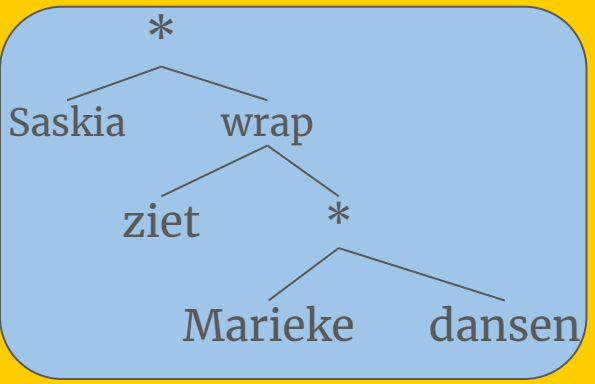
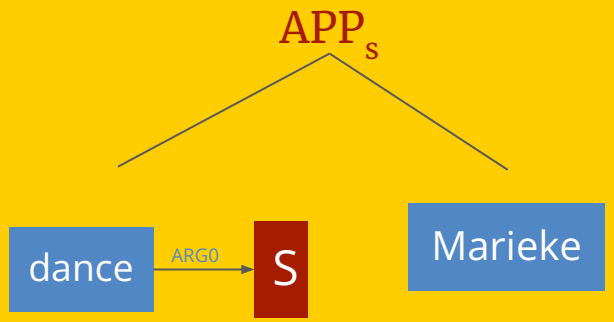
Trees as building instructions



Meaghan Fowlie
Utrecht University
séminaire ILFC
2024-09-11



Overview

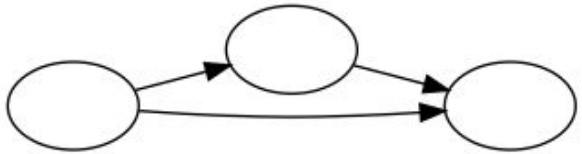
1. What is a tree?
2. Parse trees
3. Algebra terms
4. Interpreted Regular Tree Grammars (IRTGs)
5. Apply-Modify Parser

Trees and Graphs

Trees as graphs

Def: *Directed graph*: a tuple (N, E) such that:

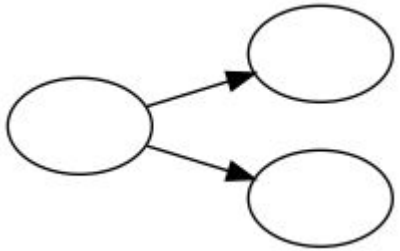
- N is a set (the *nodes* or *vertices*)
- $\mathrm{E} \subseteq \mathrm{N} \times \mathrm{N}$ (the *edges* or *arcs*)



Def: *Directed Path*: a sequence of edge-connect nodes

Def: *Tree*: A directed graph such that:

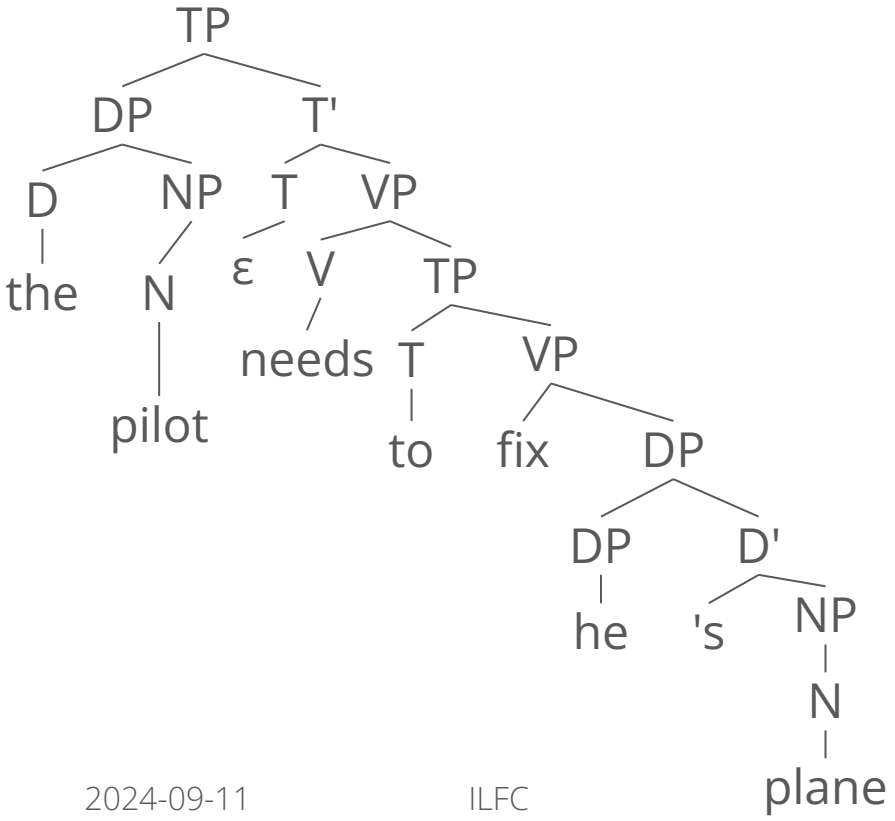
- There is exactly one node that is not reachable on a directed path from any other node (the *root*)
- There is exactly one directed path from the root to every other node



Example tree

Def: *Tree*: A directed graph such that:

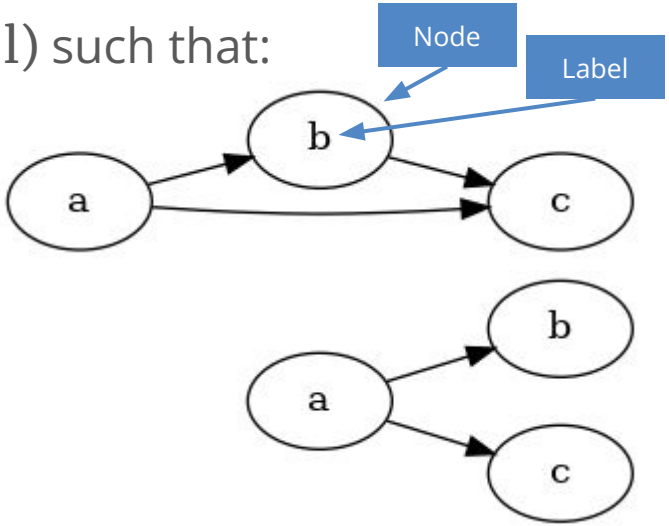
- There is exactly one node that is not reachable on a directed path from any other node (the *root*)
- There is exactly one directed path from the root to every other node



Trees as node-labelled graphs

Def: *Directed, node-labelled graph*: a tuple (N, E, L, l) such that:

- N is a set (the *nodes* or *vertices*)
- $E \subseteq N \times N$ (the *edges* or *arcs*)
- L is a set (the *labels*)
- $l: N \rightarrow_{\text{partial}} L$ (the *node-labelling function*)



Def: *Node-labelled Tree*: directed, node-labelled graph that is a tree.

- Often also: l is a total function (i.e. all nodes are labelled)

Convention: down = source to target

Ordered trees

Def: *Children/daughters, parents/mothers*

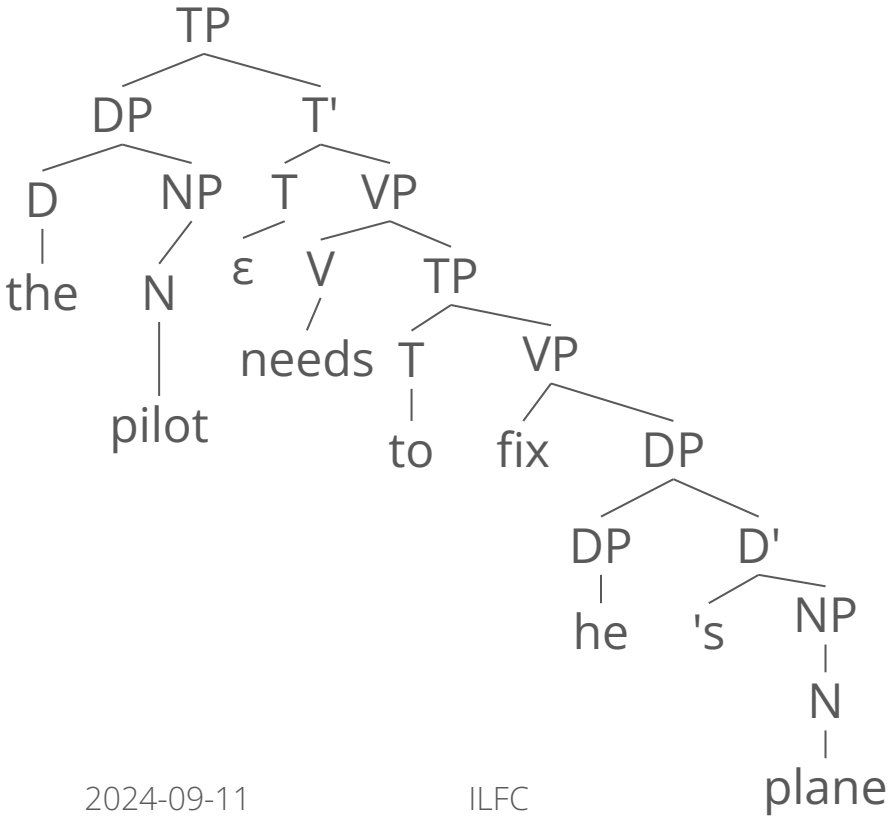
The children/daughters of a node are the nodes reached on one edge. The parent of a node is the node reached going backwards on the incoming edge. (or edges if not a tree)

