λ-Calculus in Prolog

Chapter 2.4 of Representation and Inference for Natural Language

C. Millar

Seminar fr Sprachwissenschaft

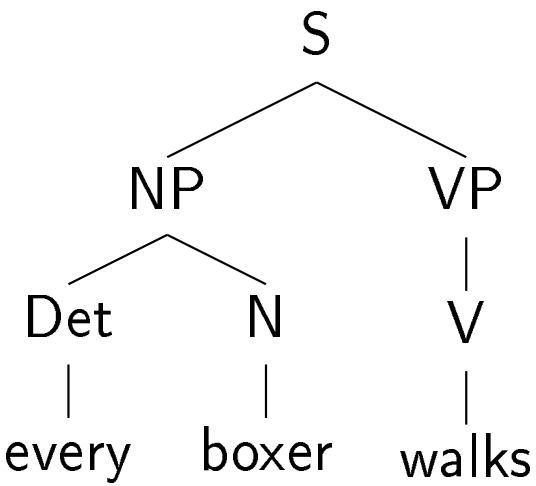
Computational Semantics SS/2008

Outline

Outline

Lambda Calculus

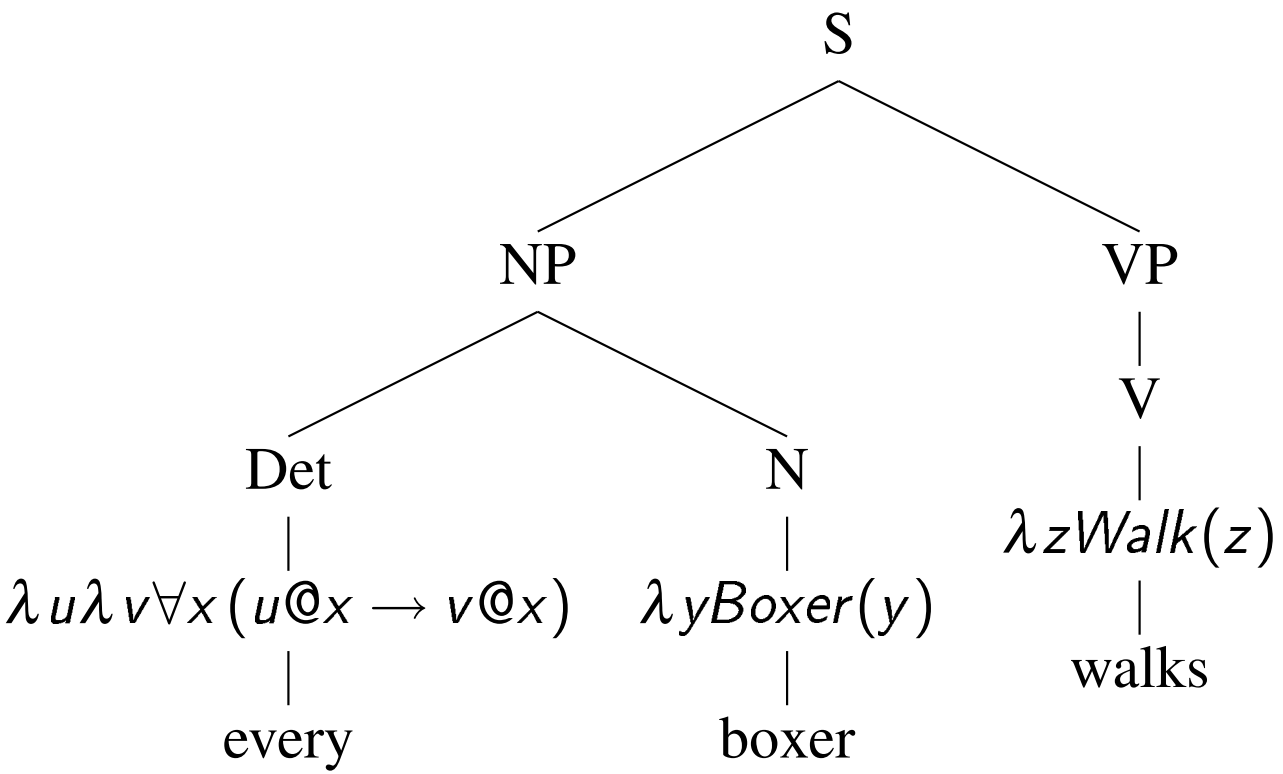
- Consider the sentence "Every boxer walks":



