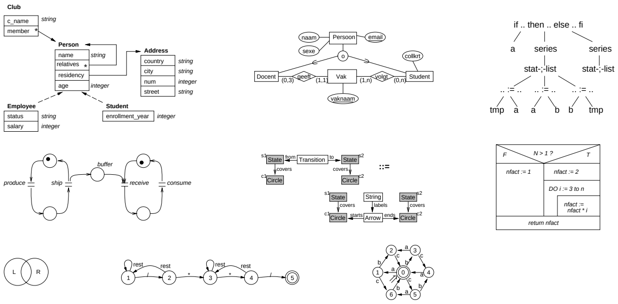
Visuele Talen – Visual Languages

Jan Rekers

Najaar 1995
Vakgroep Informatica
Universiteit van Leiden



Organisatie VL

- **College**: Geen boek, ik behandel aantal artikelen, er komen twee gast docenten

- **Praktikum**: In C++ editor voor visuele taal bouwen.

  Regelmatig inleveren, bepaalt eindcijfer voor 25%.

- **Tentamen**: Over behandelde artikelen, moet voldoende zijn, bepaalt eindcijfer voor 75%

  Voorstel voor de datum: vrijdag 22 december, 9.00 – 12.00

- **Stof**: Kopieën sheets en papers. Totaal ongeveer 300 kopieën, nu graag fl. 15

Overzicht van de colleges

- *Introduction to VL & examples*
  - 1: Introduction
  - 2: Examples of visual programming languages
  - 3: Dataflow languages

- *Syntax directed graphical editing*
  - 4: Fresco basics for the "praktikum"
  - 5: Syntax definition of MSC diagrams
  - 6: Generating visual editors

- 7 & 8: *Constraints*

  Constraint logic programming, constraint solving, constraint based graphical editing.

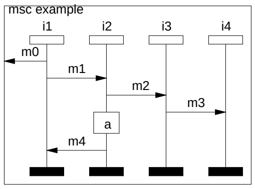
- *Definition of visual syntax, visual parsing*
  - 9: Picture layout grammars
  - 10: Attribute multiset grammars
  - 11: Graph grammar approach

- 12: *Visual reasoning*

- 13: *Overview*

Overzicht Praktikum

U bouwt een syntax gestuurde editor voor de visuele taal MSC.



Programmeren in *C++*, op de *silicon graphics*, met behulp van user interface builder *Fresco* en constraint solver *DeltaBlue*.

Inlevertijd per opdracht 1 à 2 weken, cijfer per opdracht, harde deadlines.

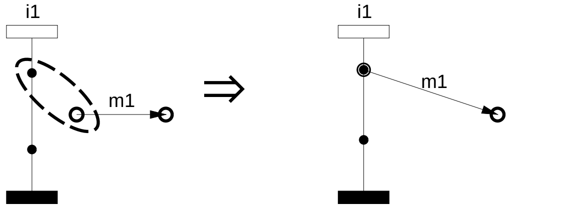
Begeleiding: *Ed van der Winden*, kamer 206, email: winden@wi

Algemeen idee: start met editor die niets van MSC weet en voeg daar allerlei MSC specifieke commando's aan toe.

De opdrachten hebben een sterk *stapel* karakter; zorg dat U van begin af aan hard meewerkt.

- 1, 2: *Fresco* installeren en leren gebruiken (hello, button, rgb)
- 3, 4: *DeltaBlue* leren gebruiken en vanuit Fresco gebruiken
- 5, 6: Aan *Figgy* MSC figuren toevoegen (white box, black box, arrow, line, text)

- 7, 8: *Poly-figures* toevoegen (hele *process instance* in één keer neerzetten) en de onderdelen koppelen via constraints
- 9, 10: MSC figuren van *stekkers* voorzien en stekkers laten koppelen via constraints



- 11, 12: De stekkers verbeteren (typering, dynamisch toevoegen, ontkoppelen, ...)
- 13: MSC constructies als tekst uitprinten en demo geven

Introduction to Visual Languages, Visual Programming, & Program Visualization

Material:

- Brad Myers, *Taxonomies of visual programming and programming visualization*, JVLC, vol. 1, pages 97-123, 1990.

  (only scan the sections 5, 6, 7)

- Bottoni, Costabile, Levialdi & Mussio, *Formalising Visual Languages*, IEEE VL'95, pages 45-52, 1995.

Motivation

- **Graphics can be easier to understand**

  The **human visual system** and visual information processing is optimized to process more-dimensional data.

  Textual representations make only little use of this fact.

  Certain complex problems are simply easier to express in more than one dimension

  – visualization of the current state of a program

  – data structure definition

  – complex data re-organization

  – design of concurrent programs

- Helpful for **non-programmers** and **novice programmers**

  Conventional programming is hard to learn and use.

  For many problems end-users would be helped if they could program

  (this explains the success of spreadsheets)

  Providing options and menu's in software packages is not enough.

  Direct manipulation interfaces enlarge the user / programmer gap

  A solution might be to use **graphics** as the programming language.

- **Higher level descriptions**

  Graphical descriptions tend to be **less precise** and give a more **high level** description of the desired actions.

  This can also be very helpful for experienced programmers.

  Design languages often apply graphical formalisms

Textual – Visual

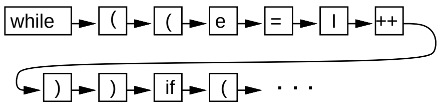
- **Textual Languages** (or **linear** languages)

  The sentences are **one-dimensional**

  Building blocks are usually (a large collection of) characters or tokens

  The only meaningful relation between building blocks is *follows*

```
while ((e = I++))
  if ( e->source()->UserData.WhoAmI ==
       e->target()->UserData.WhoAmI )
    return false;
```

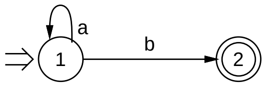


- **Visual Languages** (or **non-linear** languages)

  The **two-dimensional** aspect of the sentence matters.

  Building blocks are usually (a few) graphical objects

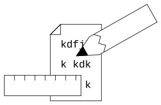
  Relations depend on the language; for example: *contains*, *above*, *touches*, *close-by*.



A gross classification can be made among visual languages by the visual **forms** they use

- **Diagram**: The objects are basic graphical objects; *Wiring* is the most important relation.

- **Iconic**: Objects are small drawings; *Overlapping* is the most important relation.



- **Formula**: Objects are mathematical symbols; *Relative position and size* is the most important relation.

$$\sum_{i=1}^{n} x_i = \int_0^1 \frac{1}{2n-1}$$

Not (per sé) visual languages:

- languages that manipulate visual objects

- languages which build visual user interfaces

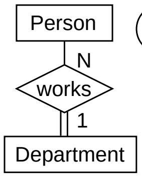
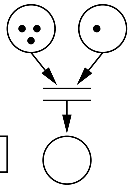
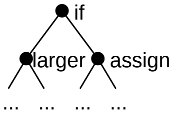
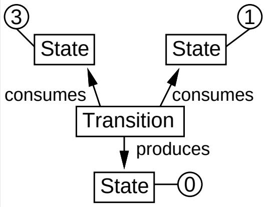
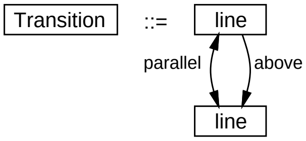
- languages that support visual interaction or visualization of data

For reasons of simplicity, we will (for now) **not** consider the following as visual languages:

- 3-D: virtual reality

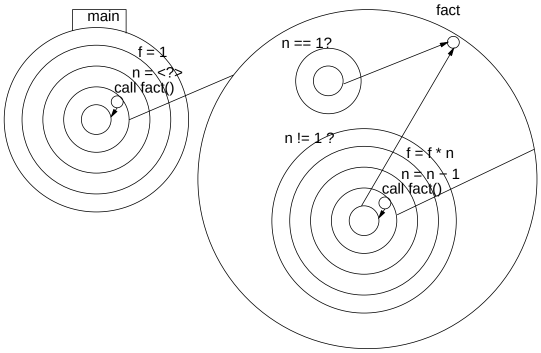
- 2-D + time: animations, iconic user interface

Equivalence of notions

Notion	**Textual**	**Visual**
Language	Pascal, C, Prolog, Lisp	EER, Petri Nets, class diagrams, PROGRES, FSA
Sentence	if a > b then x := 7; ancestor(X, Z) :– parent(X, Y), ancestor(Y, Z).	 
Underlying structure		 3 1 State State consumes consumes Transition produces State 0
Syntax Rule	S ::= if E then S	

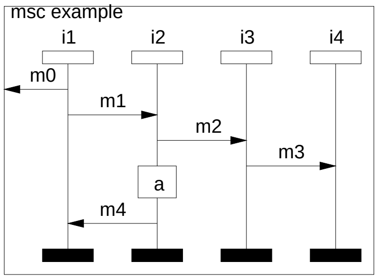
What to choose?

Text is better in some cases



```
void main() {
  f = 1;
  n = <?>; /* assign n */
  fact();
}
void fact() {
  if (n == 1) return
  else {
    f = f * n;
    n = n - 1;
    fact();
}}
```

Pictures are better in other cases:



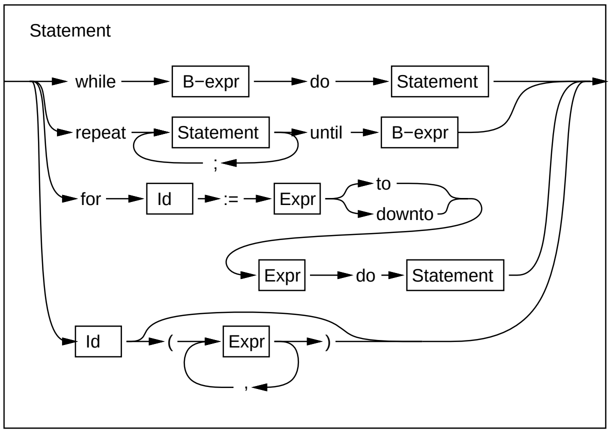
```
msc example;
  instance i1;
    out m0 to env;
    out m1 to i1;
    in m4 from i2;
  endinstance;
  instance i2;
    in m1 from i1;
    out m2 to i3;
    action a;
    out m4 to i1;
  endinstance;
```

```
instance i3;
  in m2 from i1;
  out m3 to i4;
endinstance;

instance i4;
  in m3 from i3;
  endinstance;
```

(variable names serve as edges)

Both have their advantages:



```
Statement ::=
  "while" B-expr "do" Statement |
  "repeat" Statement ( ";" Statement )* B-expr |
  "for" Id ":=" Expr ( "to" | "downto" )
    Expr "do" Statement |
  Id Parameters ;

Parameters ::=
  <empty> |
  "(" Expr ( "," Expr )* ")" ;
```

Definition VP, PV, VL

- **Program**: A set of statements that can be submitted as a unit to some computer system and used to direct the behavior of that system
- **Language**: the words and the methods of combining them used and understood by a considerable community (Webster)
- **Visual Programming Language**: Allows the user to specify a program in a two (or more) dimensional fashion.

  not VPL:

  - linear programming to define pictures (Postscript, X-11 toolkit, ...)
  - drawing packages which do not interpret the drawing (MacDraw, Idraw, ...)

- **Program Visualization**: Graphics are used to illustrate some aspect of a program or its run-time behavior

  Classification along two axis:

  - whether they illustrate the **code**, the **data** or the **algorithm**
  - whether they are **static** or **dynamic**

Every environment for a visual programming language applies graphical techniques to visualize the program.

We only use the term *Program Visualization* for textual programs.

A mapping between a visual sentence and its meaning

- Any picture can be represented by a digital image.

  A **digital image** i is a bi-dimensional string

  $$i : \{1, \ldots, r\} \times \{1, \ldots, c\} \rightarrow P$$

  with P the pixel alphabet.

- A **pictorial language** PL is a subset of the set I of possible digital images:

  $$PL \subseteq I$$

- A **description** d is a string of facts (attributed atoms) from some **Description Language** DL.

- An **interpretation** int of an image i is a function that gives its description:

  $$int : PL \rightarrow DL.$$

  An interpretation is specified by an observer of a picture, and is based on the **meaning** of a picture.

- The **materialisation** mat of a description d is a function that gives its visualization:

  $$mat : DL \rightarrow PL.$$

  A materialisation is the pictorial representation of a description.

- A **visual sentence** is a triple

  $$< i, d, < int, mat >> .$$

$< i, d, < int, mat >> :$

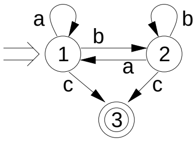
$$\begin{tikzcd} i \arrow[r, bend left, "int"] & d \arrow[l, bend left, "mat"] \end{tikzcd}$$

A visual sentence is called

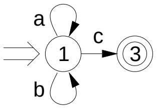
- **faithful** iff $i = mat(d)$.
- **non-faithful** if mat falls short: $mat(d) \neq i$
- **full** iff $d = int(i)$
- **non-full** if int falls short: $int(i) \neq d$
- **complete** iff it is **faithful** and **full**

A **visual language** is a set of visual sentences.

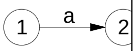
- **Full** but **non-faithful**
  - trivial: if the mat function pretty-prints the description d different from the original i
  - less trivial:

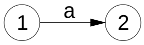
    $int($ 

    $) = (a|b)^*c$

    but $mat(\ (a|b)^*c\ ) =$ 

- **Faithful** but **non-full**: If the int function does not know the visual language,

  $mat(state(1), state(2), trans(1,2,a)) =$ 

  but $int($ 

  $) = circle(\ldots), text(\ldots), \ldots$

We usually limit ourselves to **complete** visual languages.

Still:

- **Different interpretations** might exist for a single picture i:

  $$vs_1 =< i, d_1, < int_1, mat_1 >>$$

  $$vs_2 =< i, d_2, < int_2, mat_2 >>$$

  For example, interpretation of a map depends on your intentions.

- **Different visualizations** might be used for a single description d:

  $$vs_1 =< i_1, d, < int_1, mat_1 >>$$

  $$vs_2 =< i_2, d, < int_2, mat_2 >>$$

  For example, to emphasize different aspects of d.

These were all non-constructive definitions.

- **Parsing** of visual sentences (necessary for visual programming) deals with defining an int function for a given visual language
- **Visualization** of data deals with defining an appropriate mat function for a given problem domain

A syntax directed graphical editor will have to keep i and d up-to-date with each other.

It will do so with help of int and mat.

Programming Paradigms

Visual implementations exists for many of the programming paradigms

- **Imperative** programming

  control-flow diagrams, Nassi-Shneiderman diagrams, VIPR, ...

- **Data-flow** programming

  LabView, Prograph, Show & Tell, ...

- **Constraint** based programming

  Sketchpad, Thinglab, ...

- **Rule-oriented** programming

  Vampire, many syntax definition formalisms, ...

- **Data structure definition**

  EER, class diagrams, ...

- **Event-based programming**

  Petri Nets, SDL, State Charts

- **Programming by Example**

  Pygmalion, Play, HI-visual

Many languages support more than one of the above paradigms.

Evaluation of the merits

There is **excitement** as well as **skepticism** about the prospects of VP and PV.

Textual programming seems to be more appropriate for **general-purpose programming**.

The key to success for a VL seems to be to find a good **application domain**.

Successes of VL

- helping to **teach** programming
- programming for **non-programmers**
- construction of **user-interfaces** by direct manipulation
- definition of **concurrent programs**
- visual **query** systems
- high level **design of data structures**

Problematic in VL

- **Scalability** and **abstraction**
- Lack of **formal specifications**
- Lack of **tools** to build environments
- **Poor design** of a VL
  - Visual $\neq$ better
  - Visual $\neq$ intuitive
  - A picture might be expressive, but is not precise
- **Portability** of programs
- **Integration** with textual programs

Examples of

Visual (Programming) Languages

Material:

- **Show and Tell: A visual programming language**, Kimura, Choi & Mack, in *VPE: Paradigms and Systems*, edited by Glinert.
- **The programming language aspects of ThingLab**, Borning, ACM TOPLAS, vol 3, no 4, 1981.
- **Lesson learned in the Trenches**, Graf, VL'90

Programming Paradigms

Visual implementations exists for many of the programming paradigms

- **Imperative** programming +

  control-flow diagrams, Nassi-Shneiderman diagrams, VIPR, ...

- **Data-flow** programming +

  LabView, Prograph, Show & Tell, ...

- **Constraint** based programming +

  Sketchpad, ThingLab, ...

- **Rule-oriented** programming −

  Vampire, many syntax definition formalisms, ...

- **Data structure definition** −

  EER, class diagrams, ...

- **State-Transition based programming** +

  Petri Nets, SDL, State Charts

- **Programming by Example** −

  Pygmalion, Play, HI-visual

Many languages support more than one of the above paradigms.

Imperative Visual languages or Control-Flow languages

Explicitly depicts the **flow of control** of an **imperative** program.

- terminals: instructions
- connections: flow of control

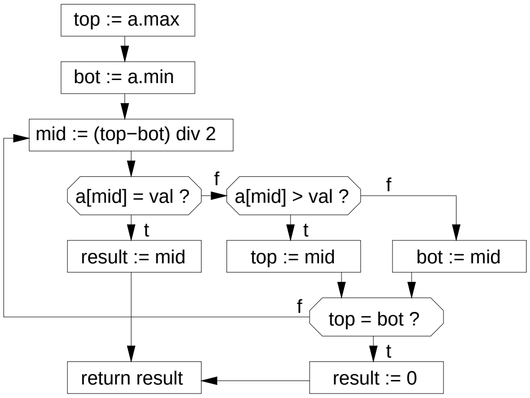
Mainly used to visualize programs, but also as visual programming language.

Problematic

- supports unstructured programming
- neglects the data structures
- textual forms turns out to be superior

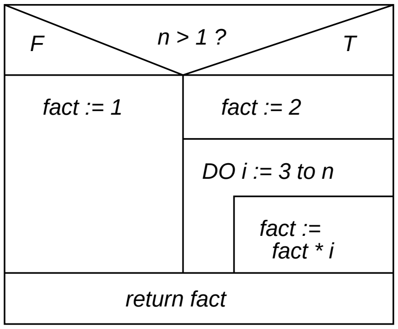
Examples

- **basic control flow diagrams**

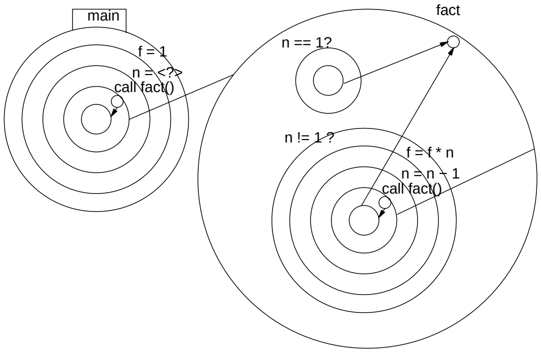


- **Nassi-Shneiderman diagrams**

Only allows well-structured programs by replacing the arrows (goto's) by containment relations.