Def: *Ordered tree*

A tree such that there is a total order on the set of children of each node

Constituency trees

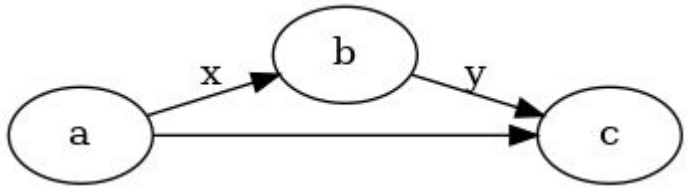
- Ordered, node-labelled tree
- Leaf nodes labelled with words (or something similar)
- Internal nodes labelled with phrase categories
- Interpretation:
  - representational
  - "Given a node *n*, find all directed paths from *n* to a leaf. Those leaf nodes, in order, have the category $l_N(n)$



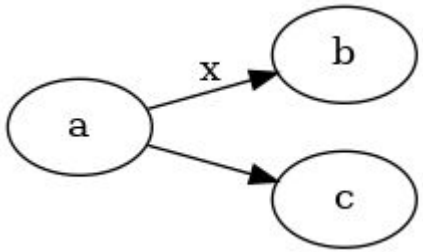
Trees as node- and edge-labelled graphs

Def: *Directed, node- and edge-labelled graph*: a tuple (N, E, L, l_N, l_E) such that:

- N is a set (the *nodes* or *vertices*)
- $E \subseteq N \times N$ (the *edges* or *arcs*)
- L is a set (the *labels*)
- $l_N: N \rightarrow_{partial} L$ (the *node-labelling function*)
- $l_E: E \rightarrow_{partial} L$ (the *edge-labelling function*)

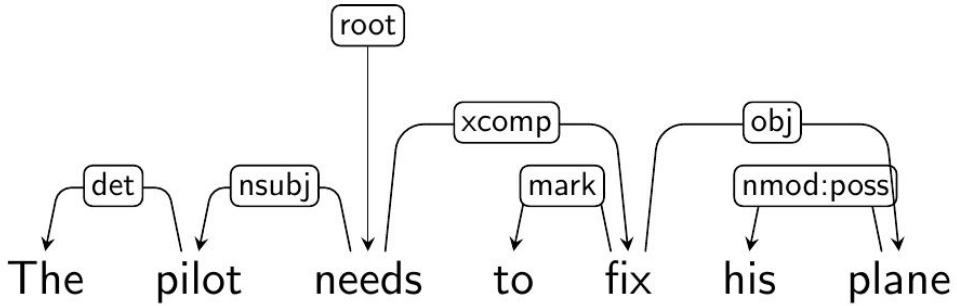


Def: *Node- and edge-labelled Tree*: directed, node- and edge-labelled graph that is a tree.



Dependency Trees

- A node- and edge-labelled tree.
  - Root node is often marked.
  - **Node** labels are **words**
  - **Edge** labels are (usually syntactic) **relationships** between words
  - Tree is usually paired with word order information
    - Conveyed as left-to-right order on the page
    - Edges have arrowheads to enable this
- Interpretation is representational: "*Pilot* is the nsubj of *needs*" etc.



Parse Trees

Constituency trees as parse trees

Same as a constituency tree, but with a different interpretation

Recall: Context-Free re-write grammar (CFG): (Σ, C, R, S)

- Σ: set of *terminal* symbols
- C: set of *non-terminal* symbols
- R: Set of rules of the form

$$LHS \rightarrow RHS$$

where $LHS \in C$ and RHS is a sequence of symbols from C and Σ

- $S \subseteq C$ is a set of *start symbols*

To use the grammar to *generate* a string of terminal symbols,

1. The strings starts as an $s \in S$

Until you have no non-terminals in the string:

2. For a nonterminal n in the string, choose a rule $r \in R$
3. Replace n with the RHS of r

Generating a string with a CFG

Let G = (Σ, C, R, S)

Let Σ (terminals) be lowercase, C (non-terminals) be uppercase, S = {A}, R =

1. A → B C
2. A → a
3. B → A C
4. B → b
5. C → c

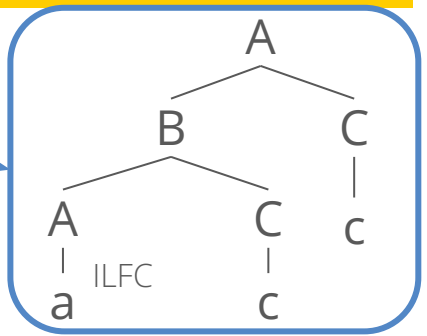
Derivation 1 of *a c c*:

	String	rule #
1.	A	start
2.	B C	1
3.	A C C	3
4.	a C C	2
5.	a c C	5
6.	a c c	5

Derivation 2 of *a c c*:

	String	rule #
1.	A	start
2.	B C	1
3.	A C C	3
4.	A C c	5
5.	a C c	2
6.	a c c	5

Parse tree:
All equivalent derivations of *a c c*

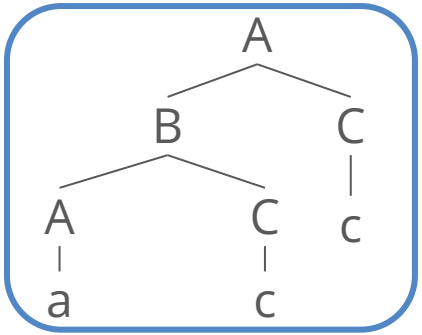


One tree, three meanings

Constituency Tree:

Interpreted representationally:

- *a c* is a string of category B.
- *c* is a string of category C.
- *a c c* is a string of category A.



Parse Tree:

Interpreted derivationally

- B was rewritten as A C
- C was rewritten as *c*
- A sequence of derivation steps led B to be rewritten as *a c*

and proof-theoretically:

- *a c c* is a sentence in the language of G

Algebras

Trees as terms over a ranked alphabet

Let Σ be a set of symbols (the *alphabet*)

Associate with each x in Σ a natural number: its *rank*.

A *term over* Σ is a sequence of symbols from Σ and {'(', ')', ',' } such that for every $x \in \Sigma$ in the term, if the rank of x is n, x is followed by a '(', n terms separated by ',', and finally ')'

e.g. let $\Sigma = \{A^{(2)}, B^{(2)}, C^{(1)}, a^{(0)}, b^{(0)}, c^{(0)}\}$ be a ranked alphabet (ranks in superscript)

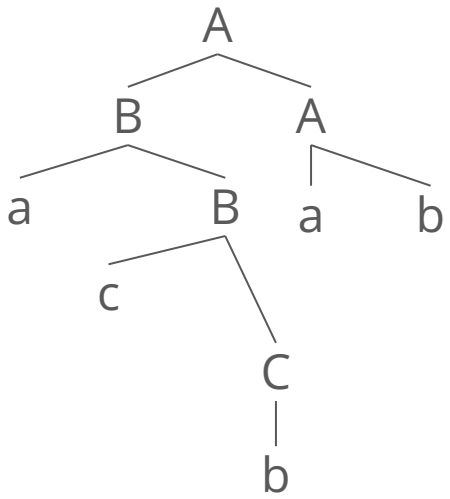
Then $A(B(a, B(c, C(b))), A(a, b))$ is a term over Σ

Or equivalently: a term is an ordered, node-labelled tree such that the nodes are labelled with elements of Σ and for every node, the number of children is equal to the rank of its label.