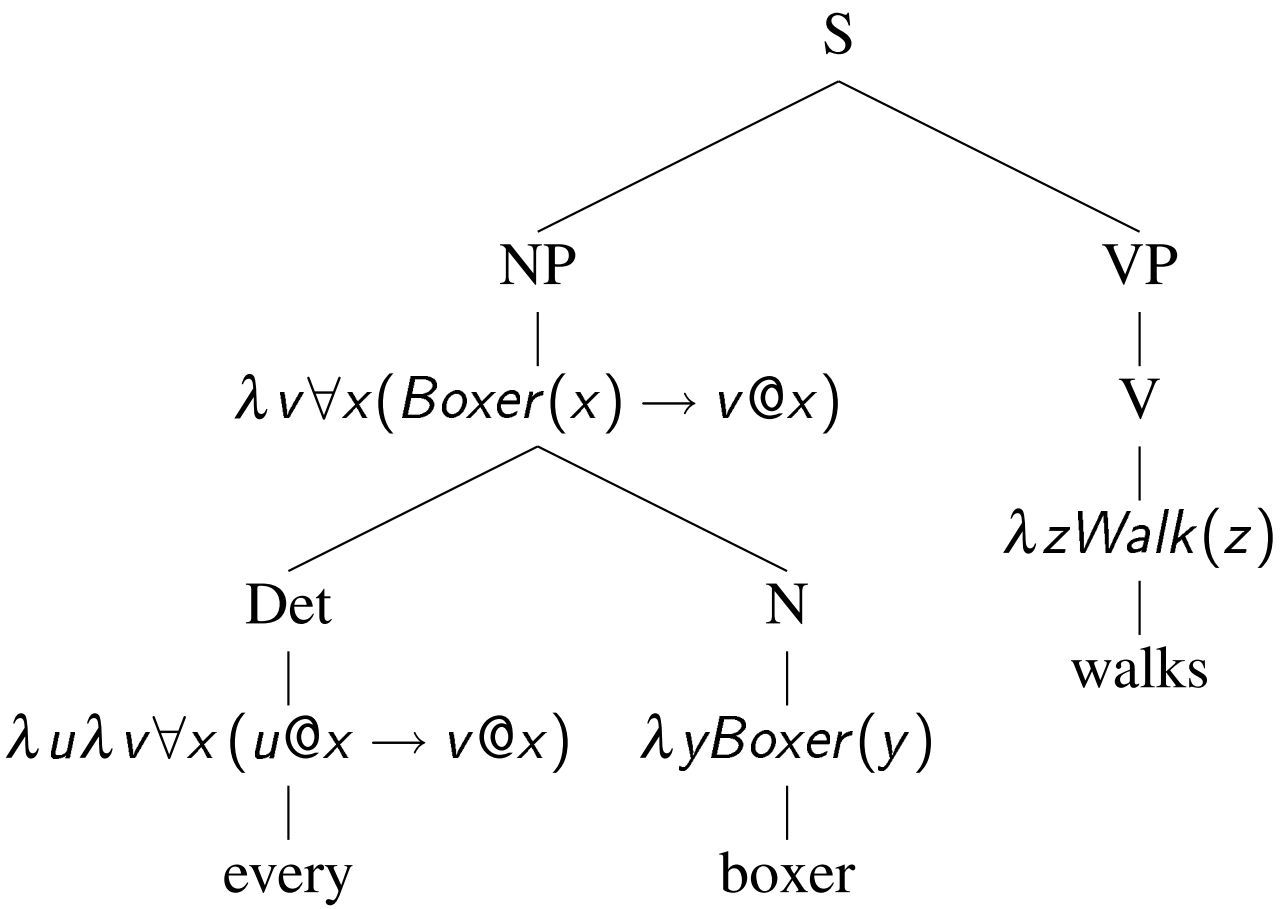
- And the lexicon:

every: $\lambda u \lambda v \forall x (u@x \rightarrow v@x)$
boxer: $\lambda y Boxer(y)$
walks: $\lambda z Walk(z)$

