Andl Grammar and other Notes

David M. Bennett FACS

Specifications and Notes for ANDL Grammar V6

# 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→global→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 are in predefined scope and 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, binary, special characters.
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 now PEG LL(\*),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

## Statements

### Assignment and Function Definition

Define a name and give it a value, which may be evaluated immediately or deferred.

|  |  |  |
| --- | --- | --- |
| Example | Notes | |
| V := length(‘asdf’) | | Assignment | |
| V => count(S join SP) | | Deferred evaluation | |
| pyth(x:0,y:0) => (x^2+y^2)^0.5  fact:0(n:0) =>  if(n<2, n, n\*fact(n-1)) | | Named function: deferred evaluation with parameters. | |
| func(x:type,y:type) => do  z := pyth(x,y)  get(x,y,z)  end | | Named function with do block to provide a named temporary. Returns value of last expr. | |
| sum(n:0) => fold(+,n) | | Named function containing a fold() becomes a folded function. It requires an iterated context. | |
| (x:type,y:type) => (x^2+y^2)^0.5 | | Anonymous lambda function? | |

Notes

Immediate assignment and deferred evaluation are similar in appearance.

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 and cannot be stored. All statements operators return VOID.]

### Type definition

Define a new type and give it a name.

|  |  |
| --- | --- |
| Example | Notes |
| def int:number  def int:number => number div 1 = number | Subtype. No components, but value must satisfy constraint predicate, if specified. [Actually there is one component called *super*.] | |
| 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. | |
| def tt{ age:0, name:’’ }  def tt:tup(rt)  def tt:tup(Employee) | Tuple. Set of uniquely named typed values. Attributes are not ordered, and each unique set of attributes is a unique type. Can be anonymous or named. Can be based on other type or value that has a heading. | |
| def rt{{ age:0, name:’’ }}  def rt:rel(tt)  def rt:rel(Employee) | Relation. Set of tuples, all the same type. Tuples are not ordered, and each unique set of attributes is a unique type. Can be anonymous or named. Can be based on other type or value that has a heading. | |

### Import

Define a relation variable and import its value from a data source.

|  |  |
| --- | --- |
| Example | Notes |
| var S(csv) | Import a CSV file into a relation, using the column labels as attribute names. | |
| var F(txt) | Import a text file into a relation with one attribute Line. | |
| var T(sql), F(odbc), F(oledb) | Import a table from an SQL, ODBC or OLEDB source respectively. Attribute names and types are obtained from the source. | |
| var V:sometype() | Import a database table that already exists, making an entry for it in the catalog. | |
| var V:sometype | Create a variable of that type. | |

Notes

The specific details of the data source and conversion strategy remain to be worked out.

The import path or connection string can be set by the compiler option #source.

### Update

Update the contents of a relation variable.

|  |  |
| --- | --- |
| Example | Notes |
| update S union sep NEWS | Perform an in situ set-wise operation on the relation. Operations are union, union sep, minus, minus sub, symdiff. |
| update R  .where(age<50)  .set{ old:=true } | Perform an in situ update on a subset of the relation. |
| update R  .where(age<50)  .delete() | Perform an in situ delete of a subset of the relation |

## Types

A type reference is a way to use types previously declared.

|  |  |
| --- | --- |
| Example | Notes |
| bool | True or false |
| number | Arbitrary precision fixed decimal |
| text | Unicode string of arbitrary length |
| time | Calendar date and time |
| binary | Arbitrary string of bytes |
| { age:number, name:text }  tup(Employee) | Tuple. Set of uniquely named typed values. Attributes are not ordered, and each unique set of attributes is a unique type. Can be anonymous or named. Can be based on other type or value that has a heading. |
| {{ age:number, name:text }}  rel(Employee) | Relation. Set of tuples, all the same type. Tuples are not ordered, and each unique set of attributes is a unique type. Can be anonymous or named. Can be based on other type or value that has a heading. |
| typename | The name of a defined type, either a basic type or one created by a type definition. |
| { age:0, name:’’ }  mul(x:0,y:0) => x\*y | Any value, including a literal or named variable. |

Note

A type reference can be the name of the type, or a variable or a literal of that type. It can only ever appear in particular places, usually after an identifier and a colon. This should not be confused with a type name, which can be used as a selector function.

