Mike Hahn – hahnbytes@gmail.com

Coopleteer

Coopleteer.com is the home of Cooplet Builder, an Android app which lets you develop cooplets, or apps written in a new programming language called Cooperscript. Cooplet users install cooplets written by other users, and use these cooplets to maintain a database of records, or to access records belonging to other users of the same cooplet. Each record can be an image or a piece of text. Text is formatted using a markup language called Coopertags, and may contain embedded images. Cooplet developers use Windows or Linux computers to develop cooplets, but the end-users use Android devices. Coopleteer is free to download, but paid subscribers receive perks such as full read/write access to cooplet databases.

Business Model

Subscribers pay $10/year or $3.60 for 3 months. They can add/modify/delete records when given write access, download records, copy/paste text contained in those records, and view normal versions of image files. Non-subscribers can only access thumbnail versions of image files, which have a maximum size of 9216 pixels (96 x 96). They cannot download records or copy/paste text. All users who join in the first 6 months, post-launch, receive a free 6-month subscription.

Searching

To search a database for matching records, the user fills out one or more fields in one or more data entry screens, one screen per table. For databases containing more than one table, a master-detail relationship exists between adjacent tables in the table list. All radio buttons/comboboxes are initially deselected, and check boxes are 3-state: on, off, don't care. Enter a period (.) to match a null string in a text field. Only alphanumeric characters are recognized in text edit boxes, other characters are treated as blanks. Keywords entered result in a hit whenever all keywords occur in the matching record. Phrases (multiple adjacent keywords) are delimited with double quotes (").

Search Results

The search results are returned in user-id/hit-count pairs. This list is sorted in descending order by hit-count. The hit-count can be view-count, zip-count, or view-zip-product-count. Whenever an end-user clicks on Zip, a record is downloaded from another user's database, and the zip-count Z is incremented. Whenever an end-user performs a command resulting in a Zip button being displayed, the view-count V is incremented. The view-zip-product-count, P = V(Z + 1).

Clicking on a list member in the list of user-id/hit-count pairs displays the first record matching the query. (When performing a global search, a list of matching cooplets is displayed at this point prior to displaying the first matching record.) By using the next/previous commands (swipe left/right), the user steps through the list of records matching the query, which is sorted in descending order by hit-count. Press and hold to select record. Press and hold bottom of screen to display list of page numbers. Leftmost page number (zero) returns to list of users (search results). Swipe up to hide list of page numbers or scroll down. Swipe down to scroll up.

Easy to Learn

The mandate of Coopleteer is to use a new programming language called Cooperscript to simplify the development of Android apps, especially for novices. The Coopleteer smartphone app is based on cooplets, which are apps written in Cooperscript. Each cooplet has its own database. Coopleteer is pronounced "coop-luh-TEER". Why should Android developers bother learning Cooperscript? Cooplet development is much easier to learn than Android development using Java, making it a novice-friendly stepping stone towards learning how to develop smartphone apps.

Indexed Search

Keywords
- keyid (4)
- text (16)

KeyPages
- keyid (4)
- pgid (8)

Phrases
- pgid (8)
- keyid (4)
- pos (2)

AppPages
- appid (4)
- pgid (8)

Monospace Mode

In monospace mode, all body text rendered to the screens of end-users is in a mono-spaced, typewriter-style font. Every character takes up 2 square cells: an upper cell and a lower cell. Superscripts and subscripts are handled by employing a vertical offset of one square cell. Header text is also mono-spaced, and each character takes up 2 oversized square cells.

Additional Formatting

The grid of characters can be subdivided into panels, which can themselves be subdivided into more panels, and so on. Any panel can contain zero or more text boxes, which may overlap each other. Vertical grid lines each take up one square cell per row of square cells. Horizontal grid lines are displayed in the same pixel row as underscore characters. Any row of square cells containing a horizontal grid line which is 2 pixels wide is taller by exactly one pixel. The following bracket characters: ( ) [ ] { } can be oriented vertically or horizontally, taking up a single column or row of at least 2 square cells, respectively. Widgets such as check boxes, radio buttons, and combo box arrows take up 4 square cells (2 by 2). Images, animations, and diagrams are contained in canvas objects, which can appear anywhere panels can appear.