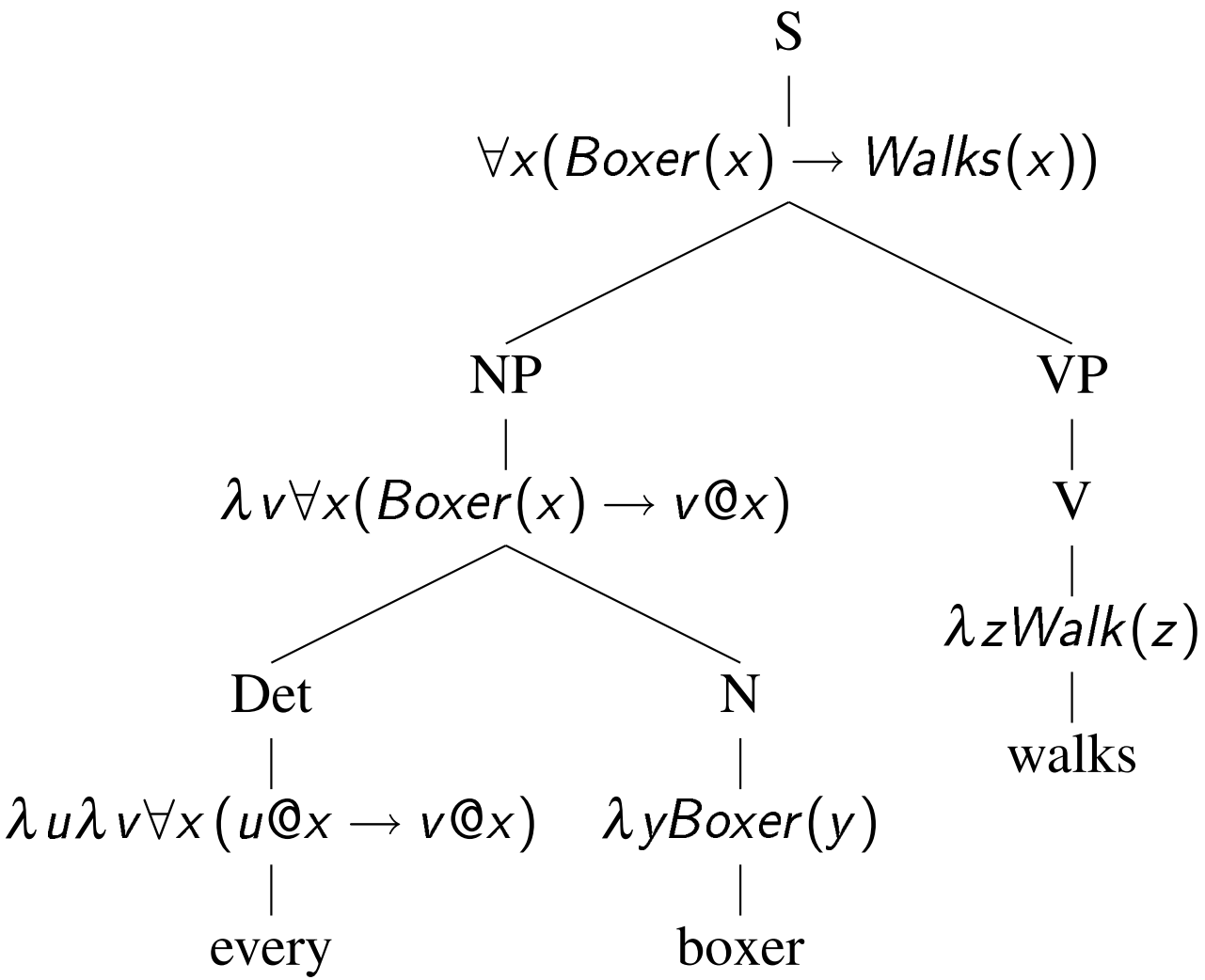
Lambda Calculus



Lambda Calculus



Lambda Calculus



Lambda Calculus

- The syntactic structure of the sentence tells us in which order to apply the rules.
- As humans (and not computers) we intuitively know how to apply these rules and what the result should be.
- What's less obvious is how we did it (β-conversion).
- $\forall x(Boxer(x) \rightarrow Walks(x))$ has the same meaning as:
- $((\lambda u \lambda v \forall x(u@x \rightarrow v@x))@\lambda y Boxer(y))@\lambda z Walk(z)$

Outline

Lambda Syntax á la Prolog

Blackburn and Bos's implementation works by:

1. Building complex lambda formulae such as:
   $((\lambda u \lambda v \forall x (u@x \rightarrow v@x))@\lambda y Boxer(y))@\lambda z Walk(z)$
2. Then β-converting those formulae.
   $\forall x(Boxer(x) \rightarrow Walks(x))$

Lambda Syntax á la Prolog

The lambda formulae are built using a DCG:

```
s(app(NP,VP))–> np(NP), vp(VP).
np(app(Det,Noun))–> det(Det), noun(Noun).
vp(IV)–> iv(IV).
det(lam(P,lam(Q,all(X,imp(app(P,X),app(Q,X))))))–> [every].
noun(lam(X,boxer(X)))–> [boxer].
iv(lam(Y,walk(Y)))–> [walks].
```

Lambda Syntax á la Prolog

Think about what this means...

