Andl Grammar and other Notes

David M. Bennett FACS

Specifications and Notes for ANDL Grammar

# Philosophy

1. Consistent use of syntax. Things that do similar things should look similar, and vice versa.
2. Functional in style (not procedural or OO). A language of expressions with (almost) no side-effects, which can support either lazy or eager evaluation. Everything that looks like a value could actually be an expression.
3. Lexical scoping. Predefined->catalog(s)->argument(s)->current tuple(s)->local. No scope resolution operators, just renaming and aliases. Scoping reads left-to-right, consistent with infix notation.
4. Limited use of reserved words. System library functions have names, but can always be overridden by user definitions.
5. Limited use of symbols. They just take too much explaining.
6. Lexical: any legal characters are permitted in an identifier and any legal characters can be used to construct a literal character string (but control characters are not recognised). Strings concatenate. Special quoting conventions for identifiers, Unicode strings, time literals, etc.
7. Types: the native abstract types are bool, number, text, time and binary; structured types are tuple, relation and user (struct). All SQL types are accepted, but are converted accordingly.
8. The compiler is hand-coded LL(1+), recursive descent, which limits certain kinds of syntax. Shouldn’t be a problem in practice. The grammar is quite small.

# Grammar

The formal grammar is documented separately.

# Notes

## General

## Symbols.

|  |  |
| --- | --- |
| Example | Notes |
| { age:number, name:text }  { age:=21, name:=‘Joe’ }  { age:=a, name:=b } | Single braces. Defines a tuple type or value, or the attributes to be found in an output relation. |
| do { x := 17 } | For now, single braces also delineate do block. |
| {{ age:number, name:text }}  {{ age:=21, name:=‘Joe’ }} | Double braces. Defines a relation type or value. |
| [ $(age>0) %(age) {\*name} ]  [ { age, name, marks } ]  [ { oldest:=max(age) } ] | Square brackets. Relational selector that parses as a postfix operator and provides an iterative scope for evaluating a predicate, open, aggregating or ordered expression. |
| 17 / (x + 3)  left(‘hello world’, 5) | Parentheses. Encloses an expression, or arguments to a called function. |
| ; | Statement delimiter |
| := => : | Used in statements; see later. |

## Literals

|  |  |
| --- | --- |
| Example | Notes |
| true  false | Bool |
| 0  123456789  123445.6789 | Number |
| “string”  ‘string’ | Strings may be double or single quoted, so it is convenient to use one style for text that contains the other. No escape sequences. |
| “hello” ‘ ‘ “world” | Adjacent strings are pasted together by the lexer to make a single string token. Guaranteed to be no grammar ambiguities |
| h'42 2a 0d 0a' | Hex: the string is a space separated list of one or more hex numbers representing individual Unicode characters. [The underlying encoding is not exposed.] |
| d'123 32 13 10' | Decimal: similar, but with space separated decimal integers. |
| b'422a2020200d0a00’ | Binary: raw string of hex-encoded binary bytes. No bit strings or octal. |
| i'~!@#$%^&\*()'  i'$' “’” ‘”’ d’10 11 12’ | Identifier: any string is a valid identifier. Strings with other prefixes can be pasted to it. |
| t'31/12/2014'  t'2014/12/31'  t'23:59:59'  t'11:59:59 pm 1/12/2014' | Time: If it parses with default regional settings, it's valid. Canonical form might be safer. |

##### Future

1. A simple mechanism to define token (lexer) shortcuts. They paste together just like strings.

Eg #define COPYR h’24b8’

i’xx’ COPYR ‘xx’ // is a valid identifier.

1. A mechanism for block text is probably needed, but not yet decided.

## Types

##### A type designator can be the name of the type or any object of that type. It always comes after a colon.

|  |  |
| --- | --- |
| Example | Notes |
| ... flag:bool ...  ... flag:true ... | True or false | |
| ... count:number ...  ... count:0 ... | Arbitrary precision fixed decimal | |
| ... name:text ...  ... name:’’ ... | Unicode string of arbitrary length | |
| ... when:time ...  ... when:t’’ ... | Calendar date and time | |
| ... data:binary ...  ... data:b’’ ... | Arbitrary string of bytes | |
| { age:number, name:text } | Tuple. Set of uniquely named typed values. Attributes are not ordered, and each unique set of attributes is a unique type. Variables are named, but types are not. | |
| {{ age:number, name:text }} | Relation. Set of tuples, all the same type. Tuples are not ordered, and each unique set of attributes is a unique type. Variables are named, but types are not. | |
| def :point(x:number, y:number)  def point(x:number, y:number) | User defined type. Named list of uniquely named typed and structured values. Components are ordered, each named type is unique. | |
| fn(x:number, name:text) => | Argument. List of uniquely named typed values. Components are ordered, and can only be referenced as part of a function call. | |
| def int:number  def int:number => number div 1 = number | Subtype. No components, but value must satisfy constraint predicate, if specified. | |