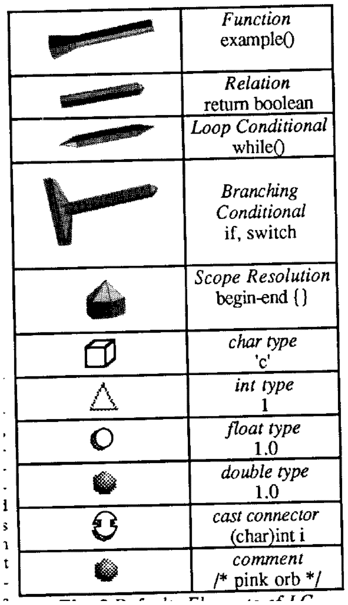
- **VIPR** (Citrin, VL'94)



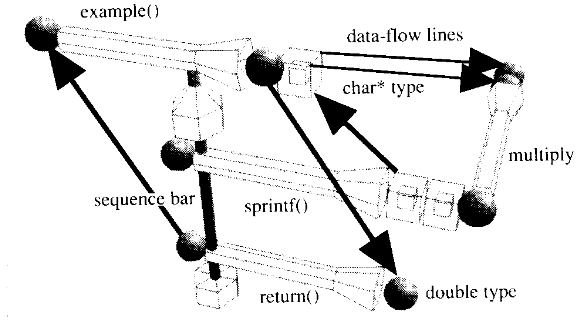
```
void main() {
  f = 1;
  n = <?>; /* assign n */
  fact();
}
void fact() {
  if (n == 1) return
  else {
    f = f * n;
    n = n - 1;
    fact();
}}
```

- **Lingua Graphica** (Styles&Pontecorvo, VL'92)

Developed at Lockheed.



	Function example()
	Relation return boolean
	Loop Conditional while()
	Branching Conditional if, switch
	Scope Resolution begin-end {}
	char type 'c'
△	*int type* 1
○	*float type* 1.0
●	*double type* 1.0
	cast connector (char)int i
●	*comment* /* pink orb */



```
double example(double num, char *str) {
  sprintf(str, "%f", (num * num));
  return(num);
}
```

Looks fancy, but the textual form is easier to comprehend.

The example contains mistakes (input for the multiply; direction of data flow arrow for `str`; constant value `"%f"` is not visible)

Data-Flow programming

Program is represented by a *directed graph*. The nodes are functions. The data flows in and out of the functions through the edges.

Flow of control depends on the availability of data (this gives intrinsic concurrency).

Well fit for data transformations problems.

To make it a more general programming language, you need to add constructs for conditionals, iteration, data structures, ...

Large library of pre-defined functions is a necessity.

The visual representation is very appropriate.

VPF (Miyao et al., VL'89)

Visualized Programming Environment for Form Manipulation Language

Special purpose programming environment for office information processing via **forms** based on **dataflow**.

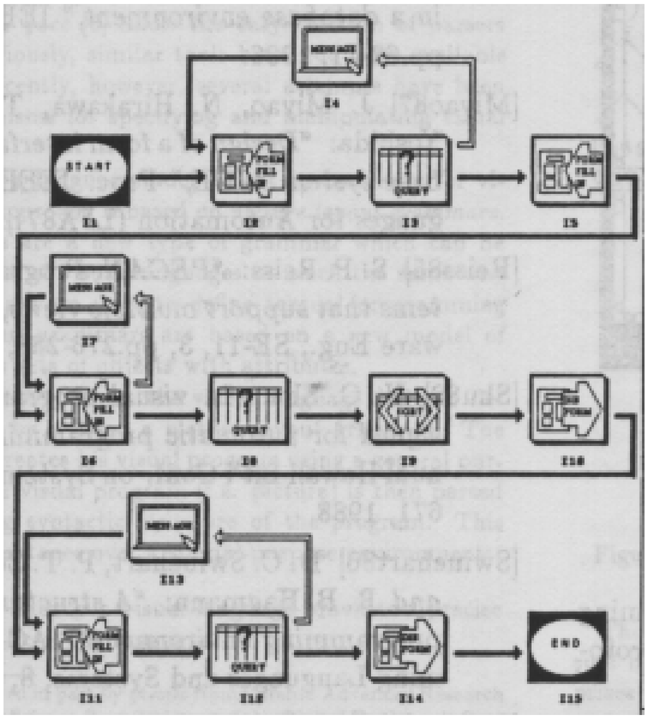
- 8 different node kinds:

  Start, *End*, *Message*, *Get*, *Sort*, *Form-fill-in* (to consult the user), *Query* (to consult the database), *Display-form* (to inform the user)

  Details of the node actions are defined in a *Form Definition Language* (user interface definition & database query language).

- 2 kind of edges: *normal* edges (black) and *error* edges (white)

- Each node has
  - single outgoing **normal** edge (but End)
  - possibly outgoing **error** edge (default to End)
  - one or more incoming edges (but Start)
- If the operation succeeds then the execution proceeds along normal edge, otherwise along the error edge



- **Programming environment** of VPF
  - Specialized editors for the different aspects
  - Mode-less operation: editing of program during execution
  - Debugging: execution possible from arbitrary icon
  - Automatic re-execution on modification of the input
- **Evaluation** of VPF
  - quite restricted application area
  - conditionals and iteration are handled via underlying actions
  - dataflow with the two kind of edges is very clear

Show and Tell Kimura et al, '90

Visual programming language based on dataflow, intended to teach programming to children

Basis construct is an (acyclic) **box-graph**.

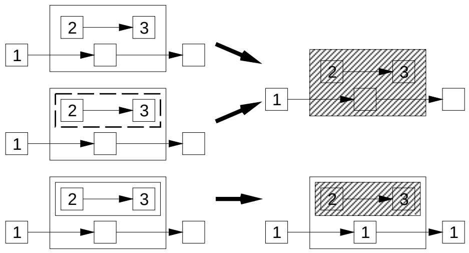
Boxes:

- may contain a **value** (integer, string, bitmap)
- may contain a *pre-defined* or *user-defined* **function**
- may be **empty** (data value has to be filled in by transfer of data values from neighboring boxes)
- may be **complex** and contain a box graph

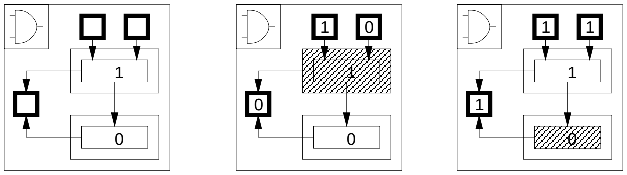
- The notion of **consistency** gives **switching** capability:
  - A box graph is **inconsistent** if two *directly* connected boxes contain *different* values.
  - Complex boxes may be **open** or **closed**:

    open: inconsistency spreads through the border to the context

    closed: inconsistency is kept within.
  - An inconsistent box is considered to be **non-existent**



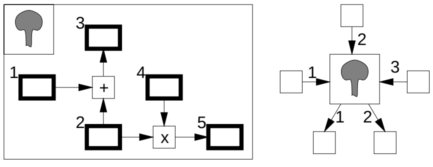
Definition of the AND function:



- **Procedural Abstraction**

  A box graph may be identified with an icon

  Binding of the parameters is done via ordering of the (x, y) positions of their boxes



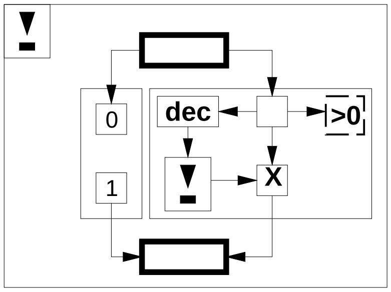
Name definition

Name use

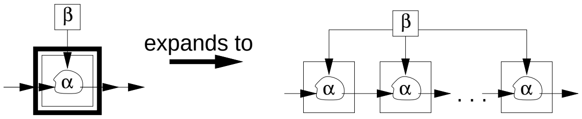
Input parameters: 1, 2, 4
Output parameters: 3, 5

Binding:
in 1 to parameter 1
in 2 to parameter 2
in 3 to parameter 4
out 1 to parameter 3
out 2 to parameter 5

- **Recursion** Factorial:



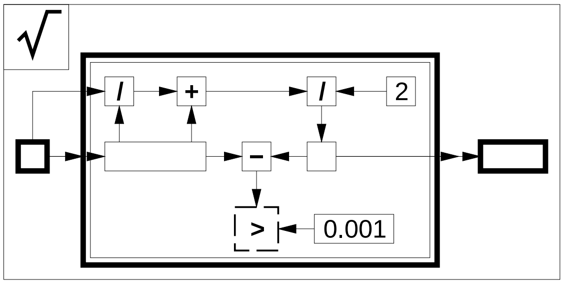
- **Iteration**



  until one of the expansions leads to an inconsistent box

Example: Newton approximation for the square root

```
newton(a):
  x := a;
  repeat
    y := x;
    x := (a/y + y) / 2;
  until y - x <= 0.001;
  return x;
```



- Show and Tell is intended for teaching:
  - visual syntax shows concepts like *subroutine*, *dataflow*, *iteration*, ...
  - value oriented programming
  - it has no variables
  - inherent concurrency
  - notion of consistency instead of Booleans
  - visualized execution in terms of the program

  But, will it allow children to program?

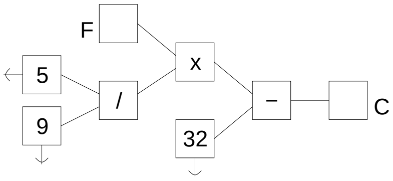
- Show and Tell offers enough constructs to be a general purpose programming language

  It does not have constructs to define datastructures, though.

Constraint-based programming

Generalization of data-flow languages: undirected dataflow, declarative.

- Program is a bi-partite graph:
  - constant and variable nodes for **values**
  - connected via **undirected edges** to
  - function nodes which represent **constraints**



Constraint solving must assign values to the *variable* nodes such that all **constraints are satisfied**.

Typical problem setting:

- I have a variable x and a screen on which the value of x is displayed

- Programmer must remember that the screen must be updated if x changes, and that x must be updated if screen representation is edited

One can **specify** this in a constraint-oriented system.

The system then ensures that the constraint is maintained.

ThingLab (Borning81)

Environment for constructing dynamic models of experiments in geometry, electric circuits, mechanical linkages, ...

Build on top of Smalltalk, object oriented, constraint based.

- **object**: defines constraints between its parts; may consist of other objects; different objects may share parts.

- A **constraint** consists of
  - **rule**: test whether the constraint holds
  - **methods**: ways to satisfy the constraint

  User provides the methods, the system chooses a way to apply them.

Primitive objects are pre-defined in Smalltalk.

```
class MidPointLine
  Superclasses
    GeometricObject
  Part Descriptions
    line: a line
    midpoint: a point
  Constraints
    midpoint = (line point1 + line point2)/2
      midpoint <- (line point1 + line point2)/2
      line point1 <- midpoint * 2 - line point2
      line point2 <- midpoint *2 - line point1
```

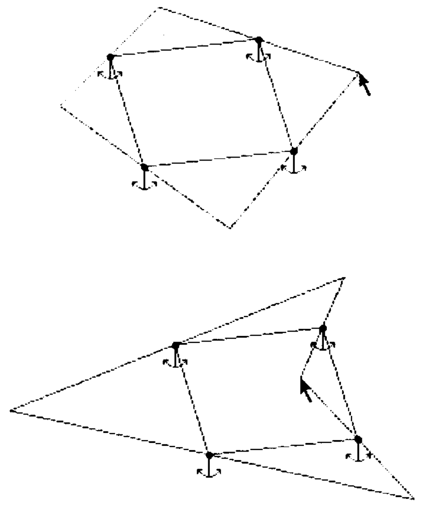
- *Points* which react to other points by merging

- *Operations* which can be attached to value and variable holders

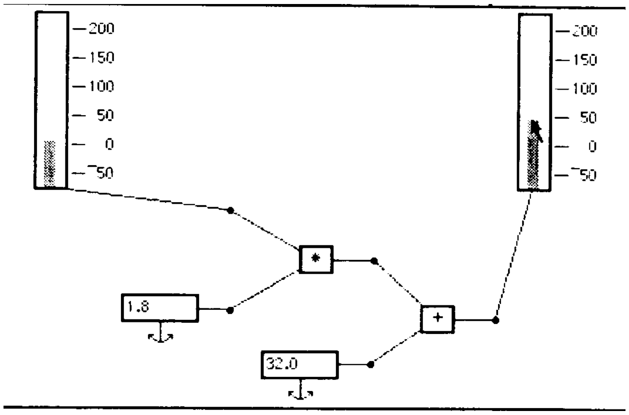
- *Electrical objects* (wire, resistor) which can be attached and obey Ohm's law

Complex objects and their constraints are edited visually by combining primitive objects.

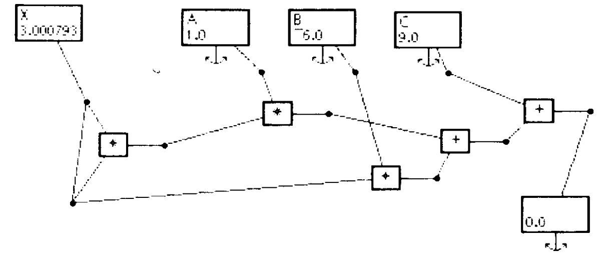
Experiments with geometric laws:



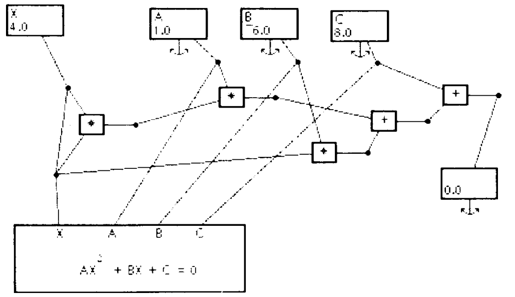
Fahrenheit-Celsius converter



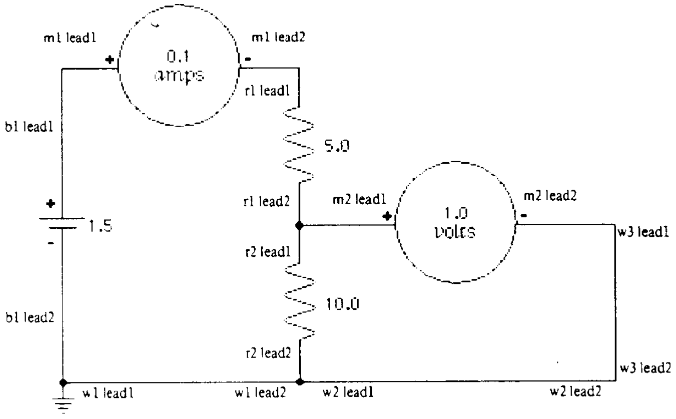
Quadratic Equation network for $ax^2+bx+c=0$



Needs *relaxation* to approximate the solution; Extension with a Quadratic solver:



A voltage divider



How does the constraint solver work?

- Certain values are “anchored” (constants)
- Last changed variable is also anchored
- All n-variable constraints for which $n-1$ values are known are solved by computing the n-th value
- All remaining variables are approximated through *relaxation* by minimizing the error function.

Constraint solver might pre-compute a plan.

Constraints may be unsolvable.

Constraint solving is inefficient as soon as relaxation must be applied.

State-Transition networks

Program is a directed graph of **states** which are connected by **transitions**.

This represents a **process**.

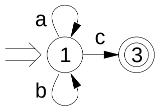
The transition to another state is triggered by an input signal.

Each process has a **current state**.

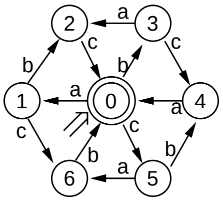
The current state of the process determines which input signals are acceptable and where to go on it.

Finite State Automata

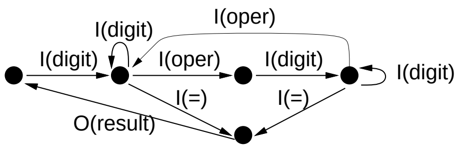
Very useful to define regular expressions.



- $(a|b)^*c$:



- $(abc|acb|bca|bac|cab|cba)^*$:
- Interface of a simple calculator:



SDL

SDL is intended to specify the **behavior** and **internal structure** of *real-time*, *interactive*, *concurrent* and *distributed* systems.

SDL has been developed to specify telecommunication system.

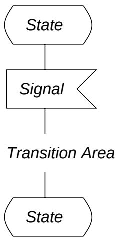
A system is described as a number of extended finite state machines: **process instances**.

These communicate with each other and with the environment of the system by exchanging **messages**.

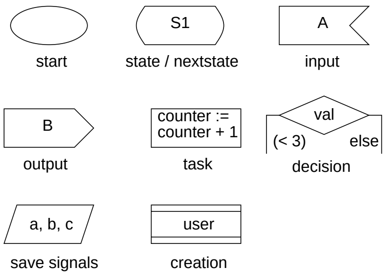
Each process instance is in a certain state. On accepting a message it makes a **transition** to another state.

During the transition it can *perform computations*, *update variables*, *make decisions*, *create new processes*, *send signals*, ...

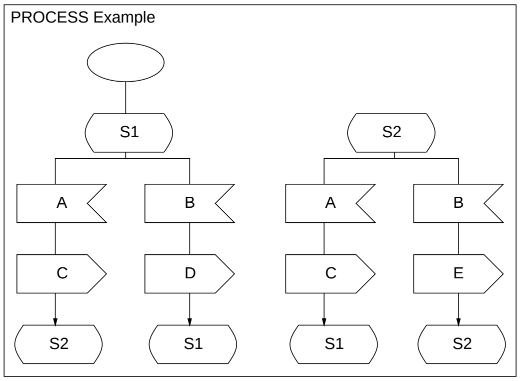
States and transitions:



The basic constructs of SDL:



An example of a simple process



Reacts on A with C.

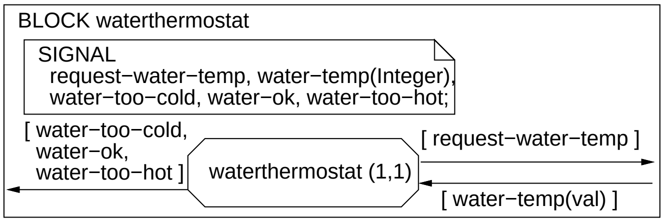
Reacts on B with D or E, depending on the number of input messages received.

