Appendix A

Grammars for Meaning Representation Languages

This appendix describes the grammars for all of the formal MRLs considered in this thesis, namely the GEOQUERY logical query language, the GEOQUERY functional query language (FUNQL), and CLANG (Section 2.1). These formal MRL grammars are used to train various semantic parsers and tactical generators, including all WASP-based systems and the PHARAOH++ tactical generator (Section 5.3.1).

A.1 The GEOQUERY Logical Query Language

The GEOQUERY logical query language was devised by Zelle (1995, Sec. 7.3) for querying a U.S. geography database called GEOQUERY. Since the database was written in Prolog, the query language is basically first-order Prolog logical forms, augmented with several meta-predicates for dealing with quantification.

There are 14 different non-terminal symbols in this grammar, of which QUERY is the start symbol. The following non-terminal symbols are for entities referenced in the GEOQUERY database:

Entity types	*Non-terminals*	*Sample productions*
City names	CITYNAME	CITYNAME → `austin`
Country names	COUNTRYNAME	COUNTRYNAME → `usa`
Place names	PLACENAME	PLACENAME → `tahoe`
(lakes, mountains, etc.)		
River names	RIVERNAME	RIVERNAME → `mississippi`
State abbreviations	STATEABBREV	STATEABBREV → `tx`
State names	STATENAME	STATENAME → `texas`
Numbers	NUM	NUM → `0`

The following non-terminals are used to disambiguate between entities that share the same name (e.g. the state of Mississippi and the Mississippi river). Note the corresponding Prolog functors (e.g. `stateid` and `riverid`):

Entity types	*Non-terminals*	*Productions*
Cities	CITY	CITY → `cityid(`CITYNAME`,`STATEABBREV`)`
		CITY → `cityid(`CITYNAME`,_)`
Countries	COUNTRY	COUNTRY → `countryid(`COUNTRYNAME`)`
Places	PLACE	PLACE → `placeid(`PLACENAME`)`
Rivers	RIVER	RIVER → `riverid(`RIVERNAME`)`
States	STATE	STATE → `stateid(`STATENAME`)`

The FORM non-terminal (short for "formula") is for the following first-order predicates, which provide most of the expressiveness of the GEOQUERY language. Note that $x_1, x_2, \ldots$ are logical variables that denote entities:

Productions	*Meaning of predicates*
FORM → `capital(`x_1`)`	x_1 is a capital (city).
FORM → `city(`x_1`)`	x_1 is a city.
FORM → `country(`x_1`)`	x_1 is a country.
FORM → `lake(`x_1`)`	x_1 is a lake.
FORM → `major(`x_1`)`	x_1 is major (as in a *major* city or a *major* river).
FORM → `mountain(`x_1`)`	x_1 is a mountain.

Productions	*Meaning of predicates*
FORM $\rightarrow$ `place`(x_1)	x_1 is a place.
FORM $\rightarrow$ `river`(x_1)	x_1 is a river.
FORM $\rightarrow$ `state`(x_1)	x_1 is a state.
FORM $\rightarrow$ `area`(x_1,x_2)	The area of x_1 is x_2.
FORM $\rightarrow$ `capital`(x_1,x_2)	The capital of x_1 is x_2.
FORM $\rightarrow$ `density`(x_1,x_2)	The population density of x_1 is x_2.
FORM $\rightarrow$ `elevation`(x_1,x_2)	The elevation of x_1 is x_2.
FORM $\rightarrow$ `elevation`(x_1,NUM)	The elevation of x_1 is NUM.
FORM $\rightarrow$ `high_point`(x_1,x_2)	The highest point of x_1 is x_2.
FORM $\rightarrow$ `higher`(x_1,x_2)	The elevation of x_1 is greater than that of x_2.
FORM $\rightarrow$ `len`(x_1,x_2)	The length of x_1 is x_2.
FORM $\rightarrow$ `loc`(x_1,x_2)	x_1 is located in x_2.
FORM $\rightarrow$ `longer`(x_1,x_2)	The length of x_1 is greater than that of x_2.
FORM $\rightarrow$ `low_point`(x_1,x_2)	The lowest point of x_1 is x_2.
FORM $\rightarrow$ `lower`(x_1,x_2)	The elevation of x_1 is less than that of x_2.
FORM $\rightarrow$ `next_to`(x_1,x_2)	x_1 is adjacent to x_2.
FORM $\rightarrow$ `population`(x_1,x_2)	The population of x_1 is x_2.
FORM $\rightarrow$ `size`(x_1,x_2)	The size of x_1 is x_2.
FORM $\rightarrow$ `traverse`(x_1,x_2)	x_1 traverses x_2.