Rich-Text Mode

In rich-text mode, a given header or paragraph of body text can consist of a single variable-width font. Paragraphs have before/after spacing, left/right indent, and line spacing (single, double, 1.5, etc.). Panels have margins on all 4 sides.

Public Fields

Public fields are Cooperscript fields (method variables) which are declared in a var block, whereas private (inner) fields are declared in an ivar block. Public fields which include getters and/or setters cannot be modified directly using an assignment statement, except within the class in which they are declared. For a field called myfield, the corresponding getter method is called get-myfield, and the corresponding setter method is called set-myfield. For a boolean field called myfield, the corresponding getter method is called is-myfield. For a boolean field called is-myfield, the corresponding getter method is called get-myfield.

About Me

I am Mike Hahn, the founder of Coopleteer.com. I was previously employed at Brooklyn Computer Systems as a Delphi Programmer and a Technical Writer (I worked there between 1996 and 2013). At the end of 2014 I quit my job as a volunteer tutor at Fred Victor on Tuesday afternoons, where for 5 years I taught math, computers, and literacy. I'm now a volunteer math/computer tutor at West Neighbourhood House. My hobbies are reading quora.com questions/answers and the news at cbc.ca. About twice a year I get together with my sister Cathy who lives in Victoria. She comes here or I go out there usually in the summer. A few months prior to starting my Coopleteer project I used to lie on the couch a lot, not being very active. Now I'm busy most of the time. I visit my brother Dave once a month or so and I also visit my friends Main and Steph once or twice a month. For 26 years I was depressed on and off (I'm a rapid cycler), but it largely vanished after I started my Coopleteer project.

Contact Info

Mike Hahn
Founder, Coopleteer.com
515-2495 Dundas St. West
Toronto, ON M6P 1X4

Country: Canada
Phone: 416-533-4417
Email: hahnbytes (AT) gmail (DOT) com
Web: www.hahnbytes.com

Implementation Steps

1. Implement Cooperscript 0.1, console-based
    - Token parsing and building program tree has already been implemented
2. Finish Cooperscript 1.0, console-based
3. Make Cooperscript Android-based
4. Begin code editor development
5. Write Coopertags design specs
6. Recruit GitHub open source coders/testers
7. Implement monospace mode
8. Implement rich-text mode
9. Integrate Cooperscript with Coopertags (monospace/rich-text modes)
10. Design core Coopleteer website (closed source)
    1. Databases
    2. Indexed search
11. Launch core website
12. Advertise using Google AdWords
13. Beta test Coopleteer
14. Implement Cooperscript SDK
    1. WYSIWYG Coopertags editor
    2. Cooperscript code editor
15. Design full Coopleteer website
16. Launch full website
17. Accept credit card payments

Cooplet Engine

Coopleteer.com consists of 2 Android apps: Cooplet Builder and Cooplet Engine. The former is used to build cooplets, and the latter is used to run cooplets which other users have already built. Users who want to browse/maintain databases which are managed by an existing cooplet do not need Cooplet Builder, they only need Cooplet Engine. Cooperscript developers need Cooplet Builder to develop cooplets. Cooplet Builder encompasses the full functionality of Cooplet Engine, so developers need not install Cooplet Engine. Both Android apps are free downloads, but many users will choose to become paid subscribers, so they can receive perks such as full read/write access to cooplet databases.

Cooperscript

Cooperscript is a Python dialect in which all operators precede their operands, and parentheses are used for all grouping (except string literals, which are delimited with double quotes). Cooperscript source files have a .COOP extension. Coopertags files have a .CPTG extension. COOPERScript stands for Compact Object-Oriented Program Editor and Runtime System. Cooperscript is open source and implemented using Java.

