while}\ B\ \mathit{do}\ C)\ \mathit{else}\ \mathit{skip}, s\rangle}\ (\text{while})$$

Remark:
There is no axiom or rule for programs of the form:

- $\langle n, s\rangle$ where n is an integer
- $\langle b, s\rangle$ where b is a boolean
- $\langle !I, s\rangle$ where $I \notin dom(s)$
- $\langle skip, s\rangle$

These are *terminal configurations*. In the case of $\langle !I, s\rangle$ where $I \notin dom(s)$ we say that the program is *blocked*.

Example:
Let P be the program `z:= !x; x:= !y; y:= !z` and s a state such that $s(z) = 0$, $s(x) = 1$, $s(y) = 2$.

There is a sequence of transitions:
$\langle P, s\rangle \to \langle z := 1; (x :=!y; y :=!z), s\rangle$
$\to \langle skip; (x :=!y; y :=!z), s[z \mapsto 1]\rangle$
$\to \langle x :=!y; y :=!z, s[z \mapsto 1]\rangle$
$\to \langle x := 2; y :=!z, s[z \mapsto 1]\rangle$
$\to \langle skip; y :=!z, s[z \mapsto 1, x \mapsto 2]\rangle$
$\to \langle y :=!z, s[z \mapsto 1, x \mapsto 2]\rangle$
$\to \langle y := 1, s[z \mapsto 1, x \mapsto 2]\rangle$
$\to \langle skip, s[z \mapsto 1, x \mapsto 2, y \mapsto 1]\rangle$

Comparison with Abstract Machine

- Before we had a precise data structure with with different stacks (Control, result and memory) and precise rules to manipulate those stacks now we have rules which are more abstract (they don't define how the implementation is done).
- To justify the transitions we need axioms and rules to justify them. To justify a rule we need more work.

Big-Step Semantics

We classify evaluation sequences in three categories:

- **Terminating:** is the sequence eventually reaches a terminal non-blocked configuration, that is configuration of the form:
  - <n,s> where n is an integer
  - <b,s> where b is a boolean
  - <skip,s>
- **Blocked (Stuck):** if the sequence eventually reaches a blocked configuration (that is, a configuration of the form <!*l*,s> where *l* is not in dom(s)
- **Divergent:** if the sequence is infinite.

Examples:

$\langle \text{while True do skip}, s\rangle$ is divergent.
$\langle \text{if } !x = 0 \text{ then skip else skip}, s\rangle$ is stuck if $dom(s)$ does not contain x.
$\langle \text{if } 4 = 0 \text{ then skip else skip}, s\rangle$ is terminating.

Transition System

$$\langle P, s\rangle \Downarrow \langle P', s'\rangle \text{ if } \langle P, s\rangle \rightarrow^* \langle P', s'\rangle \text{ where } \langle P', s'\rangle \text{ is terminal.}$$

The downward arrow only means the transition from the initial configuration <P,s> to the final configuration <P',s'> through a series of steps assuming the sequence is terminal.

Big-Step Semantics

$$\frac{}{\langle c, s\rangle \Downarrow \langle c, s\rangle,\ \textit{if } c \in Z \cup \{\textit{True}, \textit{False}\}}\ (\text{const})$$

$$\frac{}{\langle !I, s\rangle \Downarrow \langle n, s\rangle,\ \textit{if } s(I) = n}\ (\text{var})$$

$$\frac{\langle B_1, s\rangle \Downarrow \langle b_1, s'\rangle \quad \langle B_2, s'\rangle \Downarrow \langle b_2, s''\rangle}{\langle B_1 \wedge B_2, s\rangle \Downarrow \langle b, s''\rangle,\ \textit{if } b = b_1 \textit{ and } b_2}\ (\text{and})$$

$$\frac{\langle B_1, s\rangle \Downarrow \langle b_1, s'\rangle}{\langle \neg B_1, s\rangle \Downarrow \langle b, s''\rangle,\ \textit{if } b = \textit{not } b_1}\ (\text{not})$$

$$\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle E_1 \textit{ op } E_2, s\rangle \Downarrow \langle n, s''\rangle,\ \textit{if } n = n_1 \textit{ op } n_2}\ (\text{op})$$

$$\frac{\langle E_1, s\rangle \Downarrow \langle n_1, s'\rangle \quad \langle E_2, s'\rangle \Downarrow \langle n_2, s''\rangle}{\langle E_1 \textit{ bop } E_2, s\rangle \Downarrow \langle b, s''\rangle,\ \textit{if } b = n_1 \textit{ bop } n_2}\ (\text{bop})$$

Big-Step Semantics

$$\frac{}{\langle \textit{skip}, s\rangle \Downarrow \langle \textit{skip}, s\rangle}\ (\text{skip}) \qquad \frac{\langle E, s\rangle \Downarrow \langle n, s'\rangle}{\langle I := E, s\rangle \Downarrow \langle \textit{skip}, s'[I \mapsto n]\rangle}\ (:=)$$

$$\frac{\langle C_1, s\rangle \Downarrow \langle \textit{skip}, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle \textit{skip}, s''\rangle}{\langle C_1; C_2, s\rangle \Downarrow \langle \textit{skip}, s''\rangle}\ (\text{seq})$$

$$\frac{\langle B, s\rangle \Downarrow \langle \textit{True}, s'\rangle \quad \langle C_1, s'\rangle \Downarrow \langle \textit{skip}, s''\rangle}{\langle \textit{if } B \textit{ then } C_1 \textit{ else } C_2, s\rangle \Downarrow \langle \textit{skip}, s''\rangle}\ (\text{if}_\text{T})$$

$$\frac{\langle B, s\rangle \Downarrow \langle \textit{False}, s'\rangle \quad \langle C_2, s'\rangle \Downarrow \langle \textit{skip}, s''\rangle}{\langle \textit{if } B \textit{ then } C_1 \textit{ else } C_2, s\rangle \Downarrow \langle \textit{skip}, s''\rangle}\ (\text{if}_\text{F})$$

$$\frac{\langle B, s\rangle \Downarrow \langle \textit{False}, s'\rangle}{\langle \textit{while } B \textit{ do } C, s\rangle \Downarrow \langle \textit{skip}, s'\rangle}\ (\text{while}_\text{F})$$

$$\frac{\langle B, s\rangle \Downarrow \langle \textit{True}, s'\rangle \quad \langle C, s'\rangle \Downarrow \langle \textit{skip}, s''\rangle \quad \langle \textit{while } B \textit{ do } C, s''\rangle \Downarrow \langle \textit{skip}, s'''\rangle}{\langle \textit{while } B \textit{ do } C, s\rangle \Downarrow \langle \textit{skip}, s'''\rangle}\ (\text{while}_\text{T})$$

Big-Step Semantics - Example

Consider the program P : $(z :=!x; x :=!y); y :=!z$
and a state s such that $s(z) = 0$, $s(x) = 1$, $s(y) = 2$.
We can prove that $P \Downarrow \langle skip, s' \rangle$ where $s'(z) = 1, s'(x) = 2, s'(y) = 1$.
First notice that:

$$\dfrac{\dfrac{}{\langle !x, s\rangle \Downarrow \langle 1, s\rangle}}{\langle z :=!x, s\rangle \Downarrow \langle skip, s[z \mapsto 1]\rangle} \quad \dfrac{\dfrac{}{\langle !y, s[z \mapsto 1]\rangle \Downarrow \langle 2, s[z \mapsto 1]\rangle}}{\langle x :=!y, s[z \mapsto 1]\rangle \Downarrow \langle skip, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle}$$

$$\langle z :=!x; x :=!y, s\rangle \Downarrow \langle skip, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle$$

Using this we can prove:

$$\dfrac{\langle z :=!x; x :=!y, s\rangle \Downarrow \langle skip, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle \quad \dfrac{}{\langle !z, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle \Downarrow \langle 1, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle} \quad \langle y :=!z, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \end{smallmatrix}]\rangle \Downarrow \langle skip, s[\begin{smallmatrix} z \mapsto 1 \\ x \mapsto 2 \\ y \mapsto 1 \end{smallmatrix}]\rangle}{\langle P, s\rangle \Downarrow \langle skip, s'\rangle}$$

Adding Variable Declarations to SIMP

To add declarations we need to extend our syntax to be able to specify that we want to add a local variable (static scope) in memory. (*loc*)

Syntax:

We extend the syntax of SIMP with blocks.

C ::= begin loc x := E; C end

Semantics:

We add the following rule to the big-step semantics:

$$\dfrac{\langle E, s\rangle \Downarrow \langle n, s'\rangle \quad \langle C\{x \mapsto l\}, s'[l \mapsto n]\rangle \Downarrow \langle skip, s''[l \mapsto n']\rangle}{\langle \text{begin loc } x := E; C \text{ end}, s\rangle \Downarrow \langle skip, s''\rangle}$$

where

- $l \notin dom(s') \cup dom(s'') \cup locations(C)$, that is, l is a fresh name
- $C\{x \mapsto l\}$ is the program C where all the occurrences of x are replaced by l (to avoid confusion with other variables of the same name in other parts of the program).