The following m-tuples are used to constrain the combinations of entity types that the arguments of a m-place predicate can denote. See Section 4.2.5 for how to use these m-tuples for type checking:

Predicates	*Possible entity types for logical variables*
`capital`(x_1)	(CITY), (PLACE)
`city`(x_1)	(CITY)
`country`(x_1)	(COUNTRY)
`lake`(x_1)	(PLACE), (LAKE)
`major`(x_1)	(CITY), (LAKE), (RIVER)

Predicates	*Possible entity types for logical variables*
mountain(x_1)	(PLACE), (MOUNTAIN)
place(x_1)	(PLACE), (LAKE), (MOUNTAIN)
river(x_1)	(RIVER)
state(x_1)	(STATE)
area(x_1,x_2)	(CITY, NUM), (COUNTRY, NUM), (STATE, NUM)
capital(x_1,x_2)	(STATE, CITY)
density(x_1,x_2)	(CITY, NUM), (COUNTRY, NUM), (STATE, NUM)
elevation(x_1,x_2)	(PLACE, NUM), (MOUNTAIN, NUM)
elevation(x_1,NUM)	(PLACE), (MOUNTAIN)
high_point(x_1,x_2)	(COUNTRY, PLACE), (COUNTRY, MOUNTAIN), (STATE, PLACE), (STATE, MOUNTAIN)
higher(x_1,x_2)	(PLACE, PLACE), (PLACE, MOUNTAIN), (MOUNTAIN, PLACE), (MOUNTAIN, MOUNTAIN)
len(x_1,x_2)	(RIVER, NUM)
loc(x_1,x_2)	(CITY, COUNTRY), (PLACE, COUNTRY), (LAKE, COUNTRY), (MOUNTAIN, COUNTRY), (RIVER, COUNTRY), (STATE, COUNTRY), (CITY, STATE), (PLACE, STATE), (LAKE, STATE), (MOUNTAIN, STATE), (RIVER, STATE), (PLACE, CITY)
longer(x_1,x_2)	(RIVER, RIVER)
low_point(x_1,x_2)	(COUNTRY, PLACE), (COUNTRY, MOUNTAIN), (STATE, PLACE), (STATE, MOUNTAIN)
lower(x_1,x_2)	(PLACE, PLACE), (PLACE, MOUNTAIN), (MOUNTAIN, PLACE), (MOUNTAIN, MOUNTAIN)
next_to(x_1,x_2)	(STATE, RIVER), (STATE, STATE)
population(x_1,x_2)	(CITY, NUM), (COUNTRY, NUM), (STATE, NUM)
size(x_1,x_2)	(CITY, NUM), (COUNTRY, NUM), (PLACE, NUM), (LAKE, NUM), (MOUNTAIN, NUM), (RIVER, NUM), (STATE, NUM)

Predicates	*Possible entity types for logical variables*
`traverse(`x_1`,`x_2`)`	(RIVER, CITY), (RIVER, COUNTRY), (RIVER, STATE)

In addition, the `equal` predicate is used to equate logical variables to ground terms, e.g. `equal(`x_1`,cityid(austin,tx))`:

Productions	*Possible entity types for logical variables*
FORM → `equal(`x_1`,`CITY`)`	(CITY)
FORM → `equal(`x_1`,`COUNTRY`)`	(COUNTRY)
FORM → `equal(`x_1`,`PLACE`)`	(PLACE), (LAKE), (MOUNTAIN)
FORM → `equal(`x_1`,`RIVER`)`	(RIVER)
FORM → `equal(`x_1`,`STATE`)`	(STATE)

Another important production is the conjunction operator (`,`), which is used to form conjunctions of formulas:

FORM → `(`FORM`,`FORM`)`

The `not` operator is used to form negations:

FORM → `not(`FORM`)`

The FORM non-terminal is also for the following meta-predicates, which take conjunctive goals as their arguments:

Productions	*Meaning of meta-predicates*
FORM → `largest(`x_1`,`FORM`)`	The goal denoted by FORM produces only the solution maximizing the size of x_1.
FORM → `smallest(`x_1`,`FORM`)`	The goal denoted by FORM produces only the solution minimizing the size of x_1.
FORM → `highest(`x_1`,`FORM`)`	Analogous to `largest` (with elevation).
FORM → `lowest(`x_1`,`FORM`)`	Analogous to `smallest` (with elevation).

Productions	*Meaning of meta-predicates*
FORM → `longest(`x_1`,`FORM`)`	Analogous to `largest` (with length).
FORM → `shortest(`x_1`,`FORM`)`	Analogous to `smallest` (with length).
FORM → `count(`x_1`,`FORM`,`x_2`)`	x_2 is the number of bindings for x_1 satisfying the goal denoted by FORM.
FORM → `sum(`x_1`,`FORM`,`x_2`)`	x_2 is the sum of all bindings for x_1 satisfying the goal denoted by FORM.
FORM → `most(`x_1`,`x_2`,`FORM`)`	The goal denoted by FORM produces only the x_1 maximizing the count of x_2.
FORM → `fewest(`x_1`,`x_2`,`FORM`)`	The goal denoted by FORM produces only the x_1 minimizing the count of x_2.

Below are the corresponding m-tuples of entity types for type checking:

Meta-predicates	*Possible entity types for logical variables*
`largest(`x_1`,`FORM`)`	(CITY), (PLACE), (LAKE), (MOUNTAIN), (NUM), (RIVER), (STATE)
`smallest(`x_1`,`FORM`)`	(CITY), (PLACE), (LAKE), (MOUNTAIN), (NUM), (RIVER), (STATE)
`highest(`x_1`,`FORM`)`	(PLACE), (MOUNTAIN)
`lowest(`x_1`,`FORM`)`	(PLACE), (MOUNTAIN)
`longest(`x_1`,`FORM`)`	(RIVER)
`shortest(`x_1`,`FORM`)`	(RIVER)
`count(`x_1`,`FORM`,`x_2`)`	($*$, NUM)
`sum(`x_1`,`FORM`,`x_2`)`	(NUM, NUM)
`most(`x_1`,`x_2`,`FORM`)`	($*$, $*$)
`fewest(`x_1`,`x_2`,`FORM`)`	($*$, $*$)

In the above table, $*$ denotes any of these entity types: CITY, COUNTRY, PLACE, LAKE, MOUNTAIN, NUM, RIVER, STATE.

