The Computer Applications Platform, 0th Edition

December 2012

Contents

Chapter 1

Manifesto, Structure, Functions, Objects

This is the description of `cap` – a Computer Applications Platform. Here is one way to build all the software needed to use a computer including that which is traditionally termed the compiler, the operating system, applications and documentation.

Here is also a process for building software that is intended to be easy to maintain and extend, describe and distribute. As a manifesto for holistic coding, tenets of practice are introduced within the document at points where they are demonstrated. manifesto

To promote understanding, `cap` is a literate program[1] wherein regular descriptive text (such as this paragraph) is mixed with computer evaluated program text in a `monospace font`. literate programming, program text

$\boxed{\text{Tenet the First: The Documentation and the Code are One.}}$

Finally, it is hoped that `cap` embodies a pleasant system to use for general computing tasks.

[1]Donald E. Knuth, Literate Programming, Stanford, California: Center for the Study of Language and Information, 1992, CSLI Lecture Notes, no. 27

1.1 Goals

Goal — Track Goals. 01/01/2009 ✓

Goals

When requirements (or goals) are determined at the outset of a project, progress is tracked, and fulfillment is tested, a project is more likely to be successfully completed more quickly and with greater assurance of satisfactory execution than it would be without establishing and tracking goals[2].

In `cap` the goals are explicitly included, tracked, and tested or acknowledged as having been met from within the literate programming text of `cap` (this document).

The goals mirror the organizational structure as each chapter, section, and subsection provides functionality needed by the system. Section goals are found at the beginning of the section. Goals for the whole project are listed here:

Goals for the Computer Applications Platform

Goal a — Provide functionality comparable to the compiler, operating system, and applications of a conventional platform used for general computing tasks. unmet

Sub-goals specify specific goals that must be met to satisfy the main Goal, such as:

Goal a.1 — Execute on conventional hardware. unmet

A maximally useful system will run on easily acquired computers.

Goal a.2 — Re-create own executable structure from source text. unmet

The system should be self-reliant for its own development. The above Goal also captures the functionality of the typical compiler in existing systems.

Goal a.3 — Provide operating system features such as process isolation, scheduling, storage and retrieval, internetworking, input/output, and computer resource management. unmet

The above Goal encapsulates operating system activities.

[2]Barry W. Boehm, A Spiral Model of Software Development and Enhancement., ACM SIGSOFT Software Engineering Notes 11, 1986

Goal a.4 — Build and execute additional programs from additional source texts. unmet

The above Goal provides for the development of applications for the system.

Goal a.5 — Safely execute machine code compiled by other systems. unmet

CAP may not be the best tool with which to create a particular application. The system should interoperate gracefully with other programming tools and systems.

Goal b — Demonstrate holistic coding. 01/23/2009 ✓

holistic coding

The incorporation and incremental development of software development[3] extrinsic aspects that can be captured in document form is called holistic coding in `cap`. The Goals of a project are one such aspect.

Tenet the Second: Code Holistically.

Goal b.1 — Produce documentation, including a summary of Goals and project status from the source text. unmet

The above Goal integrates documentation of the system with the system.

Goal b.2 — Integrate defect tracking, unit tests, regression tests, and a summary of test results with the source text. unmet

Thorough testing reduces the number of programming defects (bugs) in a system. Testing and defect tracking is another extrinsic that is folded into the `cap` source text in adherence to holistic programming principles.

Goal b.3 — Integrate version and revision control as well as software distribution with the source text. unmet

Another common extrinsic to larger software projects is that of version and revision control, as well as packaging and distribution of the system and its applications and components. The `cap` system should internalize these aspects of software development and use.

One programming task often overlooked and then attempted in an ad-hoc

[3]Frederick P. Brooks, Jr., No Silver Bullet, 1986

manner is that of adapting a program or system to other languages and locales than it was initially developed in. `cap` itself should be easily translated and localized, and it should extend the same flexibility to other program texts written in the same way.

Goal c — Instrument source texts for localization and internationalization. unmet

The preceding Goals outlined in boxes in this section (1.1) are Goals for the system as a whole but they are not all of the Goals for `cap`. Like any large project this effort has been subdivided into smaller parts so that the system may be developed incrementally. Each chapter, section, and subsection is devoted to a specific aspect of `cap` functionality and contains its own Goals, and those Goals taken all together form a complete set which is listed in Appendix I.

The Goals, the code, and the chapters and sections of explanatory text are all developed incrementally and together, with later chapters and sections building on features and concepts introduced in earlier chapters and sections. Goals, tests, and other extrinsic matters of software development are introduced in the text where they are relevant but may also be summarized elsewhere and expanded upon later in the text. This incremental method of development is intended to make the entire system easier to understand and maintain.

Tenet the Third: A Bit at a Time.

1.2 License

Goal — Select License. 01/23/2009 ✓

Software can be copyrighted, and any work intended for distribution should have a statement on the rights reserved by the copyright holder and conferred to the recipient. `cap` is Open Source software and the MIT open source license has been chosen to maximize the freedom of recipients of the software to use and re-use it. Other software projects may have a more or less restrictive license.

open source

The purpose of this license is to allow this software and document to be used for any purpose. Printing as a book for sale for profit is discouraged, but it is not prohibited. (The authoritative print edition of `cap` may be purchased in book form from xxx publishing, ISBN xxxyyyzzz.)

1.3 Project and Sub-Project Status

Goal — Track Project Status. 01/01/2009 ✓

A large project benefits from being broken down into manageable sub-projects. In `cap` each sub-project has its own chapter and each chapter provides concepts and features that the next chapters may build upon.

The complete Computer Applications Platform 0th edition will be presented in the following chapters, including sub-sub-projects and Goals met:

Chapter:		Page — ssp — req
One	Manifesto, Structure, Functions, Objects.	5 — 3/8 — 1/11
Two	The Syntax of Syntax.	47 — 0/4 — 0/9
Three	Kernels and Runtimes.	61 — 0/4 — 0/5
Four	Documents.	63 — 0/4 — 0/0
Five	User Interface.	65 — 0/4 — 0/0
Six	Networking `cap`.	67 — 0/4 — 0/0
Seven	Applications for the Computer Platform.	69 — 0/4 — 0/0
Eight	Distributing the System.	71 — 0/4 — 0/0
Nine	Adapting to the World.	77 — 0/4 — 0/0
Appendix I	Goals.	79 — — — — —
Appendix II	Additional Architectures.	73 — 0/4 — 0/0
Appendix III	Log of Changes From The Previous Edition.	75 — — — — —

Sub-Projects (chapters) for CAP 0th edition.

This document incrementally presents both the abstract concepts of cap and also the concrete realization of cap, with the most fundamental concepts coming first, and also with the most basic realized constructs before later more complex ones. The dualism of abstract concepts on the one hand and concrete constructs on the other will be explored throughout this document. The structure of this book and also of this chapter reflects just such a decomposition of the complete whole.

This chapter has the following sections (asterisk indicates sub-sub-project:)

Section:		Page — ssp — req
1.1	Goals.	6 ———— 2/12
1.2	License.	9 ———— 1/1
1.3	Project and Sub-Project Status.	10 ———— 1/1
1.4 *	Structure, Syntax, Values.	12 ———— 1/1
1.5 *	Machine Architecture and Machine Code.	17 — 1/13— 0/1
1.6 *	The Application Binary Interface.	37 — 0/0 — 0/2
1.7 *	Functions.	42 — 0/0 — 0/2
1.8 *	Objects.	42 — 0/0 — 0/2

Sub-projects (sections) for Chapter 1: Manifesto, Structure, Functions, Objects.

1.4 Structure, Syntax, and Values

Goal — Provide Fundamental Structure. 07/01/2008 ✓

structure, tree

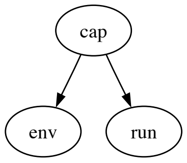
The structure[4] of the Computer Applications Platform can best be understood as a conceptual tree[5] with the label `'cap'` at the root and with the branches of the tree being subsystems, subcomponents, resources, etc.

Tenet the Fourth: There Is But One Tree.

names, programs

`cap` at the root names[6] the system as a whole. There are two branches to the tree to start with: an environment of programs and resources for programs[7] to use and also a set of programs operating in that environment. The program text below introduces the names `env` for environment and `run` for executing or dormant programs. To the right of the program text is a diagram showing the abstract tree structure created by it.

```
`cap ~ (  [ env :~ $ ];
          [ run :~ $ ];
            $ );
```



code, syntax, definition,

The above code[8] presumes a syntax[9] . The syntax shown is that of the `i` incremental programming language. See Syntax Guide to i Code on the next page for an introduction to the meanings of the symbols in the above code.

nodes, abstract syntax tree, extension

The effect of processing the above code is the insertion of new nodes containing names and other attributes in a tree initially having just one node (simply labeled `cap` .) This forms an conceptual tree which in some systems is referred to as an abstract syntax tree.