Trees as terms over a ranked alphabet

$\Sigma = \{A^{(2)}, B^{(2)}, C^{(1)}, a^{(0)}, b^{(0)}, c^{(0)}\}$

$A(B(a, B(c, C(b))), A(a, b)) \quad =$

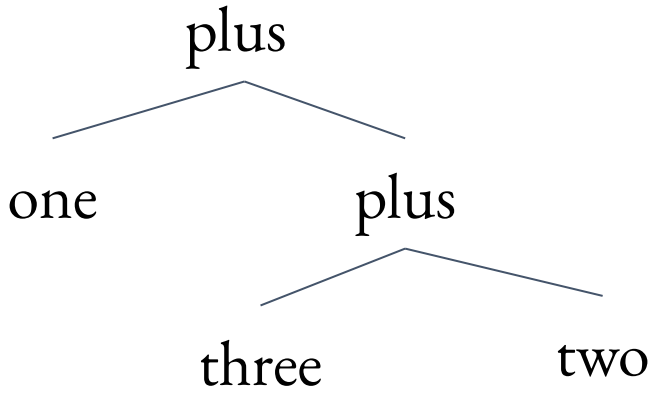


Algebras: terms as building instructions

Running example: adding natural numbers

1. a *signature:* a ranked alphabet
   a. e.g. $\{one^{(0)}, two^{(0)}, three^{(0)}, \ldots\} \cup \{\mathrm{plus}^{(2)}\}$
2. a *domain*: a set of objects
   a. e.g. $\mathbf{N} = \{1,2,3,\ldots\}$
3. an *interpretation* I, mapping each element of the signature to an operation on the domain.
   a. arity of the symbol must match the arity of the operation
      i. e.g. $I(\mathrm{plus}^{(2)})$ = + (a binary operation on numbers)
   b. arity of an operation: the number of arguments
   c. for rank 0, we call it a *constant*, and it just maps to an element of the domain (e.g. $I(one^{(0)}) = 1$)

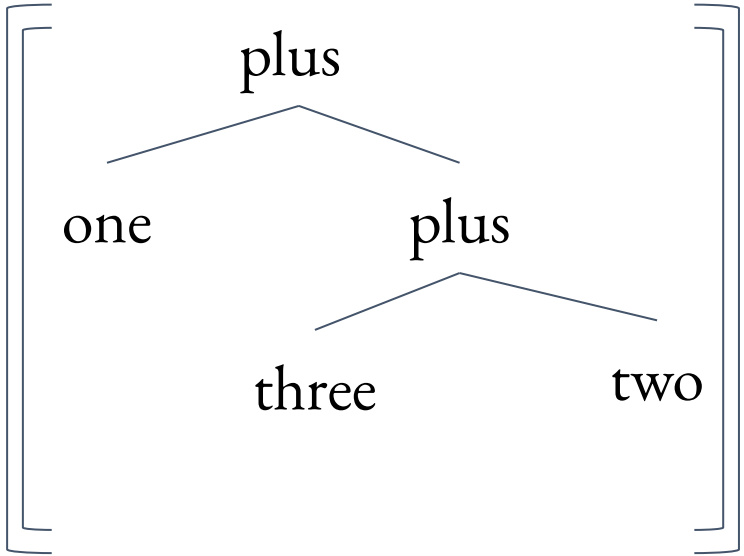
Example: interpreted algebra term

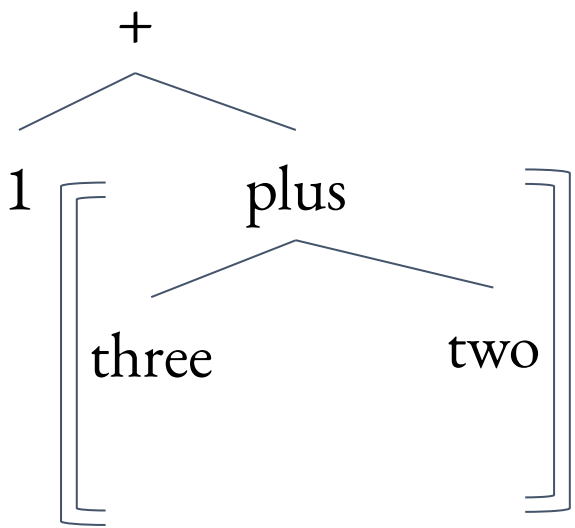


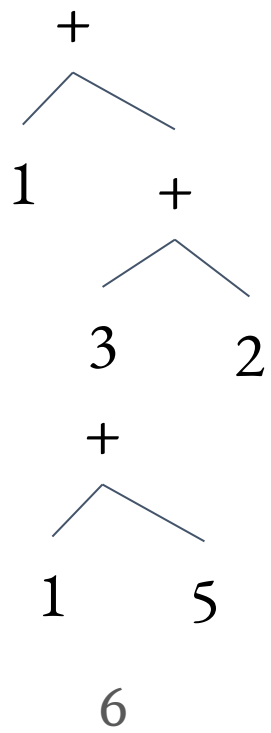
→ 1 + (3 + 2) = 6

Terms are *evaluated* bottom-up

Evaluation yields an object in the domain (here, N)



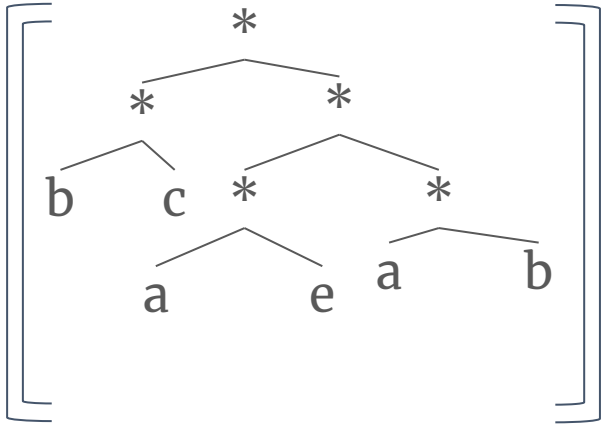




Example: string algebra

Domain: strings over a-z Signature: $\{a, b, \ldots z\}^{(0)} \cup \{*^{(2)}\}$

Interpretation: I(a) = a etc., I(*) = string concatenation



= bcaeab

Example: string-pair algebra

Domain: strings and pairs of strings over Dutch words

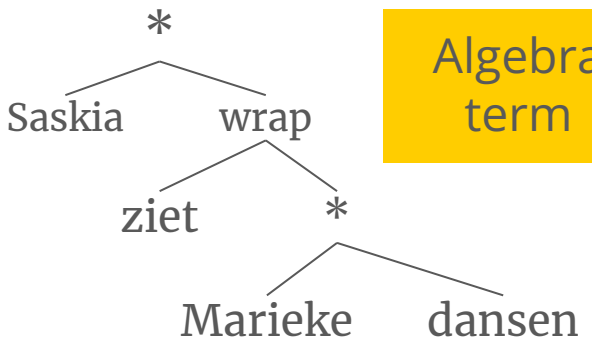
Signature: Dutch words$^{(0)}$ ∪ {*$^{(2)}$, wrap$^{(2)}$}

Interpretation: I(a) = a etc.