Finally, the start symbol, QUERY, is reserved for the `answer` meta-predicate, which serves as a wrapper for query goals (denoted by FORM):

QUERY $\rightarrow$ `answer(`x_1`,`FORM`)`

Here x_1 is the logical variable whose binding is of interest (i.e. answers the question posed). x_1 can denote entities of any type ($*$).

A.2 The GEOQUERY Functional Query Language

For semantic parsers and tactical generators that cannot handle logical variables (e.g. WASP, PHARAOH++, WASP^{-1}++), a variable-free, functional query language called FUNQL has been devised for the GEOQUERY domain (Kate et al., 2005). Below is a sample FUNQL query, together with its corresponding Prolog logical form:

What are the cities in Texas?
FUNQL: `answer(city(loc_2(stateid(texas))))`
Prolog logical form: `answer(`x_1`,(city(`x_1`),loc(`x_1`,`x_2`),`
`equal(`x_2`,stateid(texas))))`

In Section 2.1, we noted that FUNQL predicates can have a set-theoretic interpretation. For example, the term `stateid(texas)` denotes a singleton set that consists of the Texas state, and `loc_2(stateid(texas))` denotes the set of entities located in the Texas state, and so on. Here we present another interpretation of FUNQL based on the lambda calculus. Under this interpretation, each FUNQL predicate is a shorthand for a λ-function, which can be used to translate FUNQL expressions into the GEOQUERY logical query language through function application. For example, the FUNQL predicate `stateid` denotes the λ-function $\lambda n.\lambda x_1.$`equal(`x_1`,stateid(`n`))`. Hence by function application, the FUNQL term `stateid(texas)` is equivalent to the following logical form in the GEOQUERY logical query language:

λx_1.equal(x_1,stateid(texas))

Also since the FUNQL predicate loc_2 denotes $\lambda p.\lambda x_1$.(loc(x_1,x_2),$p(x_2)$), the FUNQL term loc_2(stateid(texas)) is equivalent to:

λx_1.loc(x_1,x_2),equal(x_2,stateid(texas)))

There are 13 different non-terminal symbols in the FUNQL grammar. All of them are from the GEOQUERY logical query language. Only the FORM non-terminal is not used in FUNQL. QUERY is the start symbol in the FUNQL grammar.

Below are the FUNQL productions for named entities and numbers, which are identical to those in the GEOQUERY logical query language:

Entity types	*Sample productions*	*Corresponding λ-functions*
City names	CITYNAME $\rightarrow$ austin	austin
Country names	COUNTRYNAME $\rightarrow$ usa	usa
Place names	PLACENAME $\rightarrow$ tahoe	tahoe
River names	RIVERNAME $\rightarrow$ mississippi	mississippi
State abbreviations	STATEABBREV $\rightarrow$ tx	tx
State names	STATENAME $\rightarrow$ texas	texas
Numbers	NUM $\rightarrow$ 0	0

The rest of the FUNQL productions are as follows:

Productions	*Corresponding λ-functions*
CITY $\rightarrow$ cityid(CITYNAME,STATEABBREV)	$\lambda n.\lambda a.\lambda x_1$.equal($x_1$,cityid($n$,$a$))
CITY $\rightarrow$ cityid(CITYNAME,_)	$\lambda n.\lambda x_1$.equal(x_1,cityid(n,_))
COUNTRY $\rightarrow$ countryid(COUNTRYNAME)	$\lambda n.\lambda x_1$.equal(x_1,countryid(n))
PLACE $\rightarrow$ placeid(PLACENAME)	$\lambda n.\lambda x_1$.equal(x_1,placeid(n))
RIVER $\rightarrow$ riverid(RIVERNAME)	$\lambda n.\lambda x_1$.equal(x_1,riverid(n))
STATE $\rightarrow$ stateid(STATENAME)	$\lambda n.\lambda x_1$.equal(x_1,stateid(n))