The syntactic feature of leaving a definition open to extension and then later

[4]Harold Abelson, Gerald J. Sussman: Structure and Interpretation of Computer Programs, Second Edition MIT Press 1996

[5]Donald E. Knuth, The Art of Computer Programming, Volume I: Fundamental Algorithms, 1968, Addison-Wesley, pp. 309-310

[6]Joe Lykken, What's in a name?,Symmetry Magazine, Volume 2 Issue 2 March 2005

[7]N. Wirth, Algorithms + Data Structures = Programs, Prentice Hall, 1975

[8]a.k.a. program text

[9]Roland Carl Backhouse: Syntax of Programming Languages: Theory and Practice Prentice-Hall 1979

extending it is a way to present information in an order for better explanation (pedagogical order) instead of linear program order. For example, the code introduces a place for language definitions (`lang`) within the environment and also leaves room for more environment definitions without specifying them yet:

```
`cap,env ~ (  [ lang :~ $ ]; $ );
```

Syntax Guide to i Code

Definitions come wrapped in square brackets, e.g.

```
 [ something ];
```

A colon and a tilde separate the name of the new thing from its contents, which may be references to things previously defined. Also, the name of a thing can be made of multiple words separated by white space.

```
 [ a thing encompassing :~ something ];
```

Semicolons separate multiple content items, and multiple items within the contents of a definition are grouped by parentheses.

```
 [ apple ];
 [ pear ];
 [ kiwi ];
 [ fruit basket :~ ( apple ; pear; kiwi ) ];
```

The contents of a definition may be marked by a dollar sign indicating that the content will be filled in later.

```
 [ picnic table :~ $ ];
```

A back-tick and a previously-defined reference indicate that a previously marked location is now being filled in. If it is filled in with multiple items then parentheses are needed. The last item may be a dollar-sign marker for later expansion.

```
 [ bottle of wine ];
 ‘ picnic table ~ (fruit basket; bottle of wine; $);
```

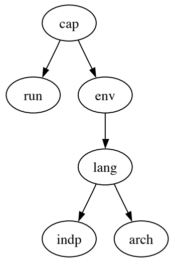
Definitions may be placed within the content area of other definitions or extension to definitions.

i syntax.

architecture dependence and independence

The Computer Applications Platform is intended to run on many different types of computers which may differ in how they operate at a low level such as their machine code or collection of attached devices. Computers in general function similarly as they are all stored program machines with memory, one or more computational units, and some form of input and output. In `cap` aspects of the system that are dependent on the architectural details of the host computer are placed in the `arch` branch of the tree and aspects of the system that are common across architectures can be found in the `indp` branch.

```
`cap,env,lang ~ (
  @[ indp :~ $ ];
  @[ arch :~ $ ];
  $ );
```



values, byte order

The three snippets of `cap` program code so far have only introduced abstract concepts and established structural relationships between them. In addition to abstract structure, definitions can introduce concrete values, which are stored in an 'image' which is kept separately. The most fundamental values are bytes and machine words of varying sizes. The concept of byte order, or endianness[10], is independent of processor architecture, and is defined next, in the branch of the tree for machine independent generation of machine code.

discontinuous, image

This next code section has more than just expansions of nodes as before (solid lines in the illustration.) It also defines nodes which refer to and include copied values of previously defined nodes (dashed line,) values for nodes (dotted line to a box,) nodes with no names but which define a type, extension of lexical scope (dotted line to an oval) and nodes that have types.

types

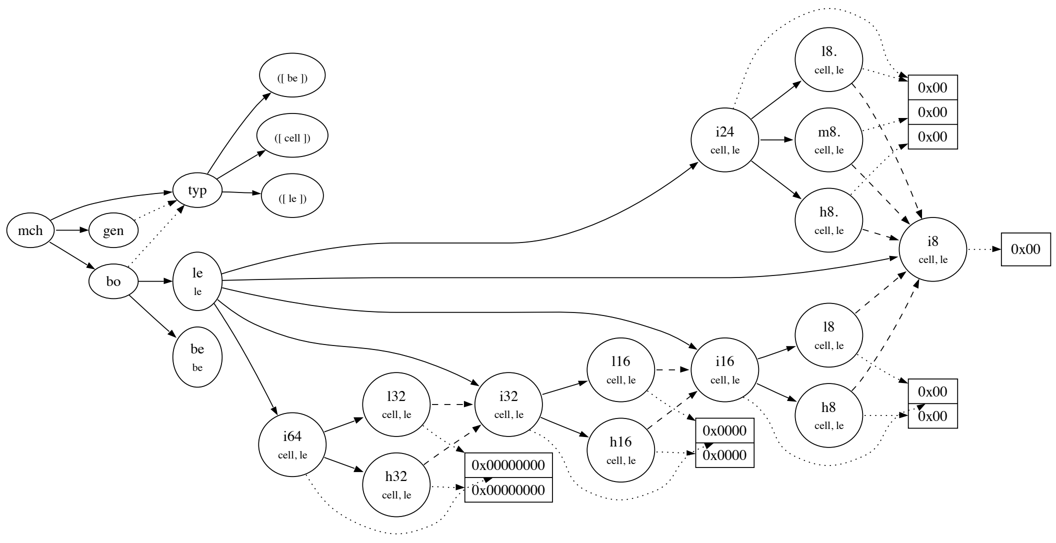
Types[11] are used in the `i` programming language to enable greater expressiveness in and increased safety for the system. The first types introduced in `cap` are for binary values stored in the system image.

The ovals in the figure represent nodes in the abstract syntax tree and square boxes represent image values. Contiguous storage of values is displayed as a 'stack' with the topmost value stored at the lowest memory address. The diagram shows that the values for i8, i16, i24, i32, and i64 are permitted to be

[10]Jonathan Swift,Gulliver's Travels and Other Works, Routledge, 1906
[11]Pierce, Types and Programming Languages, MIT Press

stored discontiguously from each other. bo stands for byte order, le stands for little-endian, and be stands for big-endian.

```
`cap,env,lang,indp ~ (
 [ mch :~ (
  [ typ :~ (
    [:([cell:~])~];
    [:([be:~])~];
    [:([le:~])~])];
  [ gen :~ ( ^typ; $ )];
  [ bo :~ ( ^typ;
   [ le :le~ (
    @()[ i8  :cell:le~ 0x00 ];
    @()[ i16 :cell:le~ ([ l8 :~ i8 ]; [ h8 :~ i8 ])];
    @()[ i24 :cell:le~ ([ l8 :~ i8 ]; [ m8 :~ i8 ]; [ h8 :~ i8 ])];
    @()[ i32 :cell:le~ ([ l16 :~ i16 ]; [ h16 :~ i16 ])];
    @()[ i64 :cell:le~ ([ l32 :~ i32 ]; [ h32 :~ i32 ])])];
   [ be :be~ (
    @()[ i8  :cell:be~ 0X00 ];
    @()[ i16 :cell:be~ ([ h8 :~ i8 ]; [ l8 :~ i8 ])];
    @()[ i24 :cell:be~ ([ h8 :~ i8 ]; [ m8 :~ i8 ]; [ l8 :~ i8 ])];
    @()[ i32 :cell:be~ ([ h16 :~ i16 ]; [ l16 :~ i16 ])];
    @()[ i64 :cell:be~ ([ h32 :~ i32 ]; [ l32 :~ i32 ])])])];$)]);
 $ );
```



When a previously defined item with a value or values is referenced for references, instances inclusion in a new definition (such as the two i16 values within an i32,) a new instance of those values is placed in the image at the new location. This is how `cap` builds binary objects that are the result of compilation in other systems. In some definitions only the sizes and offsets (relative to each other) of the values are important. In those cases the syntax directs the actual bit-patterns of the values to be disregarded when those definitions are referenced.

lexical scope

When a reference is made to previously defined constructs there may be more than one previously defined construct with that name (e.g. there are two definitions of i16 in the byte order code–one with a little-endian arrangement of its bytes, and the other with a big-endian arrangement of its bytes.) `cap` uses lexical scope[12] to manage the accessibility of items for reference.

import, promotion

For example, when defining the big-endian i32 the big-endian previous definition of i16 is naturally 'in scope' because it is a 'peer of a parent' when traversing back up the tree. The `i` syntax for lexical scope extension is used to bring the type definitions for cell, le, and be (and other types that will be defined later) into scope for the definitions of gen and bo.

[12]Harold Abelson and Gerald Jay Sussman, Lexical addressing, Structure and Interpretation of Computer Programs

Hexadecimal values can be introduced in the content area of a definition.

```
[ null byte :~ 0x00 ];
[ secret encryption key :~ 0X09F911029D74E35BD84156C5635688C0 ];
```

Hexadecimal values introduced with a lower-case x are stored in the image in little-endian format (least-significant-byte has the lowest memory address,) those with an upper-case X are stored in big-endian format.

```
[ big endian - the A1 byte comes first :~ 0XA1B2C3D4 ];
[ little endian - the d4 byte comes first :~ 0xa1b2c3d4 ];
```

Bare numbers up to 254 and down to -128 may be used and are bytes.

```
[ an unambiguous byte value :~ -4 ];
```

Types can be introduced in the type area of a definition. Also, definitions may be anonymous.

```
[:([ string ])~ ];
```

Simple ASCII strings (which are also values) can be introduced similarly. The definition may be assigned a type which was previously defined.

```
[ a greeting :  string ~ "hello" ];
```

An item by itself in the content area of a definition that begins with a character and does not match a previously defined item has the value of a short string.

```
[ an identical greeting :  string ~ hello ];
```

Values are placed in a binary image next to each other and the value of (and binary image space taken by) an encapsulating definition is the aggregate of the sub-definitions.

```
[ intl :~ [ phrases :~ [ greetings :~ (
[^ english greeting:  string ~ "Hello" ];
[^ german greeting:  string ~ "Guten Tag" ];
[^ french greeting:  string ~ "Bonjour" ])]]];
```

(intl now has the value "HelloGuten TagBonjour" which takes up 21 bytes).

A caret preceding a reference places definitions within the referenced item into the current scope for future references.

```
^ intl, phrases, greetings;
[ my null terminated string:([ cstring : ])~ ( french greeting; null
byte ) ];
```

Definitions preceeded by an at-sign have values that are not required to be contiguous with siblings or the parent values.

```
@[ separate ];
```

Definitions preceeded by an at-sign and a null space specifier have values which are ignored on reference but which have offsets of interest.

```
@()[ data ];
```

i syntax, continued.

1.5 Machine Architecture and Machine Code

Goal — Declare Machine Codes for Common Architectures. in process

The code in the previous section made increasingly refined divisions of abstract concepts moving from the general to the specific (e.g. from the `cap` system as a whole to the separation of architecture dependent from architecture independent constructs.) In addition it introduced the most basic concrete component with which the rest of the system is constructed: the byte, and then the larger cells that are assemblages of bytes.

instruction, routine

This section will continue the trend of refining the abstract, introducing machine architecture concepts such as registers, instruction formats, and routines, and also general computer programming concepts such as parameters, macros, compile-time calculation, and meta-circular definition. This section also will continue compounding the concrete by composing from bytes and words the actual machine code values for instructions and simple routines executable by real microprocessors.