Example: The following program P swaps the contents of x and y using a local variable z:

begin
  loc z $:= !x$;
  x $:= !y$;
  y $:= !z$
end

To show that the program P is correct:

- First we prove

  $$\langle x := !y; y := !l, s[l \mapsto s(x)] \rangle \Downarrow \langle skip, s[x \mapsto s(y), y \mapsto s(x)] \rangle$$

  as in the previous examples.
- Let us call s' the store $s[x \mapsto s(y), y \mapsto s(x)]$, then:

$$\frac{\overline{\langle !x, s \rangle \Downarrow \langle s(x), s \rangle} \quad \langle x := !y \; ; \; y := !l, s[l \mapsto s(x)] \rangle \Downarrow \langle skip, s' \rangle}{\langle P, s \rangle \Downarrow \langle skip, s' \rangle}$$

Exam

Possible questions, add to simp:

- Expressions like x++ and ++x
- Local declarations of variables
- Do while

Functional Programming

Functional programs consist of functions. Functions can be defined in terms of other functions (precisely defined). They focus on what is to be computer not how it should be computed. E.g. JavaScript, Scala, Haskell

+ Shorter programs, easier to understand, easier to design and maintain than imperative programs
- Slower than imperative programs

Haskell

- Modern functional programming language
- CHC is the Glasgow Haskell Compiler

Inspired by the mathematical definition of functions using equations.
I.e. `square x = x * x`
Name of function, input = what to do to produce output

Properties of Evaluation

When we evaluate we try to reduce expressions until they cannot be reduced anymore.
***Exam**, be able to show how expressions are reduced in Haskell.*

Example

`square 6` $\rightarrow$ `6 * 6` $\rightarrow$ `36`
`36` is the value or normal form of `square 6`.

Example

`((3 + 1) + (2 + 1))` $\rightarrow$ `((3 + 1) + 3)` $\rightarrow$ `(4 + 3)` $\rightarrow$ `7`
`7` is the value denoted by the expression `((3 + 1) + (2 + 1))`.

There may be several reduction sequences for an expression.

Example

The following is also a correct reduction sequence:
`((3 + 1) + (2 + 1))` $\rightarrow$ `(4 + (2 + 1))` $\rightarrow$ `(4 + 3)` $\rightarrow$ `7`

In both cases the value is the same.

1. **Unicity of normal forms:** in (pure) functional languages the value of expressions will always reduce to the same thing if it's components are the same and the expression terminates. The order doesn't matter.
2. **Non-termination:** reducing doesn't always lead to a value, sometimes reduction sequences don't terminate.

Example

Let us define a constant function: `fortytwo x = 42`
and another function: `infinity = infinity + 1`

- The evaluation of `infinity` never reaches a normal form.
- For the expression `fortytwo infinity` some reduction sequences do not terminate, but those which terminate give the value 42 (unicity of normal forms).

Strategies

Despite the normal form being unique, the order of reductions is important.

Call-by-name (normal order): reduce first the application using the def of the func and then the argument. **Always finds the value if there is one**

- Usually less efficient since because since it replaces the body before evaluating the arguments, it ends up evaluating the same expression multiple times.

Example call by name

```
Square(3+3) -> (3+3) * (3+3)-> 6 * 6 -> 36
```

We don't call Square 6 so we compute 3+3 twice instead of once at the start.

Call-by-value (applicative order): evaluate first the argument and then the application using the definition of the function, **more efficient but may fail to find a value**
Example Call by value

```
Square(3+3) -> Square(6) -> 6 * 6 -> 36
```

In this case we evaluate 3 + 3 first and then compute but it can reach non-termination (as illustrated in the `fourtytwo` and `infinity` example.

`Haskell` uses a strategy called *lazy evaluation,* it doesn't want to compute anything that is not necessary. In the `infinity` example, which guarantees that if an expression has a normal form, the evaluator will find it. It only evaluates what your really need (it uses the **call-by-name strategy**).

Functional Values

Functions are also values, even though we cannot display them or print them.
Function: a mapping that associates to each element of a given type A, called the domain of the function to an element of type B, codomain.

f : A -> B

If a function f of type A -> B is applied to an argument x of type A it gives a result (f x) of type B.

Syntax to associate a type to a function:

```
square ::  Integer → Integer
fortytwo ::  Integer → Integer
```

Takes an integer as arguments and it returns an integer as a result.

Syntax of Function Definitions

Functions are defined in terms of equations.

- `square x = x * x`
- `min x y = if x ≤ y then x else y`

We can also use **conditional equations** aka guarded equations

```
sign x
    | x < 0   = -1
    | x == 0   = 0
    | x > 0   = 1
```

The latter is equivalent to, but clearer than: `if x < 0 then -1 else if x == 0 then 0 else 1`

You cannot have an if then else that doesn't have an else.

Function Application

Application is denoted by (f x)

- `(square 3)`
  `(sign -5)`
- Application has precedence over other operations:
  - `Square 3 + 1` means `(square 3) + 1`
- Application associates left:
  - `square square 3 means (square square) 3`
    invalid (left association)
- we don't normally write outer brackets, square 3 instead of (square 3)

Recursive Definitions

In the definition of a function `f` we can use the function `f`:

```
fact ::  Integer → Integer
fact n = if n == 0 then 1 else n * (fact (n - 1))
```

Recursive definitions are evaluated by simplification (as any other expression)

```
fact 0 → if 0 == 0 then 1 else 0 * (fact (n - 1))
→ if True then 1 else 0 * (fact (n - 1)) → 1
```

Note the operational semantics of the conditional

1) Evaluate first the condition
2) If the result is true then evaluate only the expression in the left branch (then)
3) If the result is false then evaluate only the expression on the right branch (else)

Evaluation of Recursive Functions

If in the sequence above we start with fact -1 it won't terminate, to fix that:

```
fact ::  Integer → Integer
fact n
    | n > 0   = n * (fact (n - 1))
    | n == 0  = 1
    | n < 0   = error "negative argument"
```

Error is a predefined function that takes a string as argument. It causes immediate termination of the evaluator and displays the display message.

Function Definitions by Pattern Matching

Upon evaluating the argument of a function it is compared against those of the possible definitions, in the order they are provided, until either a match is found or the patterns are exhausted.
Factorial could be written like this.

```
fact ::  Integer → Integer
fact 0 = 1
fact n = n * (fact (n - 1))
```

Local Definitions

As in mathematics, we can write:

```
f x = a + 1 where a = x / 2
```

or equivalently,

```
f x = let a = x / 2 in a + 1
```

The syntax, `let ... in` and `where`, are used to introduce local definitions, available just on the right-hand side of the equation that we are writing.

We can write several local definitions:

```
f x = square (successor x)
    where square z = z * z ; successor x = x + 1
```

Evaluation of Local Definitions

$e[a]$ where $a = e'$

1. Evaluate e', obtaining a result r
2. Replace a by r in e
3. Evaluate $e\{a \mapsto r\}$

Example

```
magnitude a b = sqrt (asq + bsq)
    where asq = a * a; bsq = b * b

magnitude 3 4
→ sqrt (asq + bsq) where asq = 3 * 3; bsq = 4 * 4
→ sqrt (asq + bsq) where asq = 9; bsq = 4 * 4
→ sqrt (9 + bsq) where bsq = 4 * 4
→ sqrt (9 + bsq) where bsq = 16
→ sqrt (9 + 16) → sqrt 25 → 5
```

Arithmetic Functions

- Also functions (primitives), used in infix notations e.g. 3 + 4
- We can use them in prefix notation if we enclose them in brackets e.g. (+) 3 4
- Standard function may be made infix by quoting with backticks 10 `div` 2
- +, -, *, /, div, mod are left associative
- Application has priority:
    - Square 1 + 4 * 2 should be read as (square 1) + (4*2)

First-Class functions

Functions may take functions as arguments or return functions as values.

```
(.)  ::  (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f (g x)
```

B,c is the type of the first. a->b is the type of the second and a->c is the output of the whole thing.

We can only compose functions whose types match.

Example

```
square ::  Int -> Int
square x = x * x

quad ::  Int -> Int
quad = square .  square
```

(.) = (Int -> Int) -> (Int -> Int) -> (Int -> Int)

Curried Functions and Partial Applications

All functions in Haskell take only one argument.
Applying a function of n > 1 arguments yields another function with n -1 arguments.
This allows one to create new functions by partial application.

Example

```
plus ::  Int -> Int -> Int
plus x y = x + y
plus 3 ::  Int -> Int

plusThree ::  Int -> Int
plusThree y = 3 + y
plusThree 2 = 5

(plus 3) 2 ::  Int
(plus 3) 2 = 5
```

All arithmetic operators in Haskell are also curried functions.

Example

```
(*) ::  Int -> Int -> Int
(*) 6 ::  Int -> Int
((*) 6) 4 = 24

(+) ::  Int -> Int -> Int
(+) 1 ::  Int -> Int
((+) 1) 4 = 5
```

The alternative is to pass multiple arguments as one in a tuple but this does not allow partial application.

Example

```
plusTuple (Int, Int) -> Int
plusTuple (x, y) = x + y
```

We can convert between curried and uncurried functions using the primitive functions curry and uncurry.

```
curry ::  ((a, b) -> c) -> (a -> b -> c)
uncurry ::  (a -> b -> c) -> ((a, b) -> c)
```

Example

```
plus ::  Int -> Int -> Int
plus = curry plusTuple
```

Types

Advantages of Static Typing

Values are divided into classes called types. Carried out at compile time, it ensures that expressions that cannot be typed are considered erroneous and are rejected by the compiler without evaluation, ensuring that the program will be *free of type errors before runtime* (haskell does static typing).

+ Detects possible issues at an early stage aiding the development and maintenance of the software.
+ It is easier to document and design since it simplifies complex forms.