## Values

Tuple, relation and user values have a distinctive form and may be constant or evaluated in scope.

|  |  |
| --- | --- |
| Example | Notes |
| { age:=17, name:=’mr’ }  { age:=x, name:=t } | Tuple value | |
| {{ age:=18, name:=’miss’ }}  {{ age:number, name:=text }  (18, ‘miss’) }  {{ age:=x, name:=t }} | Relation value |
| person(21, ‘mrs’)  person(x, t) | User defined type value |
| int(1234)  date(t’31/12/2015’) | Subtype value |
| do { 17 }  do 17 end | Sequence of expressions; last is its value. |

## Assignment

Immediate assignment and deferred evaluation are similar in appearance.

|  |  |  |
| --- | --- | --- |
| Example | Notes | |
| var := expr | | Assignment | |
| var => expr | | Deferred evaluation | |
| var(x:type,y:type) => expr(x,y) | | Named function | |
| var(x:type,y:type) => do {  z := expr  expr(x,y,z)  } | | Named function with do block to provide a named temporary. Returns value of last expr. | |

The do block can contain any number of expressions, and returns the value of the last evaluated, which may be VOID. [A value of type VOID is simply ignored/discarded. Both assignment operators return VOID which prevents them from appearing elsewhere in expressions.]

Natural variants of the same syntax:

|  |  |
| --- | --- |
| (x:type,y:type) => expr(x,y) | Anonymous lambda function? |

## Variables

|  |  |
| --- | --- |
| Example | Notes |
| T := { age:=17, name:=’abc’ }  R := {{ age:=17, name:=’abc’ }) | Variable is created by assignment, using the type of the expression. | |
| def S:db(csv)  db S:csv  db S:sql  db S:{ age:=17, name:=’abc’ }  db S:old\_S | Database relation. Type and value defined by CSV file, or connection to SQL, or as specified. | |

## Expressions

Expressions come in several varieties, and may be named. The name is what appears on the left of an assignment. Only a closed expression may appear outside a relational scope.

|  |  |
| --- | --- |
| Example | Notes |
| Project | Copies an attribute from source to destination, or drops an attribute from the destination. |
| Rename | Renames an attribute, and copies it to the destination. |
| Closed | Evaluates an unscoped (closed) expression and returns a value. Relational scope not required. |
| Open | Evaluates an open expression in scope and returns a value |
| Aggregate | Evaluates an aggregating expression in scope and returns an updated value |
| Ordered/Grouped | As per aggregate, but may access other tuples in the current set. |

## Select

Select combines the functions of restrict, order, rename, project, extend, aggregate and ordered aggregate. These are the valid combinations.

|  |  |
| --- | --- |
| Example | Notes |
| [ ?(CITY = ‘Paris’) ]  WHERE(CITY = ‘Paris’) | Restrict. Takes a Boolean open expressions. Only tuples satisfying the condition contribute to the result. |
| [ $(CITY, -SNAME) ]  ORDER(CITY, -SNAME) | Order. Sort in order of given attributes, ascending or descending. |
| [ $(CITY, %SNAME) ]  ORDER(CITY, %SNAME) | Group. Aggregation resets for each value of a grouped attribute. |
| [{ idp, idp, ... }]  SELECT(idp, idp, ...) | Project. Only the named columns are in the result. |
| [{\* idp, idp, ... }]  SELECT(\* idp, idp, ...) | Project away. All except the columns idp are in the result. |
| [{\* idr := old, idr := old, ... }]  SELECT(\*idr := old, idr := old, ...) | Rename. All columns are in the result, and columns idr are renamed. |
| [{ ide := expr, idp, idr := old, }]  SELECT(ide := expr, idp, idr := old) | Mixed extend. Only the named columns are in the result, idp by project, idr by renaming and ide by expression. |
| [{\* ide := expr, idp, idr := old }]  SELECT(\*ide := expr, idp, idr := old) | Mixed extend ‘all but’. All columns except idp are in the result, columns idr are renamed, and columns ide are added. |

Extend variants compute a new attribute by evaluating an expression, which may perform an arbitrary computation and may be open or aggregating or both. If combined with ORDER it may be an ordered expression.