## Primaries

### Literals

Literal values directly translated by the compiler.

|  |  |
| --- | --- |
| 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. |
| #define COPYR h’24b8’  i’xx’ COPYR ‘xx’ | A simple mechanism to define token (lexer) shortcuts. They paste together just like strings. |
| “””  Block text  “”” | A mechanism for block text is probably needed, but not yet decided. |

### Tuple Value

A value of tuple type.

|  |  |
| --- | --- |
| Example | Notes |
| { age:=17, name:=’mr’ }  { age:=17+3, name:=’mr’&’xx } | Literal or closed tuple value |
| { age:=x, name:=t } | Open value, evaluated in scope. |
| {\*} | Open value, represents entire current tuple. |
| tup(Emp) | Open value, by conversion. Argument must be of a type with a heading. |
| TTemp(17, ‘mr’)  TTemp(age, name) | Create tuple of named type using ordered arguments. Constant or evaluated in scope. |

Note

Tuple values can be literal, constant, evaluated in global scope or evaluated in iterative scope. The latter is provided by the evaluation of an operation on a relation.

### Relation Value

A value of relation type.

|  |  |
| --- | --- |
| Example | Notes |
| {{ X:=17, Y:=24 },{ X:=90, Y:=48 }}  {{ X:=1+7, Y:=4 },{ X:=2\*3, Y:=9 }} | Literal or closed relation value. |
| {{ X:=a, Y:=b },{ X:=c, Y:=d }}  {{ age:=x, name:=t }} | Open value, evaluated in scope. For iterative tuple scope, this is usually a singleton relation. |
| {{\*}} | Open value, singleton relation represents entire current tuple. |
| rel(V1, V2, ...)  rel({ X:=a, Y:=b },{ X:=c, Y:=d })  rel(TT(17, ‘mr’), TT(21,’prof’)) | Value obtained by conversion from values of a type with a heading. All values must have same heading. Closed or Open. |
| RT(V1, V2, ...)  RT(TT(17, ‘mr’), TT(21,’prof’))  RT((17, ‘mr’), (21,’prof’)) | Create relation of named type using ordered arguments. Tuple type not required if it can be inferred. Closed or open. |

### User-defined Value

A value of a user-defined type.

|  |  |
| --- | --- |
| Example | Notes |
| person(21, ‘mrs’)  person(x, t) | User defined type value is the invocation of a selector named after the type. Closed or Open. |
| int(1234)  date(t’31/12/2015’)  date(tnow) | Subtype value is the invocation of a selector named after the type. Closed or Open. |

### Variable

A variable is a name that returns a value that may vary.

|  |  |
| --- | --- |
| Example | Notes |
| V := ‘hello world’  do  V := 42  end | A catvar (for catalog variable), at global or local scope. A name may not be redefined within its or an inner scope. | |
| {{ V := 12345 }} | A field or attribute, in a tuple. | |
| type T(V:0)  temp := T(42)  temp.V | A component, in a user-defined type. | |
| func(V:0) => V | A parameter value, an argument to an operator. | |

Note

In each case V is a variable of a different kind.

### Operator Invocation

An operator is invoked with arguments to return a value.

|  |  |
| --- | --- |
| Example | Notes |
| 3 \* (5 + 17) | Parentheses are used to override precedence. | |
| length(‘abcdef’)  left(‘abcdef’, 3)  count(S)  person(‘Smith’,17) | A function is called with arguments enclosed in parentheses, and returns a value. Arguments evaluated left to right. A typename can be used as a function to create a value of that type. | |
| not (a > 47) | A prefix operator is placed before its argument. | |
| ‘abcdef’.length  S.count  S.where(CITY = ‘Paris’) | A postfix operator is placed after its argument. The dot operator:   * Can be used to call any function with a single argument. * Can be used to extract a component or attribute value from a UDT value, tuple or singleton relation. * Must be used to call functions that provide an iterative context. | |
| 17 \* 43  S join SP | An infix operator is placed between the values that are its arguments. | |

Note

The order of evaluation is:

1. Function and parentheses, inner to outer
2. Prefix, right to left
3. Postfix, left to right
4. Infix, according to precedence, then left to right.