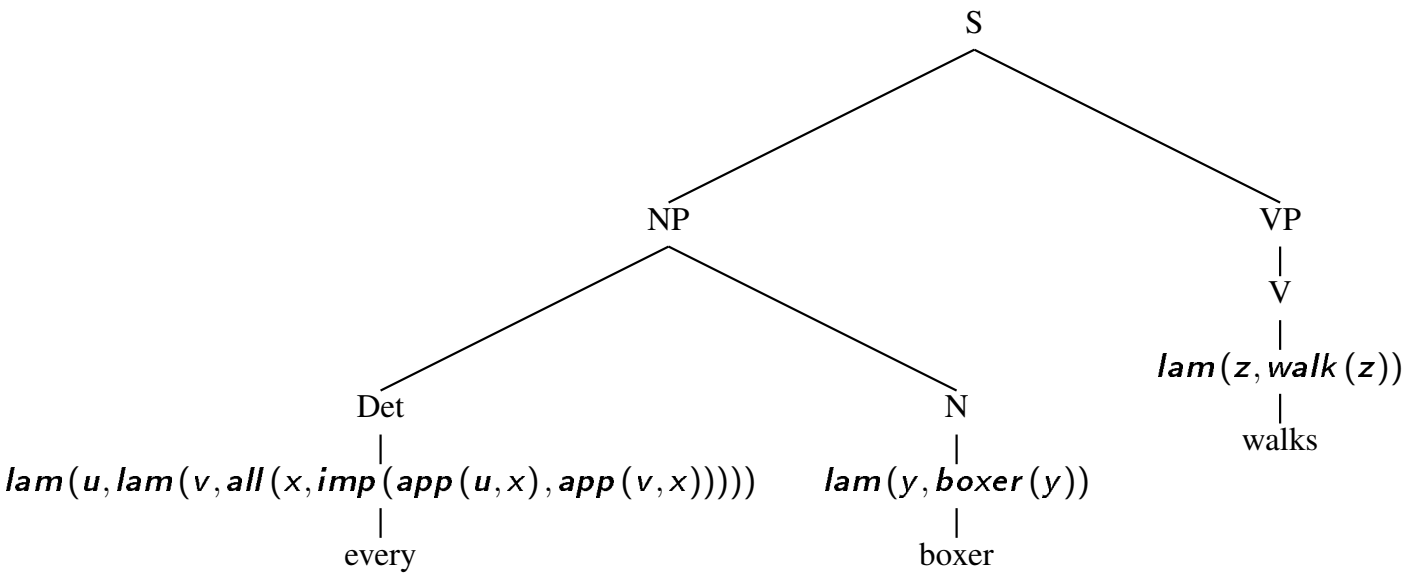
- s(app(NP,VP))–> np(NP), vp(VP).
- The semantic meaning of the S is the functional application of the NP to the VP.
- np(app(Det,Noun))–> det(Det), noun(Noun).
- The semantic meaning of the NP is the functional application of the Det to the Noun.