Types in Haskell

The set of predefined types includes:

- Primitive data types: numbers, Booleans, characters, etc…
- Constructed types: lists, tuples, etc
- Function types: i.e. Char -> Bool

The programmer can also define it own types

Polymorphism

Expressions may have more than one type, achieved by type variables that can be instantiated with other types. (haskell has this system)

Monomorphic would be expressions that have at most one type.

Polymorphic Types

Example:

Example

Functional composition `(.)` and `curry` are polymorphic functions where `a`, `b` and `c` are type variables.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
curry :: ((a, b) -> c) -> (a -> b -> c)
```

This is polymorphics as a, b and c are type variables.

Formally, the language of polymorphic types is defined as a set of terms T, built out of type variables V, (a,b,c,...), and a type constructor C, which are either atomic (Int, Bool, Char) or take other types as arguments ([], ())

$$\begin{array}{lll} \mathcal{V} & ::= & \mathtt{a}, \mathtt{b}, \mathtt{c} \ldots \\ \mathcal{C} & ::= & \mathtt{Int}, \mathtt{Bool}, \mathtt{Char}, \mathtt{->}, \mathtt{[]}, \mathtt{()} \ldots \\ \mathcal{T} & ::= & \mathcal{V} \mid \mathcal{C}\,(\mathcal{T}_1, \mathcal{T}_2, \ldots \mathcal{T}_n) \end{array}$$

Anything that is not **a -> b** where a != b is a polymorphic type.

Example

The error function is also polymorphic error.
error :: String -> a

```
fact :: Int -> Int
fact n
  | n == 0 = 1
  | n > 0 = n * (fact (n - 1))
  | otherwise = error "negative argument"
```

Here the function error is used with type String -> Int

Overloading aka ad-hoc polymorphism

Overloading: several functions, with different types share the same name

Arithmetic operations (such as addition) can be used both with integers or reals. But a polymorphic type such as

(+) :: a → a → a

Is too general but it would allow addition to be used with characters of type Char.

There are several solutions (to make it more specific):

1. Use different symbols for addition on integers and on reals
2. Enrich the languages of types, for example.
   (+) :: (Int → Int → Int) ∧ (Float → Float → Float)
3. (One used by Haskell) Define a notion of a type class, for example:
   (+) :: Num a => a → a → a
   That is, the function (+) has the type a → a → a
   Where ***a*** is an instance of the type Num.

Type Classes

Can be thought of as an interface. Types that belong to the class must implement functions specified by the class.

Common types: `Eq`, `Ord`, `Num`, `Integral`, `Enum`, `Floating`, `Show`.

Example

- `(==) :: Eq a => a -> a -> Bool`
- `(>) :: Ord a => a -> a -> Bool`
- `fromIntegral :: (Num b, Integral a) => a -> b`
- `sqrt :: Floating a => a -> a`
- `min :: Ord a => a -> a -> a`

Eq = equals, converts a to a to boolean
Ord = comparable in java can can convert from a to a and output a boolean.
Num = is any type of number
Show = like toString in java

NB, anything on the LHS of the => arrow are the constraints on the types on the type variables on the right. The single arrow is the function type constructor. The last one is always the type that the function returns.

Type Inference

Most modern functional languages don't need us to tell it the type it can *infer it,* if it exists.

- The expression is decomposed into smaller sub-expressions, and when a basic atomic expression is found, any available information is used, if it's a built in primitive type we assign it else we assign the most general type possible.
- The manner in which the sub-expressions are put together to build the expression generates constraints that their types must satisfy.
- If we can't satisfy the constraints, the expression cannot be typed.

Example

Consider the definition `square x = x * x`.
What is the type of the function `square`?

- It is a function, so the most general type is `a -> b`.
- Thus, `x :: a` and `x * x :: b`.
- `(*) :: Num a => a -> a -> a` is a primitive function.
- Therefore, `x :: Num a => a` and `x * x :: Num a => a`.
- Therefore, `a` and `b` must both be of type `Num a => a`.
- Therefore, `square :: Num a => a -> a`.

Example

Given that `square :: Int -> Int`,
what is the type of the expression `square square 3`?

- Remember that application associates to the left,
  so the expression reads `(square square) 3`.
- `square` expects its argument to be of type `Int`
  but here it is applied to itself,
  namely a function of type `Int -> Int`.
- The constraint `Int = (Int -> Int)` is thus generated
  but cannot be solved and so the expression is untypeable.

Exam, inferring a polymorphic type by hand of what Haskell will output.

Lists

Linked Lists are an important primitive data type in most functional languages

- `[] :: [a] -> a` — the empty list
- `(:) :: a -> [a] -> [a]` — add an element to a list ("cons")
- `head :: [a] -> a` — return the first element of a non-empty list
- `tail :: [a] -> [a]` — return the remainder of a non-empty list

Lists are so common they have their own special syntax:
e.g. `(1 : 2 : 3 : [])` would be abbreviated to `[1, 2, 3]`.

Example

```
[] :: [a]
(1 : 2 : []) :: [Int]      or      [1, 2] :: Int
['c', 'a', 't'] :: [Char]
[3, 3.1, 3.14, 3.141, 3.1415] :: [Double]
[[True, False], [True]] :: [[Bool]]
```

Functions on Lists

Functions on lists and other constructed types are usually defined using pattern matching

Example

```
size :: [a] -> Int
size [] = 0
size (x : xs) = 1 + size xs

size [] → 0
size [6, 5, 5, 3, 6] →* 5
```

User Defined types

We can define our own new types by declaring the data and type constructors of that type

Example

```
data Nat = Zero | Succ Nat
```

Here, `Nat` is a new recursive type and `Zero` and `Succ` are its *data constructors*.

Example

```
data Seq a = Empty | Cons a (Seq a)
```

Here, `Seq a` is a new polymorphic, recursive type, whose elements are built using one of the data constructors `Empty` or `Cons`.

`Nat` and `Seq` are *type constructors*.

Data Constructors are used to build terms of a type

I.e

- `Zero :: Nat`
- `(Suc (Suc Zero)) :: Nat`
- `Empty :: Seq a`
- `(Cons (Suc Zero) Empty) :: Seq Nat`

.

Data constructors can be used when giving definitions by pattern matching
I.e.

```
isempty Empty = True
isempty (Cons x y) = False
```

Exam,
Create a recursive data type and write a function for it

Write the definition for a data type `Tree a` representing binary trees with data stored in the leaves. Use data constructors `Leaf` and `Branch`.

Write a definition for a function `height t` which computes the height of a tree `t`. What is the type of `height`? (Hint: you will need a local definition of a function `max x y`.)

Solution:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

height :: Tree a -> Int
height (Leaf _) = 1
height (Branch l r)
    = 1 + max (height l) (height r)
  where max x y = if x > y then x else y
```

* We are creating a new data type so we have to start with our ***data*** keyword**.**
***** Then we need to create the constructor for it. (It takes one argument).
* For data constructors we give as arguments, the type that we will use. Hence why Tree ***a*** and Leaf ***a***.
* The **|** represents the different ways we can construct something of our type.
* In (Leaf _) we need to put the brackets because the function application takes precedence over most things so it will think that the input is only Leaf not Leaf x.

Induction (proving properties of our programs)

To reason with recursive types we can use the **Principle of Structural Induction**

Example

In the case of `Nat`,
to prove a property P for all the finite elements, we have to

1. Prove $P(\texttt{Zero})$ — the *base case*.
2. Prove that if $P(n)$ holds (the *induction hypothesis*), then $P(\texttt{Succ}\ n)$ holds — the *inductive case*.

This follows the familiar *Principle of Mathematical Induction:*

$$(P(0) \wedge \forall n.(P(n) \Rightarrow P(n+1))) \Leftrightarrow \forall n.P(n)$$

If a property holds for n then it should hold for the successor of n.

We can define addition on Nat by pattern matching:

```
add :: Nat -> Nat -> Nat
add Zero x = x
add (Succ x) y = Succ (add x y)
```

And prove by induction that Zero is a neutral element, so that for all natural number n , add n Zero = n

1. *Base case*: to prove `add Zero Zero = Zero`,
   we use the first equation of the definition of `add`.
2. *Inductive case*: by the second equation for `add`,
   `add (Succ n) Zero = Succ (add n Zero)`
   which is equal to `Succ n` by the induction hypothesis.

Semantics

We will use a small language SFUN to give the semantics for a language.
Evaluation strategies classify functional languages:

- Call-by-value: arguments are evaluated before function definitions
- Call-by-name (less efficient, repetitions): functions are called before evaluating arguments.

Syntax of SFUN

Set of variables, set of functions with a fixed arity (parameters it can take).

The *terms* of the language SFUN are defined by the grammar:

$$op ::= + \mid - \mid * \mid / \qquad bop ::= > \mid < \mid =$$
$$t ::= n \mid b \mid x \mid t_1\ op\ t_2 \mid t_1\ bop\ t_2 \mid \neg t_1 \mid t_1 \wedge t_2$$
$$\mid \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \mid f(t_1, \ldots, t_{\langle f \rangle})$$

- where n represents an integer value, $n \in \mathbb{Z}$
- and b represents a Boolean value, $b \in \{True, False\}$

In the case of the application of a function f such that $\langle f \rangle = 0$, then the empty parentheses may be omitted and the term written as simply f. *Division in* SFUN *is floor division.*

Note: division is floor division because we are using integers.

Variables

We have a function vars(t) that tells use the variables that occur in term t.

- $vars(x) = \{x\}$
- $vars(f(y, z)) = \{y, z\}$.