A specification of a central heather

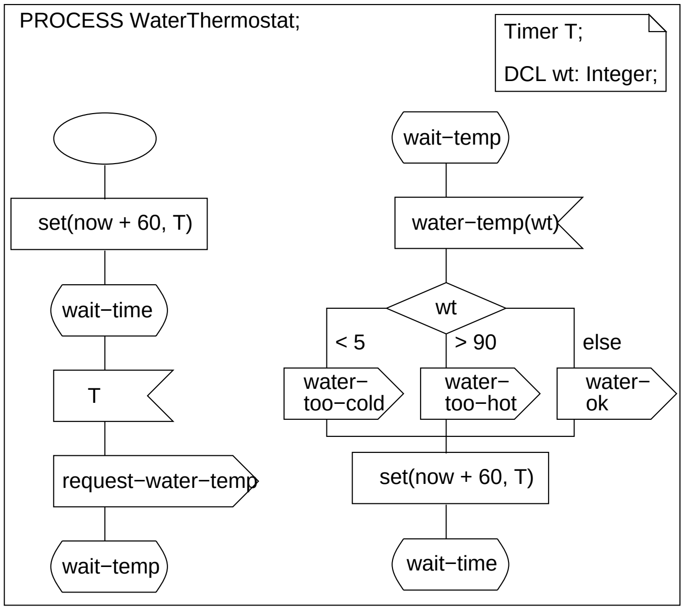
Informs the environment of the water temperature.

Uses an external process to perform the actual measurement

The definition of the process block:



```
SIGNAL
  request−water−temp, water−temp(Integer),
  water−too−cold, water−ok, water−too−hot;
```



```
Timer T;

DCL wt: Integer;
```

Informs the environment every 60 seconds of the water temperature.

SDL is quite expressive and reasonably easy to read.

SDL is quite low-level; it somehow resembles "*control-flow diagrams*".

SDL is a language of engineers.

Whether the graphical representation is appropriate for the definition of large systems is questionable.

Implementations can be generated automatically from an SDL definition.

Overview and Conclusions

I have presented a load of visual languages and environments.

Some of these are successful, others only interesting, some downright bad.

What makes a **Visual** Language a **Good** Language?

Mike Graf presents **6 lessons learned in the trenches of VL design**

- 1 - Have one unifying principle

  Sound & extendible

  A good example is Show and Tell's *consistency*

- 2 - Keep it simple

  don't use tons of different looking icons and connections

- 3 - Use animation to visualize the execution

- 4 - Design the human interface very well

  target the system to the intended users, involve them

- 5 - Quit while winning

  don't try to extend a good special purpose language into a lousy general purpose language

- 6 - First make it useful, then worry about the formal grammar

  The semantics matter, not the syntax

Design Issues in Data Flow Languages

Material:

- Visual Languages and Computing Survey: **Data Flow Visual Programming Languages**, D.D. Hils, JVLC 3, 1992.

- **Psh** – *The next generation of command line interfaces*, Glaser & Smedley, VL'95

Data Flow

A view of computation which shows the data flowing from one filter to another, being transformed as it goes.

Data flow languages are **naturally visual**.

The paradigm easily accommodates insertion of **viewers** at all point in the data flow.

The program is represented by a **directed graph** in which

- **Nodes** represent **functions**
- **Arcs** represent **flow of data** between functions
  - Arcs **in to** a node: input for the function
  - Arcs **out of** a node: output of the function

Data flow implementations

Hils describes 15 data flow languages. These are the most interesting ones:

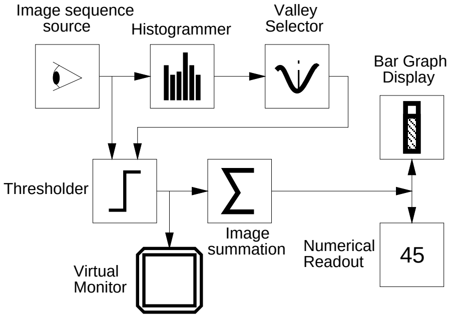
- Show and Tell (see sheets last week)
- VIVA
- Cantata
- LabView
- ProGraph

VIVA

Language for image processing.

Three kind of boxes: sources, operators, monitors.

Liveness is a goal for VIVA: it recomputes on edits, but also if new streams of input become available.



Cantata

Originally intended for image processing, but also applied for signal processing, query languages, matrix algorithms, ...

Provides library of some 240 algorithms for signal and image processing.

Iteration via control-flow constructs: a special icon in a cyclic data flow that directs the flow according to some condition.

Has procedural abstraction, but lowest level is written in C and Fortran.

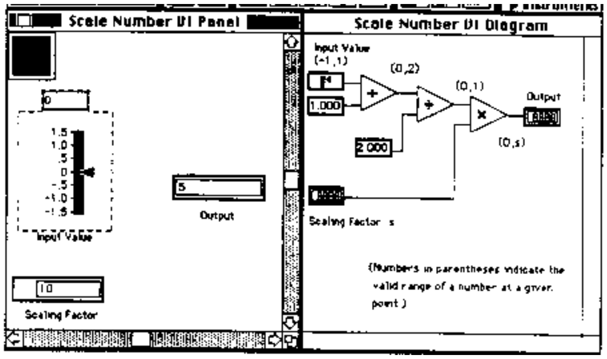
Uses a hybrid data-driven and demand driven execution model.

LabView

Intended for the collection and analysis of data from laboratory instruments.

Allows non-programmers to build **virtual instruments**: design of the front panel definition and the data flow diagram that underlies it.

Commercially very successful.

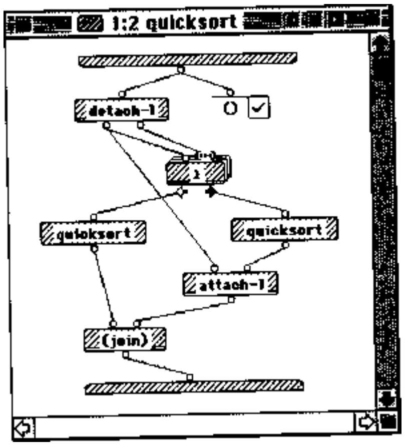


ProGraph

Intended as general purpose programming language.

Support basic object orientation.

Extended with many control-flow constructs to facilitate the programming.



Extensions to pure data flow

A data flow language is called **powerful** if a user can tackle large and complex problems with it (within its application domain).

Necessary (and sufficient?) **extensions** for a data flow language to become **powerful**:

- A large (and extendible) collection of **predefined functions**
- **Procedural abstraction**
- **Conditional choice**
- **Iteration**

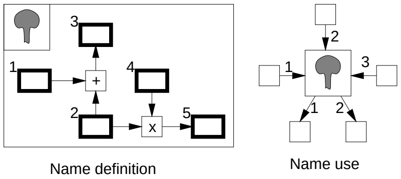
Procedural Abstraction

Fits easily in data flow model:

Define a procedure by providing a subgraph with a name (or an icon).

Use a procedure by applying a node with the same name.

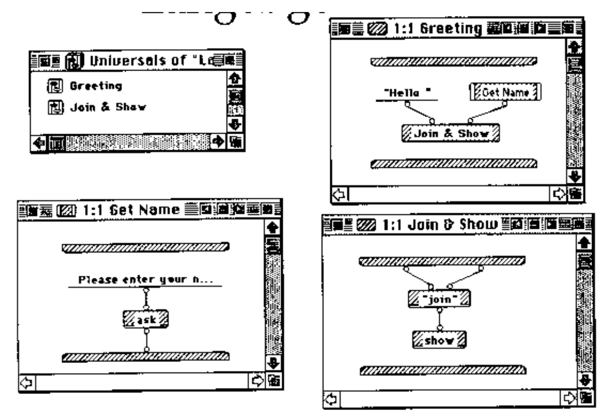
Parameter binding is quite complicated in *Show and Tell*.



Input parameters: 1, 2, 4
Output parameters: 3, 5

Binding:
in 1 to parameter 1
in 2 to parameter 2
in 3 to parameter 4
out 1 to parameter 3
out 2 to parameter 5

ProGraph uses input and output *lines* which make parameter binding much more straightforward.



Conditional choice

- **Normal** and **error** edges in VPL
- The notion of **consistency** in Show&Tell
- **Distributor** and **Selector** concept
  - **Selector**:

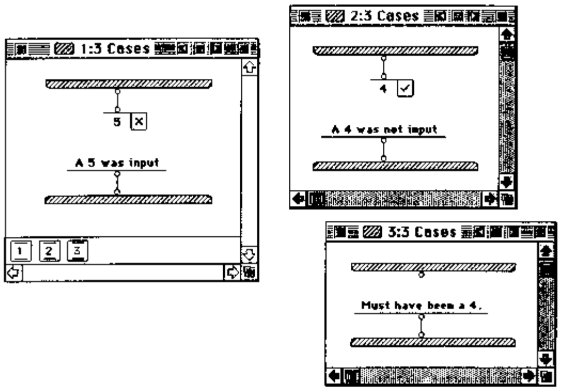
    Has input streams i_1, i_2 and c, and an output stream o.

    If c contains `true`, then a token from i_1 is propagated to o, otherwise from i_2.
  - **Distributor**:

    Has input streams i and c, and output streams o_1 and o_2.

    On the value on c, i is propagated to o_1 or o_2.

- **Case** concept in ProGraph and LabView



  - X: Next case on failure
  - V: Next case on success

Iteration / Single Assignment

Declarative visual languages, such as *data flow*, *functional* and *logic*, have **single assignment** semantics as a basic rule: *once a variable is set, it keeps that value.*

This makes programs easier to understand, to visualize, to implement.

Iteration: repeat the execution of a body of code, usually for repeated modification of the same variable.

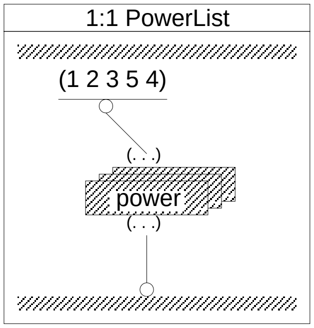
So the two notions potentially **clash**.

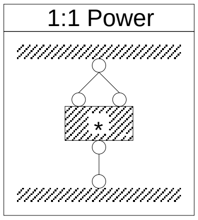
Horizontally Parallel Iteration

If the outcome of one cycle does **not affect** the outcome of the next cycle, then we speak of **horizontally parallel iteration**.

The computations are then independent, and there is **no conflict** with the single assignment rule.

ProGraph offers a special construct for this kind of iteration.





PowerList produces
(1 4 9 25 16)

Temporally Dependent Iteration

If the outcome of one cycle **depends** on outcome of earlier cycles, then we speak of **temporally dependent iteration**.

This is the most common use of iteration.

Compute n^{th} Fibonacci number:

```
fib := 1;
pred := 0;
for count := n downto 0 do
  begin
    pred := fib;
    fib := fib+pred
  end
```

The variables `count`, `fib` and `pred` violate the single assignment rule.

There are many different ways to offer this kind of iteration within the data flow framework.

Some solutions

- Offer iteration by means of **recursion**

  Does not violate the single assignment rule as every call creates new versions of the variables.

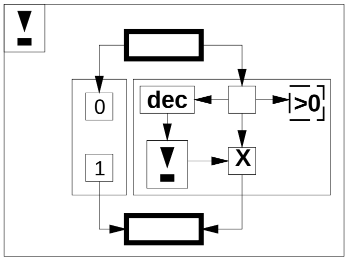
  - **Recursion**: For the i^{th} recursion, the output of the body out_i is used to compute out_{i-1}.

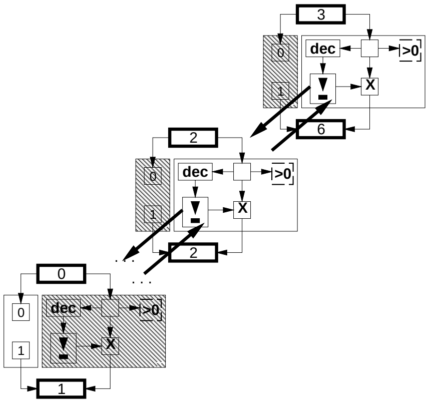
    out_0 is mapped to the final out.

  - **Iteration**: out_n is directly mapped to out, without additional computations.

  Some algorithms are most naturally expressed iteratively.

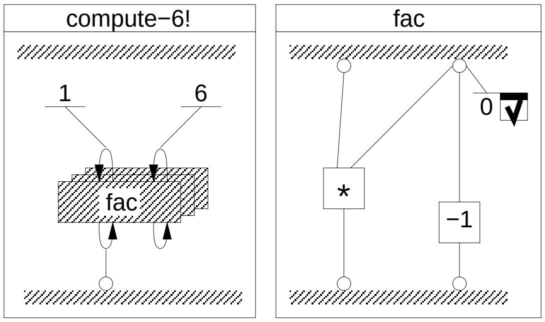
Factorial recursively in *Show and Tell*:



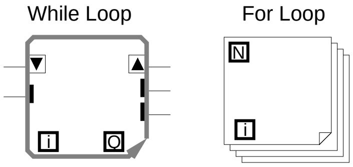


- **Bending** the single assignment rule for loop variables by introducing special **control-flow constructs**

  Factorial in ProGraph:



Special Loop nodes in LabView:



i = loop counter
Q = recirculation condition
N = total loop count

- **Temporal assignment**

  A variable is attached to a *stream of values* generated over time.

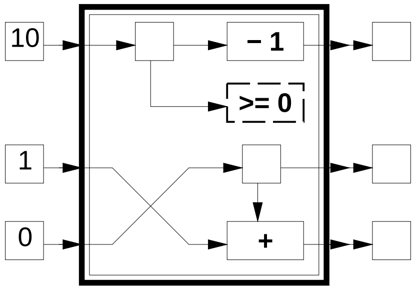
  The name then refers to the *most recent* value in this stream.

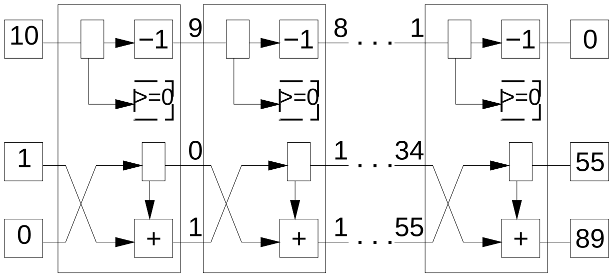
  Access to previous values by constructs such as: `thirdvalue := n attime 3`

  This *stream of values* approach fits well in the data flow paradigm.

  Show and Tell: only loop variables of the *previous* cycle are available in *current* cycle.

Computing the Fibonacci numbers in Show and Tell:





Execution model

A function **may** be executed as soon as all its input is available.

- **Data driven** execution:

  it will do so

- **Demand driven** execution:

  it will do so **when** its output is requested.

  Is more efficient.

  The user must indicate which output is interesting

  Only *Cantata* offers demand-driven execution as an option

If more than one function is to be executed, one is selected to run, or all can be run in parallel.

Level of Liveness

Tanimoto distinguishes four levels of **liveness** for visual languages:

- **1: Informative**: Visual representation is not used as instruction for a computer.
- **2: 1 + Significant**: Visual representation is executable. User gives command to execute it. (most VPL)
- **3: 2 + Responsive**: Executes automatically whenever input data or program is edited by the user. (some VPL)
- **4: 3 + Live**: Updates display continuously. For example: live video processing. (*VIVA*)

Progressive operations

A VL at level 4 offers real-time display of partially computed output values.

For example, partial executions of operations on images or sound.

Progressive operator: produces successive approximations to the correct output.

Ordinary operation: $f(x) \to y$

Progressive version: $\phi(x) \to y_1, y_2, \ldots$, which converges to y:

$$\lim_{t \to \infty} \phi_t(x) = f(x)$$

Criteria in the design of progressive operators:

- Visual quality of the y_t

  The approximation should be smoothly, not:

  $$0, 0, \ldots, 0, f(x), f(x), \ldots$$

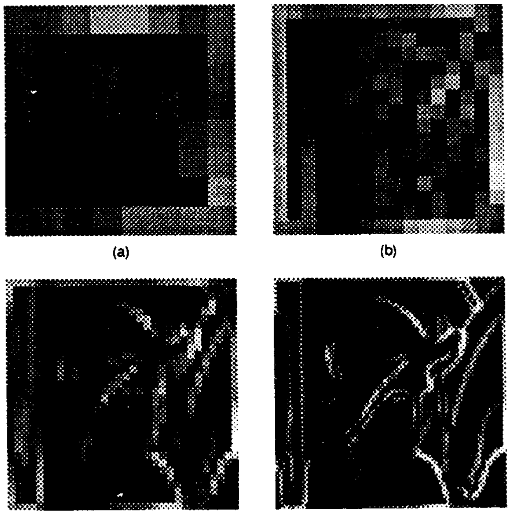
- Convergence time

  What fraction of the time taken by f does it take ϕ to reach 90% quality?

- Introduced overhead

  How much more time does it take ϕ than f to reach 100% quality?

Example: progressive **edge finding** by applying operation on $8 \times 8, 16 \times 16, \ldots$ versions of $n \times n$ bitmap:



Quality improvement: use info of level i to determine which square at level $i+1$ to handle first.

Conclusions

- The data flow paradigm is *a very simple and powerful concept*, with an intuitive visual representation.
- Data flow VPL's have been most successful for novice programmers and for specialized application domains.
- The data flow paradigm is very appropriate if application domain centers on *data manipulation* and *data transformation* (particularly image processing).
- The introduction of *progressive operators* is an interesting extension to the basic data flow paradigm.

- To make a data flow language really *useful* one needs to **extend** it with conditionals, iteration, abstraction, ...

  The commercially successful languages (ProGraph, LabView, Cantata) have been extended with all kind of **control-flow** constructs.

  Show and Tell tried harder to stay within the data flow realm.

  Show and Tell thus has fewer constructs, but whether this makes programs easier to write or inspect...?

Syntax Directed Editing of Visual Programs

Material:

- *A Definition of the Graphical Syntax of Basic Message Sequence Charts*, by J. Rekers.

Syntax Directed Editing

- What is Syntax Directed Editing?
- Syntax definition of MSC
  - What is MSC?
  - Three representations for a diagram
  - Grammar formalism
  - High level grammar
  - Low level grammar
  - Constraint definition
- Generating a syntax directed editor
- Overview

What is a Syntax Directed Graphical Editor?

Provides a **language specific** environment to **create**, to **edit**, and to **evaluate** a **graphical program**.

- For most **textual** languages, the editing environment is a simple **general text editor**.

  The user sends the program file to the compiler or interpreter in order to be evaluated.

  This provides little language specific support, much editing freedom, and a uniform interface among many languages.

  Syntax-directed editors have been a *failure* for textual languages.

- The visual counterpart of this situation would be to draw a diagram in a general graphical editor such as `Idraw`.

  The user sends the diagram on file to an appropriate interpreter, which reads and executes it.

  Little support, much freedom, uniform interface.