Top Down, Bottom Up Programming

'Refining the Abstract' is another way of saying 'Top-Down Programming'. 'Compounding the Concrete' is another way of saying 'Bottom-Up Programming'. In cap both Top-Down and Bottom-Up programming styles are used at the same time but for different aspects of structuring the system. This section is decomposed into the following sub-sections:

1.5.1 *	Macro Expansion and Processor Registers.	18 - done
1.5.2 *	No Operation, Return, Clear Temporary.	20 - done
1.5.4 *	Jumps and Calls to absolute addresses.	21 - done
1.5.6 *	Relative Jump and Relative Subroutine Call.	25 - 0/8
1.5.8 *	Register Move.	26 - 0/8
1.5.9 *	Register Load / Store.	28 - 0/8
1.5.11 *	Register to Control Register.	30 — 0/8
1.5.12 *	Zero a Register.	31 - 0/8
1.5.13 *	Integer Math.	31 — 0/8
1.5.14 *	Stack Operations.	32 - 0/8
1.5.15 *	Load Register with Constant.	34 — 0/8
1.5.15 *	Compare Register Values.	34 — 0/8
1.5.15 *	Conditional Relative Branch.	34 — 0/8

Sub-projects (sub-sections) for Section 1.5.

1.5.1 Macro Expansion and Processor Registers

Goal — Define CPU general purpose registers. 04/14/2009 ✓

Processor architectures[13] vary in many ways including how bytes are arranged in memory, how many registers they have, the format of and number of operations they perform –the machine code– and so on. This chapter prepares cap to support eight (or five) initial architectures:

processor architecture, machine code

```
`lang,indp,mch,typ~([:([ cpu ])~];
                   [:([ yb ])~];
                   [:([ nb ])~]; $);
`cap,env,lang,arch ~ (
  ^ lang,indp,mch,bo;
  @[ cpus  :~ ( x86l; x86m; x86s; armm; ppcl; ppcm; spkl; mmix ) ];
  @[ cpue  :~ (   le;   le;   le;   le;   be;   be;   be;   be ) ];
  @[ cpul  :~ (  b64;  b32;  b16;  b32;  b64;  b32;  b64;  b64 ) ];
  @[ cpuj  :~ (  b32;  b32;  b16;  b24;  b24;  b24;  b16;  b24 ) ];
  @[ cpurc :~ (   16;    8;    8;   16;   32;   32;   32;  256 ) ];
  @[ cpurb :~ (   yb;   nb;   nb;   nb;   nb;   nb;   nb;   nb ) ];
  $ );
```

While architectures differ in specific characteristics, in cap their definition follows a common format. cap exploits this common format and uses macro expansion of program text to avoid repetitive definitions that differ only in particulars.

macro expansion

The following macro is repeated once for each entry in 'cpus'. It creates within arch definitions which include in their scopes the branch of the AST reserved for machine-code generation definitions and also either the little-endian or the big-endian definition of memory cell byte order. Each is set to be expanded later with assembly language constructs for that architecture.

generator

```
`cap,env,lang,arch ~ (
<: [ <. cpus, _ .> : cpu; <. cburb .> ~ ( ^ lang,indp,mch,gen;
                                          ^ <. cpue, _ .> ;  $ )]; :>
 $ );
```

The expansion macro is equivalent to but is more concise than the following:

[13]John L. Hennessy and David A. Patterson, Computer Architecture A Quantitative Approach, 1992

```
`cap,env,lang,arch ~ (
      [ x86l : cpu; yb ~ ( ^ lang,indp,mch,gen;
                             ^ lang,indp,mch,bo,le ;  $ )];
      [ x86m : cpu; nb ~ ( ^ lang,indp,mch,gen;
                             ^ lang,indp,mch,bo,le ;  $ )];
      [ x86s : cpu; nb ~ ( ^ lang,indp,mch,gen;
                             ^ lang,indp,mch,bo,le ;  $ )];
      [ armm : cpu; nb ~ ( ^ lang,indp,mch,gen;
                             ^ lang,indp,mch,bo,le ;  $ )];
      [ ppcl : cpu; nb ~ ( ^ lang,indp,mch,gen;
                             ^ lang,indp,mch,bo,be ;  $ )];
      [ ppcm : cpu; nb ~ ( ^ lang,indp,mch,gen;
                             ^ lang,indp,mch,bo,be ;  $ )];
      [ spkl : cpu; nb ~ ( ^ lang,indp,mch,gen;
                             ^ lang,indp,mch,bo,be ;  $ )];
      [ mmix : cpu; nb ~ ( ^ lang,indp,mch,gen;
                             ^ lang,indp,mch,bo,be ;  $ )];
 $ );
```

From this point forward in this chapter, The following code sections are all expansions of the same `gen` portion of the tree, so the first line

```
‘cap,env,lang,indp,mch,gen~(
```

and the last line

```
$ );
```

of program text sections will be omitted for brevity and should be assumed in the program code through to the end of section 1.5.

The program text below introduces a simple macro which provides places within the definitions of each architecture for assembly language instructions (`asm`) register definitions (`rgs`) operation definitions (`ops`) and routines defined in a meta-circular manner (`exo`:)

```
<: `cap,env,lang,arch, <* cpus, _ *> ~ (
    @[ ^ asm :~ $ ];
    @[ ^ rgs :~ $ ];
    @[ ^ ops :~ $ ];
    @[ ^ exo :~ >$< ];
    $ );  :>
```

arguments, parameters

The following nested pair of definitions uses a macro to transform a tabular arrangement of items into a sequence of definitions which are specialized according to the macro's expansion. This macro expands on the contents of arguments which will be supplied to parameters of the definitions instead of on the contents of a previously declared item as before.

```
[ 'ty 'lo /t/ 'co 'bo :~ (

   [ 'ar  |t| 'da :~ (<:`lo,ar~ [<. co, _ .> : ty ~  <. da, _ .> ]; :> )];

  bo )];
```

The definitions can be used as follows to declare the integer register size and the number of integer and floating point a registers in a bank as well as the number of banks and whether or not there are register windows for each of the architectures. But first a datum type is needed:

```
`lang,indp,mch,typ~([:([ dat ])~]; $);
```

And then the macro is used:

```
( dat )
lang,arch /t/ ( risize; ibregs; ibs; fbregs; fbs; rgws )(
x86l      |t| (   8   ;   8   ;  2 ;    8  ;  1 ;  0   );
x86m      |t| (   4   ;   8   ;  1 ;    8  ;  1 ;  0   );
x86s      |t| (   2   ;   8   ;  1 ;    8  ;  1 ;  0   );
armm      |t| (   4   ;   16  ;  1 ;   32  ;  1 ;  0   );
ppcl      |t| (   8   ;   32  ;  1 ;   32  ;  1 ;  0   );
ppcm      |t| (   4   ;   32  ;  1 ;   32  ;  1 ;  0   );
spkl      |t| (   8   ;   32  ;  1 ;   32  ;  1 ;  1   );
mmix      |t| (   8   ;  256  ;  1 ;    0  ;  0 ;  0   ));
```

The effect of the macro when used as above is to create the equivalent of 48 lines of program text like the following:

```
   `lang,arch,x86l ~ ( [ risize : datum ~  8 ];  $ );
```

The outer definition for column headings identified by the literal `/t/` has four parameters: the `ty`pe, the `lo`cation, the `co`lumns, and the `bo`dy. The inner definition for rows of table data identified by the literal `|t|` has two: the `ar`chitecture, and the `da`ta.

literals

The `ty` parameter in `/t/` is not referenced in `/t/` itself but it is refered to in the body of |`t`|, where `ty` is an unbound reference. When `/t/` itself is *referenced* and receives its arguments the parameter `ty` receives the argument

unbound reference, lexical scope

`datum`. Each time |`t`| is *referenced* in the code above `ty` is bound to `datum` for those iterations of macro expansion. Expansion is necessarily delayed until arguments are received by parameters.

A new type identifies register definitions and another set of definitions with embedded macros will help with defining the registers for each architecture:

```
`lang,indp,mch,typ~([:([reg])~]; $);

[  'lo  /r/ 'co 'bo :~ (

  [ 'ar |i| 'da :~ (
    <:`lo,ar~[<.co,_.>:reg~ [indx:~(<.da,_.>   ; $ ) ]]; :> )];
  [ 'ar |a| 'da :~ (
    <:`lo,ar, <.co,_.>    ~([bank:~(<.da,_.>)]); :> )];
  [ 'ar |r| 'da :~ (
    <:`lo,ar~[<.co,_.>:reg~([indx:~ <.da,_.>];[bank:~0]))]];:>)];

  bo )];
```

In the above set of nested definitions, the |`r`| inner definition is used for architectures that have only one bank of registers (bank 0.) An architecture where a register may have a bank other than 0 will use the |`i`| definition for the index of the registers and then the |`a`| definition for the bank, as follows:

```
   lang,arch
        /r/(  pgc;  acc;  tmp;  stk;  frm;  lnk;  ar2;  vr2 )(
  x86l  |i|(   -1;    0;    3;    4;    5;   -1;    6;    5 );
  x86l  |a|(   -1;    0;    0;    0;    0;   -1;    0;    1 );
  x86m  |r|(   -1;    0;    3;    4;    5;   -1;    6;   -1 );
  x86s  |r|(   -1;    0;    3;    4;    5;   -1;    6;   -1 );
  armm  |r|(   15;    0;    1;   13;   11;   14;    2;    4 );
  ppcl  |r|(   -1;    3;   30;   14;    1;   29;   15;    4 );
  ppcm  |r|(   -1;    3;   30;   14;    1;   29;   15;    4 );
  spkl  |r|(   -1;    8;    4;   14;   30;   15;    9;   16 );
  mmix  |r|(   -1;    0;   16;  253;  254;   -1;    1;   17 ));
```

The numbers above tabulate which register in the regular register file for each architecture is to be used for a program counter, accumulator, stack pointer, etc. In some cases a special purpose register is not exposed in the regular register file in which case a -1 is recorded in the table. For architectures with banks of registers (e.g. the x86l) a bank value is also defined in parallel.