[[*(a, b)]] = (a,b) [[*((b,c), (a,d))]] = (b, cad)

[[*(a, (b,c))]] = (ab, c) [[*((b,c), a)]] = (b, ca)

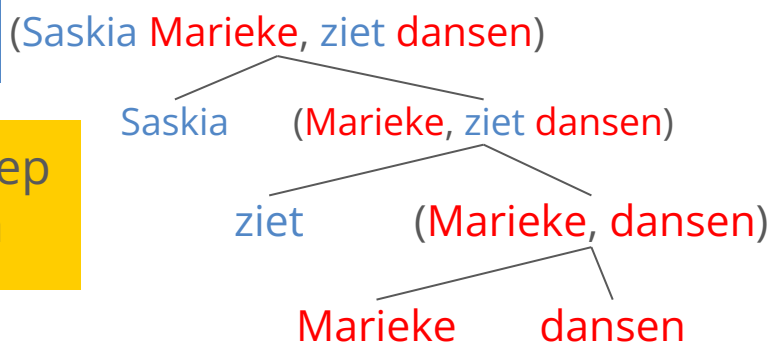
[[wrap(a, (b,c))]] = (b, ac)



Algebra term

Example: Dutch crossing dependencies
… dat Saskia Marieke ziet dansen
'… that Saskia sees Marieke dance'

Step-by-step evaluation



Interpreted Regular Tree Grammars (IRTGs)

(Koller & Kuhlmann 2011)

Issue: Algebras are often unconstrained

Recall example algebra with signature $\Sigma = \{A^{(2)}, B^{(2)}, C^{(1)}, a^{(0)}, b^{(0)}, c^{(0)}\}$

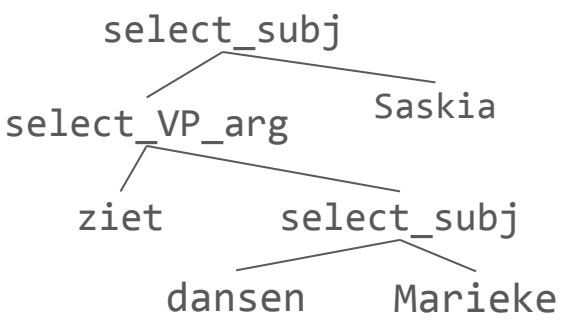
and term $A(B(a, B(c, C(b))), A(a, b))$

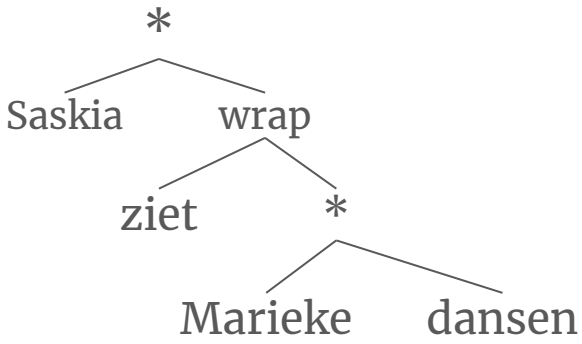
Recall example CFG with rules

1. $A \rightarrow B\ C$
2. $A \rightarrow a$
3. $B \rightarrow A\ C$
4. $B \rightarrow b$
5. $C \rightarrow c$

The algebra allows B(a, C(b)) but the grammar does not. If we want to use algebras, how can we constrain which algebra terms are "allowed"?

Interpreted Regular Tree Grammar (IRTG)





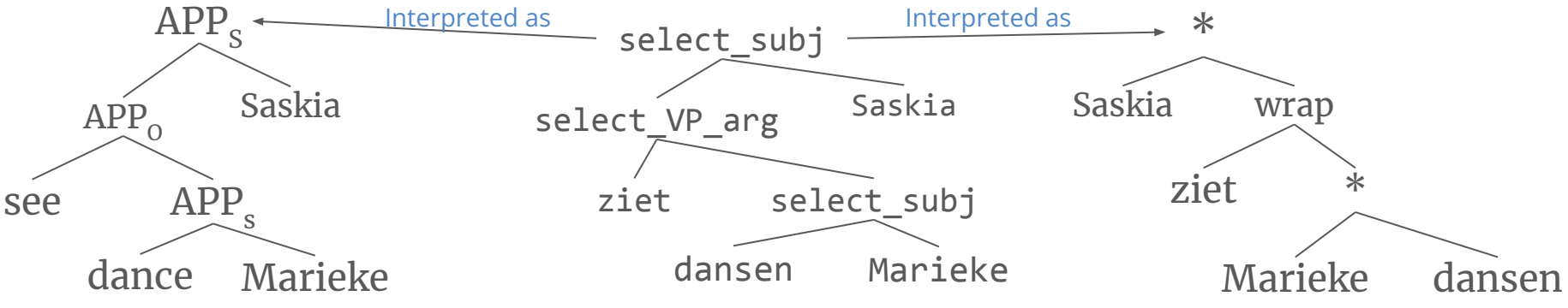
(Saskia ziet, Marieke dansen)

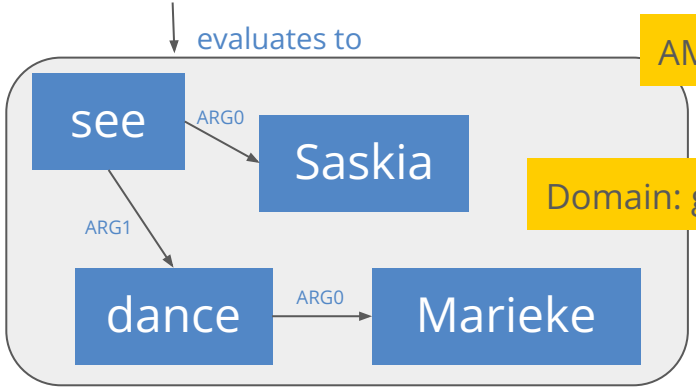
Node labels: grammar rules
(of a Regular Tree Grammar)

Node labels: elements of ranked alphabet (signature)

Not a tree (unless the domain is trees!)

IRTG with multiple interpretations





AM-algebra

Domain: graphs

String-pair algebra

evaluates to

(Saskia Marieke, ziet dansen)

Domain: strings and pairs of strings

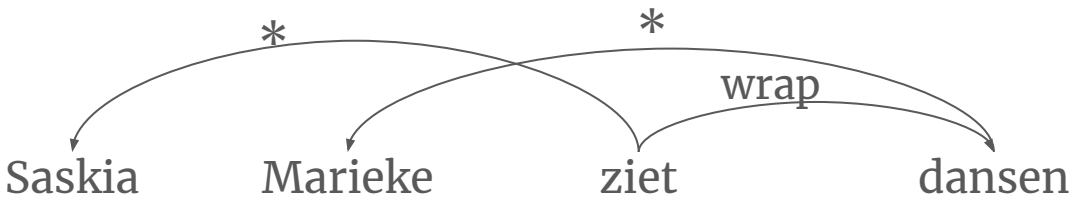
Advantages

- Synchronous grammars
- Captures generalisations, intuitions including
  - TAGs, FTAGs
  - Minimalist Grammars
  - Graph grammars
  - CFGs
- Parsing: IRTGs define a general chart parser (similar to CKY). To apply it to a new algebra, need only define the inverse of the tree homomorphism
- Generative model
- Software: Alto
- Probabilistic grammars supported

Apply-Modify Parser

(Groschwitz et al 2018)

Dependency trees as building instructions



Edge labels: binary algebra operations (alphabet elements of rank 2)

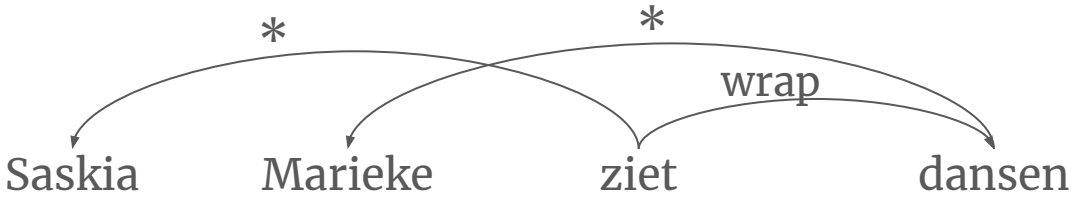
Node labels: algebra constants (alphabet elements of rank 0)

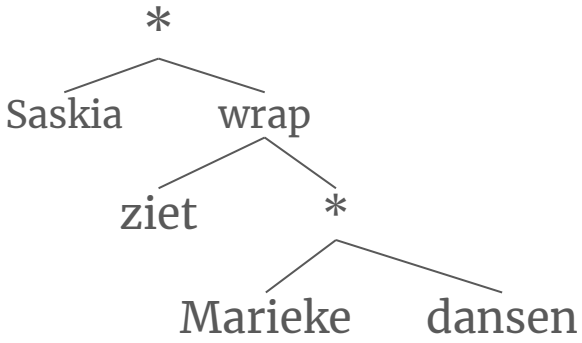
Problems: which order of arguments? Which order of operations?
Solution: e.g. choose a convention (or you might need a slightly different algebra)

Here:

edge (a,b) with label * means *(b, a)
edge (a,b) with label wrap means wrap(a,b)
wrap before *