### Special Functions

These functions take arguments that are treated specially.

|  |  |
| --- | --- |
| Example | Notes |
| if(a>b,a,b) | The arguments to if() are a bool value and two expressions of the same type. One expression is evaluated and its result returned. | |
| S .set{ N:=fold(+,1) }  P .set{ N:=fold(+,QTY) } | Requires an iterative context.  The arguments to fold() are an operator and an attribute or value of the same type.  A foldable operator has two arguments and a return value all the same type. | |
|  |  | |

### Iterated Functions

These functions are used to provide an iterative context.

|  |  |
| --- | --- |
| Example | Notes |
| S .where(STATUS > 20)  S .skip(5)  S .take(2) | Restrict. Output is restricted to tuples that satisfy the predicate. | |
| {{ z:=0 }} .recurse( {{z:=z+1}}  .where(z<10) ) | Recursive function, generates tuples from argument and adds them to the seed until no more. | |
| S .set{ CITY }  S .set{ \* CITY, STATUS }  S .set{ \* town := CITY }  S .set{ SS := STATUS \* 10 }  P .set{ CITY, QS := fold(+,QTY) } | Combines the functions of rename, project, extend. Output is a new tuple with the given attributes. | |
| S .order(CITY)  S .group(CITY) | Allows the use of special ordered and grouped functions. | |
| S .order(CITY) .set{ X:= ord() }  S .group(CITY) .set{ X:= ordg() } | Ordered function ord() is a unique ordinal number, and ordg() is the ordinal within group. | |
| P .order(CITY) .set{ X:=lag(COLOR,1) }  P .order(CITY) .set{ X:=lead(COLOR,2) }  P .order(CITY) .set{ X:=nth(COLOR,3) } | Ordered functions lag(), lead(), nth() return the value of an attribute earlier, later or elsewhere within a group, or a default value if out of range. | |
| S .where(S#=’S2’) .get(CITY) | Lift. Returns a single value from a singleton relation with a single attribute. | |

Note

The set() function is described further below.

## Expressions

Expressions belong to several different kinds.

|  |  |
| --- | --- |
| Example | Notes |
| ‘asdf’ | Literal |
| V.trim & ‘xyz’ | Closed. Free names may be parameters, locals or globals (inner to outer). |
| .where(STATUS > 20)  .set{ CITY & ‘?’ } | Open. Free names may be attributes, parameters, locals or globals (inner to outer). |
| fold(+,STATUS) | Folded, performs aggregation. First param is global operator. Second is open. |
| .order(CITY, STATUS) | Ordering. Params must be attribute names. |
| ord() | Ordered, returns a tuple ordinal. |
| nth(STATUS,1) | Ordered, selects an attribute value. First param is attribute. Second is open. |

## Special Characters.

A quick list of special characters used in the grammar.

|  |  |
| --- | --- |
| 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. |
| {{ age:number, name:text }}  {{ age:=21, name:=‘Joe’ }} | Double braces. Defines a relation type or value. |
| 17 / (x + 3)  left(‘hello world’, 5) | Parentheses. Encloses an expression, or arguments to a called function. |
| + - \* / ^ = > < | Operators |
| := => : | Used in statements; see later. |
| ; | Statement delimiter |

## Relational Operations

|  |  |
| --- | --- |
| 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. Iterative scope not required. |
| Open | Evaluates an open expression in an iterative scope and returns a value |
| Aggregate | Evaluates an aggregating expression in an iterative scope and returns a value |
| Ordered/Grouped | As per aggregate, but may access other tuples in the current set. |

## Set

Set combines restrict, order, rename, project, extend, aggregate and ordered aggregate.

|  |  |
| --- | --- |
| Example | Notes |
| .where(CITY = ‘Paris’) | Restrict. Takes a Boolean open expressions. Only tuples satisfying the condition contribute to the result. |
| .order(CITY, -SNAME) | Order. Sort in order of given attributes, ascending or descending. |
| .group(SNAME) | Group. Aggregation resets for each value of a grouped attribute. |
| .set{idp, idp, ...} | Project. Only the named columns are in the result. |
| .set{\* idp, idp, ...} | Project away. All except the columns idp are in the result. |
| .set{\*idr := old, idr := old, ...} | Rename. All columns are in the result, and columns idr are renamed. |
| .set{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. |
| .set{\* 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. |

Note

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.

## 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. |
| 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. |
| Type | Contains the definition of a user-defined type or subtype. |

Note

Values are stored as a serialised binary value, in an attribute of type binary.