Lambda Syntax á la Prolog

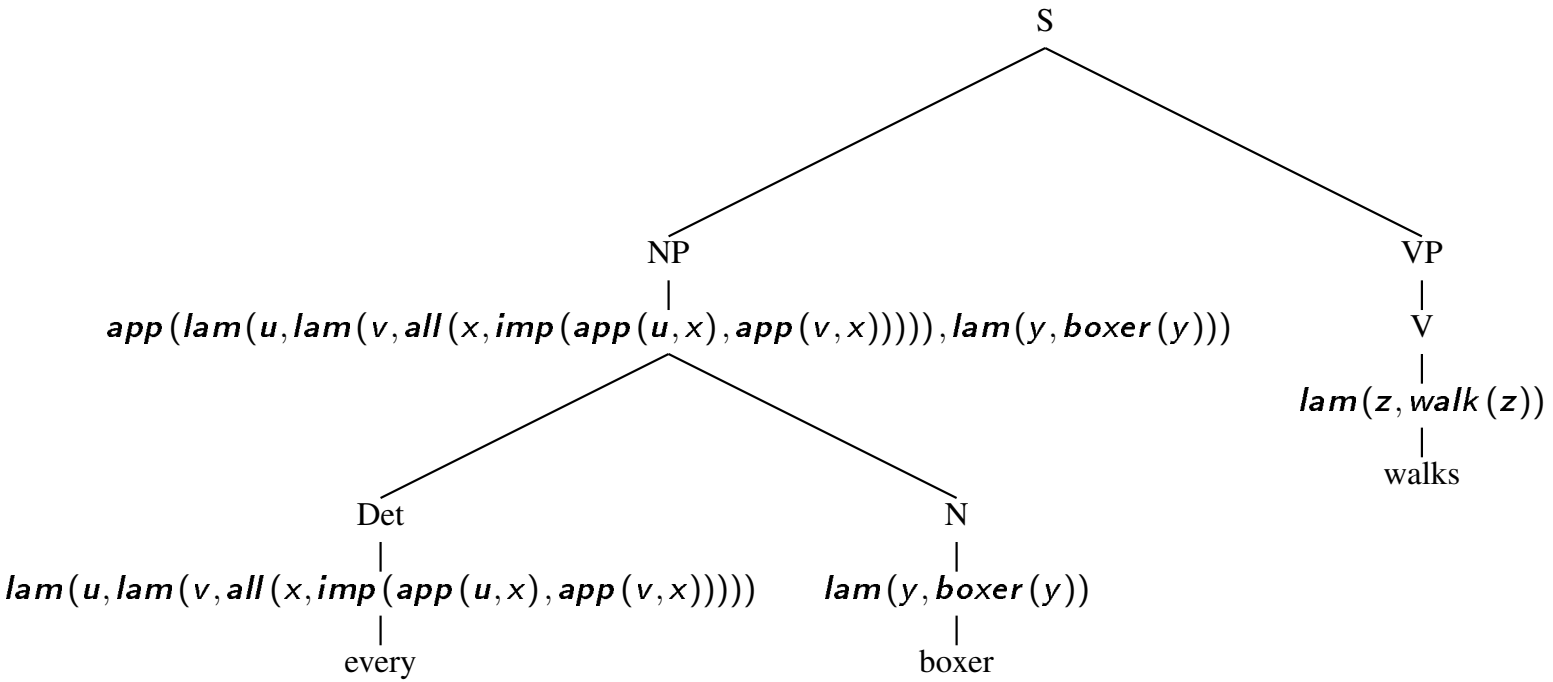
The meaning of the words is captured within the lexical entries. This meaning percolates up the tree by way of the functional applications in the DCG rules.

- det(lam(P,lam(Q,all(X,imp(app(P,X),app(Q,X))))))–> [every].
  - $\lambda u \lambda v \forall x(u@x \rightarrow v@x)$
  - lam(u, lam(v, all(x, imp(app(u, x), app(v, x)))))
- noun(lam(X,boxer(X)))–> [boxer]
  - $\lambda y Boxer(y)$
  - lam(y, boxer(y))
- iv(lam(Y,walk(Y)))–> [walks].
  - $\lambda z Walk(z)$
  - lam(z, walk(z))