- In vars(x) we have variable x, thus {x}
- vars(f(y,z)) we have variables y,z thus = {y,z}

Closed term: term such that vars(t) = { }. A term that contains no variables.

Programs in SFUN

A program in SFUN is a set of recursive equations:

$$\begin{aligned} f_1(x_1, \ldots, x_{\langle f_1 \rangle}) &= d_1 \\ &\vdots \\ f_k(x_1, \ldots, x_{\langle f_k \rangle}) &= d_k \end{aligned}$$

such that

- $d_1, \ldots, d_k$ are terms,
- $vars(d_i) \subseteq \{x_1, \ldots, x_{\langle f_i \rangle}\}$, $\forall i.\, 1 \leq i \leq k$,
- there is only one equation for each function name f_i.

Equations are recursive, thus the terms d_i may contain occurrences of $f_1, \ldots, f_k$.

Note that vars(d1) must be a subset of the variables that we used to create it, thus the right hand side. We call the right hand side terms.
Examples:

Example

$$\begin{aligned} max(x, y) &= \textbf{if } x \geq y \textbf{ then } x \textbf{ else } y \\ fact(x) &= \textbf{if } x \leq 0 \textbf{ then } 1 \textbf{ else } x * fact(x - 1) \end{aligned}$$

Example

$$\begin{aligned} square(x) &= x * x \\ quadratic(x, a, b, c) &= a * square(x) + b * x + c \end{aligned}$$

Operations Semantics of SFUN

We will assume that programs are well typed (that is that when there is supposed to be a boolean, there actually is one). We will use big-stem semantics (evaluation relation) for the program using a transition system where configurations are terms.

- The values will be **integer** and **Boolean**
- Evaluation is denoted by a downward arrow with a P.
- ***The first evaluation strategy will be call-by-value***

The evaluation relation $\Downarrow_P$ is defined inductively as follows:

$$\frac{}{n \Downarrow_P n}\,(\mathrm{n}) \qquad \frac{}{b \Downarrow_P b}\,(\mathrm{b})$$

$$\frac{t_1 \Downarrow_P n_1 \quad t_2 \Downarrow_P n_2}{t_1\ op\ t_2 \Downarrow_P n}\,(\mathrm{op})\ \textit{if}\ n_1\ op\ n_2 = n \qquad \frac{t_1 \Downarrow_P n_1 \quad t_2 \Downarrow_P n_2}{t_1\ bop\ t_2 \Downarrow_P b}\,(\mathrm{bop})\ \textit{if}\ n_1\ bop\ n_2 = b$$

$$\frac{t_1 \Downarrow_P b_1 \quad t_2 \Downarrow_P b_2}{t_1\ \wedge\ t_2 \Downarrow_P b}\,(\mathrm{and})\ \textit{if}\ b_1 \wedge b_2 = b \qquad \frac{t \Downarrow_P b_1}{\neg t \Downarrow_P b}\,(\mathrm{not})\ \textit{if}\ b = \neg b_1$$

$$\frac{t_0 \Downarrow_P \mathit{True} \quad t_1 \Downarrow_P v_1}{\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \Downarrow_P v_1}\,(\mathrm{if_t}) \qquad \frac{t_0 \Downarrow_P \mathit{False} \quad t_2 \Downarrow_P v_2}{\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 \Downarrow_P v_2}\,(\mathrm{if_f})$$

$$\frac{t_1 \Downarrow_P v_1 \quad \ldots \quad t_{\langle f_i \rangle} \Downarrow_P v_{\langle f_i \rangle} \quad d_i\{x_1 \mapsto v_1, \ldots, x_{\langle f_i \rangle} \mapsto v_{\langle f_i \rangle}\} \Downarrow_P v}{f_i(t_1, \ldots, t_{\langle f_i \rangle}) \Downarrow_P v}\,(\mathrm{fn_{val}})$$

Examples call by value

Let P be the program:

$$\begin{aligned} &\mathit{infinity} = \mathit{infinity} + 1 \\ &\mathit{fortytwo}(x) = 42 \\ &\mathit{square}(x) = x * x \end{aligned}$$

Example

The term $\mathit{fortytwo}(0)$ has the value 42, that is $\mathit{fortytwo}(0) \Downarrow_P 42$.

$$\frac{\dfrac{}{0 \Downarrow_P 0}\,(\mathrm{n}) \qquad \dfrac{}{42\{x \mapsto 0\} \Downarrow_P 42}\,(\mathrm{n})}{\mathit{fortytwo}(0) \Downarrow_P 42}\,(\mathrm{fn_V})$$

Example

The term *fortytwo*(*infinity*) does not have a value, because the evaluation of the argument *infinity* gives no value. A derivation for *infinity* cannot be constructed because it recurses infinitely on the rule (fn_{val}).

Example

The term $square(2+1)$ has the value 9, $square(2+1) \Downarrow_P 9$.

Proof.

$$\dfrac{\dfrac{\dfrac{}{2 \Downarrow_P 2}\,(\text{n}) \quad \dfrac{}{1 \Downarrow_P 1}\,(\text{n})}{2+1 \Downarrow_P 3}\,(\text{op}) \quad \dfrac{\dfrac{}{3 \Downarrow_P 3}\,(\text{n}) \quad \dfrac{}{3 \Downarrow_P 3}\,(\text{n})}{(x*x)\{x \mapsto 3\} \Downarrow_P 9}\,(\text{op})}{square(2+1) \Downarrow_P 9}\,(\text{fn}_{\text{val}})$$

Proof of unicity of normal forms

If we evaluate something, if it terminates, then the value reached is unique. We will now prove that.

Theorem

For any closed term t, if $t \Downarrow_P v_1$ and $t \Downarrow_P v_2$, then $v_1 = v_2$.

Proof.

By rule induction.

We distinguish cases according to the rule that applies to t. For any term, there is only one rule that can be applied.

Base cases (axioms):

- If t is a integer n then $n \Downarrow_P n$ using the axiom (n)
- If t is a Boolean b then $b \Downarrow_P b$ using the axiom (b)

Therefore there is only one value in both cases.

There are no more base cases, because t is closed and thus cannot contain variables.

When it says its closed it means that it's a closed term (look further up)

Inductive cases (rules):

- Assume t is the term $f(t_1, \ldots, t_{\langle f \rangle})$.
- Then, using the rule (fn_V), $f_i(t_1, \ldots, t_{\langle f_i \rangle}) \Downarrow_P v$ if and only if $t_1 \Downarrow_P v_1$, ..., $t_{\langle f_i \rangle} \Downarrow_P v_{\langle f_i \rangle}$, and $d_i\{x_1 \mapsto v_1, \ldots, x_{\langle f_i \rangle} \mapsto v_{\langle f_i \rangle}\} \Downarrow_P v$
- By the induction hypothesis, there is at most one value for each term $t_1, \ldots, t_n$ and $d_i\{x_1 \mapsto v_1, \ldots, x_{\langle f_i \rangle} \mapsto v_{\langle f_i \rangle}\}$.
- Therefore v is unique.

The cases corresponding to the other rules are similar.

Call-by-name evaluation of SFUN

TO do call by name we have to replace the following rule of the behaviour of application.

We replace it with the following rule:

$$\frac{d_i\{x_1 \mapsto t_1, \ldots, x_{\langle f_i \rangle} \mapsto t_{\langle f_i \rangle}\} \Downarrow_P v}{f_i(t_1, \ldots, t_{\langle f_i \rangle}) \Downarrow_P v}\ (\text{fn}_\text{name})$$

The reduction system still has unique values.

Theorem

For any closed term t, if $t \Downarrow_P v_1$ and $t \Downarrow_P v_2$ then $v_1 = v_2$.

Proof.

By rule induction. □

Example

Using the call-by-name semantics the term *fortytwo*(0) also has the value 42, *fortytwo*(0) $\Downarrow_P$ 42.

Example

However in contrast to call-by-value, *fortytwo*(*infinity*) $\Downarrow_P$ 42, because the argument *infinity* is discarded without being evaluated.

$$\dfrac{\dfrac{}{42\{x \mapsto \mathit{infinity}\} \Downarrow_P 42}\ (\text{n})}{\mathit{fortytwo}(\mathit{infinity}) \Downarrow_P 42}\ (\text{fn}_\text{name})$$

Example

The term $square(2+1)$ also has the value 9, $square(2+1) \Downarrow_P 9$, but note the different derivation and repeated computations in comparison to the call-by-value semantics.

$$\dfrac{\dfrac{\dfrac{}{2 \Downarrow_P 2}\,(n) \quad \dfrac{}{1 \Downarrow_P 1}\,(n)}{2+1 \Downarrow_P 3}\,(op) \quad \dfrac{\dfrac{}{2 \Downarrow_P 2}\,(n) \quad \dfrac{}{1 \Downarrow_P 1}\,(n)}{2+1 \Downarrow_P 3}\,(op)}{(x*x)\{x \mapsto 2+1\} \Downarrow_P 9}\,(op)}{square(2+1) \Downarrow_P 9}\,(\mathsf{fn}_{\mathsf{name}})$$

Types for SFUN

The grammar defining the syntax. The type system decides which of the expressions we build are good enough to be evaluated and which are errors. It's a monomorphic system.

NB,** the grammar defining the syntax allows us to write things like 1 + True, which doesn't make sense. Therefore in the definition of the semantics we will only consider **well-typed terms.