- However, for visual programming a **closer integration** between the *editing* and *execution* environment seems to be desirable.

  It then becomes possible to provide *language specific editing commands*, and to *check* and *execute* the program *during* editing.

  ⇒ environment of liveness level 3 or 4 becomes possible.

Classes of editing commands

We zoom in to the commands to build and edit a diagram:

We distinguish three kinds of edit commands:

1. **syntax directed editing**

   commands to insert, delete or change **entire language constructs** in a syntax directed fashion

2. **layout editing**

   commands to **change the layout** of a diagram

3. **free editing**

   commands to edit the underlying **graphical objects** directly

1: Syntax Directed Editing

- Syntax directed **insertion** of entire language constructs

  Examples for an editor of EER diagrams:

  - *Insert a binary relationship between these two entities*

  - *Add an attribute to this entity*

  This requires two pieces of information

  - **which** construct should be inserted?

  - **where** should it be attached to the rest?

  (in your MSC editor, these two issues have been separated by the notion *sockets*)

- Syntax directed **deletion** of entire language constructs

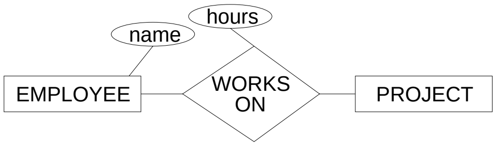
  Examples for an editor of EER diagrams:

  – *Delete this attribute*

  – *Remove this entity*

  Only requires that the user selects the construct

  The main question is: *what to do with connected constructs*?



- Syntax directed **changing** of language constructs

  Examples for an editor of EER diagrams:

  – *Make this attribute a key*

  – *Assemble these attributes in a composite attribute*

  – *Make this a weak entity*

  It is very **hard to predict** which change commands users will find useful.

  These change commands are **not strictly necessary**: every *change* can be achieved by a sequence of *delete* and *insert* commands.

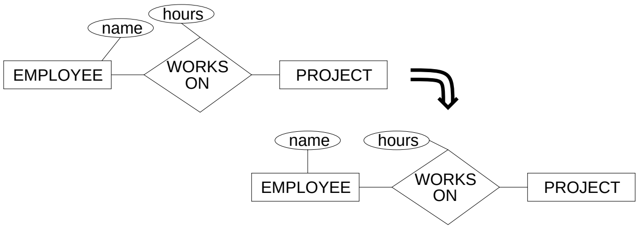
A common property of all syntax directed editing commands is that the diagram is kept **syntactically correct** at all times.

2: Layout Editing

The *syntax directed editing commands* choose some **default layout** for the construct they insert.

The user might prefer a different layout which is **semantically equivalent**.

⇒ the user should be able to change a diagram through direct manipulation such that its *interpretation* remains the same.

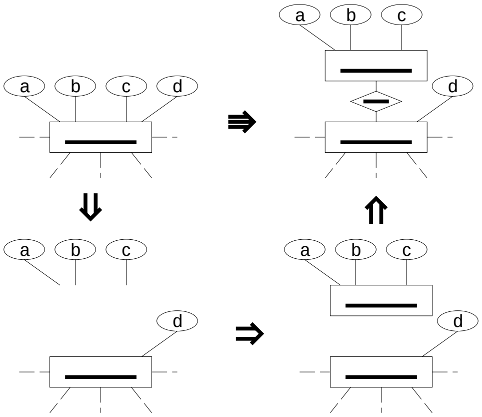


The use of **constraints** will be very helpful to implement such behavior.

3: Free Editing

The previous two kinds of editing commands keep the diagram syntactically correct at all times.

This can be very restrictive and might easily force the user to construct a certain diagram in a roundabout way.



Possible ways to alleviate this problem:

1. Anticipate the desired kind of modifications and introduce a **syntax directed change** command for every one of them.

   This solution is used in *DiaGen*. Easily makes a specification 5 times as large.

2. Temporarily **allow invalid diagrams** by allowing the user to edit the underlying graphical objects directly.

   Rediscover the structure at a certain moment

   The latter is called **graphical parsing**.

I prefer the second way.

Specific vs. Common behavior

There are many more standard editor commands which I did not discuss.

All commands are partly language specific, and partly common among different visual languages.

Users do not use a single language only: it would be very desirable if common behavior would be provided in a uniform way.

A syntax directed editor for a visual language could be:

- **programmed by hand**

  + can be very specific for the language,
  − little uniformity,
  − much low-level implementation work.

- **generated from a specification**

  − less specific,
  + more uniformity among different languages,
  + re-uses software,
  − generator must be implemented

My preference: **generation**

Specification of Visual Syntax

The visual syntax specification defines the **language dependend part** of the different editor commands.

It defines:

- the constructs used in the **syntax directed editing commands**
- the constraints which must be maintained during **layout editing**
- the structure which has to be discovered by the **graphical parser**

How to specify this syntax?

Here we use **graph grammars** as specification formalism.

The running example: MSC

Message Sequence Chart (MSC) is a graphical language for the description of interaction between entities.

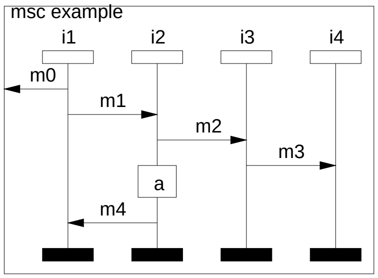
A diagram in MSC consists of **process instances** which exchange **messages** and perform **internal actions**.

An MSC depicts an **execution trace** of a system.

Having a number of such traces helps in the construction of an SDL definition of the system.

MSC has been standardized by the ITU-T (former CCITT).

An example of an MSC:



Messages are send **asynchronously**.

Time proceeds downwards **within** a process instance.

Timing relation **between** instances: a message has to be send before it can be received.

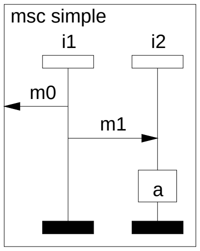
Representations

In my approach, there are **three representations** for MSC diagrams

- **Graphical Objects** (lowest level)
- **Spatial Relations**
- **Abstract Relations** (highest level)

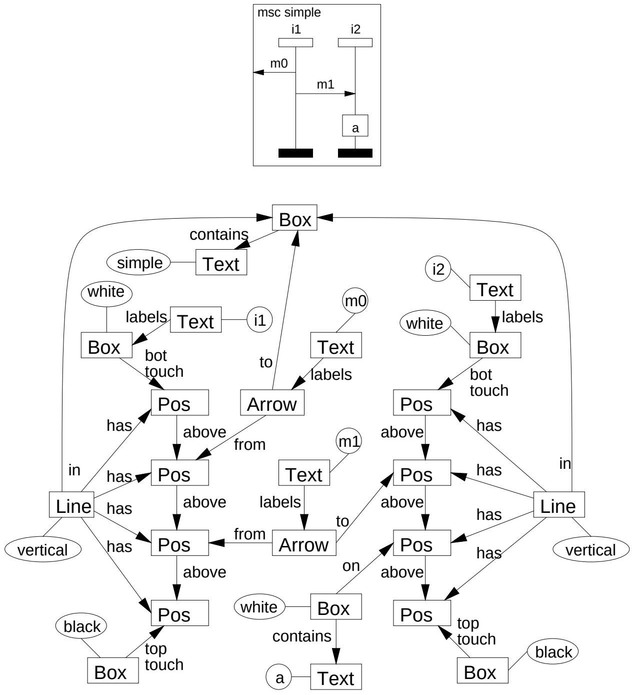
This distinction will be useful for any graphical language for which you want to define the graphical syntax.

The **graphical object** representation of

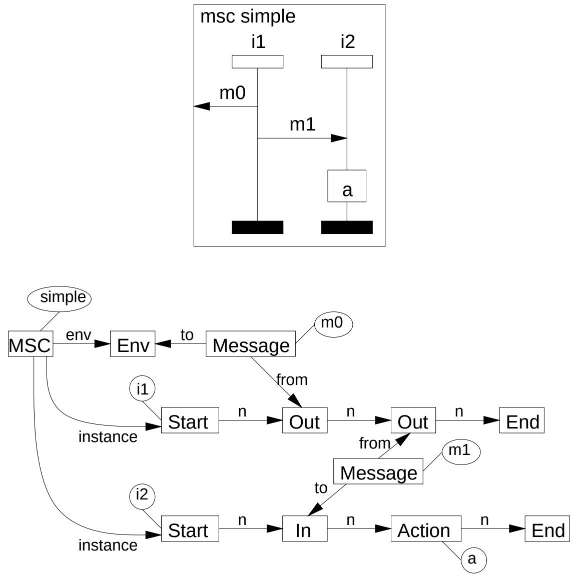


```
object(kind='Line', start=(245, 455), end=(245, 379))
object(kind='Text', ul=(197, 472), textitems=['i1'])
object(kind='Text', ul=(242, 472), textitems=['i2'])
object(kind='Rect', ll=(188, 455), ur=(213, 462))
object(kind='Rect', ll=(188, 372), ur=(213, 379))
object(kind='Rect', ll=(232, 455), ur=(258, 462))
object(kind='Rect', ll=(232, 372), ur=(258, 379))
object(kind='Line', arrowatend=1, start=(200, 436),
end=(168, 436))
. . .
```

The **spatial relation** representation of



The **abstract relation** representation of



- All **semantical processing** of MSC diagrams will be performed at the level of **abstract relations**.

  This is how a user **thinks** about a MSC diagram.

- The **display machinery** works at the level of the **graphical objects**.

  This is what the user **sees**.

- The **spatial relations** serve as **intermediate level**, to facilitate the necessary mapping between the two extremes.

  The **spatial relations** abstract from the actual positions by only coding the **constraints** which should hold between the graphical objects

The MSC syntax definition

The **abstract relations** representation and the **spatial relations** representation are graphs.

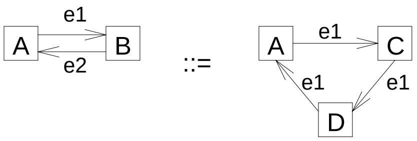
Only certain graphs represent MSC diagrams.

I will use a **graph grammar** to define the language of correct graphs at both levels.

The graph grammar formalism

A graph grammar consists of a number of **productions**.

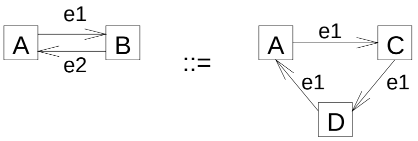
Each production is of the form $L ::= R$

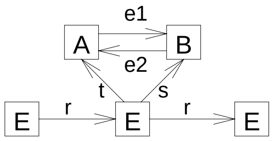


If the host graph G contains a subgraph which matches L, then the production is **applicable**.

A production is **applied** by replacing the subgraph matched by L by a copy of R.

The application of a production may **not** cause **dangling edges**.



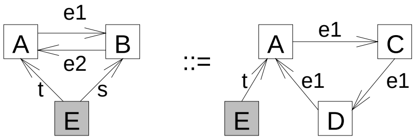


Applying

to the graph

would make edges labeled by s and t dangling, and the production is **not applicable**.

This production is **applicable**:

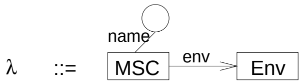


It uses grey **context elements** which have to be present, but are not affected by the application.

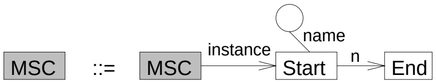
The Abstract Relations Grammar

Defines the rules by which correct **abstract relations graphs** may be **constructed**.

- p1: The Axiom production



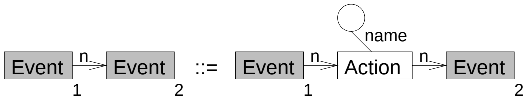
- p2: Extending a MSC with a process instance



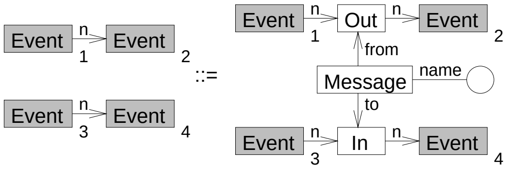
- We need the following generalization:

$Event > \{Start, In, Out, Action, End\}$

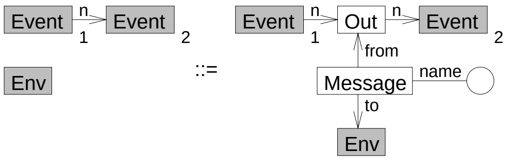
- p3: Insertion of an internal action **in** a process instance



- p4: Insertion of a message **between** two process instances

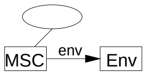


- p5: A message to the environment

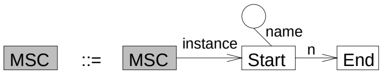


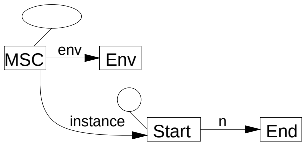
Example

Start with:



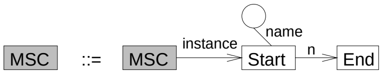
Apply p2:

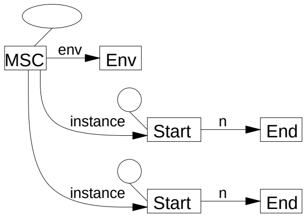




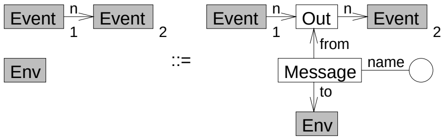
⇒

Apply p2:

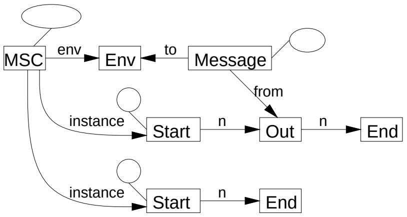




⇒

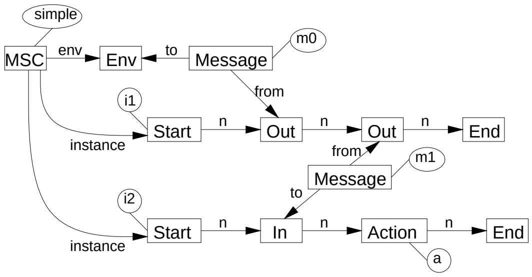


Apply p5:



⇒

Apply p4 and p3 to arrive at:



(other sequences of production applications are also possible to construct the same result graph)

The spatial relations graph

We now have a way to obtain correct MSC abstract relations graphs.

However, these graphs do not resemble MSC diagrams at all.

We need a second (low level) representation which states what the different constructs **look like**.

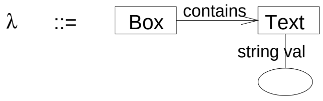
This is done by creating a **spatial relations graph** which represents a MSC in terms of **graphical objects** and the **relations** which should hold between these objects.

The spatial relations graph is not intended for human inspection, only as intermediate representation.

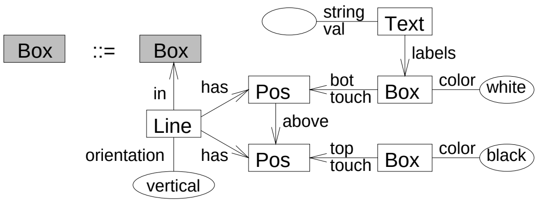
The Spatial Relations Grammar

Idea: attach to each **abstract relation** production a corresponding **spatial relation** production.

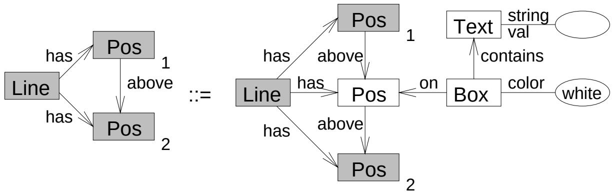
- p1: The axiom production



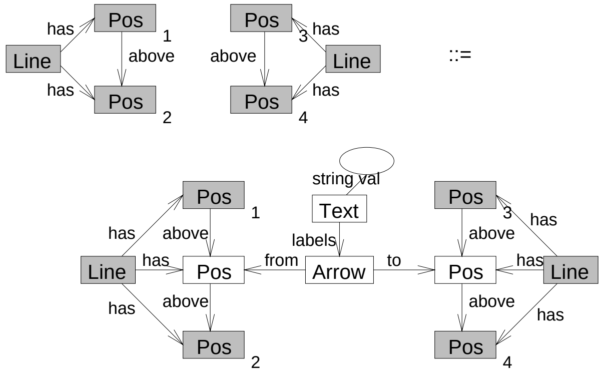
- p2: Extending a MSC with a process instance



- p3: An internal action in a process instance



- p4: A message between two process instances

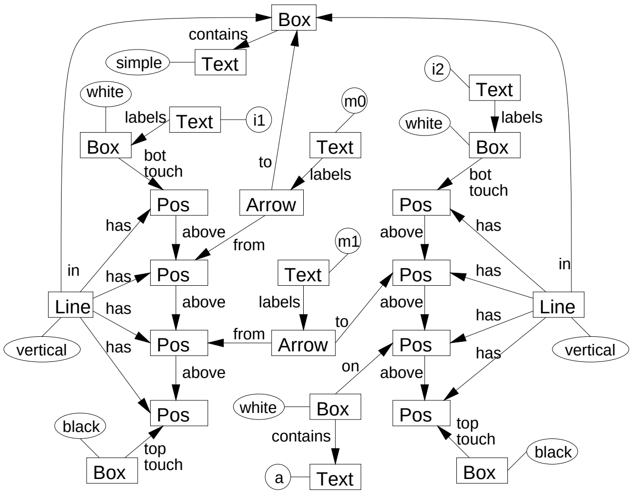


- p5, p6: More of the same.

I still need a formal definition of the **coupling** of productions, and the coupling of the generated graphs.

This might be pretty straightforward for the MSC case.

Application of [p1, p2, p2, p5, p4, p3] of the SR grammar leads to the desired spatial relations graph:



Generation of the diagram

The nodes in the SR graph are graphical objects.

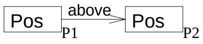
These objects have yet unspecified attributes, such as their actual position in the plane.

The spatial relations between the nodes of the SR graph define **constraints** on these attribute values.

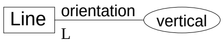
We need a **constraint solver** to derive exact values for all these attributes, on basis of the constraints.

This generates the **third representation**: a collection of **graphical objects** which can be displayed directly.

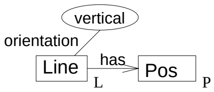
What do our spatial relations mean in terms of constraints on positions?