Productions	*Corresponding λ-functions*
CITY → `capital(all)`	λx_1.`capital(`x_1`)`
CITY → `city(all)`	λx_1.`city(`x_1`)`
COUNTRY → `country(all)`	λx_1.`country(`x_1`)`
PLACE → `lake(all)`	λx_1.`lake(`x_1`)`
PLACE → `mountain(all)`	λx_1.`mountain(`x_1`)`
PLACE → `place(all)`	λx_1.`place(`x_1`)`
RIVER → `river(all)`	λx_1.`river(`x_1`)`
STATE → `state(all)`	λx_1.`state(`x_1`)`
CITY → `capital(`CITY`)`	$\lambda p.\lambda x_1$.`(capital(`x_1`),`$p(x_1)$`)`
CITY → `capital(`PLACE`)`	$\lambda p.\lambda x_1$.`(capital(`x_1`),`$p(x_1)$`)`
CITY → `city(`CITY`)`	$\lambda p.\lambda x_1$.`(city(`x_1`),`$p(x_1)$`)`
PLACE → `lake(`PLACE`)`	$\lambda p.\lambda x_1$.`(lake(`x_1`),`$p(x_1)$`)`
CITY → `major(`CITY`)`	$\lambda p.\lambda x_1$.`(major(`x_1`),`$p(x_1)$`)`
PLACE → `major(`PLACE`)`	$\lambda p.\lambda x_1$.`(major(`x_1`),`$p(x_1)$`)`
RIVER → `major(`RIVER`)`	$\lambda p.\lambda x_1$.`(major(`x_1`),`$p(x_1)$`)`
PLACE → `mountain(`PLACE`)`	$\lambda p.\lambda x_1$.`(mountain(`x_1`),`$p(x_1)$`)`
PLACE → `place(`PLACE`)`	$\lambda p.\lambda x_1$.`(place(`x_1`),`$p(x_1)$`)`
RIVER → `river(`RIVER`)`	$\lambda p.\lambda x_1$.`(river(`x_1`),`$p(x_1)$`)`
STATE → `state(`STATE`)`	$\lambda p.\lambda x_1$.`(state(`x_1`),`$p(x_1)$`)`
NUM → `area_1(`CITY`)`	$\lambda p.\lambda x_1$.`(area(`x_2`,`x_1`),`$p(x_2)$`)`
NUM → `area_1(`COUNTRY`)`	$\lambda p.\lambda x_1$.`(area(`x_2`,`x_1`),`$p(x_2)$`)`
NUM → `area_1(`PLACE`)`	$\lambda p.\lambda x_1$.`(area(`x_2`,`x_1`),`$p(x_2)$`)`
NUM → `area_1(`STATE`)`	$\lambda p.\lambda x_1$.`(area(`x_2`,`x_1`),`$p(x_2)$`)`
CITY → `capital_1(`COUNTRY`)`	$\lambda p.\lambda x_1$.`(capital(`x_2`,`x_1`),`$p(x_2)$`)`
CITY → `capital_1(`STATE`)`	$\lambda p.\lambda x_1$.`(capital(`x_2`,`x_1`),`$p(x_2)$`)`
STATE → `capital_2(`CITY`)`	$\lambda p.\lambda x_1$.`(capital(`x_1`,`x_2`),`$p(x_2)$`)`
NUM → `density_1(`CITY`)`	$\lambda p.\lambda x_1$.`(density(`x_2`,`x_1`),`$p(x_2)$`)`
NUM → `density_1(`COUNTRY`)`	$\lambda p.\lambda x_1$.`(density(`x_2`,`x_1`),`$p(x_2)$`)`
NUM → `density_1(`STATE`)`	$\lambda p.\lambda x_1$.`(density(`x_2`,`x_1`),`$p(x_2)$`)`
NUM → `elevation_1(`PLACE`)`	$\lambda p.\lambda x_1$.`(elevation(`x_2`,`x_1`),`$p(x_2)$`)`
PLACE → `elevation_2(`NUM`)`	$\lambda n.\lambda x_1$.`elevation(`x_1`,`n`)`

Productions	*Corresponding λ-functions*
PLACE → high_point_1(STATE)	$\lambda p.\lambda x_1.$(high_point(x_2,x_1),$p(x_2)$)
STATE → high_point_2(PLACE)	$\lambda p.\lambda x_1.$(high_point(x_1,x_2),$p(x_2)$)
PLACE → higher_2(PLACE)	$\lambda p.\lambda x_1.$(higher(x_1,x_2),$p(x_2)$)
NUM → len(RIVER)	$\lambda p.\lambda x_1.$(len(x_2,x_1),$p(x_2)$)
CITY → loc_1(PLACE)	$\lambda p.\lambda x_1.$(loc(x_2,x_1),$p(x_2)$)
COUNTRY → loc_1(CITY)	$\lambda p.\lambda x_1.$(loc(x_2,x_1),$p(x_2)$)
COUNTRY → loc_1(PLACE)	$\lambda p.\lambda x_1.$(loc(x_2,x_1),$p(x_2)$)
COUNTRY → loc_1(RIVER)	$\lambda p.\lambda x_1.$(loc(x_2,x_1),$p(x_2)$)
COUNTRY → loc_1(STATE)	$\lambda p.\lambda x_1.$(loc(x_2,x_1),$p(x_2)$)
STATE → loc_1(CITY)	$\lambda p.\lambda x_1.$(loc(x_2,x_1),$p(x_2)$)
STATE → loc_1(PLACE)	$\lambda p.\lambda x_1.$(loc(x_2,x_1),$p(x_2)$)
STATE → loc_1(RIVER)	$\lambda p.\lambda x_1.$(loc(x_2,x_1),$p(x_2)$)
CITY → loc_2(COUNTRY)	$\lambda p.\lambda x_1.$(loc(x_1,x_2),$p(x_2)$)
CITY → loc_2(STATE)	$\lambda p.\lambda x_1.$(loc(x_1,x_2),$p(x_2)$)
PLACE → loc_2(CITY)	$\lambda p.\lambda x_1.$(loc(x_1,x_2),$p(x_2)$)
PLACE → loc_2(STATE)	$\lambda p.\lambda x_1.$(loc(x_1,x_2),$p(x_2)$)
PLACE → loc_2(COUNTRY)	$\lambda p.\lambda x_1.$(loc(x_1,x_2),$p(x_2)$)
RIVER → loc_2(COUNTRY)	$\lambda p.\lambda x_1.$(loc(x_1,x_2),$p(x_2)$)
RIVER → loc_2(STATE)	$\lambda p.\lambda x_1.$(loc(x_1,x_2),$p(x_2)$)
STATE → loc_2(COUNTRY)	$\lambda p.\lambda x_1.$(loc(x_1,x_2),$p(x_2)$)
RIVER → longer(RIVER)	$\lambda p.\lambda x_1.$(longer(x_1,x_2),$p(x_2)$)
PLACE → lower_2(PLACE)	$\lambda p.\lambda x_1.$(lower(x_1,x_2),$p(x_2)$)
STATE → next_to_1(STATE)	$\lambda p.\lambda x_1.$(next_to(x_2,x_1),$p(x_2)$)
STATE → next_to_2(STATE)	$\lambda p.\lambda x_1.$(next_to(x_1,x_2),$p(x_2)$)
STATE → next_to_2(RIVER)	$\lambda p.\lambda x_1.$(next_to(x_1,x_2),$p(x_2)$)
NUM → population_1(CITY)	$\lambda p.\lambda x_1.$(population(x_2,x_1),$p(x_2)$)
NUM → population_1(COUNTRY)	$\lambda p.\lambda x_1.$(population(x_2,x_1),$p(x_2)$)
NUM → population_1(STATE)	$\lambda p.\lambda x_1.$(population(x_2,x_1),$p(x_2)$)
NUM → size(CITY)	$\lambda p.\lambda x_1.$(size(x_2,x_1),$p(x_2)$)
NUM → size(COUNTRY)	$\lambda p.\lambda x_1.$(size(x_2,x_1),$p(x_2)$)
NUM → size(STATE)	$\lambda p.\lambda x_1.$(size(x_2,x_1),$p(x_2)$)