The effect of the macro when used as above is to create the equivalent of 72 lines of program text like:

```
`lang,arch,x86l~([ vr2 : reg ~ ([ indx :~ 5 ]; $ )]; $ );
```

and:

```
`lang,arch,x86l,vr2~( [ bank :~ 1 ]; $ );
```

The following macro expansion definitions create for each architecture all of the definitions for general registers of that architecture.

```
[ defregs 'a 'c : a ? nbnk ~ ( <: `arch,a~(
 [ r\<*0:x:{c-1}*> :reg~([indx:~  x      ];[bank:~   0     ])]); :> $ )];

[ defregs 'a 'c : a ? ybnk ~(  <: `arch,a~(
 [ r\<*0:x:{c-1}*> :reg~([indx:~{ x % 2 }];[bank:~{x/(c/2)}])]); :> $ )];

<: defregs <.arch,cpus,_.> <.arch,cpurc,_.>; :>
```

The above macros expand to definitions like the following:

```
`arch,armm~( [ r0 : reg ~([ indx :~ 0 ];[ bank :~ 0 ]) ]; $ );
```

and:

```
`arch,x86l~( [ r15 : reg ~([ indx :~ 7 ];[ bank :~ 1 ]) ]; $ );
```

In this scheme the registers of the Intel architecture are not named e.g. (r,e)ax, cx, dx, bx, sp, bp, si, and di but instead are named r0, r1, r2, r3, r4, r5, r6, and r7 respectively.

Some of the general purpose registers are dedicated to specific tasks. r0 on the Intel architecture, which is commonly known as rax or eax or ax, is used to accumulate the results of computations and has also been defined with the same indx and bank values as 'acc'. On the Sparc processor acc (the accumulator) is also known as r8.

Macro expansion uses sequences of items such as:

```
[ fruit :~ ( apple ; banana; pear; orange; grape; kiwi ) ];
[ beverage :~ ( soda; wine; distillate; juice ) ];
[ container :~ ( can; bottle; flask; carton ) ];
```

Program text for macro expansion is surrounded by `<:` and `:>` .

Expansion is driven by `<*` reference , _ `*>` where several generator references multiply, or `<.` reference , _ `.>` where several generator references advance lock-step.

```
<:  [ <* fruit, _ *> processor :~ $ ]; :>
```

A variable may be bound to the generated term for later referral.

```
<:  ‘ <* fruit, _x *> processor ~ (
[ makes x <* beverage, _ *> :~ ]; $ ); :>
```

Lock-step expansion requires source generators with an equal set of items each.

```
[ package ’a in ’b :  a type ~ a routine ];
<:  package <.  beverage, _ .> in <.  container, _ .>; :>
```

A dollar-sign wrapped in greater and less-than signs mark a content area for receiving externally generated constructs. Meta-circular systems eventually define the constructs.

```
[ external def :~ >$< ];
[ new def :~ external def, never before seen item ];
```

i syntax, continued.

1.5.2 No Operation, Return from Subroutine

Goal — Define No Operation, Return Instructions. 04/15/2009 ✓

constants

Computer programs are fundamentally built of machine instructions. For each machine instruction one or more byte values are included in the image for the routine being constructed. One instruction (that does nothing but increment the instruction pointer) for all architectures is the nop, or no-operation instruction. A similarly simple binary value instructs the processor to return from a subroutine. Machine instructions and routines have a type, so a type for binary machine code is needed:

```
`lang,indp,mch,typ~([:([bin])~]; $);
```

A place is made in each architecture for assembly instruction definitions:

```
<: `lang,arch,<. lang,indp,cpus, _ .> ~([ asm:~ $ ]); :>
```

The cap system introduces a type for instruction formats, a place for formats common to all architectures, and a place in each architecture for architecture specific format definitions:

```
`lang,indp,mch,typ~([:([fxp])~]; $);
`lang,indp,mch ~([ fmt:~ $ ]);
<: `lang,arch,<.lang,indp,cpus,_.> ~([ fmt:~(^lang,indp,mch,fmt; $)]); :>
```

One format that is the same for all architectures is that of the constant opcode:

```
`lang,indp,mch,fmt~( [ enc c 'z :fxp~ z ]; $ );
```

The preceding encoding format simply declares that the opcode constant be placed in the machine code routine without modification. The following definitions and macros will help define opcodes that use a previously defined encoding format (such as the 'c' format just shown.)

```
[ 'ty 'lo 'pa 'op /o/ 'co 'bo :~ (

 [ 'ar 'f1 |o| 'da :~(
   <:`lo,ar,asm~[ op,(mknm <.co,_.> <*f1,_f*> ) :ty~(
      lo,ar,fmt,(enc f <.da,_.> \(<.pa,_.>) ); :> )];
 [ 'ar 'f1 'f2 |o| 'da :~(
   <:`lo,ar,asm~[ op,(mknm <.co,_.> <*f1,_f*>\<*f2,_g*> ) :ty~(
      lo,ar,fmt,(enc f\g <.da,_.> \(<.pa,_.>) ); :> )];
  bo; )];
```

The above subordinate |o| definition includes a macro which expands in the architecture for the row, introducing definitions named by the column header item 0, those definitions containing a machine instruction encoded in the format specified by the column entry item 1, with the binary value of the column entry item 0, and with the additional parameters to the instruction encoding provided by the column header item 1. The macro can be used to create the definitions of the no-operation and the return instructions as follows:

```
(bin) lang,arch () ([mknm 'op 'f :~ op ])
         /o/(   nop      ;  return    )(
x86l (c) |o|( 0X90       ; 0XC3       );
x86m (c) |o|( 0X90       ; 0XC3       );
x86s (c) |o|( 0X90       ; 0XC3       );
armm (c) |o|( 0X0000A0E1 ; 0X1eff2fe1 );
ppcl (c) |o|( 0X60000000 ; 0X4E800020 );
ppcm (c) |o|( 0X60000000 ; 0X4E800020 );
spkl (c) |o|( 0X01000000 ; 0X81C3E008 );
mmix (c) |o|( 0XFD000000 ; 0XF8000000 ));
```

The return operation above merely redirects the processor to the next instruction in the routine from which the current subroutine was called. Processors differ in how much else the instruction may do to the processor state (manipulate the stack, shift register windows, etc.) and whatever else is required for a complete calling convention (detailed for cap in section 6 of this chapter.)

The native assembly-language equivalents of the above machine code are as follows:

```
       nop     return
x86l   nop     ret
x86m   nop     ret
x86s   nop     ret
armm   nop     bx lr
ppcl   nop     blr
ppcm   nop     blr
spkl   nop     jmpl %r15+8,%r0
mmix   nop     pop 0,0
```

Not much can be done with just the two above instructions:

```
^cap,env,lang,arch,x86m,asm;
@[ waste some time :~(  nop;  nop;  nop;  nop; )];
@[ a leaf subroutine :~( waste some time; return )];
a leaf subroutine;
```

...but the above program text will produce actual executable machine code appropriate for Intel's Pentium cpu.

1.5.3 Register Integer Operations

Goal — Define Register Integer Operations. 04/17/2009 ✓

The register integer operations such as shift, add, subtract, etc. will use different instruction formats, specialized for each architecture and for register-register-immediate (r-r-i, e.g. `r1 from r1 shl 8`) or register-register-register (r-r-r, e.g. `acc from acc sub tmp`.) Complex binary values can be constructed from simple binary values by using the 'or' binary operation to accumulate simple values that have been appropriately shifted. A couple of 'pack' definitions will help with value construction:

```
[ pack 'sz 'va 'sl 'vb :~ { sz <== (( va << sl ) | vb )} ];
[ p2  'sz 'v1 's1 'v2 's2 'vb :~ pack sz v1 s1 (pack sz v2 s2 vb ) ];
[ pf 'sz 'v1 's1 'vb :~ pack ar,fmt,sz ar,fmt,v1 ar,fmt,s1 vb  ];
```

A constant value may be required to occupy only a certain number of bits, even if it is negative (i.e. with leading ones):

```
[ chop 'sz 'ma 'vb :~ { sz <== ( ma ^| vb )} ];
```

compile time macro execution

The body of the 'pack' definition above, which includes a left shift, a binary-xor, and a fixed-size-assertion, is not passively stored but is instead immediately executed at the time the pack is referenced (as indicated by the curly-brackets) and the binary value result is made available to the referrer, rather than the sequence of instructions that comprise the mathematical operations. This feature of `i` syntax is analogous to the compile-time macro[14] facility presented by many other languages. Pack2 uses pack twice.

```
[ pr 'ar 'ra 'rb 'rd 'op :~
    p2   ar,fmt,sz1   ra,indx   ar,fmt,ra   ra,bank   ar,fmt,ba (
   p2   ar,fmt,sz1   rb,indx   ar,fmt,rb   rb,bank   ar,fmt,bb (
  p2   ar,fmt,sz1   rd,indx   ar,fmt,rd   rd,bank   ar,fmt,bd op ))];

[ pi 'ar 'ra 'ic 'rd 'op :~
      p2   ar,fmt,sz1   ra,indx   ar,fmt,ra   ra,bank   ar,fmt,ba (
     p2   ar,fmt,sz1   rd,indx   ar,fmt,rd   rd,bank   ar,fmt,bd (
    p2   ar,fmt,sz1   ic     (chop ar,fmt,sz1 ar,fmt,icm ar,fmt,ics)
        ar,fmt,cx   ar,fmt,cs   op ))];
```

Packrrr encodes into the provided base opcode value (op) the register index (ra, rb, rd) and bank (ba, bb, bd) for three registers. If the register bank is 0 then packing the bank makes no change to the instruction (xor with 0 is no-op) so architectures without banks can be treated as having a bank of 0 for all registers.