### Scope

The compiler scope levels are:

|  |  |
| --- | --- |
|  |  |
| Predefined | Built in functions, compiler symbols |
| Global | Global variables, including user-defined. Includes both ‘global’ and ‘persistent’ catalog values. |
| Tuple | Carries current heading for open expressions |
| Local | Variables created inside a block |
| Parameter | Procedure arguments |

The runtime scope levels are:

|  |  |
| --- | --- |
|  |  |
| Persist | Persistent variables |
| Global | Global variables, including operators and types |
| Local | Variables created in a block |
| Local | Procedure arguments (passed as Lookup) |
| Tuple | Current tuple during iteration (passed as Lookup) |

Note

Things just aren’t that simple!

# Notes on Syntax

## Basic Expression Evaluation

|  |  |  |
| --- | --- | --- |
| Name | Example usage | Notes |
| Scalar | true false  0 -99 123456789 3.14159 299.792458e06 x‘7fff’  ‘Smith’ '-99' 'true' 'John West' 'has"quote'  "double quoted" “pasted”’to this’  d’13 10’ x’d a’  t‘2013/12/31 23:59:59’  "mark=%exam\_mark%" | Literal, of type as inferred.  Abstract number type. Last is hex.  Single or double quote strings, can be pasted  Unicode char code points, decimal or hex  Time  Interpolation |
| Expression | ‘Smith’  Sqrt(a / b + c \* d^17)  FirstLetter(‘Smith’)  Seq(10) | Compiled expr => scalar value  0,1,2,3,4,5,6,7,8,9 |
| Open Expression | Exam\_mark / 100  FirstLetter(Name) | Compiled expr => successive scalar values  IDLIT looked up in scope dict |
| Aggregation Expression | Fold(+,mark)  Fold(+,mark^2/100) | Compiled expr => aggregated scalar value  Value looked up in scope dict to get new value, updated value stored in accumulator. |
| Tuple | { StudentId := S1, Name := ‘Smith’ }  { StudentId := S1, Mark := class\_mark / 2 } | Literal or expression  Attribute types set by implied type |
| Relation | {{ StudentId := S1, Name := Smith }}  {{ StudentId, Name }, (S1, Smith), (S2, Jones) }  {{ StudentId, Name }}  {{}}  {{:}} | Literal or expression  Second form is shorthand  As open expression, creates singleton relation from scope |

## Monadic Operations on Scalar Valued Attributes

|  |  |  |
| --- | --- | --- |
| Name | Example usage | Notes |
| 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 |
| 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 |
| 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. |
| 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 |
| Restrict | is\_called [ Name = Smith ]  is\_called [ FirstLetter(Name) = S ] | Restrict by single pass filter (O(n)) |
| Sort | course [ $(Title) {\* Title } ] | Sort relation on given attributes.  Result is a relation value with data in order. |

## Dyadic Operations using Name Matching

|  |  |  |
| --- | --- | --- |
| Name | Example usage | Notes |
| 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) |
| 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 |
| 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) |
| Group | exam\_mark [{ Marks := {{\*}} compose student }] | Grouping join. Inserts exam\_mark into student |

## Statements

|  |  |  |
| --- | --- | --- |
| Name | Example usage | Notes |
| 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 | smiths => is\_called [?(name=’Smith’)]  find(name) => is\_called [name=name]  fact:number(n:number) => if(n<2, n, n\*fact(n-1)) | Assign expression that calculates name to value.  On evaluation, returns valus.  Most useful for things like sql() query. |
| Relational update | is\_called := minus is\_called[ ?(Name = ‘Boris’) ]  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 | is\_called := ?(Name='Boris')  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 | Pred | Attribute Terms | Heading | Ident | Name | Notes |
| Aggregate | Yes | { name := fold-expr SEP } | Add |  | New | Any use of ‘fold’. Combine with extend, project, rename. Infill if no project. |
| Assign | Yes | { name := expr SEP } | Same | Exist | Exist | Parse context. Combine with rename. Infill. |
| Extend | Yes | { name := expr SEP } | Add |  | New | Anything not project or rename. Combine with project, rename. Infill if no project. |
| Ordered | Yes | Any | Any | Any | Any | Anything |
| Project | No | { [-] ident SEP } | Remove | Exist |  | Only if all terms are project. If minus infill and remove specified, else not. |
| Rename | No | { name := name SEP } | Remove, add | New | Exist | Only if terms are all rename. Always infill. |
| Restrict | Yes | -- | Same |  |  |  |

## More Notes

### 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’).