Example: dependency trees as building instructions





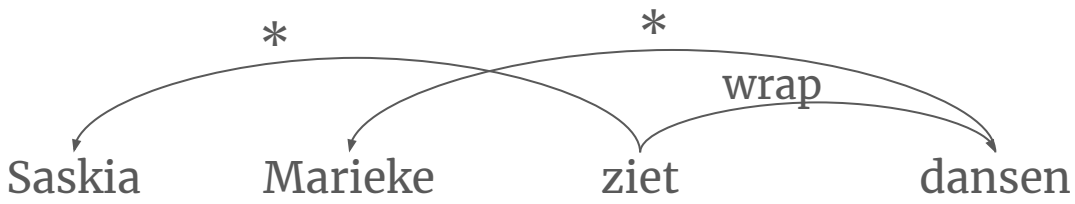
(Saskia Marieke, ziet dansen)

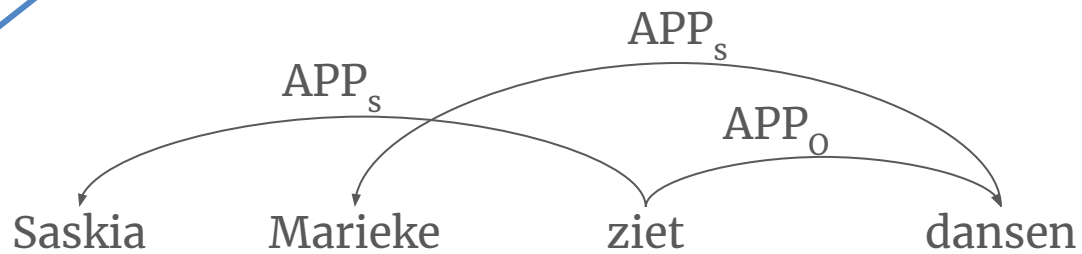
Dependency trees as building instructions: AM parser

The idea: leverage dependency parsing methods to build anything an algebra can build

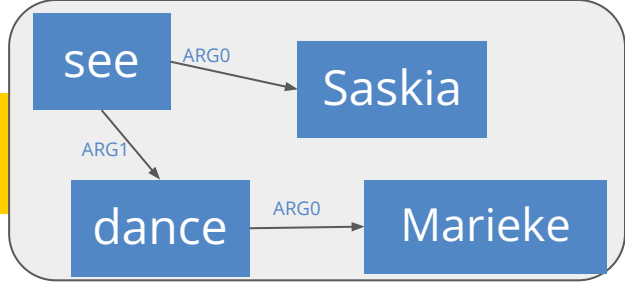
string pair algebra (strings, string pairs)

AM algebra (graphs)

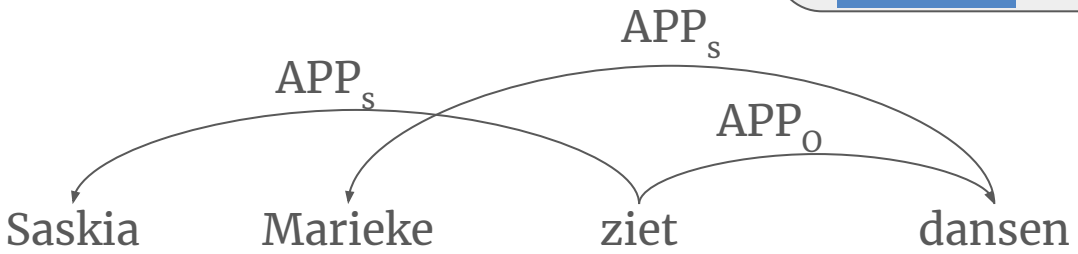




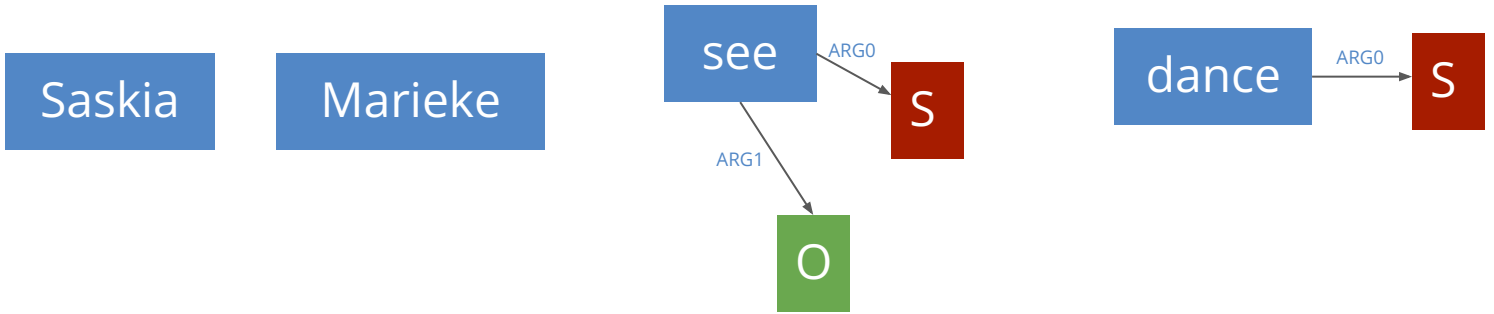
Apply-Modify Parser



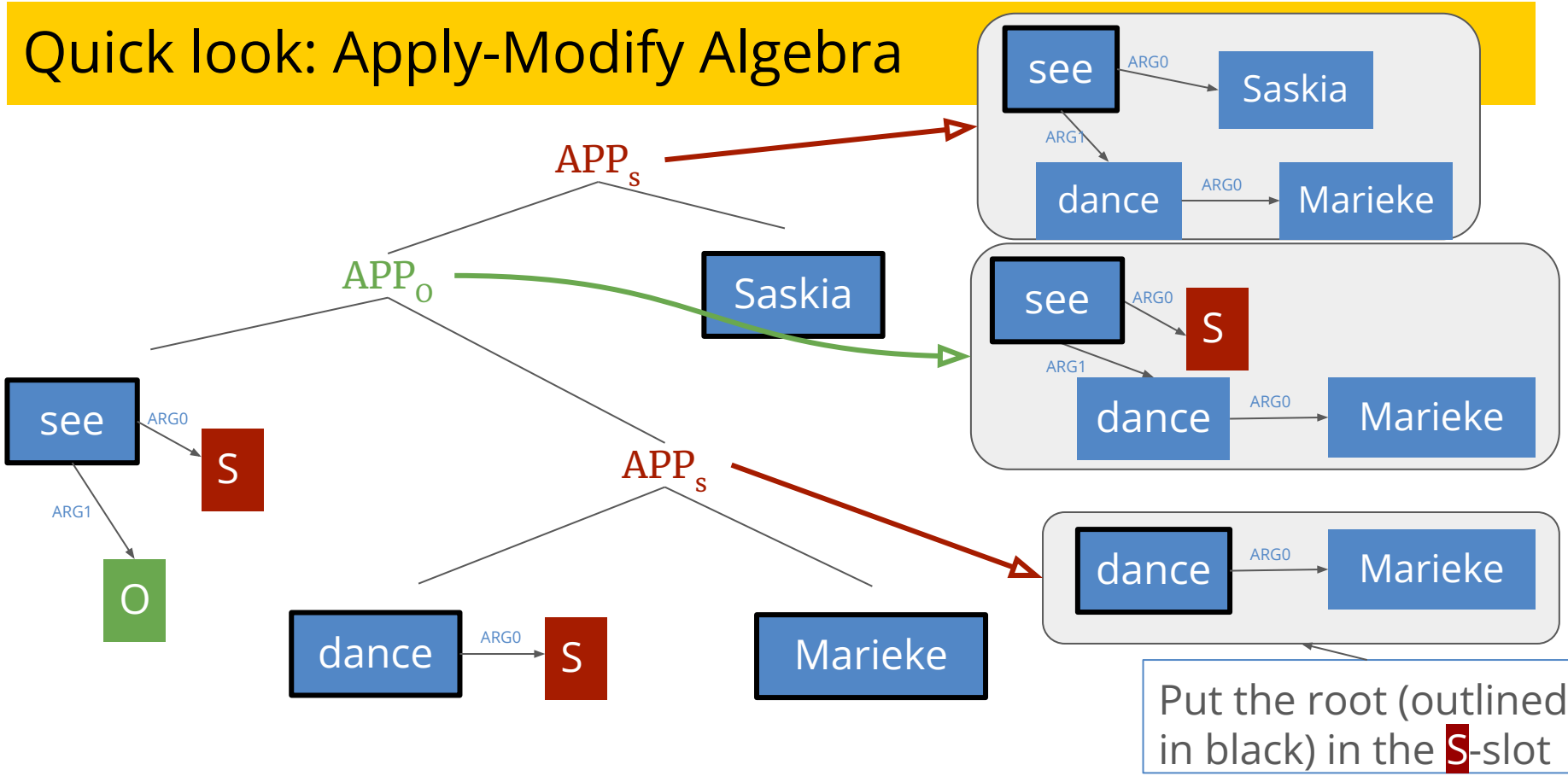
Dependency parser -> dep. tree w/ AM Algebra operations