[14]McIlroy, M.D., Macro Instruction Extensions of Computer Languages, Communications of the ACM 3 no. 4 (April 1960), pp. 214-220

A set of generalized pack definitions can be specialized on the kind of operation (e.g. register-signed-carry or immediate-unsigned-notcarrying):

```
[ri:~(r;i)];[su:~(s;u)];
<: [ fx\<*ri,_e*>\<*su,_f*>:([e\f])~];
   [ pg 'v 'r 'a 'b 'd 'o : v ? e\f ~
        p\e r a b d (pf sz1 f\x f\s o)))];  :>
```

The above macro produces four pairs of definitions like the following, one each for the rs, ru, is, and iu versions of integer arithmetic operations:

```
   [fxrs:([rs])~];
   [ pg 'v 'r 'a 'b 'd 'o : v ? rs ~
        pr r a b d (pf sz1 sx ss o)))];
```

Another table tranforming macro is customized for sub-elements of an architecture:

```
[ 'ty 'lo 'su 'pr /s/ 'co 'bo :~ (
 [ 'ar |s| 'da :~ (<:`lo,ar,su~ [<. co, _ .> : ty ~  <. da, _ .> ]; :> )];
 pr;  bo )];
```

A table contains the offsets for shifting register and bank values as well as the size of the instructions and the value, offset, and size mask for immediate values.

```
(bin) lang,arch (fmt) (^indp,bo)
     /s/ (  sz1  ;ra;ba;rb;bb;rd;bd;sx;ss;cx;cs;ux;us;nx;ns;ics;icm   )(
x86l |s| ( be,i24;0 ;8 ;3 ;10;0 ; 0;  ;  ;  ;  ;  ;  ;  ;  ;   ;      );
x86m |s| ( be,i16;0 ;0 ;3 ;0 ;0 ; 0;  ;  ;  ;  ;  ;  ;  ;  ;   ;      );
x86s |s| ( be,i16;0 ;0 ;3 ;0 ;  ; 0;  ;  ;  ;  ;  ;  ;  ;  ;   ;      );
armm |s| ( le,i32;4 ;0 ;8 ;0 ;  ; 0;  ;  ;  ;  ;  ;  ;  ;  ;   ;      );
ppcl |s| ( be,i32;5 ;0 ;0 ;0 ;22; 0;  ;  ; 0; 0;  ;  ;  ;  ; 0 ;0XFFFF);
ppcm |s| ( be,i32;17;0 ;12;0 ;22; 0;  ;  ; 0; 0;  ;  ;  ;  ; 0 ;0XFFFF);
spkl |s| ( be,i32;25;0 ;0 ;0 ;25; 0;  ;  ; 1;13;  ;  ;  ;  ; 0 ;0X1FFF);
mmix |s| ( be,i32;8 ;0 ;0 ;0 ;16; 0; 0; 0; 1;24; 2;24; 0; 0; 0 ;0XFF  ));
```

The pack definition may be used in instruction format encoding definitions as follows for register-register-register operations:

```
`lang,indp,mch,typ~([synth:([synthetic])~]; $);
<:        `lang,arch,<*cpus,_x*>,fmt~(
 [  enc <*ri,_e>\<*su,_f> 'z 'a 'b 'd :z ? cell:fxp~
     pg fx\e\f x a b d z ];
 [  enc e\f 'z 'a 'b 'd :z ? synthetic:fxp~  ]; $ ); :>
```

An add, multiply, subtract, or divide instruction may be r-r-r or r-r-i, may be signed or unsigned, and may carry or not carry overflow from a previous instruction. The following table provides base op-code values and drives the definition of assembly language operations e.g. `acc from tmp addis 254`:

```
(bin) lang,arch ('a 'b 'd) ([mknm 'op 'f :~ 'd from 'a op\f 'b])
                /o/(    add    ;   mul     ;    sub    ;    div    )(
x86l (r;i)(s;u) |o|(  0X4801C0 ;           ;  0X4809C0 ;           );
x86m (r;i)(s;u) |o|(  0X01C0   ;           ;  0X09C0   ;           );
x86s (r;i)(s;u) |o|(  0X01C0   ;           ;  0X09C0   ;           );
armm (r;i)(s;u) |o|(  0X80e0   ;           ;  0X40e0   ;           );
ppcl ( r )(s;u) |o|( 0X7C000014; 0X7C0001E3; 0X7C000050; 0X7C0003D3);
ppcl ( i )(s;u) |o|( 0X38000000; 0X1C000000; 0X20000000;   synth   );
ppcm ( r )(s;u) |o|( 0X7C000014; 0X7C0001E3; 0X7C000030; 0X7C0003D7);
ppcm ( i )(s;u) |o|( 0X30000000; 0X1C000000; 0X20000000;   synth   );
spkl (r;i)(s;u) |o|( 0X80800000; 0X80480000; 0X80A00000; 0X81680000);
mmix (r;i)(s;u) |o|( 0X22000000; 0X1A000000; 0X26000000; 0X1E000000));
```

Addition and Subtraction of signed values are carrying or without carrying overflows or underflows of the machine word size from a previous operation. Addition and subtraction will set or clear the carrying state, usually implemented as a flag in a status register. And again for five binary logic instructions:

The above macros expand to definitions like the following:

```
`arch,mmix,asm~( [ 'd from 'a addru 'b : bin ~(enc ru 0XC8000000 a b d) ]; $ );

`arch,mmix,fmt~( [ enc ru z a b d :z ? cell:fxp~ pg fxru mmix a b d z ]; $ );
```

So the mmix-specific program code `acc from ar2 addru tmp;` results in the machine code value `0XC80000000`.

1.5.4 Register Boolean Operations

Goal — Define Register Boolean Operations. 04/17/2009 ✓

The register boolean operations will reuse some of the integer operation definitions, and introduce some more:

```
(bin) lang,arch ('a 'b 'd) ([mknm 'op 'f :~ 'd from 'a op\f 'b])
          /o/(    and   ;  xor     ; orr      )(
x86l (r;i) |o|(0X4821C0  ;0X4831C0  ;0X4889C0  );
x86m (r;i) |o|(0X21C0    ;0X31C0    ;0X89C0    );
x86s (r;i) |o|(0X21C0    ;0X31C0    ;0X89C0    );
armm (r;i) |o|(0X000000e0;0X00000020;0X000080e0);
ppcl ( r ) |o|(0X7C000038;0X7c000278;0X7c000378);
ppcl ( i ) |o|(0X70000000;0X68000000;0X60000000);
ppcm ( r ) |o|(0X7C000038;0X7c000278;0X7c000378);
ppcm ( i ) |o|(0X70000000;0X68000000;0X60000000);
spkl (r;i) |o|(0X80080000;0X80180000;0X80100000);
mmix (r;i) |o|(0XC8000000;0XC6000000;0XC0000000));
```

```
(bin) lang,arch ('a 'b 'd) ([mknm 'op 'f :~ 'd from 'a op\f 'b])
          /o/(  shl     ;  shr     ;   cmp    )(
x86l (r;i) |o|(          ;          ; 0X483BC0 );
x86m (r;i) |o|(          ;          ; 0X3BC0   );
x86s (r;i) |o|(          ;          ; 0X3BC0   );
armm (r;i) |o|(          ;          ;          );
ppcl ( r ) |o|(          ;          ;0X7C000040);
ppcl ( i ) |o|(          ;          ;0X28000000);
ppcm ( r ) |o|(          ;          ;0X7C000040);
ppcm ( i ) |o|(          ;          ;0X28000000);
spkl (r;i) |o|(0X81281000;0X81301000;     0    );
mmix (r;i) |o|(0X3A000000;0X3E000000;0X32000000));
```

The following demonstrates native assembly-language equivalents of shifting left a register by a constant value, and xor-ing a register with itself:

```
       tmp from tmp shli 8;           tmp from tmp xorr tmp;

x86l - 48 c1 e3 08  shl rbx, 8        48 33 db      xor rbx,rbx
x86m - 66 c1 e3 08  shl ebx, 8        33 db         xor ebx,ebx
x86s - c1 e3 08     shl bx,  8        33 db         xor bx,bx
armm - 01 14 a0 01  moveq r1,r1,lsl#8 00 10 a0 e3   mov r1, #0
ppcl - 7b de 45 e4  sldi r30,r30,8    3b c0 00 00   li r30, 0
ppcm - 57 de 40 2e  slwi r30,r30,8    3b c0 00 00   li r30, 0
spkl - 89 29 20 08  sll %g4,8,%g4     88 10 20 00   clr %g4
mmix - 33 10 10 08  SLU $16,$16,8     C6 10 10 10   XOR $16,$16,$16
```

1.5.5 Load Immediate to Register

Goal — Load Immediate to Register. 04/17/2009 ✓

A routine of machine code instructions may require values up to the size of a machine register to be immediately loaded into registers from the instruction stream itself. Some of the architectures require several instructions to perform this task as they lack a 'load immediate register-sized value' instruction. A general routine to load a register-sized value can be synthesized from an instruction that clears a register followed by a sequence of 'orri' and 'shli' instructions.

immediate values

```
(bin) lang,arch (cri) (b; c)
     /o/(  geti `'b `'c )(
x86l |o|(   0X80cb      );
x86m |o|(   0X80cb      );
x86s |o|(   0X80cb      );
armm |o|(   0xe3811c    );
ppcl |o|(   0X63DE00    );
ppcm |o|(   0X63DE00    );
spkl |o|(   0X881120    );
mmix |o|(   0X211010    ));
```