## Metadata

Catalog

|  |  |
| --- | --- |
| Name |  |
| Type |  |

Functions

|  |  |
| --- | --- |
| Name |  |
| Type |  |

Function Parameters

|  |  |
| --- | --- |
| Name |  |
| Type |  |
| FunctionName |  |

## Catalog

The catalog contains persistent entries of four types. The catalog is persistent, so the variables it contains are persistent too.

Actual storage for database relations and for the catalog may be Andl proprietary or an RDBMS such as Sqlite.

|  |  |
| --- | --- |
| Entry Type | Description |
| Variable | Contains a value of scalar, tuple, relation or user-defined type. The value is represented by a serialised binary value. |
| Database relation | Contains a link to storage for a database relation. |
| Operator | Contains a value of type code, which implements an executable function with arguments and a return value. |
| User defined type  Subtype | Type definition used by other catalog entries. |

# Notes on Syntax

## Basic Expression Evaluation

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Example usage | Notes | API function |
| Scalar | true false  0 -99 7fff 123456789 123'456'789 -3.14159 299.792458e06  Smith exam\_mark '-99' 'true' 'John West' 'has"quote'  "double quoted" "mark=%exam\_mark%" "100%%" "quote%34%"  2013/12/31 23:59:59 | Literal, of type as inferred.  Abstract number type  Bare word or single quote is an IDLIT, looked up as attribute.  Double quote is a string. With interpolation: IDLIT, char escape  Time | Scalar values |
| Expression | Smith  Sqrt(a / b + c \* d^17)  FirstLetter(Smith)  Sequence(0,100,2) | Compiled expr => scalar value | val eval(expr) |
| Open Expression | Exam\_mark / 100  FirstLetter(Name) | Compiled expr => successive scalar values  IDLIT looked up in scope dict | val eval(expr, scope) |
| Agregation Expression | Fold(+,mark)  Fold(+,mark^2/100) | Compiled expr => aggregated scalar value  IDLIT looked up in scope dict to get old and new value | val eval(expr, scope, old) |
| Heading | { StudentId, Name }  { ^ Name }  \*exam\_mark | Literal (known at compile time)  Use ^ in scope to mean ‘all but’.  Include entire heading | Hdg values |
| Tuple | { StudentId := S1, Name := ‘Smith’ }  { StudentId := S1, Mark := class\_mark / 2 } | Literal or expression  Attribute types set by implied type | tup maketup(hdg, expr[d]) |
| Relation | {{ StudentId := S1, Name := Smith }}  {{ StudentId, Name }, {:= S1, Smith }, {:= S2, Jones }}  {{ StudentId, Name }}  {{}} [false] | Literal or expression  Second form is shorthand  As open expression, creates singleton relation from scope | rel makerel(tup[c]) |

## Monadic Operations on Scalar Valued Attributes

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Example usage | Notes | API function |
| Aggregate | exam\_mark [ { StudentId, totalmark := fold(+,Mark) } ]  exam\_mark [ { StudentId, marks := fold(union,{{ Mark := Mark }}) } ] | Project by single pass copy with dup check and update (O(nlogn))  No implicit inclusion of other attributes  May be combined with rename | rel aggregate(rel, hdg, expr[e]) |
| Extend | is\_called [ \*{ Initial := FirstLetter(Name) } ] | Extend by single pass copy, no dup check (O(n))  By explicit inclusion.  Open expression evaluated in scope of each tuple | rel extend(rel, *hdg*, expr[e]) |
| Project | exam\_mark [ { StudentId, Mark } ]  exam\_mark [ \*{ CourseId, StudentId } ]  exam\_mark [ { StudentId, Level := (Mark – 50) / 5 } | Project by single pass copy, with dup check (O(nlogn))  By inclusion or exclusion; may have extras. | rel project(rel, hdg)  rel project(rel, hdg, expr[c]) |
| Rename | exam\_mark [ \*{ SID := StudentId } ]  exam\_mark [ { SID := StudentId, CID := CourseId} ] | Rename by header update (O(1))  By inclusion or exclusion, but only a rename if all mentioned | rel rename(rel, expr) |
| Restrict | is\_called [ Name = Smith ]  is\_called [ FirstLetter(Name) = S ] | Restrict by single pass filter (O(n)) | rel restrict(rel, expr) |
| Sort | course [ {\* Title } sort ] | Sort relation on given attributes.  Result is a relation from which retrieved data will be in order. |  |