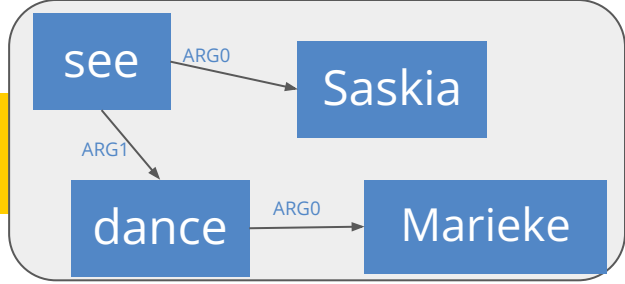
Supertagger chooses AM Algebra constants (graphs) for words



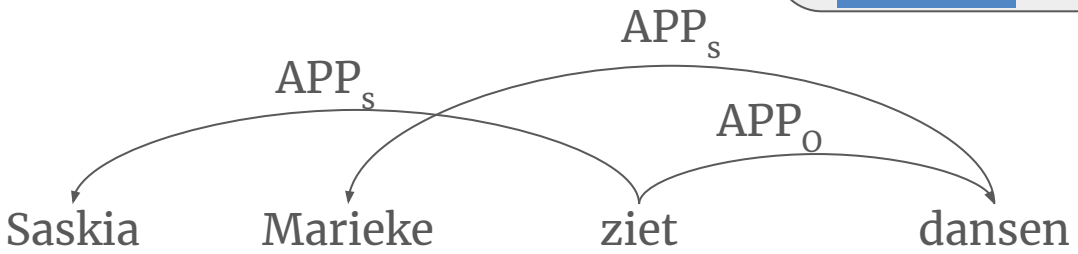
Quick look: Apply-Modify Algebra



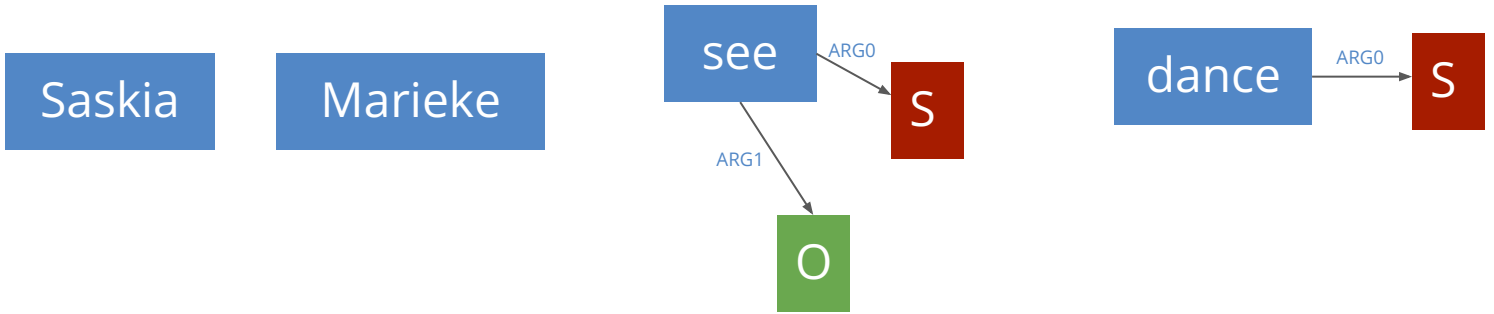
Apply-Modify Parser



Dependency parser -> dep. tree w/ AM Algebra operations



Supertagger chooses AM Algebra constants (graphs) for words



Does it work?

Yes, quite well!

Competitive results on multiple graph banks

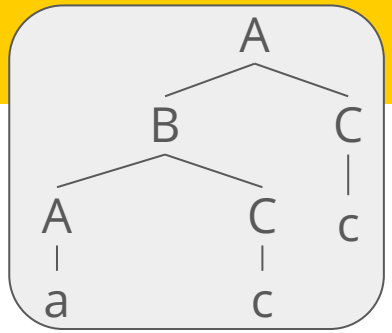
SoTA on compositional generalisation tasks

Fast and accurate algorithms

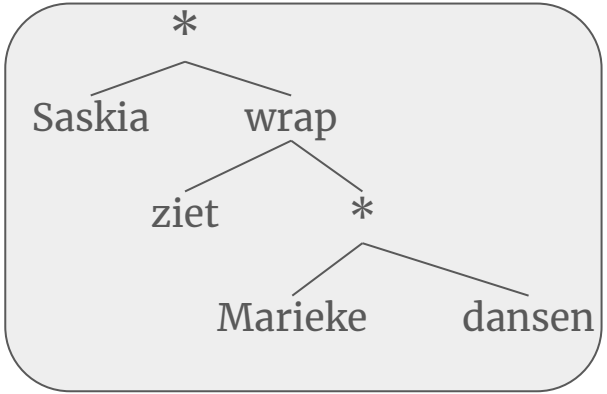
Summing up

Summary

1. Trees as derivational records in a CFG:



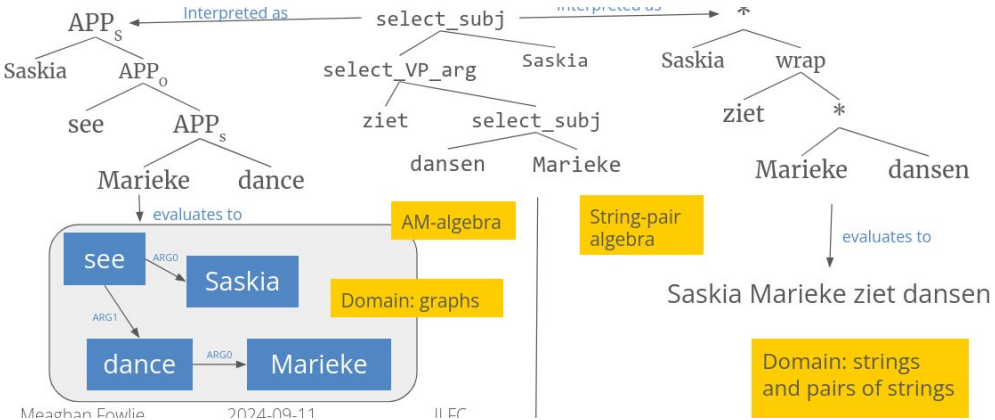
2. Trees as algebra terms
   a. Internal Node labels interpreted as functions
   b. Leaf node labels interpreted as objects in domain