The native assembly-language equivalents of the above machine code are as follows:

```
             geti tmp b8 0xFF
     x86l   or bl, 0xFF
     x86m   or bl, 0xFF
     x86s   or bl, 0xFF
     armm   orr r1,r1, 0xFF
     ppcl   ori r30,r30, 0xFF
     ppcm   ori r30,r30, 0xFF
     spkl   or %g4, 0xFF, %g4
     mmix  ADD $16,$16, 0xFF
```

With the clear, byte-or-immediate, and shift instructions on the temporary register just defined, the temporary register can now be loaded with any value up to the capacity of the register. Since a sequence of instructions (a routine) will be used to perform such a multi-byte load two more macros will create synthetic instructions for loading two-, four-, and eight-byte values into the temporary register of each architecture, but first some definitions to drive the macros:

```
`arch,indp,mch~([ bsq  :~  ( b8; b16; b32 )];
                [ bszh  :~ (  h8; h16; h32 )];
                [ bszl  :~ (  l8; l16; l32 )];
                [ bssq :~ ( b16; b32; b64 )];
                [ bst :~ ( b8; b16; b32; b64 )]; $ );
```

the geti b8 instruction was created previously. The following macro creates orti instructions for sizes b16, b32, and b64 by concatenating 'geti b8' and shift instructions.

```
<: `lang,arch, <*arch,cpus,_*> ,asm~(
   [ geti <. indp,mch,bssq, _ .> 'a :bin~(
     geti <. indp,mch,bsq, _ .> a,<. indp,mch,bszh, _ .> ;
     geti <. indp,mch,bsq, _ .> a,<. indp,mch,bszl, _ .> )]; $ ) :>
```

Loading a value using the 'or' operation depends on the register being cleared first. The following macro creates new synthetic instructions that first clear the register and then get the immediate value:

```
<: `lang,arch, <*arch,cpus,_*> ,asm~(
   [ ldti <. indp,mch,bst, _ .> 'a :bin~(
     clrt; geti <. indp,mch,bst, _ .> a  )]; $ ) :>
```

The ldti synthetic assembly language instruction might be used as follows:

```
ldti 0X11223344;
```

and would, when used to create armm or x86m machine code, be implemented as the byte values and equivalent native assembly language as follows:

```
00 10 a0 e3  mov   r1, #0        db 33        xor ebx,ebx
11 1c 81 e3  orr   r1,r1, 0x11   80 cb 11     or  bl, 0x11
01 14 a0 01  moveq r1,r1,lsl#8   66 c1 e3 08  shl ebx, 8
22 1c 81 e3  orr   r1,r1, 0x22   80 cb 22     or  bl, 0x22
01 14 a0 01  moveq r1,r1,lsl#8   66 c1 e3 08  shl ebx, 8
33 1c 81 e3  orr   r1,r1, 0x33   80 cb 33     or  bl, 0x33
01 14 a0 01  moveq r1,r1,lsl#8   66 c1 e3 08  shl ebx, 8
44 1c 81 e3  orr   r1,r1, 0x44   80 cb 44     or  bl, 0x44
01 14 a0 01  moveq r1,r1,lsl#8   66 c1 e3 08  shl ebx, 8
```

There are more concise ways to load a constant value from the instruction stream into a register for these architectures but this method is general across all architectures and optimization may be later applied.

1.5.6 Jumps and Calls to absolute addresses

Goal — Define Jump and Call Instructions. 04/20/2009 ✓

The above `no operation` and `return from subroutine` definitions have values that are the same regardless of how or where they are used. For definitions which have binary values that depend on the context of their use, the system requires a way to identify what the definition depends on. In `i` (and many other programming languages) the parameters to a definition serve this function.

For example, in order to create a a jump to an absolute location in memory (as shown below) the definition needs to have passed in to it as a parameter the location in memory that is the destination of the jump.

```
[ 'ty 'lo 'ls /o/ 'co 'bo :~ (

  [ 'ar |o| 'da :~ (
     <:`lo,ar,ls ~ ([<. co, _ .> : <. ty, _ .> ~  <. da, _ .> ]; $ ); :> )];

  bo )];
```

```
        ((   bin                  );(   bin       )) lang,arch
(asm)/o/(  jabt                    ;  cabt         )(
x86l |o|(( 0X48FFE3               );( 0X48FFE3    ));
x86m |o|(( 0XFFE3                 );( 0XFFE3      ));
x86s |o|(( 0XFFE3                 );( 0XFFE3      ));
armm |o|(( 0xe1a0f001             );( 0x0fe0e1a0; 0xe1a0f001 ));
ppcl |o|(( 0x7fc803a6; 0x4e800020 );( 0x7fa802a6; 0x7fc803a6; 0x4e800020 ));
ppcm |o|(( 0x7fc803a6; 0x4e800020 );( 0x7fa802a6; 0x7fc803a6; 0x4e800020 ));
spkl |o|(( 0X89C02000             );( 0X89C3E000  ));
mmix |o|(( 0X9F101000             );( 0XBF001000  )));
```

The native assembly-language equivalents of the above machine code are as follows:

```
       jmp abs tmp   call abs tmp
x86l   jmp *%rbx     call *%rbx
x86m   jmp *%ebx     call *%ebx
x86s   jmp *%bx      call *%bx
armm   mov pc, r1    mov lr, pc
                     mov pc, r1
ppcl   mtlr r1       mflr r29
       blr           mtlr r30
                     blr
ppcm   mtlr r1       mflr r29
       blr           mtlr r30
                     blr
spkl   jmpl %g4,%r0  jmpl %g4,%r15
mmix   GO $16,$16,0  PUSHGO $0,$16,0
```

Combining the load immediate and jump (or call) absolute via temp register instructions, we can fabricate a jump or call to any memory location:

```
<: `lang,arch, <.arch,cpus,_.> ,asm~(
 [ jmpa  'a :bin~
  ( ldti <.arch,cpul,_.> ({ loc of a <.arch,cpuj,_.> }) ; jabt )];
 [ calla 'a :bin~
  ( ldti <.arch,cpul,_.> ({ loc of a <.arch,cpuj,_.> }) ; cabt )]; $ ) :>
```

parameter type functions

When definitions have parameters, type functions can be applied to the parameters. When parameters have type functions, the type functions serve as constraints which limit the applicability of that definition when it is referenced. In the case of the definitions above, the definitions are limited to apply only when the parameter refers to a value of four bytes (for the armm) or two to eight bytes (for the x86 family.)

A subroutine call is similar to a jump except that the location of execution previous to the jump is saved so that computation may continue from that point once the subroutine is done.

The above instructions allow creating loops and calling leaf subroutines as follows:

```
jmpa ( main routine );
[ sub1 ];
a leaf subroutine;
[ sub2 ];
a leaf subroutine;
[ main routine ];
calla sub1;
calla sub2;
jmpa ( main routine);
```

The Intel assembly for an absolute call is the same as for a jump, with call value replaced with jmp. The call instruction saves the return address to the stack where it will be found and used by the return from subroutine instruction. The Arm, on the other hand, uses a link register instead of the stack for subroutine return so the PC is saved to the link register before the absolute jump to the subroutine is performed, as translated below:

Type labels may be introduced and referenced between the colon and tilde sign. Type labels by default have the same scope as the enclosing definition.

```
 [ apple :  ([ fruit ]) ~ ];
 [ pear :  fruit ~ ];
 [ beef :  ([ meat ]) ~ ];
 [ pork :  meat ~ ];
```

Single quote marks denote the parameters in definition names. A type function returning true or false in the type section of a definition limits the applicability of a definition to when the function returns true.

```
 [ eats 'a :  a ?  fruit ~ [ vegetarian ] ];
 [ eats 'a :  a ?  meat ~ [ carnivore ] ];
```

i syntax, continued.

1.5.7 Relative Jump, Relative Subroutine Call

Goal — Define Relative Jump and Call Instructions. 04/21/2009 ✓

For a relative jump to another binary code location, the definition must determine the number of in bytes between the location of the branch instruction and the location being branched to. The parameter is required to be a reference to binary code. A macro computation provides the offset between the location in the image of the referenced routine and of the location in the image of the instance of the branch to be constructed. macro computation

A subroutine call also uses relative addressing. When the subroutine is called the location to return to is stored. The return instruction(s) previously defined merely instruct the CPU to resume processing at that saved location.

```
  [ mmixjop  'o : o positive ~ 0xf0 ];
  [ mmixjop  'o : o negative ~ 0xf1 ];
  [ mmixpjop 'o : o positive ~ 0xf2 ];
  [ mmixpjop 'o : o negative ~ 0xf3 ];
  [ spkljop  'o : o positive ~ 0X1080 ];
  [ spkljop  'o : o negative ~ 0X10BF ];
  [ spklpjop 'o : o positive ~ 0x4000 ];
  [ spklpjop 'o : o negative ~ 0x7FFF ];
        ((   bin               );(   bin       )) lang,arch
(asm)/o/(  jrels 'o             ;  crels 'o        )(
x86l |o|(( 0xe9; o             );( 0xe8; o         ));
x86m |o|(( 0xe9; o             );( 0xe8; o         ));
x86s |o|(( 0xe9; o             );( 0xe8; o         ));
armm |o|(( o; 0xea             );( o; 0xeb         ));
ppcl |o|(( 0x48;  o            );( 0x48; { o && 1 }));
ppcm |o|(( 0x48;  o            );( 0x48; { o && 1 }));
spkl |o|(((spkljop o ); o; nop );((spklpjop o ); o; nop )));
mmix |o|(((mmixjop o ); o      );((mmixpjop o ); o )));
```

The native assembly-language equivalents of the above machine code are as follows:

```
           jrels o         crels o
     x86l  jmp < o >      call < o >
     x86m  jmp < o >      call < o >
     x86s  jmp < o >      call < o >
     armm  b < o >        bl < o >
     ppcl  b < o >        bl < o >
     ppcm  b < o >        bl < o >
     spkl  ba < o >       call < o >
           nop            nop
     mmix  JMP < o >      PUSHJ < o >
```