The *base types*, β, and the *types* τ of SFUN are defined as follows:

$$\beta ::= \mathsf{int} \mid \mathsf{bool}$$
$$\tau ::= (\beta_1, \ldots, \beta_n) \to \beta$$

In the case of a type τ such that $n = 0$ (the type of a function with no arguments), τ may be written simply as β.

We have base types which are integers or booleans and we have function definitions which are a set of base types as arguments that evaluate to another base type.

The set of well-typed terms can be defined using a relation:

$$\Gamma \vdash_\varepsilon t : \tau$$

- Γ is a *variable environment*, or simply *environment*, given as a finite partial function from variables to base types
- ε is a *function environment* assigning to each function name a type respecting its arity: that is, if $\langle f \rangle = 0$ then $\varepsilon(f) = \beta$ and if $\langle f \rangle = n$ where $n \geq 1$, then $\varepsilon(f) = (\beta_1, \ldots, \beta_n) \to \beta$
- t is a SFUN term
- τ is a type

The relation $\Gamma \vdash_\varepsilon t : \tau$ can be read:

> *"If the variable x has type $\Gamma(x)$ for each $x \in dom(\Gamma)$ and the functions $f_1, \ldots, f_k$ have types $\varepsilon(f_1), \ldots, \varepsilon(f_k)$ then the term t has type τ."*

- The term that we are typing is ***t*** and ***tau*** which is the type of the term.
- **Gamma** are the conditions that give t the type tau. Gamma contains the information about variables. Variable x has type tau, variable y has type something else.
- **Epsilon** stores the functions in our program.

If each variable in gamma has a particular type and each function in epsilon has a particular type then term t has type tau.

The well-typedness relation is inductively defined by the following system of axioms and rules. Note that in the case of the application of a function f, if $\langle f \rangle = 0$ then the rule reduces to an axiom, because there are no arguments to check.

$$\frac{}{\Gamma \vdash_\varepsilon b : \text{bool}} \qquad \frac{}{\Gamma \vdash_\varepsilon n : \text{int}} \qquad \frac{}{\Gamma \vdash_\varepsilon x : \beta} \; \textit{if } \Gamma(x) = \beta$$

$$\frac{\Gamma \vdash_\varepsilon t_1 : \text{int} \quad \Gamma \vdash_\varepsilon t_2 : \text{int}}{\Gamma \vdash_\varepsilon t_1 \; op \; t_2 : \text{int}} \qquad \frac{\Gamma \vdash_\varepsilon t_1 : \text{int} \quad \Gamma \vdash_\varepsilon t_2 : \text{int}}{\Gamma \vdash_\varepsilon t_1 \; bop \; t_2 : \text{bool}} \qquad \frac{\Gamma \vdash_\varepsilon t : \text{bool}}{\Gamma \vdash_\varepsilon \neg t : \text{bool}}$$

$$\frac{\Gamma \vdash_\varepsilon t_1 : \text{bool} \quad \Gamma \vdash_\varepsilon t_2 : \text{bool}}{\Gamma \vdash_\varepsilon t_1 \wedge t_2 : \text{bool}} \qquad \frac{\Gamma \vdash_\varepsilon t_0 : \text{bool} \quad \Gamma \vdash_\varepsilon t_1 : \tau \quad \Gamma \vdash_\varepsilon t_2 : \tau}{\Gamma \vdash_\varepsilon \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 : \tau}$$

$$\frac{\Gamma \vdash_\varepsilon t_1 : \beta_1 \quad \ldots \quad \Gamma \vdash_\varepsilon t_{\langle f \rangle} : \beta_{\langle f \rangle}}{\Gamma \vdash_\varepsilon f(t_1, \ldots, t_{\langle f \rangle}) : \beta} \; \textit{if } \varepsilon(f) = (\beta_1, \ldots, \beta_{\langle f \rangle}) \to \beta$$

If you have a function application, all arguments must have a type. Epsilon must say that f is a function that has that type.

Example

Given that $\Gamma(x) = \text{int}$ and $\varepsilon(fact) = (\text{int}) \to \text{int}$ give the derivation for the typing of term $\textbf{if } x \leq 0 \textbf{ then } 1 \textbf{ else } x * fact(x-1)$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_\varepsilon x : \text{int}} \quad \dfrac{}{\Gamma \vdash_\varepsilon 0 : \text{int}}}{\Gamma \vdash_\varepsilon x \leq 0 : \text{bool}} \qquad \dfrac{}{\Gamma \vdash_\varepsilon 1 : \text{int}} \qquad \dfrac{\dfrac{}{\Gamma \vdash_\varepsilon x : \text{int}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash_\varepsilon x : \text{int}} \quad \dfrac{}{\Gamma \vdash_\varepsilon 1 : \text{int}}}{\Gamma \vdash_\varepsilon x - 1 : \text{int}}}{\Gamma \vdash_\varepsilon fact(x-1) : \text{int}}}{\Gamma \vdash_\varepsilon x * fact(x-1) : \text{int}}}{\Gamma \vdash_\varepsilon \textbf{if } x \leq 0 \textbf{ then } 1 \textbf{ else } x * fact(x-1) : \text{int}}$$

Typing SFUN programs

Given a program P in SFUN

$$\begin{aligned} f_1(x_1, \ldots, x_{\langle f_1 \rangle}) &= t_1 \\ &\vdots \\ f_k(x_1, \ldots, x_{\langle f_k \rangle}) &= t_k \end{aligned}$$

and a function environment ε,
P is typeable, if for each equation $f_i(x_1, \ldots, x_{\langle f_i \rangle}) = t_i$,
there exists an environment Γ_i and a type τ_i, such that

- $\Gamma_i \vdash_\varepsilon f_i(x_1, \ldots, x_{\langle f_i \rangle}) : \tau_i$
- $\Gamma_i \vdash_\varepsilon t_i : \tau_i$

To be able to type a program we need:

- The program itself. A list of equations
- And Epsilon which tells us the types of the functions we are given.
- We work at checking the type not at working them out, thus we need predefined functions.
- We have to be able to type the left and the right hand side using the same gamma.

Example of typing SFUN Programs

Example

Given the following program and function environment
show that the program is typable.

$$\begin{aligned} \mathit{infinity} &= \mathit{infinity} + 1 \\ \mathit{fortytwo}(x) &= 42 \\ \mathit{square}(x) &= x * x \end{aligned}$$

$$\begin{aligned} \varepsilon(\mathit{infinity}) &= \mathsf{int} \\ \varepsilon(\mathit{fortytwo}) &= (\mathsf{int}) \to \mathsf{int} \\ \varepsilon(\mathit{square}) &= (\mathsf{int}) \to \mathsf{int} \end{aligned}$$

Both sides of each equation are typeable with type int,
the first in an empty variable environment
and the latter two using an environment Γ, such that $\Gamma(x) = \mathsf{int}$.

Example

Given the following program and function environment show that the program is typable.

$$\begin{array}{lcl} \mathit{mod}(x,y) & = & \text{if } x-y<0 \text{ then } x \text{ else } \mathit{mod}(x-y,y) \\ \mathit{even}(x) & = & \mathit{mod}(x,2)=0 \end{array}$$

$$\begin{array}{lcl} \varepsilon(\mathit{mod}) & = & (\text{int},\text{int}) \to \text{int} \\ \varepsilon(\mathit{even}) & = & (\text{int}) \to \text{bool} \end{array}$$

The two sides of the equation for *mod* both have type int in a variable environment Γ_1, such that $\Gamma_1(x) = \text{int}$ and $\Gamma_1(y) = \text{int}$.
The two sides of the equation for *even* both have type bool using an environment Γ_2 where $\Gamma_2(x) = \text{int}$.

$$\dfrac{\dfrac{}{\Gamma_1 \vdash_\varepsilon x : \text{int}}\,(\text{var}) \quad \dfrac{}{\Gamma_1 \vdash_\varepsilon y : \text{int}}\,(\text{var})}{\Gamma_1 \vdash_\varepsilon \mathit{mod}(x,y) : \text{int}}\,(\text{fn})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash_\varepsilon x : \text{int}} \quad \dfrac{}{\Gamma_1 \vdash_\varepsilon y : \text{int}}}{\Gamma_1 \vdash_\varepsilon x-y : \text{int}} \quad \dfrac{}{\Gamma_1 \vdash_\varepsilon 0 : \text{int}}}{\Gamma_1 \vdash_\varepsilon (x-y)<0 : \text{bool}} \quad \dfrac{}{\Gamma_1 \vdash_\varepsilon x : \text{int}} \quad \dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash_\varepsilon x : \text{int}} \quad \dfrac{}{\Gamma_1 \vdash_\varepsilon y : \text{int}}}{\Gamma_1 \vdash_\varepsilon x-y : \text{int}} \quad \dfrac{}{\Gamma_1 \vdash_\varepsilon y : \text{int}}}{\Gamma_1 \vdash_\varepsilon \mathit{mod}(x-y,y) : \text{int}}}{\Gamma_1 \vdash_\varepsilon \text{if } (x-y)<0 \text{ then } x \text{ else } \mathit{mod}(x-y,y) : \text{int}}$$

$$\dfrac{\dfrac{\dfrac{}{\Gamma_2 \vdash_\varepsilon x : \text{int}}\,(\text{var})}{\Gamma_2 \vdash_\varepsilon \mathit{even}(x) : \text{bool}}\,(\text{fn}) \quad \dfrac{\dfrac{\dfrac{}{\Gamma_2 \vdash_\varepsilon x : \text{int}}\,(\text{var}) \quad \dfrac{}{\Gamma_2 \vdash_\varepsilon 2 : \text{int}}\,(\text{n})}{\Gamma_2 \vdash_\varepsilon \mathit{mod}(x,2) : \text{int}}\,(\text{fn}) \quad \dfrac{}{\Gamma_2 \vdash_\varepsilon 0 : \text{int}}\,(\text{n})}{\Gamma_2 \vdash_\varepsilon \mathit{mod}(x,2)=0 : \text{bool}}\,(\text{bop})$$

Proving properties of SFUN programs

We can us proof by induction on SFUN programs directly.

$$fact(x) = \text{if } x \leq 0 \text{ then } 1 \text{ else } x * fact(x-1)$$

For all natural numbers, n, prove that $fact(n) = n!$

Example

- Base case: $n = 0$
  by the left-hand side of the conditional, $fact(0) = 1 = 0!$
- Induction case: assume $fact(n) = n!$ and prove for $(n+1)$
  by the right-hand side, $fact(n+1) = (n+1) * fact((n+1)-1)$
  which equals $(n+1) * fact(n)$
  and by the inductive hypothesis equals $(n+1) * n! = (n+1)!$

Logic Programming

We use logic to express knowledge, describe a problem. Use inference to compute, manipulate knowledge and obtain a solution to a problem.