Summary Continued

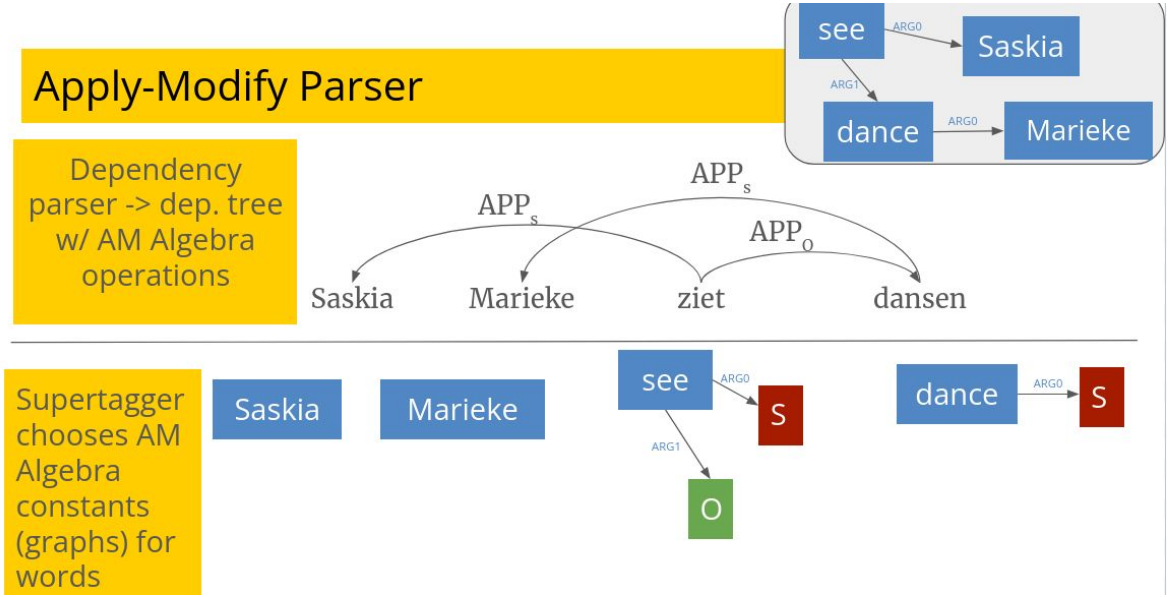
3. Interpreted Regular Tree Grammars

   a. Derivation trees of rules
   b. DT interpreted into algebra terms



More Summary

4. Apply-Modify Parser: Dependency trees as building instructions



References

IRTGs: Alexander Koller and Marco Kuhlmann. 2011. A Generalized View on Parsing and Translation. In *Proceedings of the 12th International Conference on Parsing Technologies*, pages 2–13, Dublin, Ireland. Association for Computational Linguistics.

Code: https://github.com/coli-saar/alto/

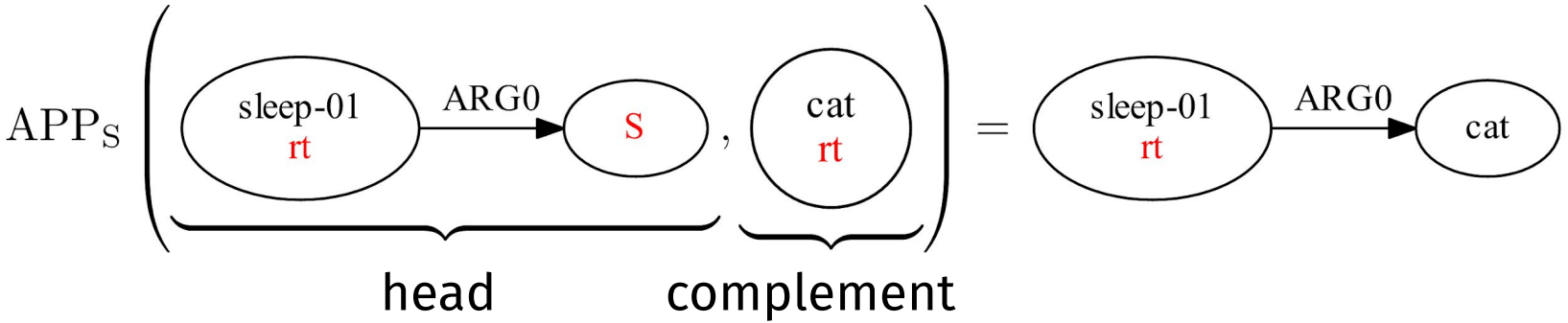
AM Parser: Jonas Groschwitz, Matthias Lindemann, Meaghan Fowlie, Mark Johnson, and Alexander Koller. 2018. AMR dependency parsing with a typed semantic algebra. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1831–1841, Melbourne, Australia. Association for Computational Linguistics.

Code: https://github.com/coli-saar/am-parser

Appendix

Apply Operation

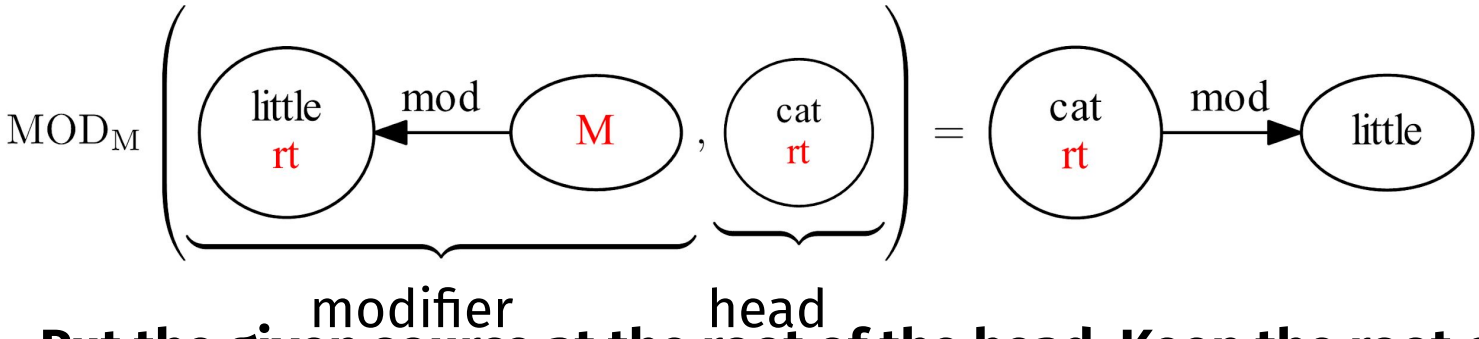
- Combines a **head** with a **complement**.



- **Put the root of the complement into the given source. Combine shared sources.**

Modify Operation

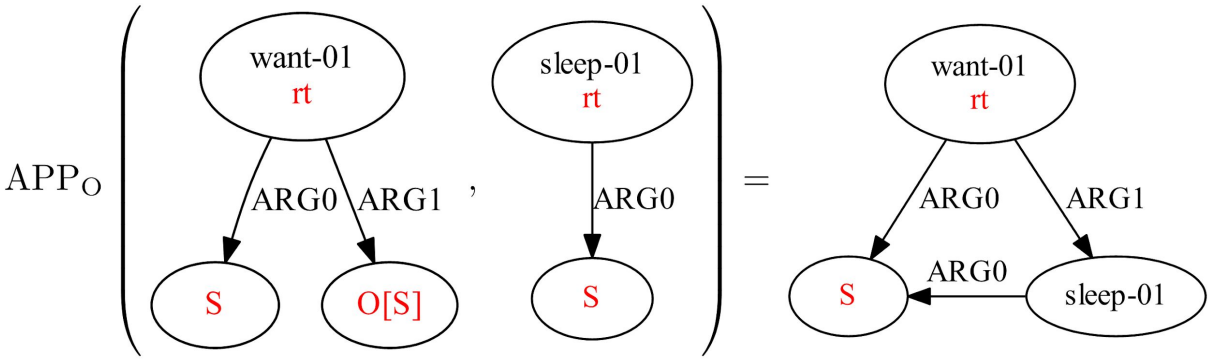
- Combines a **modifier** with a **head**.



- **Put the given source at the root of the head. Keep the root of the head.**
- Warning: these are old pictures; nowadays we always put the head on the left (for APP and MOD)

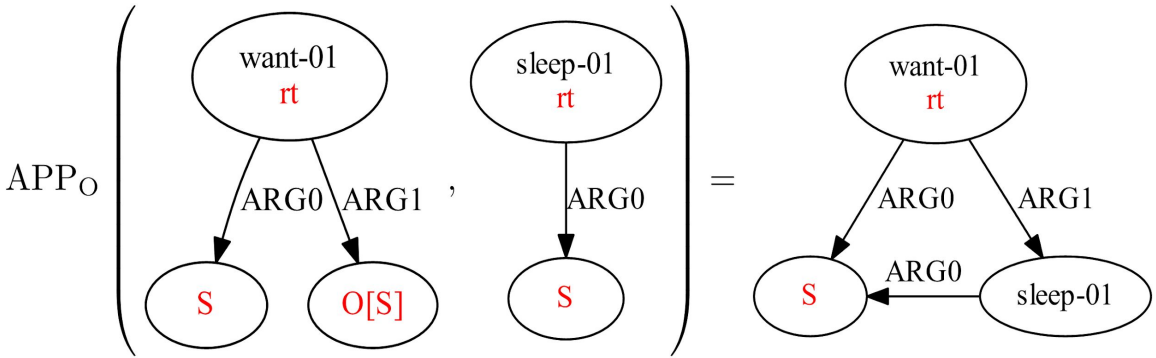
Requests

- When we use APP_O, we want the complement to still have an S-source so that it will merge with the S-source of G[want]
  - This information is **lexical**: *want* is a subject-control verb
  - -> Make it part of G[want]'s type
- The *request* at O is [S]
- At S, we require the complement to have the empty type (only rt-source), so the request at S is []
- Shorthand: annotate the sources in the graph (no annotation means request is [])

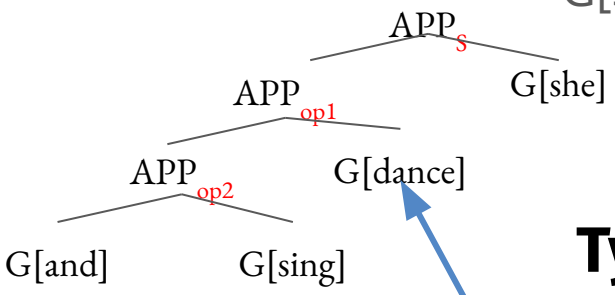


Annotated S-graphs and their *types*

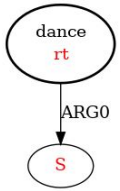
- The *type* of an *annotated s-graph* is its sources and their requests
- Defined recursively: the request at a source is the type of the graph that should fill it. (including *its* requests!)