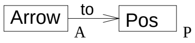
$$\Rightarrow \quad P_1.y > P_2.y \qquad (1)$$



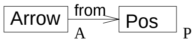
$$\Rightarrow \quad L.start.x = L.end.x \wedge$$
$$L.start.y < L.end.y \qquad (2)$$



$$\Rightarrow \quad P.x = L.start.x \wedge$$
$$P.y \geq L.start.y \wedge$$
$$P.y \leq L.end.y \qquad (3)$$



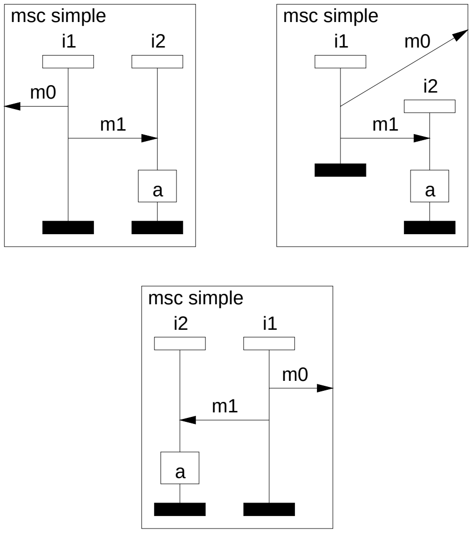
$$\Rightarrow \quad A.end = P \qquad (4)$$



$$\Rightarrow \quad A.start = P \qquad (5)$$

$\vdots$

The constraint solver should then produce a diagram like one of the following:



Constraint **hierarchies** could be used to differentiate between **required** and **prefered** constraints.

Specification of the Editor

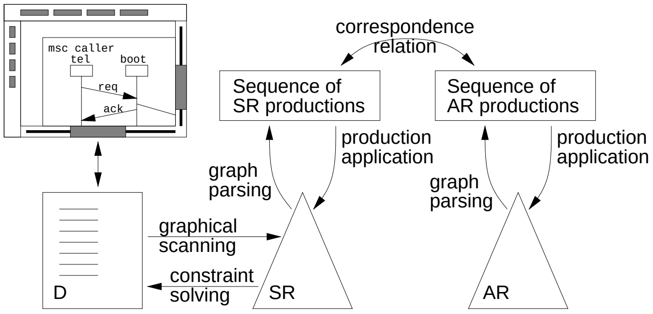
This completes the definition of the visual syntax of MSC diagrams.

How can such a syntax definition be used to implement a syntax directed editor for the defined visual language?

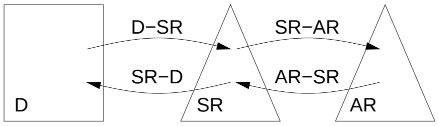
The syntax definition should specify the language dependent parts of the editing operations.

The editor maintains the three diagram representations and the mapping between them.

These are the **operations** which relate the three representations to each other:



This defines the following **mappings**:



1: Syntax Directed Editing

- Syntax directed **insertion**:
  - The user selects an AR production $L ::= R$ from a menu

    The user selects a region s in D

    SR-AR(*D-SR*(s)) gives a region s_{AR} in AR

    The editor finds a match for L in s_{AR}

    It applies (L, R) to AR

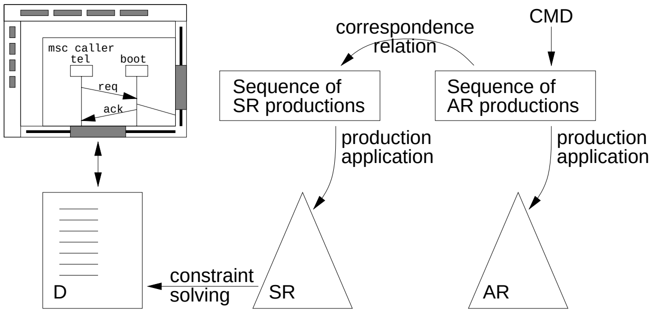
    It applies the corresponding SR production to SR

    The constraint solver updates D

- Syntax directed **deletion**: put $R ::= L$ in the production menu also.

- Syntax directed **change**: not supported

Syntax directed editing commands use the following part of the available **operations** on the three representations:



2: Layout Editing

If the user drags a graphical object then the editor might implement this in two ways

1. Ask the constraint solver for the area within which the object may safely move

   Restrict the movements to that area only

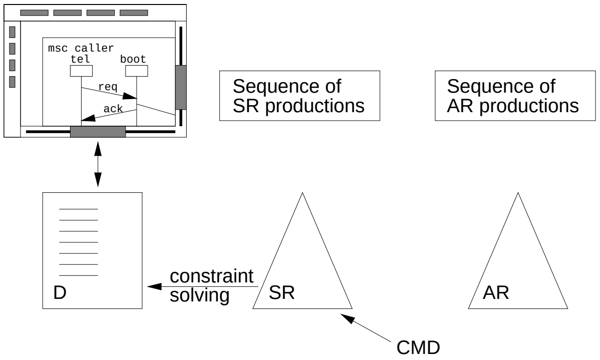
2. Translate each drag-event into a high-priority additional constraint for the moved object

   Call the constraint solver to compute a new layout

The 2nd way requires efficient, incremental constraint solving

The constraints are maintained ⇒ the SR graph remains the same ⇒ the AR graph remains the same.

Layout editing commands use the following part of the available **operations** on the three representations:



3: Free Editing

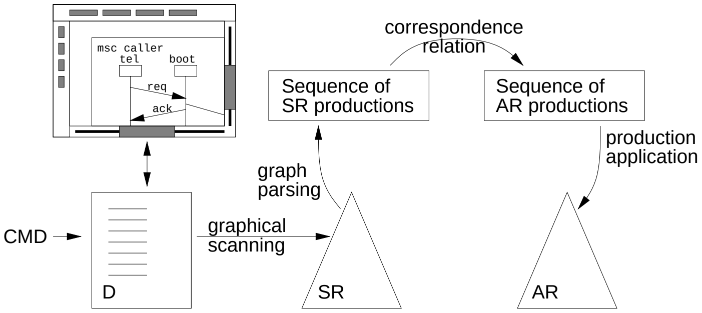
- The user enters a free-editing mode
- All modifications are in terms of graphical objects and directly update D
- The user indicates that he is done
  - **graphical scanning** creates a new SR graph

    graph parsing creates a new sequence of SR productions

    the **correspondence relation** gives a sequence of AR productions

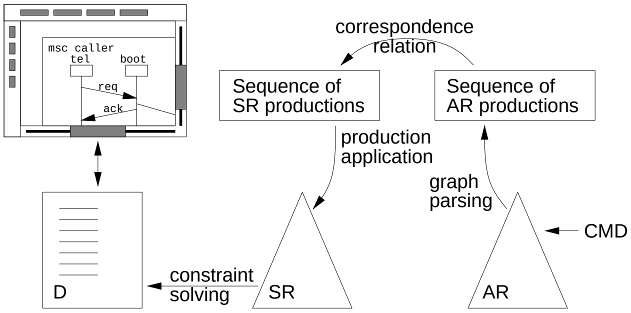
    application of these gives a new AR graph
- If all this succeeds, then the user may indeed leave the free-editing mode

Free editing commands use the following part of the available **operations** on the three representations:



Semantic oriented commands operate on the Abstract Relations Graph directly.

These commands use the following part of the available **operations** on the three representations:



Overview

- Syntax directed editing is desirable for visual languages
- Three kinds of commands to build and edit a diagram
- A definition of the visual syntax specifies the behavior of such an editor
- Three internal representations: *graphical objects*, *Spatial Relations* graph, *Abstract Relations* graph
- The allowed AR and SR graphs are defined by graph grammars
- Here applied on the language of MSC diagrams, but widely applicable

Constraints

Material:

- *Algorithms for Constraint Satisfaction Problems - a Survey*, Vipin Kumar, AI magazine, 1992.
- *An Incremental Constraint Solver*, Freeman-Benson, Maloney, Borning, Comm. ACM, vol. 33, no. 1, 1990.

Motivation

Constraints provide a very expressive way to formulate complex dependency relations between objects.

Closely related to visual languages:

- There are various visual languages to program in terms of constraints (*Thinglab* for example)
- All VL environments have to print visual sentences, but generating a good layout is hard

  Constraints are a convenient way to express layout preferences; the constraint solver then performs the layout. (*MSC*, *VODL*)
- You use the constraint solver DeltaBlue and need some background information.

Overview

Constraint systems are a very convenient mechanism to specify relations among objects, but their *power lead to a weakness*:

One can specify problems that are very difficult to solve.

This requires very general, powerful constraint solvers.

The generality increases the run-time, however.

⇒ You need a **spectrum of constraint solvers**, each trading generality versus efficiency differently.

The spectrum

- **Red**

  Most general and slowest, based on *graph rewriting*.
- **Orange**

  Based on *Simplex* method.

  Good at solving linear programming problems, allows for hierarchies of equality and inequality constraints
- **Yellow**

  Bases on *relaxation*, an iterative *hill-climbing* technique.

  Slower than Orange, but can handle non-linear equations

- **Green**

  Solves constraints over finite domains with combination of local propagation and backtracking.

  Paper by **Kumar**

- **Blue**

  Based on local propagation, fast, equality constraints only, cannot handle cycles in constraint hierarchies.

  Paper by **Freeman-Benson**, **Maloney**, and **Borning** on DeltaBlue

Algorithms for CSP – Kumar

Constraint Satisfaction Problem (CSP):

- Given:
  - a set of **variables** with
    - a **finite**, **discrete domain** per variable
  - a set of **constraints**:
    - defined over a **subset** of the variables
    - limits the **combinations** of values the variables may take
- Goal:
  - find an **assignment** of values to the variables that **satisfies** all constraints

For simplicity of the presentation, we restrict ourselves to **binary** CSP's, in which every constraint is either *unary* or *binary*.

A binary CSP can be represented by a **constraint graph**:

- **variables** are represented by **nodes**
- **binary constraints** are represented by **arcs** between variables
- **unary constraints** are represented by **cyclic arcs**

Any CSP with n-ary constraints can be converted to an equivalent binary CSP:

- Create a new variable whose domain is the set of n-tuples which satisfy the constraint.
- And connect the old variables to it with constraints which project a value out of the tuples.

A constraint $X + Y = Z$ with $D_X = \{1, 2, 3\}$ and $D_Y = \{4, 5, 6\}$:

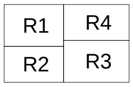
Create a new variable W with domain:

```
{ <1,4,5> , <1,5,6> , <1,6,7> ,
  <2,4,6> , <2,5,7> , <2,6,8> ,
  <3,4,7> , <3,5,8> , <3,6,9> }
```

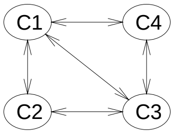
Constraint XW projects the first value out of W, YW the second, ZW the third.

Example

The map coloring problem for these regions:



is expressed by the constraint graph:



with every node the domain $\{red, blue, green\}$, and every arc the constraint $source \neq dest$.

Generate and Test – GT

Every CSP can be solved by the following (too simple) method:

Systematically generate each combination of values for the variables, and test whether all constraints are met.

Complexity of the order of the Cartesian product of all domain sizes.

Better is: check each constraint at the moment all of its variables have been assigned a value.

Backtracking – BT

Performs a **depth-first search** in the space of potential CSP solutions.

- Instantiate the variables one by one by choosing a value from their domain.
- As soon as all variables of a constraint are instantiated:
  - check the constraint
  - if violated, backtrack over the last instantiated variable
- if all variables have been instantiated: a solution has been found.

Is more efficient than generate-and-test method, but its complexity still is exponential.

Suffers from **trashing**: search in *different* parts of the space fail for the *same* reason.

The two main causes for trashing:

- **Node inconsistency**:

  domain D_i of variable V_i contains value a, but a does not satisfy a *unary* constraint on V_i:

  assignment of a to V_i will repeatedly fail.

Can be avoided by making domain D_i smaller.

This leads to **node consistency**.

- **Arc inconsistency**:

  Suppose variables are instantiated in the order

  $$V_1, V_2, \ldots, V_i, \ldots, V_j, \ldots, V_n$$

  Suppose that assignment of a to V_i disallows any value to V_j according to some constraint between V_i and V_j.

  This failure will be repeated for each combination of values that the variables $V_k\,(i < k < j)$ may take.

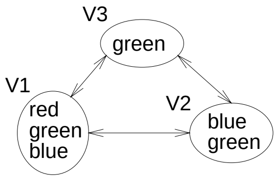
Can be avoided by **constraint propagation**.

Propagating constraints

Avoid **trashing** by making sure that each arc (V_i, V_j) of the constraint graph is **consistent** before the search starts.

Definition: Arc (V_i, V_j) for constraint C is **consistent** iff for every value x in the current domain of V_i some value y in the domain of V_j exists such that C is met.

Example:



Arc (V_3, V_2) is consistent, arc (V_2, V_3) is not consistent.

Arc (V_i, V_j) can be made consistent by deleting those values from D_i for which the above condition does not hold.

```
procedure REVISE $(V_i, V_j)$:
  delete := false
  for each $x \in D_i$ do
    if $\neg\exists v_j \in D_j$ such that $(x, v_j)$ consistent
    then
      delete $x$ from $D_i$
      delete := true
    endif
  endfor
  return delete
```

REVISE(V_2, V_3) removes *blue* from the domain of V_2.

Two loops, so complexity $O(d^2)$ with d the average domain size.

Once domain D_i is reduced, every arc (V_k, V_i) needs to be REVISE'd once more.

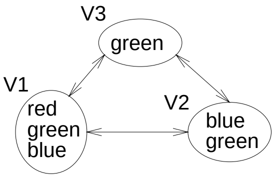
Algorithm to make a constraint graph **arc-consistent**:

```
procedure AC-3
  $Q := \{(V_i, V_j) \in arcs(G), i \neq j\}$
  while $Q \neq \emptyset$ do
    select and delete some arc $(V_i, V_j)$ from $Q$
    if REVISE($V_i, V_j$) then
      $Q := Q \cup \{(V_k, V_i) \in arcs(G), k \neq i, k \neq j\}$
    endif
  endwhile
```

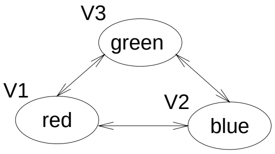
You do not need to reconsider (V_j, V_i) as elements deleted from V_i would never have given support for elements in V_j anyway.

The best *arc-consistency* algorithm has a worst case complexity of $O(ed^2)$, with e the number of binary constraints.

Applying AC-3 to the following constraint graph



transforms it in the graph



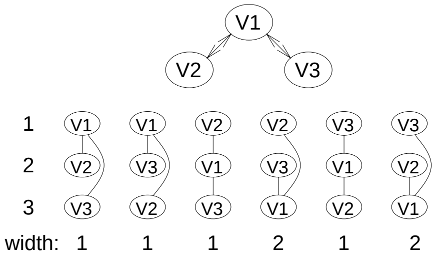
which has a single solution that is found without backtracking.

Making the graph **node-consistent** and **arc-consistent** reduces the search space, but backtracking might remain necessary.

Various techniques exist to take constraint propagation further (**path consistency**), but these often are more expensive than the backtracking.

Which arc-consistent constraint graphs can be solved without backtracking?

That partly depends on the order in which the variables are visited:



There are $n!$ different orders in which the constraint solver may visit the variables.

Each of these linear orderings constitutes a **ordered constraint graph**.

width of a constraint graph:

- The *width* of a **node** in a ordered constraint graph:

  the number of arcs that lead from previous nodes to this node.

- The *width* of a **ordered constraint graph**:

  the *maximum* width of its nodes

- The *width* of a **constraint graph**:

  the *minimum* width of its ordered constraint graphs

If you achieve node- and arc-consistency and the *width* of the constraint graph is 1, then no searching is necessary:

- You instantiate the variables according to a ordering with width 1

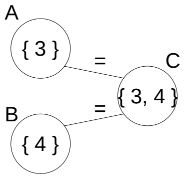
- For every variable you add there exists a value that satisfies all constraints

  (as you have node- and edge-consistency)

- This implies that you never have to backtrack to find an instantiation for all variables.

(tree structured constraints always have an ordering of width 1)

Suppose you have an arc-consistent constraint graph with width 2, then backtracking might still be necessary:



The arcs (A,C) and (B,C) are consistent, but the two constraints are incompatible with each other.

The instantiation of C fails, and backtracking is still necessary.

Till now: constraint propagation is performed **before** the actual constraint solving.

It can be advantageous to do it **during** constraint solving also.:

- As soon as a variable has been instantiated: perform constraint propagation on the remaining variables.
- Care must be taken that these additional domain restrictions are undone again at an eventual backtrack

This is done in the **Really Full Lookahead (RFL)** constraint solving method.

As constraint propagation is an expensive technique, various algorithms only **partly** perform it. (**GT**, **BT**, FC, PL, FL, **RFL**)

Reason maintenance

Ordinary backtracking not only suffers from *trashing*, but also easily performs *double work*.

Say we have the following partial assignment:

V_1	V_2	V_3	V_4
a_1	b_3	c_1	

For the cases $V_2 = b_1$ and $V_2 = b_2$ we could not find a value for V_3.

Now we discover that no value for V_4 can be found that does not conflict with $V_1 = a_1$.

Ordinary backtracking will backtrack over V_2 and V_3 first.

After assigning a new value to V_1, it will have to rediscover the problem with b_1 and b_2.

Dependence directed backtracking (or intelligent backtracking) avoids this double work.

It stores a *justification* for choosing a certain pair of values, and re-uses this reasoning at a new backtrack branch.

This mechanism avoids the redundant work, but requires complex administration and reasoning.

Simple backtracking often turns out to be more efficient.

Heuristic methods

In the backtracking method, the algorithm can choose

- the **order** in which the **variables** are instantiated, and
- the **order** in which the **values** are tried.

Considerable efficiency improvements can be achieved by making sensible choices here.

These methods are all based on **heuristics**.

1. Select the variable with the **fewest remaining alternatives** first.

   This reduces the bushiness of the search tree by moving failures up in the tree.

   Prunes unsuccessful branches as soon as possible.

2. Select the variable which participates in the **highest number of constraints** first.

   Same reasoning as above.

3. Transform the constraint graph into a **tree** by **removing some variables** from it.

   This *cutset* has to be chosen as small as possible.

   First assign values to the variables in the *cutset*, next solve the constraint tree without backtracking.

4. Transform the constraint graph into a **tree** by **removing some arcs** from it.

   Solve the constraint tree first, without backtracking.

   This provides a first approximation of the values.

   Next repair the chosen values incrementally such that they also satisfy the remaining constraints.

5. In instantiating a variable, select those values first that **maximize the number of options** for the next assignments.

   This enlarges the change that the left-most search path is successful.

Conclusions

CSP's can be solved by backtracking, but at considerable costs.

Various ways to improve the efficiency:

- **Constraint propagation** makes the search space smaller and avoids trashing
- **Reason maintenance** remembers why certain combinations of values failed
- **Heuristic methods** direct the search such that a solution is found earlier

All these techniques complicate the basic steps.

Too many of these optimizations might turn out worse than simple backtracking.