Special Characters

- ( ) grouping
- – used in identifiers
- ; end of stmt.
- : dot operator
- " string delimiter
- \ escape char.
- # comment
- Extra:
  - _ used in identifiers
  - $ string prefix char.
  - { } block comment
  - {{ }} Coopertags comment

Version 0.1

- No inheritance
- No interfaces
- No IDE

Differences from Python

- Parentheses, not whitespace
- Integration with Coopertags
- Operators come before their operands
- Information hiding (public/private)
- Single, not multiple inheritance
- Adds interfaces ("scool" defs.)
- Drops iterators and generators
- Adds lambdas
- Adds quote and list-compile functions, treating code as data
- Adds cons, car and cdr functionality

Grammar Notation

- Non-terminal symbol: <symbol>
- Optional text in brackets: [*text*]
- Repeats zero or more times: [*text*]…
- Repeats one or more times: <symbol>…
- Pipe separates alternatives: *opt1* | *opt2*
- Comments in *italics*

Keyboard Aid

This optional feature enables hyphens, open parentheses, and close parentheses to be entered by typing semicolons, commas, and periods, respectively. When enabled, keyboard aid can be temporarily suppressed by using the Ctrl key in conjunction with typing semicolons, commas, and periods (no character substitution takes place). By convention, hyphens are used to separate words in multi-word identifiers, but semicolons are easier to type than hyphens. Similarly, commas and periods are easier to type than parentheses. Typing semicolon converts previous hyphen to a semicolon, and previous semicolon to a hyphen (use the Ctrl key to override this behaviour). Typing semicolon after close parenthesis simply inserts semicolon. The close delim switch automatically inserts a closing parenthesis/double quote when the open delimiter is inserted.

Coopertags

Coopertags is a simplified markup language used to replace HTML (also used to define screen layouts). Arbitrary Coopertags code can be embedded in the Cooperscript echo statement. Coopertags syntax, where asterisk (*) means repetition, is defined as follows:

- Tags:
  - [tag]
  - [tag: body]
  - [tag (fld val)*: body]
- Body:
  - text
  - [: text]*
  - [(fld val)*: text]*
- Call Cooperscript code:
  - [expr: <expr>]
  - [exec: <stmt>...]
  - [coop: <path>]

Cooperscript Grammar

White space occurs between tokens (parentheses and semicolons need no adjacent white space):

<source file>:
- do ([<imp>]... [<def glb>] [<def>]... [<class>]...)

<imp>:
  <import stmt> **;**

<import stmt>:
  import <module>...
  from <rel module> import <mod list>
  from <rel module> import all

<module>:
  <name>
  (**:** <name><name>...)
  (as <name><name>)
  (as (**:** <name><name>...) <name>)

<mod list>:
  <id as>...

<id as>:
  <mod id>
  (as <mod id><name>)

<mod id>:
  <mod name>
  <class name>
  <func name>
  <var name>

<rel module>:
  (**:** [<num>] [<name>]...)
  <name> *// ?*

<class>:
- <cls typ><name> [<base class>] [<does>] [<vars>] [<ivars>] do (<def>…) **;**
- abclass <name> [<base class>] [<does>] [<vars>] [<ivars>] do (<anydef>…) **;**
- <scool><name> [<does>] [<const list>] do ([<abdef>]... [<defimp>]...) **;**
- enum <name><elist> **;**
- ienum <name><elist> **;**

<cls typ>:
  class
  iclass

<does>:
  (does <scool name>...)

<scool name>:
<base class>:
  <name>
  (**:** <name><name>…)

<const list>:
  (const <const pair>...)

<const pair>:
  (<name><const expr>)

<scool>:
  scool
  iscool

<def glb>:
  gdefun [<vars>] [<ivars>] do <block> **;**

<def>:
- <defun> (<name> [<parms>]) [<vars>] [<dec>] do <block> **;**

<defimp>:
- defimp (<name> [<parms>]) [<vars>] [<dec>] do <block> **;**

<abdef>:
  abdefun (<name> [<parms>]) [<dec>] **;**

<defun>:
  defun
  idefun

<anydef>:
  <def>
  <abdef>

<vars>:
  (var [<id>]...)

<ivars>:
  (ivar [<id>]...)

<parms>:
  [<id>]... [<parm>]... [(* <id>)] [(** <id>)]

<parm>:
  (<set op><id><const expr>)

<dec>:
  (decor <dec expr>...)