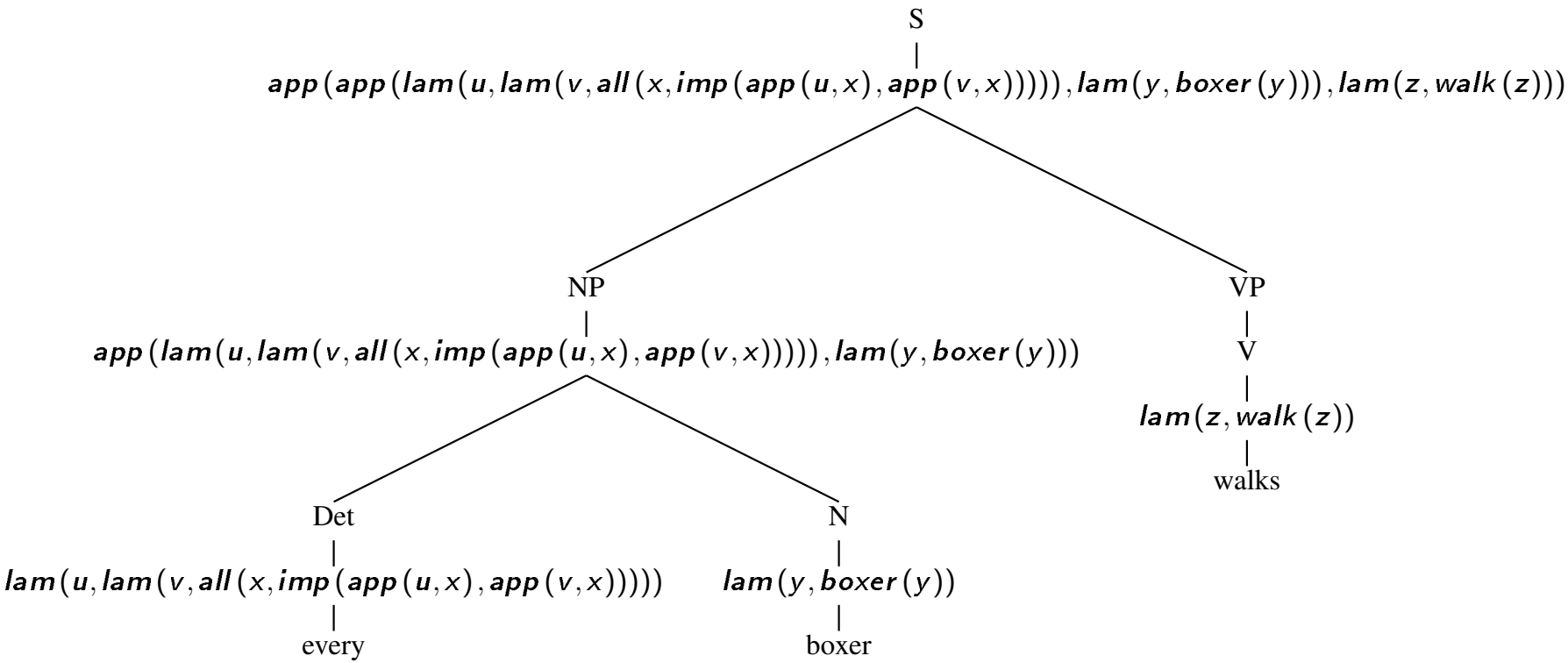
Lambda Syntax á la Prolog



Lambda Syntax á la Prolog



Lambda Syntax á la Prolog



Outline

β-conversion á la Proglog

Now that we know how Blackburn and Bos represent the Lambda Calculus in Prolog let's examine how they β-convert those expressions to valid 1st order logic formulae.

Our Initial call is to betaConvert/2 which calls betaConvert/3 with a new empty stack.

```
betaConvert(X,Y):-
  betaConvert(X,Y,[])
```

If the stack is empty and X is a variable then we have reached our base clause.

```
betaConvert(X,Y,[]):-
  var(X), !,
  Y=X.
```

β-conversion á la Proglog

If "Expression" is not a variable and "Expression" is some application function then push "Argument" onto the stack, α-convert "Functor" and β-convert the α-converted functor "Converted".

We'll look at α-conversion later.

```
betaConvert(Expression,Result,Stack):-
  nonvar(Expression),
  Expression = app(Functor,Argument),
  alphaConvert(Functor,Converted),
  betaConvert(Converted,Result,[Argument|Stack]), !.
```

β-conversion á la Proglog

If "Expression" is not a variable and "Expression" is some lambda function then pop "X" off the stack and substitute the free variable in the lambda expression for "X" then β-convert "Formula".

```
betaConvert(Expression,Result,[X|Stack]):-
   nonvar(Expression),
   Expression = lam(X,Formula),
   betaConvert(Formula,Result,Stack), !.
```

β-conversion á la Proglog

If the stack is empty and "Formula" is not a variable then we use compose/3 to breakdown the "Formula" into a Functor and a list of sub-formulae "Formulas" which are then β-converted using betaConvertList/2 which calls betaConvert/2 on each sub-formula.

When betaConvertList/2 terminates compose/3 is used to build "Result" which will be passed up through all previous calls and eventually become the result of betaConvert/2.

```
betaConvert(Formula,Result,[]):-
  nonvar(Formula), !,
  compose(Formula,Functor,Formulas),
  betaConvertList(Formulas,ResultFormulas),
  compose(Result,Functor,ResultFormulas).
```

Outline

α-conversion á la Proglog

- α-conversion is such a simple idea that as humans (and not computers) it may not be obvious why we need it.
- α-conversion prevents free variables in the argument from being accidentally bound to lambdas of quantifiers in the functor.
- α-conversion changes all bound variables in the functor to variables not used in the argument.