DeltaBlue
an incremental constraint solver

The group of Alan Borning is active to apply constraint solving in user interfaces, to support direct manipulation.

They have developed a collection of constraint solvers: Blue, DeltaBlue, SkyBlue, UltraViolet, ...

These are all very efficient, but only allow for a quite restricted class of constraint systems.

Their constraint solver allows for *priorities* among the constraints, only allows for *equality* constraints, and *cannot solve cyclic constraints*.

That is, once a value has been chosen for a variable, it should not be revised anymore.

Constraints, variables, methods

A constraint system consists of a **set of constraints** C, and a **set of variables** V.

A constraint is a **n-ary** relation among a subset of V.

Each constraints has a **set of methods** M, any of which may be executed to cause the constraint to be **satisfied**.

Each method uses some of the constraint's variables as input and computes the remainder as output.

A method may only be executed when *all of its inputs*, and *none of its outputs* have been determined by other constraints.

For example, the constraint $a + b = c$ has the following associated methods:

$$c := a + b \qquad b := c - a \qquad a := c - b$$

Execution of each of these will satisfy the constraint.

It is the task of the constraint solving technique to decide

- which constraints to satisfy,
- which method will be used to satisfy the constraint, and
- in what order these methods will be applied.

Constraint hierarchies

Each constraint is labeled with a **strength**.

There is an ordering between strengths.

$$required > strong > medium > weak$$

The constraints with strength *required* are special, as they **must** be satisfied.

For the other strengths, it is only *prefered* that they be satisfied.

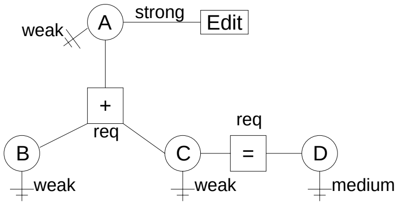
Why are constraint hierarchies useful?

Say we have $A + B = C$ as constraint and we edit the value of A.

What should the constraint solver do to re-satisfy the constraint?

Update B, update C, refuse the edit of A, or some combination?

Constraint hierarchies allow us to specify our preferences **declaratively**.



Types of constraints

- **Ordinary** constraint

  Relates set of variables, methods to compute value for each variable

- **Stay** constraint

  On one variable; prevents weaker constraints to change the value

- **Input** constraint

  On one variable, method puts external data in the data flow graph

- **Edit** constraint

  On one variable, method sets variable to a constant value

Solutions

A **solution** to a given constraint system is a mapping from variables to values.

An **admissible solution** is a solution that satisfies all *required* constraints.

The collection of **admissible solutions**:

$$S_0 = \{x \mid \forall c \in C_0 : x \text{ satisfies } c\}$$

with C_0 the set of *required* constraints.

Some comparator *better* compares admissible solutions:

The collection of **best solutions**:

$$S = \{x \mid x \in S_0 \land \forall y \in S_0 : \neg better(y, x, C)\}$$

DeltaBlue uses the comparator "*locally-predicate-better*":

Admissible solution A is better than B if

- For each level $0 \ldots k-1$, A and B satisfy exactly the same constraints, and

- A satisfies the same constraints as B on level k, and at least one more.

This comparator only checks whether a constraint is satisfied or not.

Another comparator could be one that measures the error of every constraint, and computes some weighted average of these errors.

Plans

Give the constraints and their strengths, a **plan** is constructed.

A **plan** consists of a *linear list of methods*.

Execution of a plan: sequentially execute the methods.

Each *method execution* satisfies *one constraint* and assigns *one variable* with its value.

The *entire execution* results in a *solution* for the constraint system.

If some variables are determined by **input** constraints, then a plan can be *executed repeatedly*, to update all related variables accordingly → fast, interactive feedback.

Difference Blue and DeltaBlue

In Blue, the plan for a constraint system is computed from scratch.

In DeltaBlue, the plan is incrementally computer from a previous plan.

DeltaBlue requires more administration, but will be faster on small updates to the constraint hierarchy.

Both methods: the plan remains valid as long as constraints do not change, and a plan can be re-executed when an input value changes.

Efficiency measure: a plan does not need to update variables which are determined by a *stay* constraints.

How to construct a plan?

A plan represents the choice

- which constraints should be satisfied
- which methods are used to do so
- in what order are these methods applied

How to make these choices?

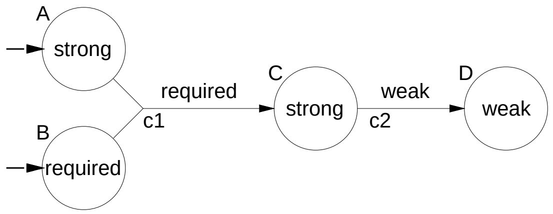
We associate a **walkabout strength** with each variable.

The **walkabout strength** is the strength of the weakest constraint that could be revoked to satisfy another constraint.

The **walkabout strength** is based on the **current plan** for the constraint system.

In a plan, variable v is determined by method m of constraint c.

The *walkabout strength* of v is the **minimum** of the strength of c and the walkabout strengths of the inputs of m.



D is *weak*, as c_1 is *weak*; C is *strong* as A is *strong*.

Weaker walkabout strengths propagate through stronger constraints, but not vice versa.

Adding a variable

A variable must be created explicitly before any constraint can be defined on it.

On creating a variable, the programmer must provide it with an initial value.

The variable automatically obtains an invisible, *very weak* **stay** constraint.

This provides the new variable with its initial walkabout strength.

(DeltaBlue cannot handle under-constrained systems where it has freedom in assigning values to variables)

Adding a constraint

Say, we add constraint c with strength s on variable v.

In the current plan, v is determined by method m' of constraint c'.

This gives v a walkabout strength s'.

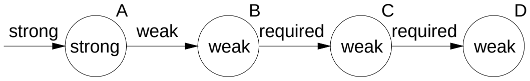
If $s \leq s'$ nothing needs to be done.

If $s > s'$, then c' has to be revoked and c is invoked.

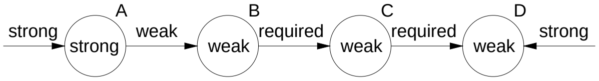
The new walkabout strength of v then also has to be propagated through the rest of the network.

Example

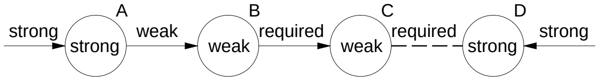
Initially we have the following plan:



Now we add a *strong* constraint on D.



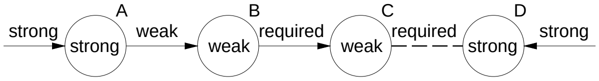
$strong > weak$, so the new constraint is invoked:



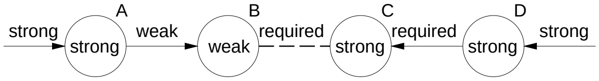
Do note that the walkabout strength of D has changed.

Now we have to propagate the new situation further.

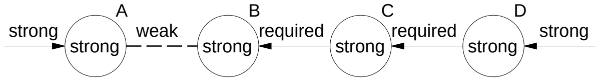
We had:



On C: $required > weak$, so we get:



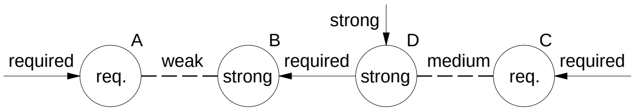
On further propagation:



Here propagation stops, as $weak < strong$.

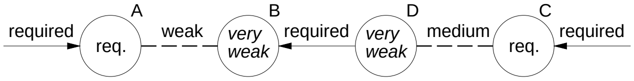
Removing a constraint

Initially we have:

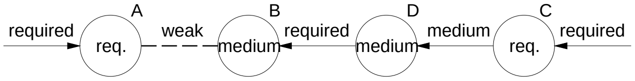


We remove the *strong* constraint on D.

The default *very weak* **stay** constraints temporarily come into action.



But now the inactive constraints are stronger:



(removing a constraint which currently is not active has no influence on the plan)

Complexity

Updating a plan is of worst case complexity $O(M)$, with M the number of constraints.

Creating a plan incrementally for M constraints takes $O(M^2)$ steps.

Executing a plan takes $O(M)$ steps.

This is very efficient and can indeed be applied in interactive environments.

The price paid is that Blue is restricted to non-cyclic constraint systems.

Comparison

- **Green** (Kumar)

  Finite, discrete domains.

  All constraint systems allowed

  Solving via backtracking and constraint propagation

  Exponential behavior

- **Blue** (Borning)

  No domain limitation

  Each constraint provides methods which satisfy it.

  Constraint hierarchies to express preferences

  Linear behavior

  Every step taken by the solver leads to the solution

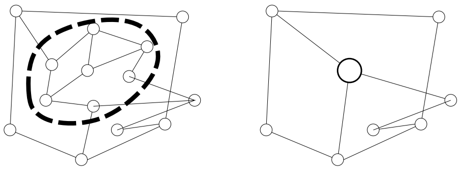
Extensions to DeltaBlue

SkyBlue extends DeltaBlue by allowing constraints to have methods that have *multiple outputs*.

Very convenient to model transformations like between cartesian and polar coordinates:

$$(x, y) \Leftrightarrow (\phi, r)$$

Makes it possible to take a subgraph of a constraint graph together into a single node.



The "method" could then be another kind of constraint solver.

UltraViolet allows for a larger class of constraints than DeltaBlue does:

- It uses local propagation where possible
- It resorts to a cycle solver when necessary

  The cycle solver can solve linear equations; similar to Simplex method (orange)

Parsing Visual Languages with Picture Layout Grammars

Material:

- *Parsing Visual Languages with Picture Layout Grammars*, Golin, JVLC, 1991.

Overview

I will treat three parsing algorithms for visual languages

- *Golin* - **Picture Layout Grammars**
- *Marriott* - **Constraint Multiset Grammars**
- *Rekers & Schürr* - **Graph Grammar** based approach

Golin and Marriott's approaches are based on **Attribute Multiset Grammars**.

This week:

- Explain what parsing for visual languages is about and show some *picture layout grammars*
- Define *attribute multiset grammars*
- Explain Golin's parsing method for *picture layout grammars*

Visual Languages

A **visual program** is a **picture** with a well-defined **structure** and **meaning**.

The **picture** itself

- is a flat collection of graphical symbols
- which have attributes that tell how they are arranged in a two-dimensional fashion.

A **visual language** is a set of pictures.

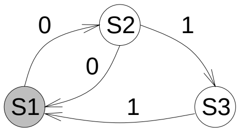
The **structure** of a picture is defined by the **syntax of its visual language**.

This syntax definition: (1) serves as basis for assigning **meaning** to the picture, (2) can be used to control a **parsing** algorithm, and (3) generates the visual **language**.

Syntax definition and parsing for *textual* languages is well developed; for *visual* languages much need to be done still.

Much of the theory can be re-used, but the **multi-dimensional nature** of pictures poses the following additional problems:

- **No natural linear ordering** between the symbols
- **A wide variety of relations** instead of the simple concatenation of textual languages.
- The underlying structure is a **directed graph**, rather than a tree.



Picture representation

We represent a visual program as an **attributed multiset**: an unordered collection of attributed visual symbols.

- The **class** of a symbol corresponds to its type (*arrow*, *circle*, ...)
- The **attributes** of a symbol specify its features (*location*, *color*, *text value*, ...)

Visual languages are then sets of attributed multisets.

Assumption (**visibility**): everything needed to understand the picture is available in its attributed multiset representation.

Syntax definition

The attributed multiset representation of a picture is a **flat** structure.

If we view the picture as an **element of a visual language**, then it has a **complex** structure.

This structure corresponds to the significant **relationships** between the symbols.

This language dependent structure is defined by **productions**.

State → contains(circle, text)

The operator "**contains**" specifies the kind of composition of its constituents.

For VL the composition has to be **explicit**.

An example grammar

A **picture layout grammar** of (a simple version of) State Transition Diagrams:

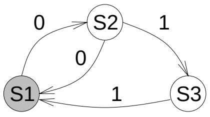
1: STD → StateList

2: StateList → State
3: StateList → (State, StateList)

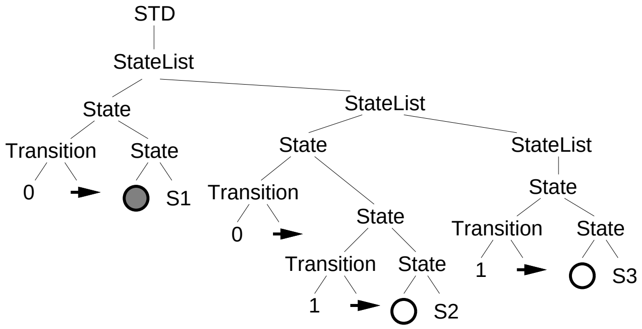
4: State → contains(circle, text)
5: State → leaves(State, Transition)

6: Transition → labels(arrow, text)

According to this grammar, the diagram



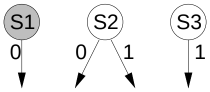
might have the following **parse tree** as structure:



Context symbols

A tree is not really fit to describe the two-dimensional adjacency relationships.

The same tree would describe this, incorrect, STD:



A transition clearly belongs to *two* states. This is impossible in a tree.

The grammar formalism is extended with **context symbols**.

These are terminal symbols which *must be present* to apply the production, but are *not consumed* by the application.

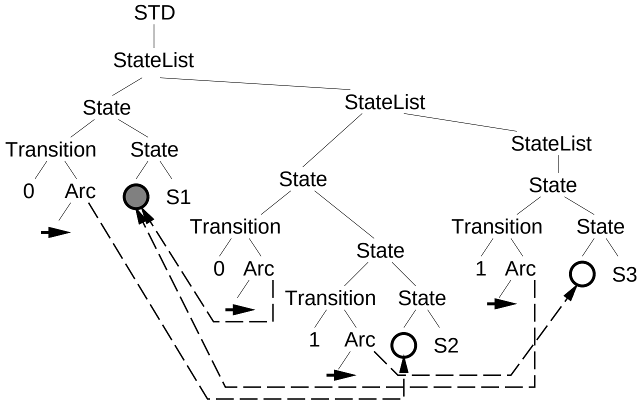
1: STD → StateList

2: StateList → State
3: StateList → (State, StateList)

4: State → contains(circle, text)
5: State → leaves(State, Transition)

6: Transition → labels(Arc, text)
7: Arc → enters(arrow, circle)

With these context symbols, the "*parse tree*" becomes a "*parse tree-DAG*":



Exercise

1. In State Transition Diagrams one distinguishes *ordinary states* (single circle) from *final states* (double circle).

   Extend the grammar such that *final states* are allowed also.

2. Treat grey circles differently as being a *start state* of the DFA.

My solution

DoubleCircle $\rightarrow$ **contains(circle, circle)**
FinalState $\rightarrow$ **contains(DoubleCircle, text)**
State $\rightarrow$ **FinalState**

GreyCircle $\rightarrow$ **isgrey(circle)**
StartState $\rightarrow$ **contains(GreyCircle, text)**
State $\rightarrow$ **StartState**

Ambiguous derivations

The grammar allows the *states* to be composed in any order.

The parse tree-DAG gives the *states* a specific order

$\Rightarrow$ There will be **many ambiguous derivations**, each choosing a different order.

This will make parsing much more expensive.

Note that the ambiguity is due to the grammar, the picture itself is not ambiguous.

Hard problem: write a grammar which generates as little ambiguities as possible.

Attributed Multiset Grammars

The formal model underlying PLG are **Attributed Multiset Grammars** (AMG's).

AMG is quite similar to *attribute grammars*, but the implicit notion of sequence is replaced by explicit constraints on the attribute values.

An **AMG** is a six-tuple (N, Σ, s, I, D, P) with:

N: finite set of non-terminal symbols
Σ: finite set of terminal symbols
$s \in N$: startsymbol
I: attribute names
D: attribute domains
P: set of **productions**

A **production** is a triple (R, SF, C) with:

R: rewrite rule of the form $A \rightarrow M_1/\lambda$:
  $A \in N$: left hand side (LHS)
  M_1/λ: right hand side (RHS)
  $M_1 \subset (N \cup \Sigma)$: multiset of ordinary symbols
  $\lambda \subset \Sigma$: multiset of context symbols
SF: semantic function
C: constraint on the application of R

M is **analyzable**: M has a **derivation tree** T:

leaf nodes of T: M
root node of T labeled by s
each interior node n is labeled by
  $p = (R, SF, C)$:
    labels(RHS(R)) = labels(children n)
    C(children n) = true
    attributes n = SF(children n)

The **language** recognized by G:

$$\mathcal{S}(G) = \{M \mid M \in \Sigma^* \wedge M \text{ analyzable over } G\}$$

PLG are based on AMG, but put some **restrictions** on the allowed grammars

- The attributes may only take a finite number of different values

  (otherwise the parser might not terminate)

- No two terminal symbols may have the same class and location

  This is implied by *visibility* property

- The RHS of a production may at maximum consist of **two** symbols.

  (only s, ss and sc, with s an ordinary symbol and c a context symbol)

  This is easily solved by replacing each n-ary production by a sequence of binary productions.

The PLG rule **Arc** → **enters(arrow, circle)** corresponds to the AMG rule

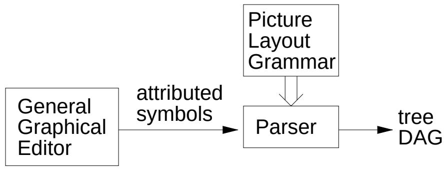
Arc → { arrow } / { circle }
  Arc.lx = arrow.lx
  Arc.ly = arrow.ly
  Arc.rx = arrow.rx
  Arc.ry = arrow.ry
where
  (arrow.rx, arrow.ry) on border of circle

This shows that the definition of the *composition operators* is pretty important.

A library is provided, users may extend them.

The PLG Parsing Algorithm

The PLG parser:



It works in **three phases**:

1. Create a **Factored Multiple Derivation** (FMD) structure which represent *all* possible analysises of the input

2. Filter all faulty parts out of FMD and extract a single derivation

3. Attribute evaluation to implement semantics (not discussed)

The FMD structure

The FMD stores all possible derivations in a single structure by taking all "parsing fragments" together.