<block>:
  ([<stmt-semi>]…)

<stmt-semi>:
    <stmt> **;**

<jump stmt>:
    <continue stmt>
    <break stmt>
    <return stmt>
    return <expr>
    <raise stmt>

<raise stmt>:
    raise [<expr> [from <expr>]]

<stmt>:
    <if stmt>
    <while stmt>
    <for stmt>
    <try stmt>
    <asst stmt>
    <del stmt>
    <jump stmt>
    <call stmt>
    <print stmt>

<call expr>:
- (<name> [<arg list>])
- (**:** <obj expr> [<colon expr>]…
  (<method name> [<arg list>]))
- (call <expr> [<arg list>])

<call stmt>:
- <name> [<arg list>]
- **:** <obj expr> [<colon expr>]…
  (<method name> [<arg list>])
- call <expr> [<arg list>]

<colon expr>:
    <name>
    (<name> [<arg list>])

<arg list>:
    [<expr>]... [(<set op><id><expr>)]...

<dec expr>:
    <name>
    (<name><id>...)
    (**:** <name><id>...)
    (**:** <name>... (<id>...))

<dot op>: *// 'dot', ':', both OK*
    dot | **:**

<del stmt>:
    del <expr>

<asst stmt>:
    <asst op><target expr><expr>
    <set op> (tuple <target expr>...) <expr>

<asst op>:
    set | addset | minusset | mpyset | divset |
    idivset | modset |
    shlset | shrset | shruset |
    andbset | xorbset | orbset |
    andset | xorset | orset |
    = | += | -= | *= | /= |
    //= | %= |
    <<= | >>= | >>>= |
    &= | ^= | '|=' |
    &&= | ^^= | '||='

<set op>:
    set | =

<target expr>:
    <name>
    (**:** <name> [<colon expr>]… <name>)
    (slice <arr><expr> [<expr>])
    (slice <arr><expr> all)
    (<crop><cons expr>)

<arr>: *// string or array/lyst*
    <name>
    <expr>

<obj expr>:
    <name>
    <call expr>

<if stmt>:
- if <expr> do <block> [elif <expr> do <block>]… [
  else do <block>]

<while stmt>:
    while <expr> do <block>
    while do <block> until <expr>

<for stmt>:
    for <name> in <expr> do <block>

<try stmt>:
- try do <block> <except clause>… [else do
  <block>] [eotry do <block>]
- try do <block> eotry do <block>

<except clause>:
    except <name> [as <name>] do <block>

<return stmt>:
    return

<break stmt>:
    break

<continue stmt>:
    continue

<paren stmt>:
    (<stmt>)

<qblock>:
    (quote [<paren stmt>]...)

<expr>:
    <keyword const>
    <literal>
    <name>
    (<unary op><expr>)
    (<bin op><expr><expr>)
    (<multi op><expr><expr>…)
    (<quest><expr><expr><expr>)
    <lambda>
    (quote <expr>...)
    <renum expr>
    <cons expr>
    <tuple expr>
    <lyst expr>
    <dict expr>
    <bitarray expr>
    <string expr>
    <bytezero expr>
    <bytes expr>
    <target expr>
    <obj expr>
    <cast>

<quest>:
    quest | ?

<unary op>:
    minus | notbitz | not |
    - | ~ | !

<bin op>:
    <arith op>
    <comparison op>
    <shift op>
    <bitwise op>
    <boolean op>

<arith op>:
    div | idiv | mod | mpy | add | minus |
    / | // | % | * | + | -

<comparison op>:
    ge | le | gt | lt | eq | ne | is | in |
    >= | <= | > | < | == | !=

<shift op>:
    shl | shr | shru |
    << | >> | >>>

Note: some operators delimited with single quotes for clarity (quotes omitted in source code)

<bitwise op>:
    andbitz | xorbitz | orbitz |
    & | ^ | '|'

<boolean op>:
    and | xor | or |
    && | ^^ | '||'

<multi op>:
    mpy | add | strdo | strcat |
    and | xor | andbitz | xorbitz |
    or | orbitz |
    * | + | % | + |
    && | ^^ | & | ^ |
    '||' | '|'

<const expr>:
    <literal>
    <keyword const>

<literal>:
    <num lit>
    <str lit>
    <bytes lit>

<cons expr>:
    (cons <expr> [<expr>])
    (<crop><cons expr>)

<tuple expr>:
    (tuple <expr>…)

<lyst expr>:
    (lyst [<expr>]…)

<dict expr>:
    (dict [<pair>]…)

<bitarray expr>:
    (bitarray <enum name> [<elist>])
    (bitarray <enum name><idpair>...)