## Dyadic Operations using Name Matching

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Example usage | Notes | API function |
| Join | course join exam\_mark  course semijoin exam\_mark | Natural Join by matching values on common attributes.  Combined with projection on L/C/R attributes. R is reverse. Valid are:   * JOIN * COMPOSE (LR) * DIVIDE (L) * SEMIJOIN (LC) * RSEMIJOIN (RC) | rel join(rel, rel, hdg) |
| Anti-join | course ajoin exam\_mark  course rajoin exam\_mark | Natural Anti-join by keeping non-matching values on common attributes.  Combined with projection on L/C/R attributes. R is reverse. Valid are:   * AJOIN * RAJOIN * AJOINL * RAJOINL | rel antijoin(rel, rel, hdg) |
| Set | course union exam\_mark  course minus exam\_mark | Set operation on tuples using only common attributes.  Here code L/C/R signifies that tuples from the corresponding relation are kept. R is reverse. Valid are:   * UNION (LCR) * INTERSECT (C) * SYMDIFF (LR) * MINUS (L) * RMINUS (R) | rel union(rel, rel, hdg, expr) |
| Group | exam\_mark [{ Marks := {{\*}} compose student }] | Grouping join. Inserts exam\_mark into student |  |

## Statements

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Example usage | Notes | API function |
| Assignment | student\_name := ‘Smith’  smiths := is\_called [?(name=’Smith’)] | Assign name to value, replacing old value (of same type) if needed.  Both scalars and relations. |  |
| Lazy Assignment | is\_called => source(‘sql:’+conn\_string, ‘is\_called’)  smiths = is\_called [?(name=’Smith’)]  find(name) => is\_called [name=name] | Assign expression that calculates name to value.  On evaluation, returns valus.  Most useful for things like sql() query. |  |
| Relational update | X => is\_called := minus is\_called[ ?(Name = ‘Boris’) ]  X => is\_called := union {{ StudentId := S9, Name := Jones }} | Relational operation resulting in update  Any set operation permitted (same heading)  Effect is to delete some rows then insert some rows. |  |
| Selector update | X => is\_called := [ Name := 'Boris' ]  X => is\_called := [ Name := ‘Smith’ { Name := ‘Jones’ }] | Selector-based operation resulting in update.  Rows satisfying predicate are always deleted.  Project format gives rows to insert with changed values if any. |  |

## Common tail analysis

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Operation | Predicate | Attribute Terms | Heading | Ident | Name | Notes |
| Aggregate | pred-expr | { name := fold-expr SEP } | Add |  | New | Any use of ‘fold’. Combine with extend, project, rename. Infill if no project. |
| Assign | pred-expr | { name := expr SEP } | Same | Exist | Exist | Parse context. Combine with rename. Infill. |
| Extend | pred-expr | { name := expr SEP } | Add |  | New | Anything not project or rename. Combine with project, rename. Infill if no project. |
| Project | -- | { [-] ident SEP } | Remove | Exist |  | Only if all terms are project. If minus infill and remove specified, else not. |
| Rename | -- | { name := name SEP } | Remove, add | New | Exist | Only if terms are all rename. Always infill. |
| Restrict | pred-expr | -- | Same |  |  |  |

## Neat Ideas

### Bulk renaming

Rename every attribute of a relation using a prefix/suffix/regex to simplify managing conflicting attribute names. Especially good for self-joins.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Bulk Rename | { b\* := a\* }  { \*\_2 := \* } | Rename all, using prefix or suffix. |

### Infill

All selectors default to a result that includes only the columns mentioned. The STAR (‘\*’) enables infill, so that all columns not mentioned are included unchanged.

*Project* defaults to including only the specified attributes, but if infill is enabled then specified attributes are removed (as per ‘all but’).

For *aggregate*, *assign* and *extend* if infill is enabled then specified *project* columns removed (as per ‘all but’).

### Aggregation

At compile time, any expression with fold() requires reduction.

At runtime, algorithm 1 (sorted) goes something like this.

1. Create sorted index on all result columns other than those requiring reduction.
2. Process input table in sorted order.
3. For each group, evaluate expression once.
4. A reduce function takes 2 args: a function, and a list of column values (possibly empty).

At runtime, algorithm 2 (hash index) goes something like this.

1. Extend: create row with default values suitable for indexing
2. Initialise accumulators using first data row.
3. Update accumulators with each data row.
4. Finalise, calculate final value.

### Scope

The predefined scope levels are:

|  |  |
| --- | --- |
|  |  |
| Predefined | Built in functions, compiler symbols |
| Catalog | Global variables, including user-defined |
| Parameter | Variables provided by host system |
| Expression | Carries current heading |
| Tuple | Makes current heading searchable |

1. Predefined. System iden

A new scope is

Each expression starts a new scope