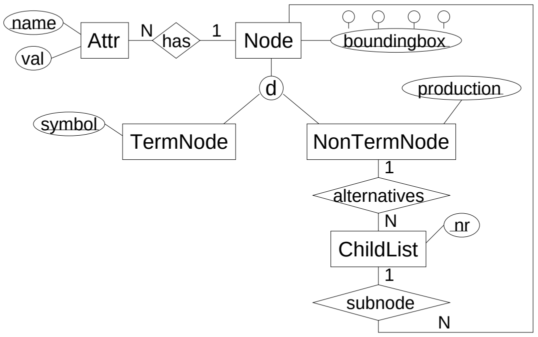
- Nodes are labeled by their symbol
  - The leaf nodes are *terminal* nodes
  - The interior nodes are *non-terminal* nodes

- Each node has a vector of *attribute* values; at least lx, by, rx, ty

- Non-terminal nodes which are labeled by the same production and have the same attribute values are *grouped* into a single node

Each **non-terminal** node

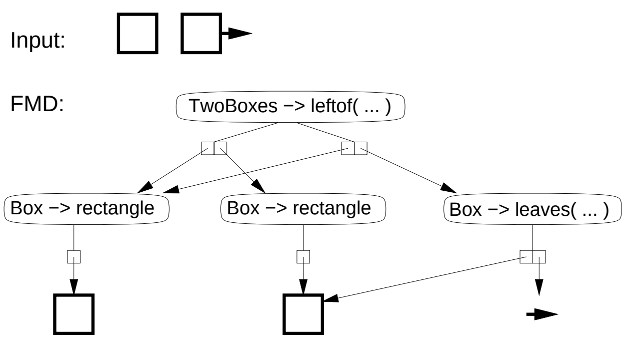
- has an associated *production*
- has one or more *childlists*:
  - Each childlist is a list of sub-nodes, corresponding to the RHS
  - More than one childlist: several possible derivations



Example

Grammar:

TwoBoxes → **leftof(Box, Box)**
Box → **rectangle**
Box → **leaves(rectangle, arrow)**



Based on assumption that the composition operator **leaves** gives **Box** same bounding-box as **rectangle**.

Phase 1 of PLG parsing

Builds the FMD.

Takes attributed multiset M as input, and constructs FMD structure according to productions in G.

Algorithm works **bottom-up**, resembles CYK parsing.

Builds FMD structure and keeps two queues of pointers to nodes in this FMD structure: *to-do* and *done*:

- Newly created nodes are added to *to-do* and are later moved to *done*.
- The queue *to-do* is initialized to the input nodes in M.
- Algorithm terminates when *to-do* is empty.

```
Build($M$, $P$):
for each $b \in M$ do
   Add a terminal node for $b$ to $todo$ and $FMD$
while $todo \neq \emptyset$ do
   $next :=$ some element of $todo$
   $X := symbol(next)$
   for each $p \in P$ such that $X \in RHS(p)$ do
      if $p = A \rightarrow \{X\}$ then
         if constraints satisfied then
            Add$(p, \{next\})$
      else
         for each occurrence of $X$ in $RHS(p)$ do
            let $Y$ be the other symbol in $RHS(p)$
            for each $old \in done$ such that $symbol(old) = Y$ do
               if constraints satisfied then
                  Add$(p, \{next, old\})$
   move $next$ from $todo$ to $done$
return $FMD$

Add$(p, subnodes)$
$new :=$ create a node for $p$
$childlist(new) := subnodes$
$attr(new) := SF(subnodes)$
for each node $n \in (todo \cup done)$ do
   if $symbol(n) = LHS(p) \wedge attr(n) = attr(new)$ then
      $childlist(n) := childlist(n) \cup subnodes$
      discard $new$
      return
add $new$ to $FMD$
add $new$ to $todo$
```

Remarks

- The building phase terminates as the attributes may only take finitely many distinct values.

  (I would rather see a restriction on teh kind of grammars)

- The *todo* queue has no prefered order

  (Isn't necessary here)

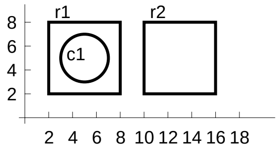
- There is no structure to help searching in $todo \cup done$

  (A hashing structure would help)

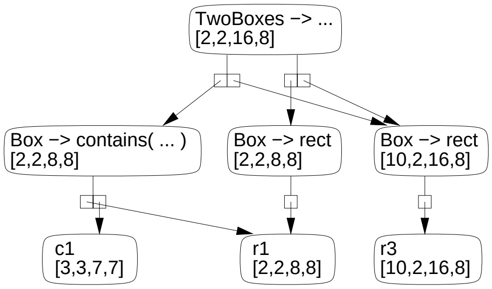
Example of Build

Box → rectangle
Box → contains(rectangle, circle)
TwoBoxes → leftof(Box, Box)

Input:



FMD:



TwoBoxes –> ...
[2,2,16,8]

Box –> contains(...)
[2,2,8,8]

Box –> rect
[2,2,8,8]

Box –> rect
[10,2,16,8]

c1
[3,3,7,7]

r1
[2,2,8,8]

r3
[10,2,16,8]

Phase 2 of PLG parsing

Extracts single derivation out of the FMD

Works **top-down**

Intuitively:

- Start at root node and walk down

- If a node has several childlists: **select** one

- If the result is a *valid* tree-DAG: **return** it.

  Else: **backtrack** over the most recent select choice.

What is a **valid** tree-DAG?

1. all constraints are satisfied

   (automatically met by the way in which the FMD is constructed)

2. the root is labeled by the start symbol

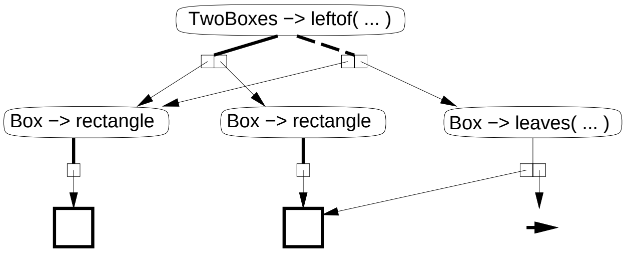
   (automatically met by the way in which the FMD is walked)

3. the input M is the *yield* of the tree-DAG

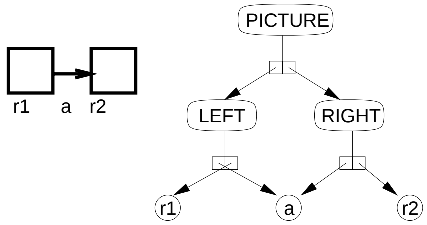
   (the "*yield*" is the part of the input it covers)

4. no node appears more than once in the tree-DAG

- Violation of condition 3:



- Violation of condition 4:
  PICTURE → leftof(LEFT, RIGHT)
  LEFT → leaves(arrow, rectangle)
  RIGHT → enters(arrow, rectangle)



The **actual** implementation of phase 2 does **not backtrack**:

Instead,

- it walks the FMD three times:
  - to compute for each node which terminal nodes it covers
  - to remove childlists that cover less terminals then their parent node does

    (I have my doubt about the correctness of this step)
  - to remove childlists of which the subnodes cover overlapping parts of the input
- it walks the remaining FMD a final time and the first tree-DAG it encounters is valid

Golin does not give a proof of correctness of the parsing algorithm, but it all *looks fine*.

Theoretical complexity: $O(n^9)$

Complexitiy in practical cases: $O(n^2)$ to $O(n^3)$

The parsing algorithm has been implemented and has been used to define the syntax of a variety of visual languages.

The restriction to 2 symbols and the hidden power of the combinator operations makes PLG grammars very hard to read.

Parsing Visual Languages with **Constraint Multiset Grammars**

Material:

- *Parsing Visual Languages*, Chok & Marriott, Australasian Computer Science Conference, 1995.

Overview

- Introduction and example grammar
- Semantics of the productions
- The CMG parsing algorithm
- Comparison with Picture Layout Grammars

Constraint Multiset Grammars

Constraint Multiset Grammars (CMG's) are a variation on **Attribute Multiset Grammars** (AMG's):

- Conditions may test on the **existence** or **non-existence** of remote symbols

  This extends the context mechanism of AMG's

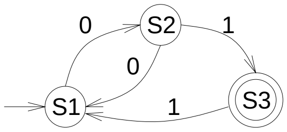
- The RHS of a production may use a **collection primitive** which assembles **all** symbols which match a condition

  Alleviates the "ordering" problem which causes ambiguities

- The associated parsing algorithm only works for **cycle free** and **deterministic** CMG's, and only needs **one phase**.

Example grammar

Example CMG for State Transition Diagrams



Three kinds of states: **normal**, **final** and **start**.

A CMG production that describes a **final state**:

```
State(point mid, integer radius,
      string name, string kind) ::=
  C1: circle, C2: circle, T: text
  where (
    C1.mid == C2.mid &&
    C1.radius != C2.radius &&
    close(T.mid, C1.mid)
  ) {
    mid = C1.mid;
    radius = max(C1.radius, C2.radius);
    name = T.text;
    kind = "final"
}
```

To describe what a **start state** looks like we first describe an **unlabeled arrow**:

```
start_arc(point end) ::=
  A: arrow
  where (
    not exists R:text where (
      close(R.mid, A.mid)
    )
  ) {
    end = A.end;
}
```

(note the use of a negative constraint)

First attempt for **start state**:

```
State(point mid, integer radius,
      string name, string kind) ::=
  A: start_arc, C: circle, T: text
  where (
    on_circle(A.end, C.mid, C.radius) &&
    close(T.mid, C.mid)
  ) {
    mid = C.mid;
    radius = C.radius
    name = T.text;
    kind = "start"
}
```

First attempt for **normal state**:

```
State(point mid, integer radius,
      string name, string kind) ::=
  C: circle, T: text
  where (
    close(T.mid, C.mid)
  ) {
    mid = C.mid;
    radius = C.radius
    name = T.text;
    kind = "normal"
}
```

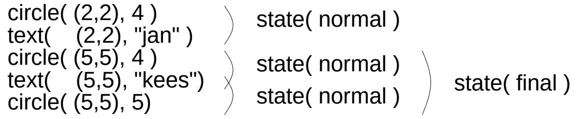
However, the pattern for a *normal* state overlaps with the patterns for *final* state and *start* state.

This makes the grammar **non-deterministic**.

Overlapping of productions:

```
State(string kind) ::=
  C: circle, T: text
  where ( . . . ) {
    kind = "normal"
}

State(string kind) ::=
  C1: circle, C2: circle, T: text
  where ( . . . ) {
    kind = "final"
}
```


circle((2,2), 4)
text((2,2), "jan")
circle((5,5), 4)
text((5,5), "kees")
circle((5,5), 5)
state(normal)
state(normal)
state(normal)
state(final)

Golin needs the second phase to determine which production applications can form part of a correct reduction.

Marriott avoids this by the **deterministic** requirement. This is more restrictive than *non-ambiguous*.

We need additional negative constraints:

1. `not exists M: circle where (`
   `  M.mid == C.mid`
   `)`

2. `not exists A: start_arc where (`
   `  on_circle(A.end, C.mid, C.radius)`
   `)`

- To make the grammar deterministic:
  - add constraints 1 and 2 to the production for **normal state**
- To disallow states which are both of kind **final** and **start**:
  - add 1 to the production for **start** state
  - add 2 to the production for **final** state

A **labeled arrow**:

```
arc(point start, point end, string text) ::=
  A: arrow, T: text
  where (
    close(A.mid, T.mid)
  ) {
    start = A.start;
    end = A.end;
    text = T.text;
}
```

A **transition**:

```
transition(string start, string label, string end) ::=
  A: arc
  where (
    exists S1: state, S2:state
    where (
      on_circle(A.start, S1.mid, S1.radius) &&
      on_circle(A.end, S2.mid, S2.radius)
    )
  ) {
    start = S1.name;
    label = A.text;
    end = S2.name;
}
```

We finally need to take the recognized **state** and **transition** symbols together in a single symbol.

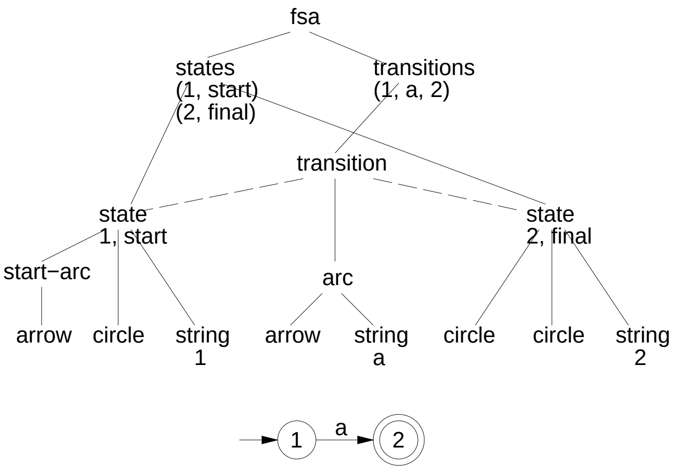
The order in which these symbols are collected does not matter: we use the **collection primitive** *“all”*.

```
states(string states) ::=
  all S: state
  where (true) {
    states = "[" << "(" << S.name <<
                    "," << S.kind << ")" << "]";
}

transitions(string transitions) ::=
  all T: transition
  where (true) {
    transitions = "[" << "(" << T.start <<
                          "," << T.label <<
                          "," << T.end <<
                          ")" << "]";
}

fsa(string states, string transitions) ::=
  S: states, T: transitions
  where (true) {
    states = S.states;
    transitions = T.transitions
}
```

A derivation



Remarks

- The CMG version of the syntax of STD’s is *much easier to read and write* than the **picture layout grammar** version

- Grammars should be as **declarative** as possible.

  This aspect is compromised by the **deterministic** requirement.

  One needs negative constraints in one production to prevent it to be applied on parts that should be matched by other productions.

  This is more like *programming* the parser than like *specifying* a graphical syntax.

  Still, this restriction makes parsing **efficient** and allows for **incremental** parsing.

Semantics

- The simplest kind of productions are **local**:

$$\begin{array}{lll} T(\vec{A}) & ::= & V_1 : T_1, \ldots, V_n : T_n \\ & & \mathit{where}\,(C) \\ & & \vec{A} = \vec{E} \end{array}$$

  with C a **conjunction of primitive constraints** over the attributes of $V_1, \ldots, V_n$.

- Productions may use **existentially quantified** (or **context** or **remote**) variables. The constraint C then is of the form:

$$\mathit{exists}\, V_1' : T_1', \ldots, V_m' : T_m'\, \mathit{where}\,(C')$$

- Productions may use **negative constraints**. The constraint C then is of the form:

$$\mathit{not\ exists}\, V_1' : T_1', \ldots, V_m' : T_m'\, \mathit{where}\,(C')$$

The application of a **local** production:

Find a match for the RHS of P in S such that C holds, and replace the matched tokens by an instance of the LHS.

Or, more formally:

Let P be a production

$$T(\vec{A}) ::= V_1 : T_1, \ldots, V_n : T_n \; where \, (C) \; \vec{A} = \vec{E}$$

then $S \overset{P}{\Rightarrow} S'$ if there is an **assignment** θ from the variables $V_1, \ldots, V_n$ to tokens in S such that:

- $\{\theta(V_1), \ldots, \theta(V_n)\} \subseteq S$,

- θ satisfies C, and

- $S' = S \cup T(\vec{A}) \setminus \{\theta(V_1), \ldots, \theta(V_n)\}$

Constraint multiset grammars deal in a quite **simple way** with existentially quantified variables.

The above semantics of reduction are extended to **existentially quantified** variables in the following way:

Next to the current sentence S, we also keep a collection of **previously reduced tokens** *R. Existentially quantified variables may match tokens in R as well.*

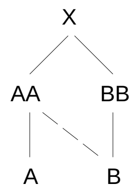
For example, this means that the production for *transition* is applicable regardless whether the associated *states* are still in the current sentence or not.

This works well in 90% of the cases.

Grammar:

```
AA() ::= a: A where exists b: B
BB() ::= b: B
X() ::= a: AA, b: BB
```

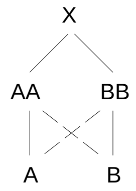
Fine derivation:



Grammar:

```
AA() ::= a: A where exists b: B
BB() ::= b: B where exists a: A
X() ::= a: AA, b: BB
```

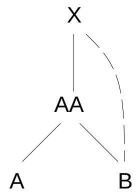
Questionable derivation:



Grammar:

```
AA() ::= a: A, b: B
X() ::= a: AA where exists b: B
```

Incorrect derivation:



Reduction with existentially quantified variables:

Production P:

$$\begin{array}{lll} T(\vec{A}) & ::= & V_1 : T_1, \ldots, V_n : T_n \\ & & where \, exists \, V'_1 : T'_1, \ldots, V'_m : T'_m \\ & & where \, (C) \\ & & \vec{A} = \vec{E} \end{array}$$

Reduction $\langle S, R\rangle \overset{P}{\Rightarrow} \langle S', R'\rangle$ if there is an **assignment** θ such that:

- $\{\theta(V_1), \ldots, \theta(V_n)\} \subseteq S$,

- $\{\theta(V'_1), \ldots, \theta(V'_m)\} \subseteq S \cup R$,

- θ satisfies C,

- $S' = S \cup T(\vec{A}) \setminus \{\theta(V_1), \ldots, \theta(V_n)\}$, and

- $R' = R \cup \{\theta(V_1), \ldots, \theta(V_n)\}$

Productions with **negative constraints** check for the **non-existence** of tokens in $S \cup R$.

However, what should happen if the application of the production (indirectly) generates these tokens itself?

```
BB() ::= a:A where not exists b:BB
```

or

```
AA() ::= a: A where not exists b: BB
BB() ::= b: B where not exists a: AA
X() ::= a: AA, b: BB
```

The application of these productions invalidates itself.

Constraint multiset grammars are restricted to be **stratified**:

Types may only depend negatively on types which are strictly lower in the dependency graph.

A **stratification** ψ assigns an integer to each type such that for each production

$$\begin{array}{lll} T(\vec{A}) & ::= & V_1 : T_1, \ldots, V_n : T_n \\ & & \mathit{where\,exists}\, V_1' : T_1', \ldots, V_m' : T_m' \\ & & \mathit{where}\,(C) \\ & & \vec{A} = \vec{E} \end{array}$$

the following conditions hold:

- $\psi(T) \geq \psi(T_i)$ for $i = 1, \ldots, n$,
- $\psi(T) \geq \psi(T_i')$ for $i = 1, \ldots, m$, and
- for each variable $V' : T'$ appearing in a **negative constraint** in C, or in an **all** primitive:

  $\psi(T) > \psi(T')$.

A CMG is **stratifiable** if it has a **stratification**.

A **reduction sequence** for a stratifiable grammar G and a sentence S:

$$\langle S, \emptyset \rangle \overset{P_0}{\Rightarrow} \langle S_1, R_1 \rangle \overset{P_1}{\Rightarrow} \ldots \overset{P_n}{\Rightarrow} \langle S_n, R_n \rangle$$

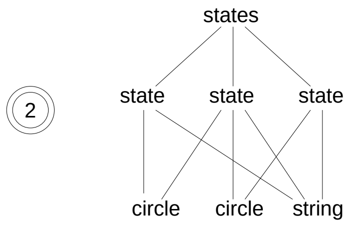
Here P_i is a production in G.

