Andl Grammar

This is version 4 of the Andl grammar, influenced by many sources. The compiler is hand-written recursive descent, so this grammar may contain errors.

Notes

1. The symbols `{}[]()+|` are part of the EBNF and are not literals. Where used as terminals the bracket symbols are shown as `LC RC LB RB LP RP`.
2. Other terminal symbols represent themselves.
3. `EOL` and `EOF` represent end of line and end of file. Not shown here are multiple points in the grammar where optional `EOL` is treated as whitespace, and ignored. The final `EOL` in a `DO` block is also optional.
4. The terminal `CMA` represents a comma separator within a repetition. A trailing comma is permitted unless it would create an ambiguity.
5. Other uppercase words are terminals. In the language they are the same words but in lower case.

Statements

```
main        ::= {stmt EOL} EOF
stmt        ::= deferred
              | decls
              | define
              | update
              | assign
              | expr
assign      ::= new-id := expr
decls       ::= DEF { (user-type|connect) CMA }
user-type   ::= : new-id LP decl-list RP
connect     ::= new-id : DB LP [source-id] RP
define      ::= new-id [arg-list] => (expr|update)
arg-list    ::= LP decl-list RP
update      ::= rel-id := tran-op
              | rel-id := dyadic-op rel-expr
```

Expression and primary

```
do-block    ::= DO LC {stmt EOL} RC
expr        ::= primary {bin-op expr}
primary     ::= sim-prim {trn-op|dot-op}
sim-prim    ::= var-id                  // ident
              | att-id                  // attribute
              | lit-val                 // literal
              | rel-val                 // {{ ... }}
              | tup-val                 // { ... }
              | func-call               // ident(...)
              | un-op sim-prim          // - primary
              | LP expr RP              // ( expr )
bin-op      ::= infix-op
              | cmp-op
              | dyadic-op
dot-op      ::= . dot-id
func-call   ::= func-id LP {expr CMA} RP
              | IF LP pred-expr CMA expr CMA expr
              | FOLD LP fold-op CMA expr RP
```

```
func-id     ::= builtin-id
              | def-id
              | win-func
lit-val     ::= bool-lit
              | num-lit
              | text-lit
              | time-lit
              | bin-lit
```

Relation and tuple primaries

```
rel-val     ::= LC {tup-val CMA}+ RC
              | LC heading {LC {expr CMA} RC} RC
              | LC LC * RC RC
tup-val     ::= LC {(proj-term|ext-term) CMA} RC
              | LC * RC
heading     ::= LC {decl-term CMA}+ RC
              | LC : RC
```

Transform operator

```
trn-op      ::= LB [pred-term] [ord-term] [attr-expr] RB
pred-term   ::= ? LP pred-expr RP
ord-term    ::= $ LP {[%] [-] att-id CMA}+ RP
attr-expr   ::= LC [*] {attr-term CMA} RC
attr-term   ::= ren-term
              | proj-term
              | ext-term
              | agg-term
ren-term    ::= new-id := att-id
proj-term   ::= att-id
ext-term    ::= new-id := open-expr
agg-term    ::= new-id := fold-expr
```

Types

```
decl-list   ::= {decl-term CMA}
decl-term   ::= new-id : type-term
type-term   ::= type-id
              | simp-prim
type-id     ::= sys-type-id
              | usr-type-id
sys-type-id ::= BOOL|TEXT|NUMBER|TIME|BINARY
```

Descriptive non-terminals

```
pred-expr   ::= ? predicate expr of type BOOL ?
open-expr   ::= ? expr that may contain an att-id ?
fold-expr   ::= ? open-expr containing at least one FOLD ?
fold-op     ::= ? any operator or function with exactly two
              arguments of the same type ?
```

Descriptive terminals

```
new-id      ::= ? ident definable in this scope ?
dot-id      ::= ? name of single arg function ?
var-id      ::= ? name of variable created by assign or define
              ?
def-id      ::= ? name of function created by define ?
usr-type-id ::= ? name of user-defined type created by user-
              type ?
```

Predefined names

```
builtin-id  ::= ? name of builtin function including:
                type text format pp length fill trim left right
                before after toupper tolower now date dateymd
                year month day dow daysdiff time count degree
                schema seq read ?
un-op       ::= ? unary operators including: + - not ?
infix-op    ::= ? scalar operators including: + - * / ^ div mod
                & max min ?
cmp-op      ::= ? comparison operators including: eq ne ge gt
                le lt and or xor = <> < > <= >= <> =~ sub sep sup
                ?
dyadic-op   ::= ? relational operators including: join compose
                divide rdivide semijoin rsemijoin ajoin rajoin
                ajoinl rajoinr matching notmatching union
                intersect symdiff minus rminus ?
win-func    ::= ? window functions including: ord ordg lead lag
                nth rank ?
source-id   ::= ? a connection source, including csv txt con
                file ?
```

Terminals

```
bool-lit    ::= ? the literal values true and false ?
str-lit     ::= ? string literal consisting of any sequence of
                quote, squote, dquote or hquote ?
bin-lit     ::= ? binary literal b'aabbcc' where aa, bb... are
                hex bytes ?
num-lit     ::= ? digit string with optional decimal point ?
hex-lit     ::= ? $ then numeric digit then sequence of hex
                digits ?
time-lit    ::= ? t'2015/12/31 23:59:59' where the date and
                time are in the locale-dependent format ?
squote      ::= ? '<any>' single quoted string (no escapes) ?
quote       ::= ? "<any>" double quoted string (no escapes) ?
iquote      ::= ? i'<any>' as per squote ?
dquote      ::= ? d'xx xx' where xx are space separated Unicode
                code points in decimal ?
hquote      ::= ? h'xx xx' where xx are space separated Unicode
                code points in hex ?
ident       ::= ? character string, first must be "a-zA-
                Z_$@#^", subsequent may be "%&?!~`|" ?
                | ? iquote followed by any sequence of quote,
                squote, dquote or hquote ?
operator    ::= ? one or two of " -+=<>:*~" ?
line        ::= ? sequence of Unicode characters as provided by
                input source, with all control characters removed
                ?
white       ::= ? the space character (code 32) ?
comment     ::= ? from // to end of line, treated as part of
                EOL ?
EOL         ::= ? token inserted to represent end of line ?
EOF         ::= ? token inserted to represent end of input ?
```