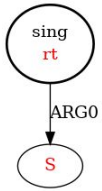


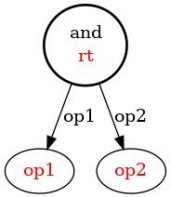
Graph Types



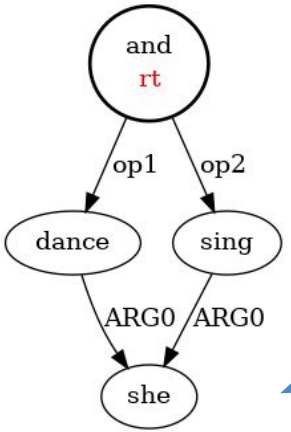
Constants: G[she] G[dance] G[sing] G[and]







Types: [S] [S] (op1 op2 → S) []



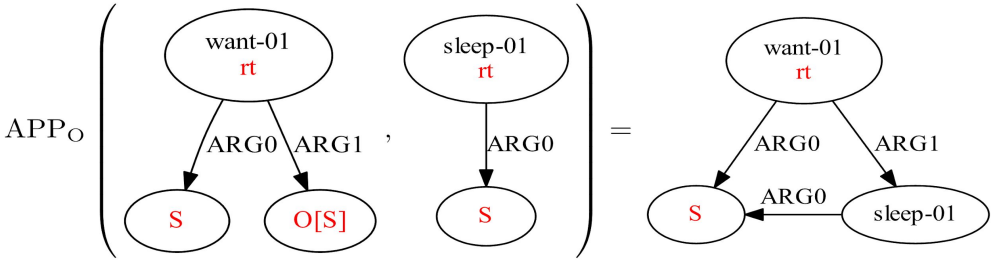
Complement has type [S]

Causes these nodes to merge

request at op1 is [S]
request at op2 is [S]

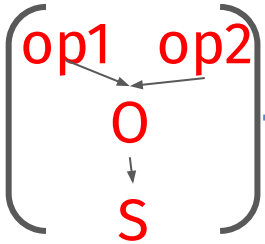
More complex types

Subject control verbs like *want* request an S-source at O

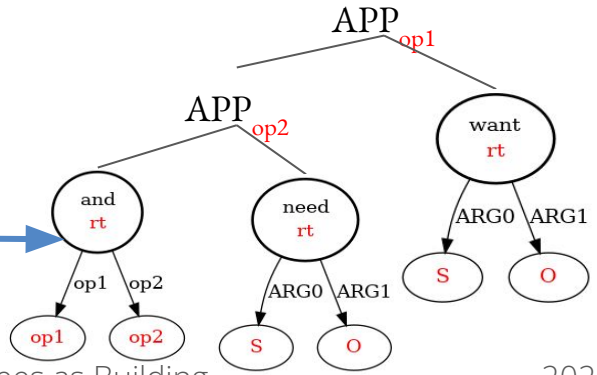


[O -> S] [S] [S]

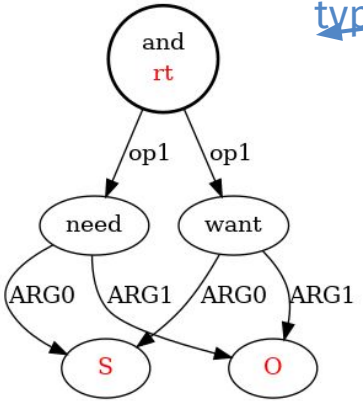
What if we coordinate 2?



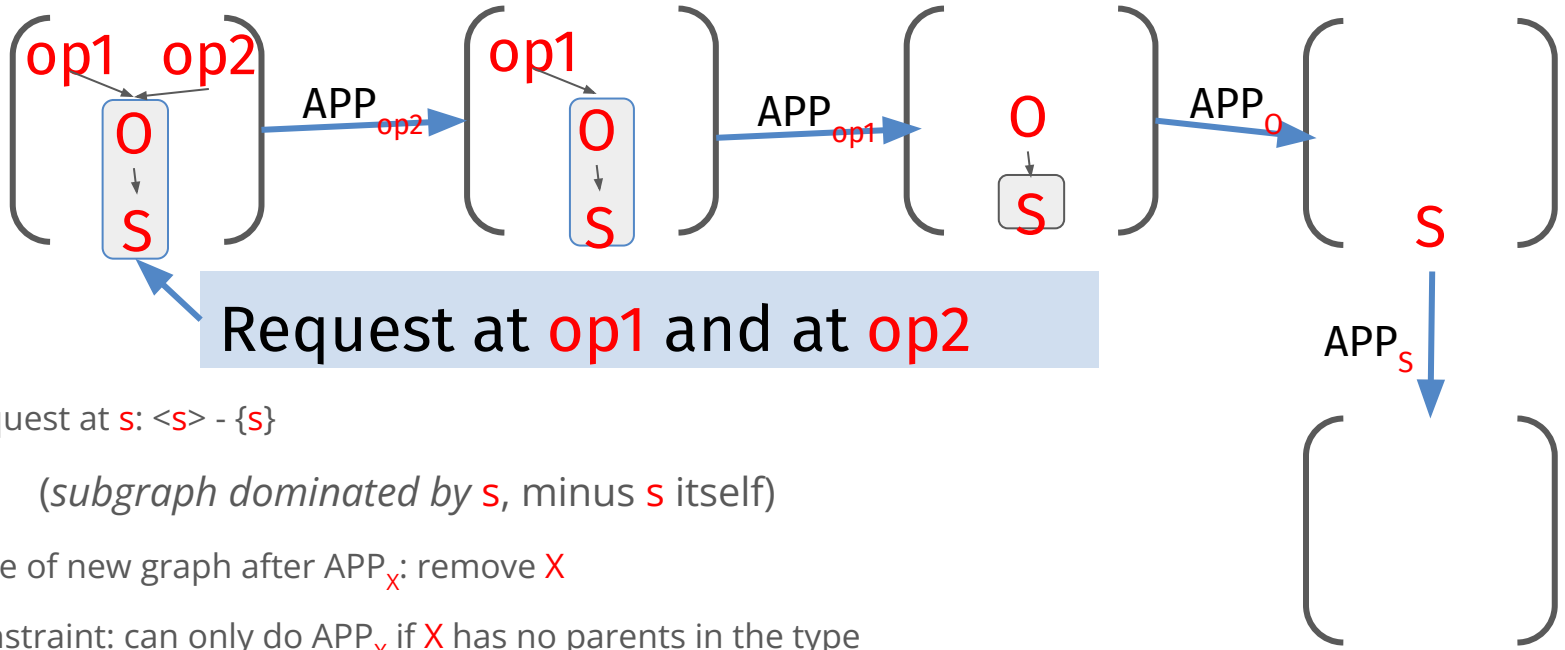
type



type [O->S]



Reading requests off the type



Request at s: <s> - {s}

(*subgraph dominated by* s, minus s itself)

Type of new graph after APP_X: remove X

Constraint: can only do APP_X if X has no parents in the type

Here: could do APP_{op1} and APP_{op2} in either order, BUT everything else is determined

Apply and Modify with types

Apply$_X$(head, complement):

- is allowed only if, in the type of the head, the request at X is the type of the complement
- the type of the new graph is the type of the head, with X removed.

Modify$_X$(head, modifier) is allowed only if:

- The type of the modifier, without X, is a subgraph of the type of the head
- (i.e. modifying the head doesn't change its type)