α-conversion á la Proglog

Consider what can go wrong:

every man: $\lambda u \forall y\,(Man(y) \rightarrow u@y)$
loves a woman: $\lambda x \exists y\,(Woman(y) \wedge Love(x,y))$
$\lambda u \forall y\,(Man(y) \rightarrow u@y)@\lambda x \exists y\,(Woman(y) \wedge Love(x,y))$
$\forall y\,(Man(y) \rightarrow \lambda x \exists y\,(Woman(y) \wedge Love(x,y))\,@y)$

$\forall y\,(Man(y) \rightarrow \exists y\,(Woman(y) \wedge Love(y,y)))$

α-conversion á la Proglog

How we fix it:

every man: $\lambda u \forall y\,(Man(y) \rightarrow u@y)$
loves a woman: $\lambda x \exists y\,(Woman(y) \wedge Love(x,y))$
$\lambda u \forall y\,(Man(y) \rightarrow u@y)@\lambda x \exists y\,(Woman(y) \wedge Love(x,y))$

STOP α-convert

$\lambda v \forall z\,(Man(z) \rightarrow v@z)@\lambda x \exists y\,(Woman(y) \wedge Love(x,y))$
$\forall z\,(Man(z) \rightarrow \lambda x \exists y\,(Woman(y) \wedge Love(x,y))@z)$

every man loves a
woman:$\forall z\,(Man(z) \rightarrow \exists y\,(Woman(y) \wedge Love(z,y)))$

Outline

α-conversion á la Proglog

Blackburn and Bos implement predicate which recursively decomposes formulas that need to be α-converted and relabels the bound variables. To accomplish this two lists are needed to keep track of the free and bound variables.

```
alphaConvert(F1,F2):-
   alphaConvert(F1,[],[]-_,F2).
```

The first list is a list of substitutions.
The second is a difference list of free variables.

α-conversion á la Proglog

Now lets look at what passes for a base clause:

```
alphaConvert(X,Sub,Free1-Free2,Y):-
  var(X),
  (
    memberList(sub(Z,Y),Sub),
    X==Z, !,
    Free2=Free1
  ;
    Y=X,
    Free2=[X|Free1]
  ).
```

If X is a variable and part of the list of substitutions then we substitute X for the new variable Y.

Otherwise X is a free variable and we add X to the list of free

α-conversion á la Proglog

If the previous explanation of the base clause was confusing consider this: The base clause only performs substitutions if they are on the list of substitutions. We haven't looked at how the list of substitutions gets built yet!

The list of substitutions is built on the following slide.

α-conversion á la Proglog

If "Expression" is not a variable and it equals the predicate "some/2" then add the pair X,Y to the substitution list and perform the first instance of substitution here (later substitutions will occur when we reach the base clause). Note that the old "Expression" just hangs around un-used... a new "some/2" which uses Y, the next availiable prolog variable, replaces it.

```
alphaConvert(Expression,Sub,Free1-Free2,some(Y,F2)):-
   nonvar(Expression),
   Expression = some(X,F1),
   alphaConvert(F1,[sub(X,Y)|Sub],Free1-Free2,F2).
```

Blackburn and Bos impliment a predicate like this for "some/2", "every/2" and "lam/2". We'll skip the others because their the same.

α-conversion á la Proglog

Now we need a mechanisim to α-convert subexpressions. If "F1" is not a variable and it's not "some/2", "every/2" or "lam/2" then we use "compose/3" to break F1 down into a functor "Symbol" and a list of arguments "Args1". The predicate "alphaConvertList/4" is used to α-convert each argument using "alphaConvert/4". When "alphaConvertList/4" terminates with the Result in Args2 "compose/3" is used to construct the new α-converted formula with the same functor.

```
alphaConvert(F1,Sub,Free1-Free2,F2):-
  nonvar(F1),
  \+ F1 = some(_,_),
  ... ,
  compose(F1,Symbol,Args1),
  alphaConvertList(Args1,Sub,Free1-Free2,Args2),
```

END

Questions?

Blackburn, P and Bos J. Representation and Inference for Natural Language
CSLI Stanford, California.