<elist>:
    <id>...
    <intpair>...
    <chpair>...

<intpair>
    // integer constant
    <int const>
    (<int const><int const>)

<chpair>
    // one-char. string
    <char lit>
    (<char lit><char lit>)

<idpair>
    <idt>
    (<id><id>)

<pair>:
    // expr1 is a string
    (<expr1><expr2>)
    (<str lit><expr>)

<renum expr>
    (renumize <expr><ren id>...)
    (renumize <expr><ren int>...)
    (renumize <expr><ren ch>...)

<ren id>:
    (0 <id>)
    (1 <id>)
    (1 <id><id>)

<ren int>:
<ren ch>:
    // expr is <dec int> | <char lit>
    (0 <expr>)
    (0 <expr><expr>)
    (1 <expr>)
    (1 <expr><expr>)

<cast>:
    (cast <type><expr>)

<print stmt>:   *// built-in func*
    print <expr>…
    println [<expr>]…
    echo <expr>…

<lambda>:
    (lambda ([<id>]...) <expr>)
    (lambda ([<id>]...) do <block>)
    (lambdaq ([<id>]...) do <qblock>)
    // must pass qblock thru compile func

No white space allowed between tokens, for rest of Cooperscript Grammar

<white space>:
    <white token>...

<white token>:
    <white char>
    <line-comment>
    <blk-comment>

<line-comment>:
    # [<char>]... <new-line>

<blk-comment>:
    { [<char>]... }

<white char>:
    <space> | <tab> | <new-line>

<name>:
- [<underscore>]… <letter> [<alnum>]…
  [<hyphen-alnum>]… [<underscore>]…

<hyphen-alnum>:
    <hyphen><alnum>…

<alnum>:
    <letter>
    <digit>

In plain English, names begin and end with zero or more underscores. In between is a letter followed by zero or more alphanumeric characters. Names may also contain hyphens, where each hyphen is preceded and succeeded by an alphanumeric character.

<num lit>:
    <dec int>
    <long int>
    <oct int>
    <hex int>
    <bin int>
    <float>

<dec int>:
    [<hyphen>] 0
    [<hyphen>] <any digit except 0> [<digit>]…

<long int>:
    <dec int> L

<float>:
    <dec int><fraction> [<exponent>]
    <dec int><exponent>

<fraction>:
    <dot> [<digit>]…

<exponent>:
    <e> [<sign>] <digit>…
<e>:
    e | E

<sign>:
    + | -

<keyword const>:
    none
    true
    false

<oct int>:
    0o <octal digit>…

<hex int>:
    0x <hex digit>…
    0X <hex digit>…

<bin int>:
    0b <zero or one>…
    0B <zero or one>…

<octal digit>:
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<hex digit>:
    <digit>
    A | B | C | D | E | F
    a | b | c | d | e | f

<str lit>:
    [$ <str prefix>] <quoted str>

<str prefix>:
    r | u | R | U

<quoted str>:
    " [<str item>]... "

<str item>:
    <str char>
    <escaped str char>
    <str newline>

<str char>:
    any source char. except "\", newline, or
    end quote

<str newline>:
    \ <newline> [<white space>] "

<bytes lit>:
    $ <byte prefix><quoted bytes>

<byte prefix>: *// any case/order*
    b | br

<quoted bytes>:
    " [<bytes item>]... "

<bytes item>:
    <bytes char>
    <escaped char>
    <str newline>

<bytes char>:
    any ASCII char. except "\", newline, or
    end quote

<escaped char>:
    \\ *backslash*
    \" *double quote*
    \} *close brace*
    \a *bell*
    \b *backspace*
    \f *formfeed*
    \n *new line*
    \r *carriage return*
    \t *tab*
    \v *vertical tab*
    \ooo *octal value = ooo*
    \xhh *hex value = hh*

<escaped str char>:
    <escaped char>
    \N{name} *Unicode char. = name*
    \uxxxx *hex value (16-bit) = xxxx*

<crop>:
    c <crmid>... r

<crmid>:
    a | d