Productions	*Corresponding λ-functions*
CITY $\rightarrow$ `traverse_1(`RIVER`)`	$\lambda p.\lambda x_1.$`(traverse(`x_2`,`x_1`),`$p(x_2)$`)`
COUNTRY $\rightarrow$ `traverse_1(`RIVER`)`	$\lambda p.\lambda x_1.$`(traverse(`x_2`,`x_1`),`$p(x_2)$`)`
STATE $\rightarrow$ `traverse_1(`RIVER`)`	$\lambda p.\lambda x_1.$`(traverse(`x_2`,`x_1`),`$p(x_2)$`)`
RIVER $\rightarrow$ `traverse_2(`CITY`)`	$\lambda p.\lambda x_1.$`(traverse(`x_1`,`x_2`),`$p(x_2)$`)`
RIVER $\rightarrow$ `traverse_2(`COUNTRY`)`	$\lambda p.\lambda x_1.$`(traverse(`x_1`,`x_2`),`$p(x_2)$`)`
RIVER $\rightarrow$ `traverse_2(`STATE`)`	$\lambda p.\lambda x_1.$`(traverse(`x_1`,`x_2`),`$p(x_2)$`)`
CITY $\rightarrow$ `largest(`CITY`)`	$\lambda p.\lambda x_1.$`largest(`x_1`,`$p(x_1)$`)`
PLACE $\rightarrow$ `largest(`PLACE`)`	$\lambda p.\lambda x_1.$`largest(`x_1`,`$p(x_1)$`)`
STATE $\rightarrow$ `largest(`STATE`)`	$\lambda p.\lambda x_1.$`largest(`x_1`,`$p(x_1)$`)`
STATE $\rightarrow$ `largest_one(area_1(`STATE`))`	$\lambda p.\lambda x_1.$`largest(`x_2`, (area(`x_1`,`x_2`),`$p(x_1)$`))`
CITY $\rightarrow$ `largest_one(density_1(`CITY`))`	$\lambda p.\lambda x_1.$`largest(`x_2`, (density(`x_1`,`x_2`),`$p(x_1)$`))`
STATE $\rightarrow$ `largest_one(density_1(`STATE`))`	$\lambda p.\lambda x_1.$`largest(`x_2`, (density(`x_1`,`x_2`),`$p(x_1)$`))`
CITY $\rightarrow$ `largest_one(population_1(`CITY`))`	$\lambda p.\lambda x_1.$`largest(`x_2`, (population(`x_1`,`x_2`),`$p(x_1)$`))`
STATE $\rightarrow$ `largest_one(population_1(`STATE`))`	$\lambda p.\lambda x_1.$`largest(`x_2`, (population(`x_1`,`x_2`),`$p(x_1)$`))`
CITY $\rightarrow$ `smallest(`CITY`)`	$\lambda p.\lambda x_1.$`smallest(`x_1`,`$p(x_1)$`)`
NUM $\rightarrow$ `smallest(`NUM`)`	$\lambda p.\lambda x_1.$`smallest(`x_1`,`$p(x_1)$`)`
PLACE $\rightarrow$ `smallest(`PLACE`)`	$\lambda p.\lambda x_1.$`smallest(`x_1`,`$p(x_1)$`)`
STATE $\rightarrow$ `smallest(`STATE`)`	$\lambda p.\lambda x_1.$`smallest(`x_1`,`$p(x_1)$`)`
STATE $\rightarrow$ `smallest_one(area_1(`STATE`))`	$\lambda p.\lambda x_1.$`smallest(`x_2`, (area(`x_1`,`x_2`),`$p(x_1)$`))`
STATE $\rightarrow$ `smallest_one(density_1(`STATE`))`	$\lambda p.\lambda x_1.$`smallest(`x_2`, (density(`x_1`,`x_2`),`$p(x_1)$`))`
CITY $\rightarrow$ `smallest_one(population_1(`CITY`))`	$\lambda p.\lambda x_1.$`smallest(`x_2`, (population(`x_1`,`x_2`),`$p(x_1)$`))`
STATE $\rightarrow$ `smallest_one(population_1(`STATE`))`	$\lambda p.\lambda x_1.$`smallest(`x_2`, (population(`x_1`,`x_2`),`$p(x_1)$`))`