A macro will create definitions which compute the offset:

```
<: `lang,arch, <.arch,cpus,_.> ,asm~(
   [ jmpr  'a :bin~([l]; jrels ({ offset a to l <.arch,cpuj,_.> }) )];
   [ callr 'a :bin~([l]; crels ({ offset a to l <.arch,cpuj,_.> }) )]; $ ) :>
```

Relative jumps are limited to offsets less (sometimes far less) than the addressable address space of the architecture. The following example code may look like the previous one but because the jumps and calls are relative the binary routine produced by the code below can be relocated without adjusting the destinations of the calls and jumps:

```
jmpr ( main routine );
[ sub1 ];
a leaf subroutine;
[ sub2 ];
a leaf subroutine;
[ main routine ];
callr sub1;
callr sub2;
jmpr ( main routine);
```

Up to this point in the description of `cap` everything is defined before it is referred to. Now, however, the definitions are using functions that have not been defined previously in the source text (e.g. 'location of a' in the above code.) This will be resolved by borrowing those routines from an already existing `cap` system.[15] The procedures used but not yet defined will be defined later in this text. A system that is implemented in terms of its own functionality is meta-circular.

meta-circular

`'r ?  't`	fails if r is not type t
`'a positive`	fails if the value of a is not positive
`'a negative`	fails if the value of a is not negative
`'a == 'b`	fails if the value of a is not the same value as b
`'a land 'b`	fails if either a or b fail
`'a lor 'b`	fails if both a and b fail
`offset 'a to 'l 's`	presents the difference between the locations of a and l in a value of size s
`'a - 'b`	presents a - b
`'a + 'b`	presents a + b
`'a / 'b`	presents a / b
`'a percent 'b`	presents a mod b
`'a band 'b`	presents a bit-wise-and b
`'a bor 'b`	presents a bit-wise-or b
`'a << 'b`	presents a shift by b
`i64le <= 'a`	presents the 64-bit value of a
`i32le <= 'a`	presents the 32-bit value of a
`i24le <= 'a`	presents the 24-bit value of a
`i16le <= 'a`	presents the 16-bit value of a
`i8 <= 'a`	presents the 8-bit value of a

Functions used but not yet defined.

[15]Ken Thompson, Reflections on trusting trust, Communications of the ACM, 27(8):761-763, August 1984

Curly-brackets enclose items which are to be instantiated in a separate image location and then to have execution passed to them so that the computation described within the curly brackets can happen at the time of construction (instantiation) of the enclosing definition.

```
 [ general 'a :  a ?  foo ~ { munge a }; do b ];
 [ general 'a :  a ?  bar ~ { chomp a }; do b ];
```

More explanation.

```
 [ general 'a :  a ?  foo ~ { munge a }; do b ];
```

i syntax, continued.

1.5.8 Register Move

Goal — Define Register Move Instructions. 04/23/2009 ✓

Binary code for moving values between registers in the x86l architecture is as simple to construct as a jump definition, but providing a separate definition for each combination of source and destination registers would be tedious. A compile time calculation based on the register number and bank number will require much less coding.

Register index and bank values are often packed in with a constant to create machine codes for operations on registers. Two macros will assist with the packing:

```
[ p2i a sa b in sz      :~{sz <== (b+(a,indx << sa))}];
[ p3i a sa b sb c in sz :~{sz <== (c+((a,indx << sa)+(b,indx << sb)))}];
[ p3b a sa b sb c in sz :~{sz <== (c+((a,bank << sa)+(b,bank << sb)))}];
```

With definitions for general purpose registers and macros to pack values operations can be defined, including loading to a register via another register:

```
( (( a ? reg ) && ( b ? reg)) ; bin )
lang,arch/t/(              'a to 'b                        )(
x86l,asm |t|((         p3b b 2 a 0 0x48 in i8;
                 0x89; p3i b 3 a 0 0xc0 in i8              ));
x86m,asm |t|((    0x89; p3i b 3 a 0 0xc0 in i8              ));
x86s,asm |t|((    0x89; p3i b 3 a 0 0xc0 in i8              ));
armm,asm |t|(( b,indx ; p2i a 4 0x00 in i8; 0xe1a0;         ));
ppcl,asm |t|((         p3i a 5 b 0 0x6000 in i16be ; 0x0000 ));
ppcm,asm |t|((         p3i a 5 b 0 0x6000 in i16be ; 0x0000 ));
spkl,asm |t|((         p3i a 25 b 15 0x8010200 in i32be     ));
mmix,asm |t|((    0xC1; b,indx; a,indx; 0x00                )));
```

The native assembly-language equivalents of the above machine code are as follows:

```
        a to b
x86l  movq %< b >, %< a >
x86m  movd %< b >, %< a >
x86s  mov  %< b >, %< a >
armm  mov  < b >, < a >
ppcl  ori  < b >, < a >, 0
ppcm  ori  < b >, < a >, 0
spkl  or %< a >, 0, %< b >
mmix  OR $< b >,$< a >,0
```

The above instructions can be used in the following manner, but only for the registers that are defined for the architecture the routine is being crafted for. An x86 processor must not be asked to move data to or from r31, for example.

```
r1 to r3;
acc to tmp;
tmp to r5;
```

1.5.9 Register Load / Store

Goal — Define Register To Memory Instructions. 05/05/2009 ✓

The load and store instructions for each architecture share some common characteristics. A couple of tables collect the particulars of how each architecture encodes its instructions, a 'build-load-store' definition describes a way to construct a load or store instruction from the table and then a macro drives the creation of the instructions for each architecture.

The first table provides the base binary value of the machine instruction to load or store an 8, 16, 32, or 64-bit value for each architecture.

```
<: `lang,arch,<. lang,indp,cpus, _ .> ~([ ldst:~ $ ]); :>

        ( s==b8  ; s==b8  ; s==b16;  s==b16 ; s==b32 ; s==b32 ; s==b64 ; s==b64 )
lang,arch (ldst)
     /o/( st 's  ; ld 's  ; st 's  ; ld 's  ; st 's  ; ld 's  ; st 's  ; ld 's  )(
x86l |o|(0X904088;0X90408A;0X664189;0X66418B;0X674889;0X67488B;0X904889;0X90488B);
x86m |o|( 0X9088 ; 0X908A ; 0X6689 ; 0X668B ; 0X6789 ; 0X678B ;  ()    ;   ()   );
x86s |o|( 0X90   ; 0X90   ; 0X89   ; 0X8B   ;   ()   ;   ()   ;  ()    ;   ()   );
armm |o|(0xe5c000;0xe5d000;0xe1c000;0xe1d000;0xe50000;0xe51000;  ()    ;   ()   );
ppcl |o|( 0XA000 ; 0X8800 ; 0XE800 ; 0X8000 ; 0XB000 ; 0X9800 ; 0XF800 ; 0X9000 );
ppcm |o|( 0XA000 ; 0X8800 ; 0XE800 ; 0X8000 ; 0XB000 ; 0X9800 ;  ()    ;   ()   );
spkl |o|(0XC02120;0XC00120;0XC03020;0XC01020;0XC02020;0XC00020;0XC03820;0XC01820);
mmix |o|(0XA30000;0X830000;0XA70000;0X870000;0XAB0000;0X8B0000;0XAF0000;0X8F0000));
```

The second table contains the amounts by which to shift the index and bank values of registers for their placement in the machine instruction for their role as the source/destination register or the register holding the memory location:

```
        (dat;dat;dat;dat;dat;dat ;dat;dat;dat; dat ; dat ) lang,arch
(ldst)/o/(dp; ar; br; ab; bb;cpad; sp; sn;ns ;lopsz;ofsz )(
x86l  |o|( 1;  0;  3;  8; 10;0x40;  0;  0; 0 ;i24le; i8  );
x86m  |o|( 1;  0;  3;  0;  0;0x40;  0;  0; 0 ;i16le; i8  );
x86s  |o|( 1;  0;  3;  0;  0;0x40;  0;  0; 0 ; i8  ; i8  );
armm  |o|( 0;  4;  8;  0;  0; () ;  1;  0;15 ;i24le; i8  );
ppcl  |o|( 1;  5;  0;  0;  0; () ;  0;  0; 0 ;i16be;i16be);
ppcm  |o|( 1;  5;  0;  0;  0; () ;  0;  0; 0 ;i16be;i16be);
spkl  |o|( 1; 17;  6;  0;  0; () ;  0;  0; 0 ;i24be;i24be);
mmix  |o|( 1;  8;  0;  0;  0; () ;  0;  0; 0 ;i24le; i8  ));
```

The columns in the above table have the following meanings:

```
dp - data position
ar - 'a' register index shift by    br - 'b' register index shift by
ab - 'a' register bank shift by     bb - 'b' register bank shift by
cr - 'c' register index shift by    lopsz  - load/store operand size
```

A load or store instruction for an architecture is made by packing a sequence of binary values with each of the above characteristics, each one appropriately coded and shifted. Packing items can be done with the following definition, which is also aliased for easier expansion:

```
^([ buildldst 'cpu 'lsz 'op 'a via 'b offset 'c :~ $ ])~bl;
```

A definition for pack is made privately within buildldst. The pack definition shifts left the value it receives and adds that to whatever it gets in its third parameter. All values are assumed to fit in a ’lopsz’ value as defined by the ’cpu’ architecture received from the containing buldldst.

```
`bl~( @[ negflag 'c : positive c ~ cpu,ldst,sp ];
      @[ negflag 'c : negative c ~ cpu,ldst,sn ];
      @[ pack 'va 'sl 'vb :~ { cpu,ldst,lopsz <== (( va << sl ) + vb )} ]; $ );
```