+ It has a ***declarative*** style, the program says what should be computed, rather than how it is computed.
+ Precise and simple semantincs
+ The same formalism can be used to specify a problem, write a program and prove properties of it.
+ The same program can be used in many ways.
- Supporting arithmetic and input/output operations (file handling) is provided at the expense of the declarative semantics.
- Most logic languages are restricted to a fragment of classical first-order logic.

Domain of computation

Term

- Variables represented by X, Y, Z …
- Function symbols, with fixed arities represented by f, g, h, … or a,b,c … for constants of arity zero

A term is either a variable, or has the form f(t1,....,t2) where f is a function symbol of arity n and t1, … , tn are terms.

Example

If a is a constant, f a binary function, and g a unary function, and X, Y are variables, then the following are possible terms:

- X
- a
- $g(a)$
- $f(X, g(a))$
- Y
- $f(f(X, g(a)), Y)$
- $g(f(f(X, g(a)), Y))$

Literals

Let p, q, r … represent predicate symbols, each with a fixed arity.
If p is a predicate of arity n and t1, … , tn are terms, then p(t1, … , tn) is an atomic formula, A, B, …
A literal is an atomic formula A, or a negated atomic formula ¬A.

Example

If *rainy* and *snowy* are unary predicates, *temperature* is a binary predicate, *celsius* is a unary function symbol, *tuesday* and *zero* are constants, and X is a variable, then the following are possible literals:

- $temperature(tuesday, celsius(zero))$
- $\neg rainy(tuesday)$
- $snowy(X)$

Clauses

A **Horn clause** is a disjunction of literals in which at most one (one or zero) can be positive.

- A Horn clause with **one** positive literal (not zero) is called a definite clause [A ∨ ¬B_1 ∨ . . . ∨ ¬B_n] can be read as A if B1 and Bn.
- A **definite clause** is called a **fact** otherwise it's a **rule**.
- Programs are set of definite clauses.
- A Horn clause that contains only negative literals [¬B_1 ∨ ... ∨ ¬B_n], is called a **goal** or **query**.

Example

If *rainy* and *snowy* are unary predicates, *temperature*, is a binary predicate, *celsius* is a unary function symbol, *tuesday* and *zero* are constants, and X is a variable, then the following are possible clauses:

- $rainy(tuesday)$
- $temperature(tuesday, celsius(zero))$
- $snowy(X) \vee \neg raining(X) \vee \neg temperature(X, celsius(zero))$
- $\neg rainy(X) \vee \neg snowy(X)$

Substitutions

Values are terms, associated to variables by means of automatically generated *substitutions*.

- A *substitution* is a partial mapping from variables to terms, with a finite domain.
- A substitution σ is written as a mapping $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$.
- $dom(\sigma)$ denotes the domain of the substitution $\{X_1, \ldots, X_n\}$.
- A substitution σ is applied to a term t or a literal L by simultaneously replacing each variable occurring in $dom(\sigma)$ by the corresponding term. The resulting term is denoted $t\sigma$ or $L\sigma$.

Example

- $t = f(f(X, g(a)), Y)$
- $\sigma = \{X \mapsto g(Y), Y \mapsto a\}$
- $t\sigma = f(f(g(Y), g(a)), a)$

Example

- $L = \neg p(X, X, f(g(a), Y))$
- $\sigma = \{X \mapsto f(a, b), Y \mapsto Z, Z \mapsto b\}$
- $L\sigma = \neg p(f(a, b), f(a, b), f(g(a), Z))$

* Note that since all replacements are done at once, Y gets replaced by Z but Z never gets replaced by b.

Prolog Syntax

Clauses

Set of predicates defined as a list of facts and rules. The order the clauses are defined is critical to the evaluation of the program.

- Variables begin with an upper-case letter or underscore
- Function and predicate symbols begin with a lower-case letter

A rule $A \vee \neg B_1 \vee \ldots \vee \neg B_n$ is written:

```
a :- b1, ..., bn.
```

A fact A is written:

```
a.
```

A goal $\neg B_1 \vee \ldots \vee \neg B_n$ is written:

```
:- b1, ..., bn.
```

Tip: ***:-*** looks like an arrow. Thus a implies b1, ... , bn. What is true goes on the left hand side and what is not true on the right.

Example

```
based(prolog, logic).
based(haskell, maths).
likes(claire, maths).
likes(max, logic).
likes(X, P) :- based(P, Y), likes(X, Y).
```

This program consists of four facts and a final rule. Sample goals might include:

- `:- likes(claire, haskell).`
- `:- likes(max, P).`
- `:- likes(Z, prolog).`

X likes P if P is based on Y and X likes Y. [5th example]

Built-in predicates

The programmer can freely choose the names of variables, function symbols and predicate symbols. However there are some built-in predicates:

- Boolean operators: =, >, <
- Arithmetic operators: + -, *, /
- Arithmetic evaluation: ***is***, forces arithmetic evaluation within the system.
- Tuples
- I/O: write, nl

Example

- `(+(3, 2)` or `3 + 2`
- `(>(X, 2)` or `X > 2`
- `is(X, *(+(3, 2), 2))` or `X is (3 + 2) * 2`
- `,(cat, dog)` or `(cat, dog)`
- `write("error")`

Example

```
rainy(tuesday).
temperature(tuesday, celsius(-4)).
snowy(X) :- rainy(X),
    temperature(X, celsius(Y)), Y <= 0.
```

Example

```
fact(0, 1).
fact(X, N) :- X > 0, Y is X - 1,
    fact(Y, M), N is X * M
```

In logic languages, base cases are facts and our logic case turns into rules

Example

```
hanoi(X) :- move(X, left, right, middle).
move(1, X, Y, _) :- write([X, "->", Y]), nl.
move(N, X, Y, Z) :- M is N - 1,
    move(M, X, Z, Y),
    move(1, X, Y, Z),
    move(M, Z, Y, X).
```

Recall the binary predicate `temperature(X, Y)` that expresses the temperature `Y` on day `X`, and let `follows(X, Y)` be a binary predicate indicating that day `X` follows day `Y`.

- Write a predicate `hot(X)` that decides whether the temperature on a day is over 30 degrees celsius.
- Using `hot`, write a second predicate `heatwave(M, N)` that expresses that it has been been above 30 degrees celsius on consecutive days from `M` to `N`.
- Test your answer on the facts:

```
follows(wed, tue).
follows(thu, wed).
follows(fri, thu).
temperature(tue, celsius(32)).
temperature(wed, celsius(31)).
temperature(thu, celsius(34)).
temperature(fri, celsius(29)).
```

- ```
  hot(Day) :-
      temperature(Day, celsius(Temp)), Temp > 30.
  ```
- ```
  heatwave(Day, Day) :- hot(Day).
  heatwave(First, Last) :- hot(First),
      follows(Next, First), heatwave(Next, Last).
  ```

Lists

Linked lists have special syntax.

- The constant [ ] denotes the empty list.
- The built-in predicate | is the “cons” operator that joins an element X to the front of a list L, [X | L]
- [X | [Y | [Z | [ ]]] is abbreviated to [X, Y, Z].
- H is called the head of the list [H | T] - left hand side of cons operator.
- T is called the *tail* of the list [H | T] - right hand side of cons operator.
- Built-in predicates for common operations: member/2, length/2, sort/2 . . .
    - The slash number is the arity of a function. Member, length, sort take two arguments.

The notation p/n indicates that the predicate symbol p has arity n.

Example

The predicate `append(S, T, U)` expresses that the result of appending the list `T` onto the end of list `S` is the list `U`.

```
append([], L, L).
append([X | L], Y, [X | Z]) :- append(L, Y, Z).
```

The following goals represent questions to be solved using the definitions given in the program:

```
:- append([0], [1, 2], U)
:- append(X, [1, 2], U)
:- append([1, 2], X, [0])
```

If you take list S and add to its end list T, then the result of joining them will be U.
The last goal is impossible since it is saying that given a list 1,2 add X and give me one that has only zero. This is impossible since we already have 1,2 in the list.