Productions	*Corresponding λ-functions*
PLACE $\rightarrow$ `highest(PLACE)`	$\lambda p.\lambda x_1.$`highest(`x_1`,`$p(x_1)$`)`
PLACE $\rightarrow$ `lowest(PLACE)`	$\lambda p.\lambda x_1.$`lowest(`x_1`,`$p(x_1)$`)`
RIVER $\rightarrow$ `longest(RIVER)`	$\lambda p.\lambda x_1.$`longest(`x_1`,`$p(x_1)$`)`
RIVER $\rightarrow$ `shortest(RIVER)`	$\lambda p.\lambda x_1.$`shortest(`x_1`,`$p(x_1)$`)`
NUM $\rightarrow$ `count(CITY)`	$\lambda p.\lambda x_1.$`count(`x_2`,`$p(x_2)$`,`x_1`)`
NUM $\rightarrow$ `count(PLACE)`	$\lambda p.\lambda x_1.$`count(`x_2`,`$p(x_2)$`,`x_1`)`
NUM $\rightarrow$ `count(RIVER)`	$\lambda p.\lambda x_1.$`count(`x_2`,`$p(x_2)$`,`x_1`)`
NUM $\rightarrow$ `count(STATE)`	$\lambda p.\lambda x_1.$`count(`x_2`,`$p(x_2)$`,`x_1`)`
NUM $\rightarrow$ `sum(NUM)`	$\lambda p.\lambda x_1.$`sum(`x_2`,`$p(x_2)$`,`x_1`)`
CITY $\rightarrow$ `most(CITY)`	$\lambda p'.\lambda x_1.$`most(`x_1`,`x'`,`$p'(x_1)$`)`, where p' contains one and only one free variable, x'
PLACE $\rightarrow$ `most(PLACE)`	$\lambda p'.\lambda x_1.$`most(`x_1`,`x'`,`$p'(x_1)$`)`
RIVER $\rightarrow$ `most(RIVER)`	$\lambda p'.\lambda x_1.$`most(`x_1`,`x'`,`$p'(x_1)$`)`
STATE $\rightarrow$ `most(STATE)`	$\lambda p'.\lambda x_1.$`most(`x_1`,`x'`,`$p'(x_1)$`)`
CITY $\rightarrow$ `fewest(CITY)`	$\lambda p'.\lambda x_1.$`fewest(`x_1`,`x'`,`$p'(x_1)$`)`
PLACE $\rightarrow$ `fewest(PLACE)`	$\lambda p'.\lambda x_1.$`fewest(`x_1`,`x'`,`$p'(x_1)$`)`
RIVER $\rightarrow$ `fewest(RIVER)`	$\lambda p'.\lambda x_1.$`fewest(`x_1`,`x'`,`$p'(x_1)$`)`
STATE $\rightarrow$ `fewest(STATE)`	$\lambda p'.\lambda x_1.$`fewest(`x_1`,`x'`,`$p'(x_1)$`)`
CITY $\rightarrow$ `intersection(CITY,CITY)`	$\lambda p_1.\lambda p_2.\lambda x_1.$`(`$p_1(x_1)$`,`$p_2(x_1)$`)`
PLACE $\rightarrow$ `intersection(PLACE,PLACE)`	$\lambda p_1.\lambda p_2.\lambda x_1.$`(`$p_1(x_1)$`,`$p_2(x_1)$`)`
RIVER $\rightarrow$ `intersection(RIVER,RIVER)`	$\lambda p_1.\lambda p_2.\lambda x_1.$`(`$p_1(x_1)$`,`$p_2(x_1)$`)`
STATE $\rightarrow$ `intersection(STATE,STATE)`	$\lambda p_1.\lambda p_2.\lambda x_1.$`(`$p_1(x_1)$`,`$p_2(x_1)$`)`
CITY $\rightarrow$ `exclude(CITY,CITY)`	$\lambda p_1.\lambda p_2.\lambda x_1.$`(`$p_1(x_1)$`,not(`$p_2(x_1)$`))`
PLACE $\rightarrow$ `exclude(PLACE,PLACE)`	$\lambda p_1.\lambda p_2.\lambda x_1.$`(`$p_1(x_1)$`,not(`$p_2(x_1)$`))`
RIVER $\rightarrow$ `exclude(RIVER,RIVER)`	$\lambda p_1.\lambda p_2.\lambda x_1.$`(`$p_1(x_1)$`,not(`$p_2(x_1)$`))`
STATE $\rightarrow$ `exclude(STATE,STATE)`	$\lambda p_1.\lambda p_2.\lambda x_1.$`(`$p_1(x_1)$`,not(`$p_2(x_1)$`))`