Productions should be applied such that the stratification number of their LHS never decreases: Given that each P_i has a symbol of type T_i as LHS, it must be the case that $\psi(T_i) \leq \psi(T_{i+1})$:

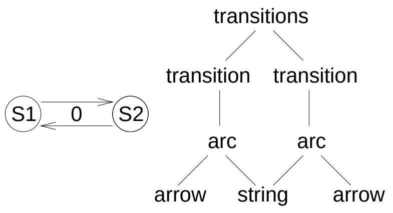
If $S_n = \{s\}$ (with s the start symbol) then the reduction sequence is **successful**.

The **language** of stratifiable CMG is the collection of sentences which have a successful reduction sequence.

According to this definition of **successful reduction sequence**, there is nothing wrong with this derivation:



Neither with this one:



Overlapping derivations

To disallow such overlapping derivations, we need a **general rule**:

We add an attribute *coverset* to each token which contains the terminals which it **generates**:

- Each terminal token t gets the singleton set $\{t\}$ as *coverset*;
- each non-terminal token gets the union of the *coverset*s of its input tokens as *coverset*.

Rule: *Productions may only apply if the input tokens have* **disjoint** *coversets.*

Then we get, with P a local production

$T(\vec{A}) ::= V_1 : T_1, \ldots, V_n : T_n \, where \, (C) \, \vec{A} = \vec{E}$:

$S \overset{P}{\Rightarrow} S'$ if there is an **assignment** θ from the variables $V_1, \ldots, V_n$ to tokens in S such that:

- $\{\theta(V_1), \ldots, \theta(V_n)\} \subseteq S$,
- for all $i, j \in 1, \ldots, n, i \neq j : coverset(\theta(V_i)) \cap coverset(\theta(V_j)) = \emptyset$,
- θ satisfies C,
- $cvs = coverset(\theta(V_1)) \cup \ldots \cup coverset(\theta(V_n))$,
- $S' = S \cup T(\vec{A}, cvs) \setminus \{\theta(V_1), \ldots, \theta(V_n)\}$

Existentially quantified variables do not participate in the coverset check.

Additional restrictions

- The grammar should be **cycle-free**

  Derivation sequences of the form

  $$\ldots \Rightarrow \langle S, R \rangle \Rightarrow \ldots \Rightarrow \langle S, R' \rangle \Rightarrow \ldots$$

  are not permitted.

  Can simply be checked on the grammar.

- The grammar should be **deterministic**:

  A stratifiable CMG G is deterministic if every terminal sentence has a single, maximal reduction sequence for G, without checking the coverset condition.

  This means that every step taken by the parser should be a step in the direction of the result.

  Unfortunately, this restriction **cannot** be checked on the grammar.

The parsing algorithm

Productions are classified in groups which have the same *stratification* number.

Productions are applied as long as possible per increasing stratification level.

The algorithm uses a *token database* D.

```
routine Parser($M$)
  $D := \emptyset$
  for all $t \in M$ do
    add $t$ to $D$
  for $S$ := 1 to maxstrata do
    repeat
      changed := false
      for each $P \in$ stratum($S$) do
        if EvaluateRule($P, D$) then
          changed := true
    until changed = false
  return $D$
```

Routine *EvaluateRule* applies P as many times as it can in D.

```
**routine** *EvaluateRule*($P, D$)
  $T$ := the variables in production $P$
  $C$ := the constraints of production $P$
  *changed* := false
  **repeat**
    $A$ := *FindNextCombination*($T, D$)
    **if** *SatisfiesConstraints*($C, A$) **then**
      *InsertToken*($P, D$)
      *DeleteTokenList*($A, D$)
      *changed* := true
  **until** $A$ is empty
  **return** *changed*
```

Routine *DeleteTokenList* does not actually delete its tokens, but only **marks** them as deleted.

This allows *FindNextCombination* to match both current and already deleted tokens.

The above only gives an **impression** of what the parser has to do.

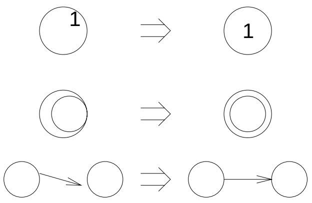
It should have been more precise on:

- The check on **overlapping coversets**, and the creation of the **new coverset**.
- How to deal with the **different kind of variables** (*normal*, *exists*, *not exists*, *all*)
- How to deal with re-application of the **same production on the same tokens**

Recent work on CMG's

At VL'95, Sitt Sen Chok described a syntax directed graphical editor, based on CMG and constraint solving:

- User draws diagram **freely**
- Incremental parse of the diagram with **sloppy checking** of constraints
- Constraint **solver** prints **beautified version** of the diagram according to parse tree



Comparison Golin - Marriott

- Both have Attribute Multiset Grammars as basis
- Both have a parsing algorithm that only works for **certain classes** of grammars.
  - Golin: finitely many different attribute values
  - Marriott: deterministic grammars
- Neither can **check** these conditions on a given grammar
- Marriott's dealing with context symbols is too simple.

- Marriott is **more restrictive** and thus avoids the second phase of the parsing algorithm.

  Marriott's parsing algorithm is **efficient** and is geared towards **incremental** parsing

- Golin's grammar formalism PLG results in hard-to-read grammars

- Both use constraints to **check** the applicability of a production on matched tokens, but do not use them to **direct** the search for matching tokens.

These are efficient parsing algorithms for **restricted** grammars; we also need an (efficient) parsing algorithm for **full** Attribute / Constraint Multiset Grammars.

Parsing Visual Languages with **Graph Grammars**

Material:

- *A graph grammar approach to graphical parsing*, Rekers & Schürr, VL'95

Overview

- Implicit versus explicit relations

- Highlights of the parsing algorithm

- Comparison with CMG approach

Constraints test for Relations

The general form of a CMG production (without existentially quantified symbols) is:

$$\begin{array}{lll} T(\vec{A}) & ::= & V_1 : T_1, \ldots, V_n : T_n \\ & & \mathit{where}\,(C) \\ & & \vec{A} = \vec{E} \end{array}$$

The constraint C puts conditions on the allowed attribute values.

These conditions are mostly used to **relate** the objects which are bound to $V_1, \ldots, V_n$.

Typical examples of constraints are:

```
C1.mid == C2.mid
close(T.mid, C1.mid)
on_circle(A.end, C.mid, C.radius)
```

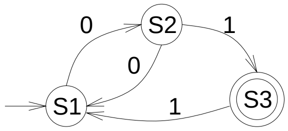
The relations which are checked by the constraints are **implicit** consequences of the attribute values.

This has as consequence that:

- If the parser tries to apply a production, it
  - finds all combinations of tokens which match $V_1, \ldots, V_n$
  - checks C for each of the matches.
- If the production is applied, the parser creates a new token with type T.

  This means that all previously checked productions which have a token of type T in their RHS may find new matches.

```
State(point mid, integer radius,
      string name, string kind) ::=
  C1: circle, C2: circle, T: text
  where (
    C1.mid == C2.mid &&
    C1.radius != C2.radius &&
    close(T.mid, C1.mid)
  ) {
    mid = C1.mid;
    radius = max(C1.radius, C2.radius);
    name = T.text;
    kind = "final"
}
```



This simple diagram contains 7 strings and 4 circles.

That means that the CMG parser has $7*4*4 = 112$ combinations to consider.

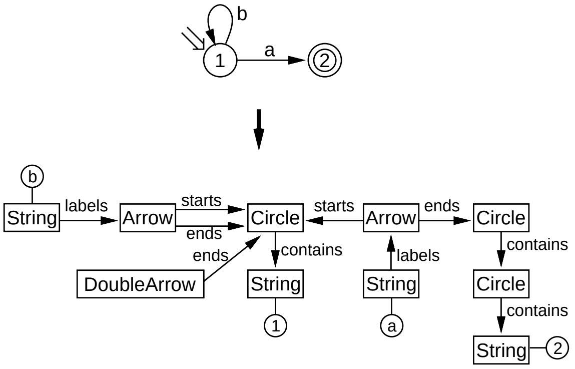
The Graph Grammar approach

In the **graph grammar** approach one makes these relations **explicit** by representing them as actual **edges** between objects.

- These edges are **initially** created by a **graphical scanner**
- The parser can **follow these edges** to find matches
- The application of a **production** not only creates a new object, but also **creates edges** with other objects.

  This tells the parser which matches to reconsider.

The spatial relations graph



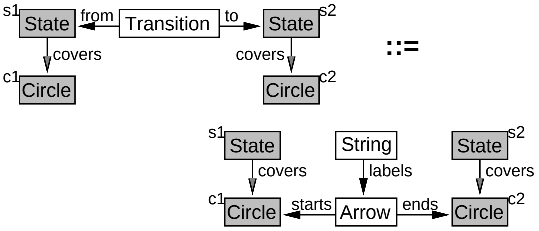
The **graphical scanner** receives a collection of attributed graphical object and creates a *spatial relations graph*:

- *Nodes*: the **graphical objects**
- *Edges*: the **spatial relations** of interest which hold between the objects

Graph Grammar Productions

A **production** is a tuple (L, R) of graphs.

The intersection $L \cap R$ is the preserved **context**.



- **Graph parsing**: search graph for match of R, replace by copy of L
- **Graph generation**: search graph for match of L, replace by copy of R

Recorded in a **production instance**

In graph grammar productions

- Edges have to be **created explicitly**

  The new symbols in the LHS are connected to context symbols with edges.

- Edges have to be **deleted explicitly**

  A production is applicable only if it mentions **all edges** the ordinary symbols in the RHS have.

  (this is due to the **dangling-edge rule**)

This symmetry makes the grammar more **declarative**, but also requires the grammar to be very **precise**.

This certainly makes it harder to write a grammar.

The parser has to perform **two tasks**: (1) **finding matches** and (2) **assembling derivations**.

1: **Bottom-up phase** (*element level*)

- **generate** as many **production instances** as possible

  this generates the collection of building blocks for **all** possible derivations

↓

2: **Top-down phase** (*production instance level*)

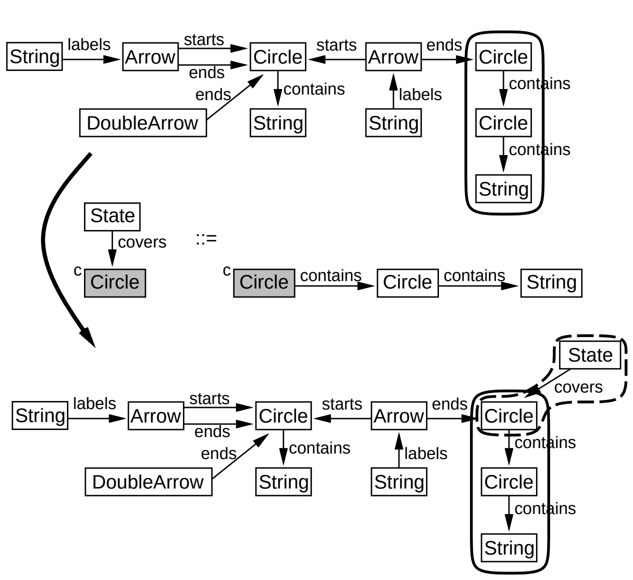
- select a subset of production instances which forms a **viable derivation** for the graph

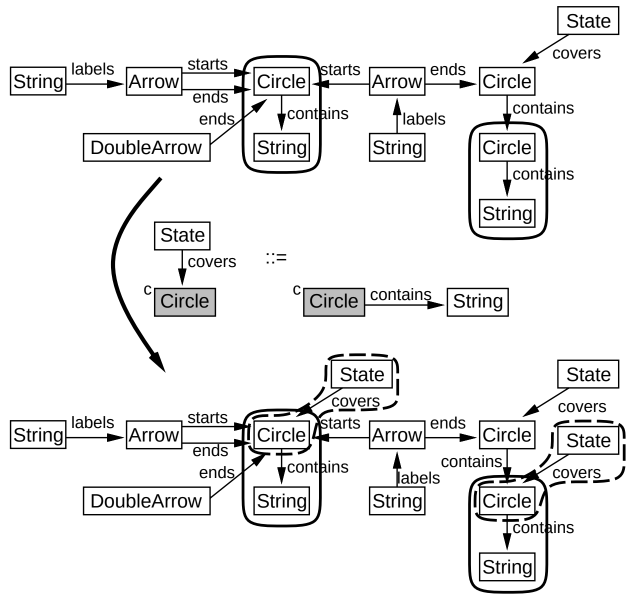
The bottom-up phase

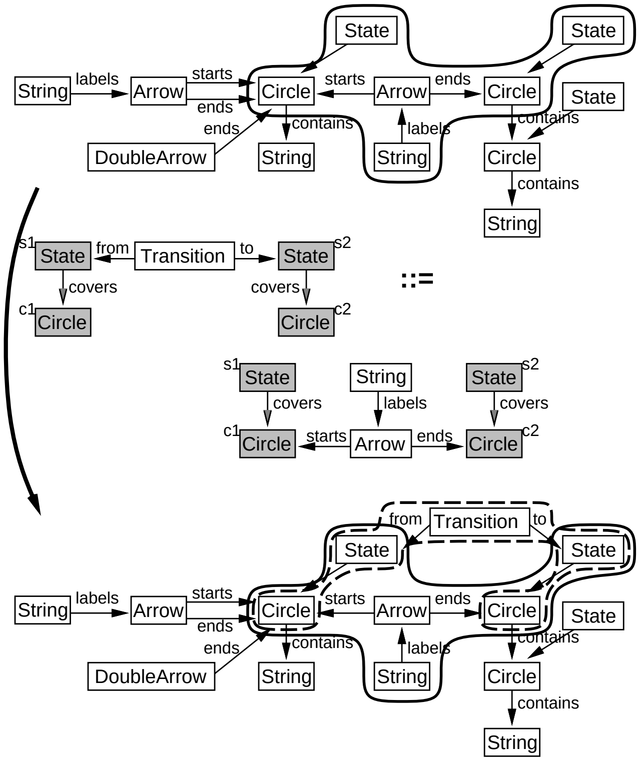
- Start with the graph G
- **Repeat** until no **new** matches can be found:
  - Search G for a match of some right-hand side
  - If found:
    - ∗ **extend** G with **left-hand side**

      (does **not** delete the **right-hand side**!)
    - ∗ generate **production instance**
- Return all **production instances**

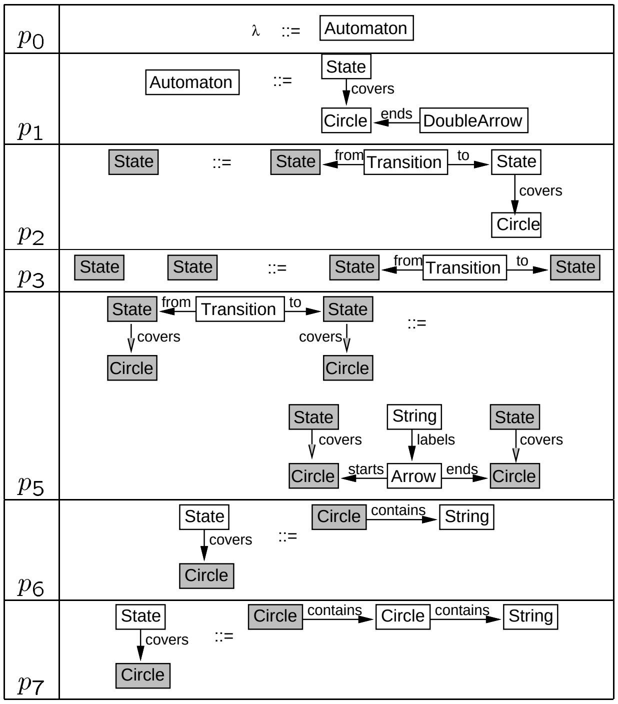
Examples







The entire grammar:



Optimizations

- Each **right-hand side** is represented by a single **linear search plan**

  Leads to the notion of **dotted rule**

- These matches are **step-by-step extended**

- Many **partial matches** are simultaneously maintained

- **Stratification** gives a good ordering on the productions

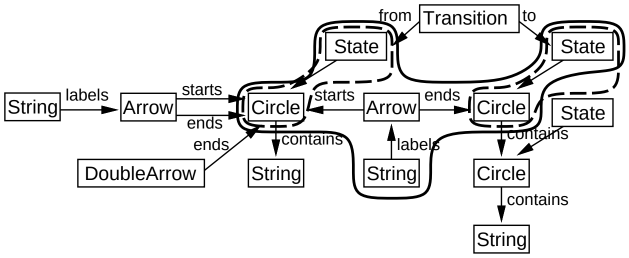
Our bottom-up phase is straightforward as it does not have to deal with *derivations*.

Dependencies

Production instances might **depend** on each other:

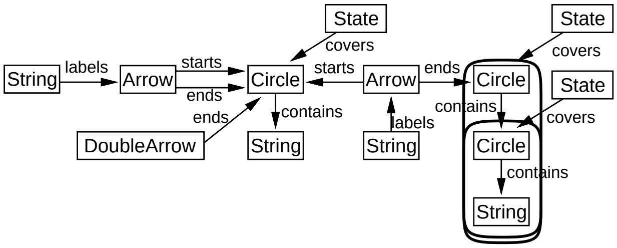
- **Above**:

  If production instance pi adds an element x to the graph, which is matched by the right-hand side of pi', then pi' **above** pi.



- **Excludes**:

  If both pi and pi' consume a same element x of the graph, then pi **excludes** pi' and vice versa.



The top-down phase

Combines production instances into a **derivation** for G by selecting a **subset** of them.

It starts with the axiom graph A and applies production instances $pi_1, \ldots, pi_n$ to it such that

- $A \overset{pi_1}{\Rightarrow} \ldots \overset{pi_n}{\Rightarrow} G$

- $\neg\exists pi, pi' : pi$ **excludes** pi'

- $\forall pi_i, pi_j$: if pi_i **above** pi_j then $i < j$

Production instances are **selected on basis of the dependencies alone**.

- The **order** within $pi_1, \ldots, pi_n$ is insignificant, as long as the **above** relation is respected (equivalent derivations).

- Once pi is added to a derivation, it is no longer possible to add pi', if pi **excludes** pi'.

  ⇒ it is a **choice** to add pi instead of pi'

  We deal with these choices in a pseudo-parallel fashion.

By knowing the dependencies **beforehand**:

- we know the **choice points**

- we can **postpone** choice points

Comparison

In CMG, parsing is optimized by requiring deterministic grammars.

In our approach, parsing is optimized by having explicit relationships instead of constraints on attribute values.

Our approach allows for a far larger class of grammars, but requires very precise productions.

An interesting combination would be to use the CMG algorithm as bottom-up phase, and part of our algorithm as top-down phase.

This would be a **less efficient**, but **complete** parsing algorithm for the full class of Attribute / Constraint Multiset Grammars.