Write clauses to define predicates `member/2`, `length/2` and `reverse/2`, such that:

- `member(X, L)` decides whether `X` is contained in list `L`.
- `length(L, N)` expresses the length of list `L` in variable `N`.
- `reverse(L, R)` decides whether list `R` is the reverse of list `L`.
  Hint: write an auxilliary predicate `reverseAux(L, R, A)` which uses a third list argument `A` as an accumulator.

- ```
  member(X, [X | _]).
  member(X, [Y | T]) :- X \= Y, member(X, T).
  ```
- ```
  length([], 0).
  length([_| T], N) :- length(T, M), N is 1 + M.
  ```
- ```
  reverse(L, R) :- reverseAux(L, R, []).
  reverseAux([], A, A).
  reverseAux([X | L], R, A) :-
      reverseAux(L, R, [X | A]).
  ```
  ```

Semantics

Unification

- A *unification problem* $\mathcal{U}$ is a set of equations between terms containing variables.

$$\{s_1 = t_1, \ldots, s_n = t_n\}$$

- A solution to $\mathcal{U}$, also called a *unifier*, is a substitution σ, such that when when applied to every term in $\mathcal{U}$, for each equation $s_i = t_i$, the terms $s_i\sigma$ and $t_i\sigma$ are syntactically identical
- The *most general unifier* (*mgu*) of $\mathcal{U}$ is a unifier σ such that any other unifier ρ is an instance of σ

We want to find a substitution that when applied to both the LHS and the RHS the result is the same.

Unification Algorithm

Simplifies equations by applying rules non deterministically.

- The unification algorithm finds the *mgu* for a unification problem if a solution exists, or otherwise fails, indicating that there are no solutions
- To find the *mgu* the algorithm simplifies the set of equations using a set of transformation rules
- At each step, either a new set of equations is produced or a failure case arises
- The algorithm terminates and outputs the *mgu* when no rule may be applied

Rules

- Input: a finite set of term equations, $\{s_1 = t_1, \ldots, s_n = t_n\}$
- Output: the *mgu* of those equations or failure.

$$
\begin{array}{lrcl}
(1) & f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n), E & \rightarrow & s_1 = t_1, \ldots, s_n = t_n, E \\
(2) & f(s_1, \ldots, s_n) = g(t_1, \ldots, t_m), E & \rightarrow & \textit{failure} \\
(3) & X = X, E & \rightarrow & E \\
(4) & t = X, E & \rightarrow & X = t, E \\
 & & & \text{if } t \text{ is not a variable} \\
(5) & X = t, E & \rightarrow & X = t, E\{X \mapsto t\} \\
 & & & \text{if } X \text{ not in } t \text{ and } X \text{ in } E \\
(6) & X = t, E & \rightarrow & \textit{failure} \\
 & & & \text{if } X \text{ in } t \text{ and } X \neq t
\end{array}
$$

If we have two constants equaling each other we can eliminate them by applying rule 1. E.g. b = b.

More on the unification algorithm

- The unification algorithm applies the rules in a non-deterministic way until no rule can be applied or a failure case arises.
- In the case of success, by changing each $=$ in the final set of equations to $\mapsto$ we obtain the *mgu* of the initial set of terms.
- Note that we are working with *sets* of equations, therefore their order in the problem is unimportant
- The test in case (6) is called an *occur-check*. For example, $X = f(X)$ is a failure case. This test is time consuming, and for this reason in some systems it is not implemented.
- Cases (1) and (2) apply also to constants: in the first case the equation is eliminated and in the second failure arises.

Exam Question: When is a substitution called the most general unifier of two atomic formulae?
A substitution σ is called a unifier of φ1 and φ2 if φ1σ = φ2σ. It is the most general unifier if any other unifier is an instance of σ. The unification algorithm described in the notes finds the most general unifier of the given terms.

Examples

Example

{ f(a, a) = f(X, a) }

- using rule (1) with the first equation: $\{a = X, a = a\}$
- using rule (4) with the first equation: $\{X = a, a = a\}$
- using rule (1) with the second equation: $\{X = a\}$

$\{X \mapsto a\}$

If we follow { X -> a } and substitute X by a then we see that we arrive to the original equation f(a,a).

Example

{[X | L] = [0], Y = [1, 2], [X | Z] = U}

- using rule (1) with the first equation: {X = 0, L = [], Y = [1, 2], [X | Z] = U}
- using rule (5) with the first equation: {X = 0, L = [], Y = [1, 2], [0 | Z] = U}
- using rule (4) with the last equation: {X = 0, L = [], Y = [1, 2], U = [0 | Z]}

{X ↦ 0, L ↦ [], Y ↦ [1, 2], U ↦ [0 | Z]}

$\{f(g(a), b) = f(X, b),\ X = g(Z),\ f(a, Y) = f(Z, Y)\}$

- Rule (1), 1: $\{g(a) = X,\ b = b,\ X = g(Z),\ f(a, Y) = f(Z, Y)\}$
- Rule (4), 1: $\{X = g(a),\ b = b,\ X = g(Z),\ f(a, Y) = f(Z, Y)\}$
- Rule (5), 1: $\{X = g(a),\ b = b,\ g(a) = g(Z),\ f(a, Y) = f(Z, Y)\}$
- Rule (1), 2: $\{X = g(a),\ g(a) = g(Z),\ f(a, Y) = f(Z, Y)\}$
- Rule (1), 2: $\{X = g(a),\ a = Z,\ f(a, Y) = f(Z, Y)\}$
- Rule (4), 2: $\{X = g(a),\ Z = a,\ f(a, Y) = f(Z, Y)\}$
- Rule (5), 2: $\{X = g(a),\ Z = a,\ f(a, Y) = f(a, Y)\}$
- Rule (1), 3: $\{X = g(a),\ Z = a,\ f(a, Y) = f(a, Y)\}$
- Rule (1), 3: $\{X = g(a),\ Z = a,\ a = a,\ Y = Y\}$
- Rule (1), 3: $\{X = g(a),\ Z = a,\ Y = Y\}$
- Rule (3), 3: $\{X = g(a),\ Z = a\}$

$\{X \mapsto g(a),\ Z \mapsto a\}$

The overall process works in the following way:

- We start with the leftmost equation. We look for equivalence and operate on it.
- If we can get to a state of Variable = term, where the variable is not within the term and the variable exists elsewhere in the expression, we substitute the info.
- Then we check for constants being equal, and repeat the same process.

These rules are important because they are crucial to the principle of resolution.

The Principle of Resolution

Computers solve using resolution which is a mechanical process where we try to create a proof by contradiction. To prove a goal, we take our program and we negate our goal and try to show that it leads to a contradiction.
Contradiction: when we have a particular literal and its contradiction true at the same time.

- In order to prove a goal $B_1, \ldots, B_n$ with respect to a set P of program clauses, resolution seeks to show that $P, \neg B_1, \ldots, \neg B_n$ leads to a contradiction
- A contradiction is obtained when a literal and its negation are stated at the same time, that is, both A and $\neg A$
- If a contradiction does not arise directly, a new goal is derived by unifying the current goal with a program clause
- The derived goals are called *resolvents*

SLD Resolution

We use a particular form of resolution called SLD-resolution. We are going to be working with definite clauses (at most one positive literals). We always choose a resolution step in a fixed way (Prolog always chooses the first one). Linear means that we always use the most recently created resolvent.

- 'S' *selection rule*: at each resolution step a fixed computation rule is applied to select which atom from the goal will be next resolved upon
- 'L' *linear*: at each resolution step the most recently derived resolvent is used as the next goal
- 'D' *definite*: all the program clauses are definite clauses

Properties of SLD

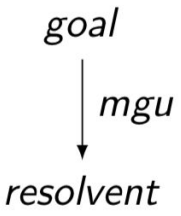
1. SLD-resolution is refutation-complete: given a program and a goal, if a contradiction can be derived, then SLD-resolution will eventually generate it
2. Independence of the selection rule: if there exists a solution to a goal, SLD-resolution will find it, regardless of the selection rule employed for choosing which atom is next resolved upon

Computing resolvents with SLD-resolution

- Given a goal clause $G_1, G_2, \ldots, G_k$, we select an atom G_i of form $p(s_1, \ldots, s_n)$
- We select a program clause $p(t_1, \ldots, t_n) \leftarrow B_1, \ldots, B_m$ such that $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$ are unifiable using the *mgu* σ
- We then obtain the resolvent $B_1\sigma, \ldots, B_m\sigma, G_1\sigma, \ldots, G_k\sigma$ from which $G_i\sigma$ has been eliminated
- The idea is to continue deriving new resolvents until an empty one is found, which indicates that a contradiction has been found
- When an empty resolvent is generated, the composition of the substitutions applied at each resolution step, restricted to the variables of the query, is the *solution* to the goal

SLD-resolution trees

We represent each resolution step graphically as follows:



- Since there might be several clauses in the program that can be used to generate a resolvent from a goal, we obtain an *SLD-resolution tree*
- Every branch in the SLD-tree that leads to an empty resolvent, denoted by □, yields a solution
- If a goal unifies with no program clause, the branch is a failure
- An SLD-resolution tree may thus have success branches, failure branches, but also infinite branches

SLD-resolution in Prolog

- Prolog always selects the leftmost literal in the goal
- Prolog uses the clauses in the program in the order they are written, that is, from top to bottom
- SLD-resolution is complete, but Prolog's implementation of SLD-resolution is not complete because of its search strategy in the SLD-tree. Prolog uses depth-first search, which is not complete, but is efficient in respect to the time until a first solution is found.

Example

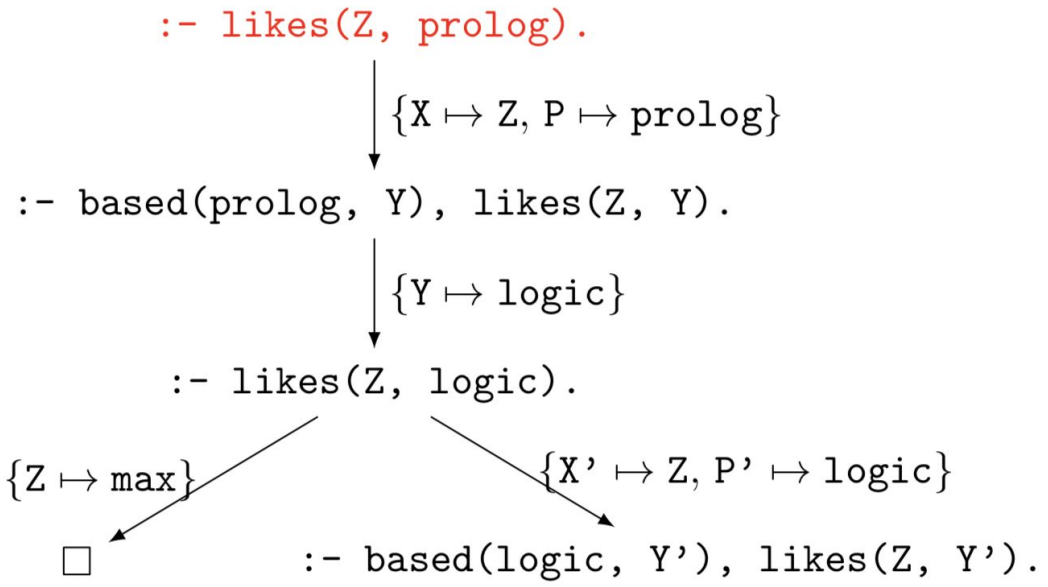
```
based(prolog, logic).
based(haskell, maths).
likes(max, logic).
likes(claire, maths).
likes(X, P) :- based(P, Y), likes(X, Y).
```

- You can't match :- likes (Z,prolog) with based(prolog, logic) because the root is different.
- You can't match it with the fact likes(max,logic), it succeeds with the root ***likes*** but when you compare ***Z*** with ***max*** and ***prolog*** with ***logic*** you get a clash between the constants ***prolog*** with ***logic***.
- The only way we can unify likes(Z, prolog) is with likes(X,P).
- We then match X to Z and P to prolog.

Example

- `:- likes(Z, prolog).`
- Using the last clause and the *mgu* $\{\texttt{X} \mapsto \texttt{Z}, \texttt{P} \mapsto \texttt{prolog}\}$, we obtain the resolvent `:- based(prolog, Y), likes(Z, Y).`
- Now using the first clause and the *mgu* $\{\texttt{Y} \mapsto \texttt{logic}\}$, we obtain the new resolvent `:- likes(Z, logic).`
- Since we can now unify with the fact `likes(max, logic).` using the substitution $\{\texttt{Z} \mapsto \texttt{max}\}$, we can obtain an empty resolvent
- The composition of substitutions generated is $\{\texttt{X} \mapsto \texttt{max}, \texttt{P} \mapsto \texttt{prolog}, \texttt{Y} \mapsto \texttt{logic}, \texttt{Z} \mapsto \texttt{max}\}$
- And the solution to the initial query is $\{\texttt{Z} \mapsto \texttt{max}\}$

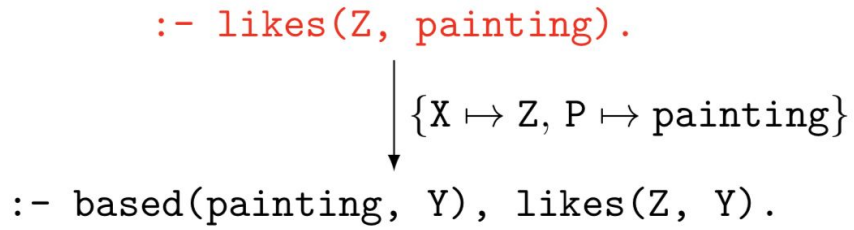
Example



* Whenever you find a successful output in prolog you can tell it to go back to the previous substitution and see if that solves your program (seen in the bottom right).
* Everytime you use a rule, you should make a copy of it since when you do the substitutions, the variables in the rule are local to the rule.

Failure Example

Example



Back-tracking

When a branch ends Prolog can back-track over the tree to search alternative branches. The precious resolution step is taken back and the next possible resolvent is generated.

- Failure branches back track automatically
- Solution branches can be backtracked from on request
- Backtracking can be stopped by using the ***cut*** predicate, written ***!***, which means **don't backtrack beyond this point.**

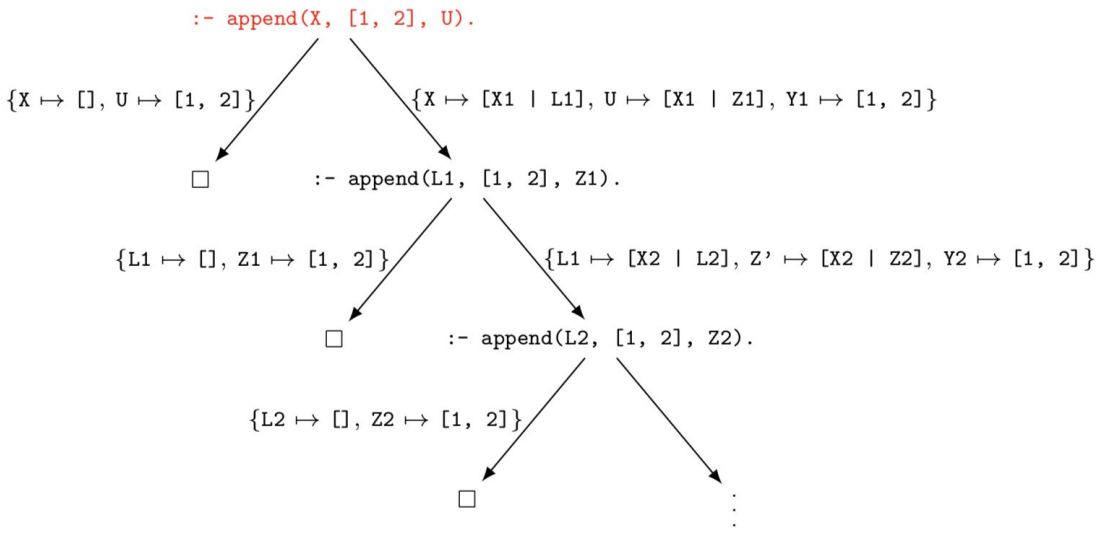
  Example