Productions	*Corresponding λ-functions*
QUERY $\rightarrow$ `answer(`CITY`)`	$\lambda p.$`answer(`x_1`,`$p(x_1)$`)`
QUERY $\rightarrow$ `answer(`COUNTRY`)`	$\lambda p.$`answer(`x_1`,`$p(x_1)$`)`
QUERY $\rightarrow$ `answer(`NUM`)`	$\lambda p.$`answer(`x_1`,`$p(x_1)$`)`
QUERY $\rightarrow$ `answer(`PLACE`)`	$\lambda p.$`answer(`x_1`,`$p(x_1)$`)`
QUERY $\rightarrow$ `answer(`RIVER`)`	$\lambda p.$`answer(`x_1`,`$p(x_1)$`)`
QUERY $\rightarrow$ `answer(`STATE`)`	$\lambda p.$`answer(`x_1`,`$p(x_1)$`)`

A.3 CLANG: The ROBOCUP Coach Language

In the ROBOCUP Coach Competition, teams compete to provide effective instructions to advice-taking agents in the simulated soccer domain. Coaching instructions are provided in a formal coach language called CLANG (Chen et al., 2003, Sec. 7.7).

The CLANG grammar described here basically follows the one described in Chen et al. (2003). We have slightly modified CLANG to introduce a few concepts that are not easily describable in the original CLANG language. These new constructs are marked with asterisks (*).

In CLANG, coaching instructions come in the form of *if-then rules*. Each if-then rule consists of a *condition* and a *directive*:

RULE $\rightarrow$ `(`CONDITION DIRECTIVE`)`

Possible conditions are:

Productions	*Meaning of predicates*
CONDITION $\rightarrow$ `(true)`	Always true.
CONDITION $\rightarrow$ `(false)`	Always false.

Productions	*Meaning of predicates*
Condition → (`ppos` Player Unum$_1$ Unum$_2$ Region)	At least Unum$_1$ and at most Unum$_2$ of Player is in Region.
Condition → (`ppos-any` Player Region)*	Some of Player is in Region.
Condition → (`ppos-none our` Region)*	None of our players is in Region.
Condition → (`ppos-none opp` Region)*	None of the opponents is in Region.
Condition → (`bpos` Region)	The ball is in Region.
Condition → (`bowner` Player)	Player owns the ball.
Condition → (`playm bko`)	Specific play modes (Chen et al., 2003).
Condition → (`playm time_over`)	
Condition → (`playm play_on`)	
Condition → (`playm ko_our`)	
Condition → (`playm ko_opp`)	
Condition → (`playm ki_our`)	
Condition → (`playm ki_opp`)	
Condition → (`playm fk_our`)	
Condition → (`playm fk_opp`)	
Condition → (`playm ck_our`)	
Condition → (`playm ck_opp`)	
Condition → (`playm gk_our`)	
Condition → (`playm gk_opp`)	
Condition → (`playm gc_our`)	
Condition → (`playm gc_opp`)	
Condition → (`playm ag_our`)	
Condition → (`playm ag_opp`)	
Condition → "Ident"	Condition named Ident. See `definec`.
Condition → (< Num$_1$ Num$_2$)	Num$_1$ is smaller than Num$_2$. Both Num$_1$ and Num$_2$ can be identifiers.
Condition → (> Num$_1$ Num$_2$)	Num$_1$ is greater than Num$_2$.
Condition → (<= Num$_1$ Num$_2$)	Num$_1$ is not greater than Num$_2$.
Condition → (== Num$_1$ Num$_2$)	Num$_1$ is equal to Num$_2$.
Condition → (>= Num$_1$ Num$_2$)	Num$_1$ is not smaller than Num$_2$.

Productions	*Meaning of predicates*
CONDITION → `(!=` NUM_1 NUM_2`)`	NUM_1 is not equal to NUM_2.
CONDITION → `(and` CONDITION_1 CONDITION_2`)`	CONDITION_1 and CONDITION_2.
CONDITION → `(or` CONDITION_1 CONDITION_2`)`	CONDITION_1 or CONDITION_2.
CONDITION → `(not` CONDITION`)`	CONDITION is not true.

Directives are lists of actions for individual players to take:

Productions	*Meaning of predicates*
DIRECTIVE → `(do` PLAYER ACTION`)`	PLAYER should take ACTION.
DIRECTIVE → `(dont` PLAYER ACTION`)`	PLAYER should avoid taking ACTION.

Possible actions are:

Productions	*Meaning of predicates*
ACTION → `(pos` REGION`)`	Go to REGION.
ACTION → `(home` REGION`)`	Set default position to REGION.
ACTION → `(mark` PLAYER`)`	Mark PLAYER (usually opponents).
ACTION → `(markl` REGION`)`	Mark the passing lane from current ball position to REGION.
ACTION → `(markl` PLAYER`)`	Mark the passing lane from current ball position to position of PLAYER (usually opponents).
ACTION → `(oline` REGION`)`	Set offside-trap line to REGION.
ACTION → `(pass` REGION`)`	Pass the ball to REGION.
ACTION → `(pass` PLAYER`)`	Pass the ball to PLAYER.
ACTION → `(dribble` REGION`)`	Dribble the ball to REGION.
ACTION → `(clear` REGION`)`	Clear the ball to REGION.
ACTION → `(shoot)`	Shoot the ball.
ACTION → `(hold)`	Hold the ball.
ACTION → `(intercept)`	Intercept the ball.
ACTION → `(tackle` PLAYER`)`	Tackle PLAYER.