  ```
  append([], L, L).
  append([X | L], Y, [X | Z]) :- append(L, Y, Z).
  ```

  The goal `:- append(X, [1, 2], U).` produces the following solutions:

  - $\{X \mapsto [], U \mapsto [1, 2]\}$.
  - $\{X \mapsto [X1], U \mapsto [X1, 1, 2]\}$.
  - $\{X \mapsto [X1, X2], U \mapsto [X1, X2, 1, 2]\}$.
  - ...
-

Example



-

Example of non-termination

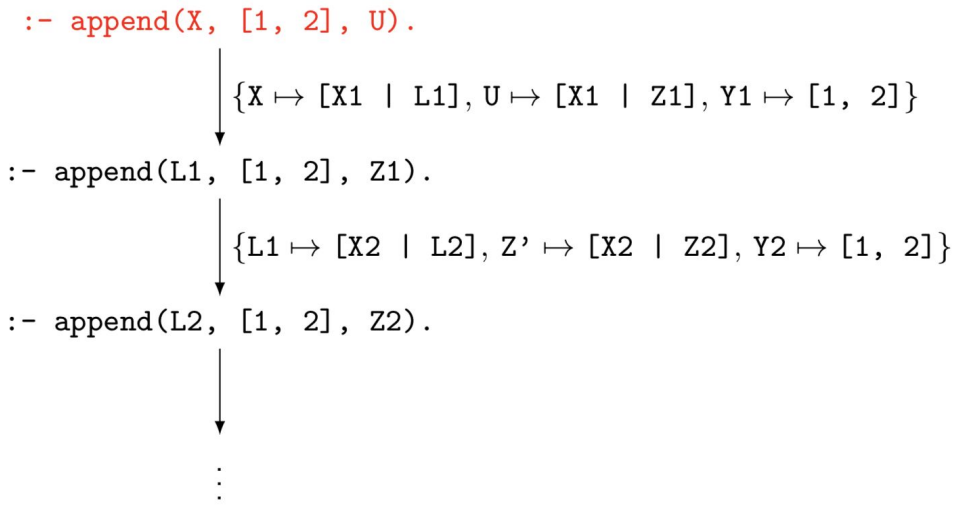
Since Prolog searches the tree using the program clauses in the order in which they are written, and in depth-first manner, we have to put the base clauses before the recursive clauses. If we go into an infinite search, the computation will never terminate and we will not generate other possible solutions. This is why the order in which we write the rules in our program is important.

Example

```
append([X|L],Y,[X|Z]) :- append(L,Y,Z).
append([],L,L).
```

This program produces no solutions and will result in an error, because Prolog will attempt to construct an infinite branch by using the recursive clause in each resolution step.

Example



Exam

3rd question is on Logic Programming. There is always one question on unifying terms using the unification algorithm.

There is always one question taking a prolog goal and evaluating it in the context of some program. Writing SLD trees.