The following productions are for specifying players: (UNUM stands for "uniform numbers", i.e. 1 to 11)

Productions	*Meaning of predicates*
PLAYER → `(player our` {UNUM}`)`**	Our player UNUM.
PLAYER → `(player our` {UNUM_1 UNUM_2}`)`**	Our players UNUM_1 and UNUM_2.
PLAYER → `(player our` {UNUM_1 UNUM_2 UNUM_3}`)`**	Our players UNUM_1, UNUM_2 and UNUM_3.
PLAYER → `(player our` {UNUM_1 UNUM_2 UNUM_3 UNUM_4}`)`**	Our players UNUM_1, UNUM_2, UNUM_3 and UNUM_4.
PLAYER → `(player opp` {UNUM}`)`**	Opponent player UNUM.
PLAYER → `(player our {0})`**	Our team.
PLAYER → `(player opp {0})`**	Opponent's team.
PLAYER → `(player-range our` UNUM_1 UNUM_2`)`*	Our players UNUM_1 to UNUM_2.
PLAYER → `(player-range opp` UNUM_1 UNUM_2`)`*	Opponent players UNUM_1 to UNUM_2.
PLAYER → `(player-except our` {UNUM}`)`*	Our team except player UNUM
PLAYER → `(player-except opp` {UNUM}`)`*	Opponent's team except player UNUM

Productions marked with double asterisks (**) are slight variations of existing constructs in the original CLANG grammar (e.g. as in `(bowner our {4})`). The new `player` predicate is introduced for uniformity. To specify regions, we can use the following productions:

Productions	*Meaning of predicates*
REGION → POINT	A POINT.
REGION → `(rec` POINT_1 POINT_2`)`	A rectangle with opposite corners POINT_1 and POINT_2.
REGION → `(tri` POINT_1 POINT_2 POINT_3`)`	A triangle with corners POINT_1, POINT_2 and POINT_3.

Productions	*Meaning of predicates*
REGION → `(arc` POINT NUM_1 NUM_2 NUM_3 NUM_4`)`	A donut arc (Chen et al., 2003).
REGION → `(circle` POINT NUM`)`*	A circle of center POINT and radius NUM.
REGION → `(null)`	The empty region.
REGION → `(reg` REGION_1 REGION_2`)`	The union of REGION_1 and REGION_2.
REGION → `(reg-exclude` REGION_1 REGION_2`)`*	REGION_1 excluding REGION_2.
REGION → `(field)`*	The field.
REGION → `(half` TEAM`)`*	The TEAM's half of field. TEAM can be either `our` or `opp`.
REGION → `(penalty-area` TEAM`)`*	The TEAM's penalty area.
REGION → `(goal-area` TEAM`)`*	The TEAM's goal area.
REGION → `(midfield)`*	The midfield.
REGION → `(midfield` TEAM`)`*	The TEAM's midfield.
REGION → `(near-goal-line` TEAM`)`*	Near TEAM's goal line.
REGION → `(from-goal-line` TEAM NUM_1 NUM_2`)`*	NUM_1 to NUM_2 meters from TEAM's goal line.
REGION → `(left` REGION`)`*	The left half of REGION (from our team's perspective).
REGION → `(right` REGION`)`*	The right half of REGION.
REGION → `(left-quarter` REGION`)`*	The left quarter of REGION.
REGION → `(right-quarter` REGION`)`*	The right quarter of REGION.
REGION → `"`IDENT`"`	Region named IDENT. See `definer`.

To specify points, we can use the following productions:

Productions	*Meaning of predicates*
POINT → `(pt` NUM_1 NUM_2`)`	The xy-coordinates $(\text{NUM}_1, \text{NUM}_2)$.
POINT → `(pt ball)`	The current ball position.
POINT → POINT_1 `+` POINT_2	Coordinate-wise addition.
POINT → POINT_1 `-` POINT_2	Coordinate-wise subtraction.
POINT → POINT_1 `*` POINT_2	Coordinate-wise multiplication.

Productions	*Meaning of predicates*
POINT → POINT$_1$ `/` POINT$_2$	Coordinate-wise division.
POINT → `(pt-with-ball-attraction` POINT$_1$ POINT$_2$`)`*	POINT$_1$ `+ ((pt ball) *` POINT$_2$`)`.
POINT → `(front-of-goal` TEAM`)`*	Directly in front of TEAM's goal.
POINT → `(from-goal` TEAM NUM`)`*	NUM meters in front of TEAM's goal.

The following CLANG statements can be used to define names for conditions and regions. These names (IDENT) can be used to simplify the definition of if-then rules:

STATEMENT → `(definec "`IDENT`"` CONDITION`)`
STATEMENT → `(definer "`IDENT`"` REGION`)`

Note that an if-then rule is also a CLANG statement:

STATEMENT → RULE

STATEMENT is the start symbol in the CLANG grammar.