That definition is used to pack the opcode received (which is not shifted, and which specifies already whether the operation is a load or a store and whether the value moved is 8, 16, 32, or 64 bits in size) and the next thing,

```
`bl~(  pack op 0 ($) );
```

which is the index of the register to receive or supply the value. After that is packed the bank for that register.

```
`bl~(  pack a,indx cpu,ldst,ar ($) );
`bl~(  pack a,bank cpu,ldst,ab ($) );
```

Same for the register that contains the memory location:

```
`bl~(  pack b,indx cpu,ldst,br ($) );
`bl~(  pack b,bank cpu,ldst,bb ($) );
```

The negative or positive offset flag is similarly packed depending on whether the offset is positive or negative:

```
`bl~(  pack (negflag c) cpu,ldst,ns ($) );
```

The first four of the following definitions allow for a definition to tailor its results on whether or not the offset constant comes at the begin or the end of the instruction, and whether or not the absolute value of the offset should be stored. The last two drive the macro to create load and store instructions.

```
[ pos 'dp 'c 'b 'p : dp == 0 ~ ( p; c; b )];
[ pos 'dp 'c 'b 'p : dp == 1 ~ ( b; p; c )];
[ absc 'az 'dp 'c : dp == 0 ~ { az <== c } ];
[ absc 'az 'dp 'c : dp == 1 ~ { az <== ( abs c )}];
[ altlsn :~( load ; store )];
[ altlsd :~( to ; from )];
[ altls :~( ld ; st )];
```

A macro will create load/store instructions for each architecture using the pack definition.

```
<: `lang,arch, <*arch,cpus,_a*> ,asm~(
[ <. altlsn,_ .> 'lsz <<: altlsd,_ :>> 'r via 'm offset 'c :~
  pos `arch,a,ldst,dp ( absc  `arch,a,ldst,ofsz `arch,a,ldst,sp c )
  ( buildldst a lsz `arch,a,ldst,(<<:altls,_:>> lsz ) r via m offset c )
    `arch,a,ldst,cpad ]; $) :>
```

The above instructions can be used in the following manner, but only for the registers that are defined for the architecture the routine is being crafted for. An x86 processor must not be asked to load data to r31, for example.

```
    load b32 to r3 via r5 offset 8;
    store b64 from acc via tmp offset -64;
    load b8 to acc via frm offset 0;
```

1.5.10 Integer Register Operations

Goal — Define Integer Register Operations. partial

There are a common set of intger register operations that most architectures support. As with register loads and stores, these operations are composed in a regular way. Some tables will collate specific encodings, construction definitions will facilitate composition, and a macro will create the assembly operation definitions.

The first table provides the base binary value of the machine instruction for a particular operation.

```
     `lang,indp,mch,typ~([:([op])~]; $);

         ( op; op; op; op; op )  (lang,arch)
(ops)/o/(   add    ;   adc    ;   sub    ;    sbc   ;   cmp    )(
x86l |o|( 0X4801C0 ; 0X4811C0 ; 0X4809C0 ; 0X4819C0 ; 0X483BC0 );
x86m |o|( 0X01C0   ; 0X11C0   ; 0X09C0   ; 0X19C0   ; 0X3BC0   );
x86s |o|( 0X01C0   ; 0X11C0   ; 0X09C0   ; 0X19C0   ; 0X3BC0   );
armm |o|( 0xe080   ; 0x       ; 0xe040   ; 0xe0c0   ;          );
ppcl |o|( 0x       ;0x        ;          ;          ;0x        );
ppcm |o|( 0x       ;0x        ;0x        ;0x        ;0x        );
spkl |o|( 0x       ;0x        ;0x        ;0x        ;0x        );
mmix |o|(0X20000000;    ()    ;0X24000000;    ()    ;          ));

         ( op; op; op; op; op )  (lang,arch)
(ops)/o/(    and    ;    xor   ; orr      ;    shl   ;    shr   )(
x86l |o|( 0X4821C0  ;0X4831C0  ;0X4889C0  ;0x        ;0x        );
x86m |o|( 0X21C0    ;0X31C0    ;0X89C0    ;0x        ;0x        );
x86s |o|( 0X21C0    ;0X31C0    ;0X89C0    ;0x        ;0x        );
armm |o|( 0xe000    ;0x2000    ;0xe080    ;0x        ;0x        );
ppcl |o|( 0x        ;0x        ;0x        ;0x        ;0x        );
ppcm |o|( 0x        ;0x        ;0x        ;0x        ;0x        );
spkl |o|( 0x        ;0x        ;0x        ;0x        ;0x        );
mmix |o|( 0XC8000000;0XC6000000;0xC0000000;0x        ;0X24000000));
```

The second table contains the amounts by which to shift the index and bank values of the two source registers and the destination register. Some architectures (e.g. the x86) require that the second source register be the same as the destination register.

```
           (dat;dat;dat;dat;dat;dat ;dat;dat;dat; dat ; dat ) lang,arch
   (iop) /o/( ar; br; cr; ab; bb; cb; h3;   ;   ;iopsz;     )(
   x86l  |o|(  0;  3;  0;  8; 10;  0;  0;   ;   ;i24be;     );
   x86m  |o|(  0;  3;  0;  0;  0;  0;  0;   ;   ;i16be;     );
   x86s  |o|(  0;  3;  0;  0;  0;  0;  0;   ;   ;i16be;     );
   armm  |o|(   ;   ;   ;  0;  0;  0;  1;   ;   ;     ;     );
   ppcl  |o|(   ;   ;   ;  0;  0;  0;  1;   ;   ;     ;     );
   ppcm  |o|(   ;   ;   ;  0;  0;  0;  1;   ;   ;     ;     );
   spkl  |o|(   ;   ;   ;  0;  0;  0;  1;   ;   ;     ;     );
   mmix  |o|( 16;  8;  0;  0;  0;  0;  1;   ;   ;i32be;     ));
```

An integer operation for an architecture is made by packing a sequence of binary values with each of the above characteristics, each one appropriately coded and shifted. Packing items can be done with the following definition, which is also aliased for easier expansion:

```
^([ buildintop 'cpu 'lsz 'op 'a via 'b offset 'c :~ $ ])~bi;
```

A definition for pack is made privately within buildintop. The pack definition shifts left the value it receives and adds that to whatever it gets in its third parameter. All values are assumed to fit in a 'lopsz' value as defined by the 'cpu' architecture received from the containing buldldst.

```
`bi~( @[ pack 'va 'sl 'vb :~ { cpu,ldst,lopsz <== (( va << sl ) + vb )} ]; $ );
```

That definition is used to pack the opcode received (which is not shifted, and which specifies already whether the operation is a load or a store and whether the value moved is 8, 16, 32, or 64 bits in size) and the next thing,

```
`bi~(  pack op 0 ($) );
```

which is the index of the register to receive or supply the value. After that is packed the bank for that register.

```
`bi~(  pack a,indx cpu,ldst,ar ($) );
`bi~(  pack a,bank cpu,ldst,ab ($) );
`bl~(  pack b,indx cpu,ldst,br ($) );
`bl~(  pack b,bank cpu,ldst,bb ($) );
`bl~(  pack b,indx cpu,ldst,br ($) );
`bl~(  pack b,bank cpu,ldst,bb ($) );
```

With the above table the instructions can be defined:

```
  lang,arch/t/(  'a <-- 'a  'o 'b                               )(
  armm,asm |t|((b, {i8 <== ( o,ord * 16 )};
                {i8 <== ( a,ord + o,b2 )}; o,b1                   ));
```

The above instructions can be used in the following manner, but only for the registers that are defined for the architecture the routine is being crafted for. An x86 processor must not be asked to xor r31 and r24, for example.

```
r7  and r9  to r5;
acc xor tmp to acc;
r1  cmp 0   to r2;
acc sub 4   to acc;
r31 xor r31 to r31;
```

1.5.11 Stack Oprerations

Goal — Define Stack Operation Instructions. 01/30/2009 ✓

And the stack is generally considered a good idea:

```
   `lang,indp,mch,typ~([:([sasmop])~]; $);

( sasmop )
lang,arch    /t/(  pop  ;  push )(
x86l,ops     |t|( 0x58  ; 0x50  );
x86m,ops     |t|( 0x58  ; 0x50  );
x86s,ops     |t|( 0x58  ; 0x50  );
armm,ops     |t|( 0xee8bd; 0xe92d);
ppcl,ops     |t|( 0x??  ; 0x??  );
ppcm,ops     |t|( 0x??  ; 0x??  );
spkl,ops     |t|( 0x??  ; 0x??  );
mmix,ops     |t|( 0x????; 0x????));

((( a ? reg )&&( o ? sasmop )) ; bin )
lang,arch/t/(  'o register 'a                                    )(
x86l,asm |t|(( {i8 <== ( 0x40 +  a,bank ) };{i8 <== ( o +  a,ord ) }));
x86m,asm |t|(( {i8 <== ( o +  a,ord ) }                             ));
x86s,asm |t|(( {i8 <== ( o +  a,ord ) }                             ));
armm,asm |t|(( {i16le <== ( 0x01 << a,ord ); o  }                   ));
ppcl,asm |t|(( {i8 <== ( 0x?? +  a,bank ) };{i8 <== ( o +  a,ord ) }));
ppcm,asm |t|(( {i8 <== ( ??         ) }                             ));
spkl,asm |t|(( {i8 <== ( ??         ) }                             ));
mmix,asm |t|(( {i16le <== ( 0x?? << a,ord ); o  }                   )));
```

1.5.12 Load Register with Constant

Goal — Define Load Constant Instructions. unmet

The previous two assembly language definitions redirect the flow of execution for the binary code they are instantiated within. The next load constant values into registers:

```
`cap,env,lang,arch,gen ~  (

   instruction ( load 'a with 'b )
    with (( a ? reg )&&( b # 8 )) over (
   for x86l is ( {i8 <== ( 0x48 +  a,bank ) };
                   {i8 <== ( 0xB8 +  a,ord ) }; b  ));
 $ );
```

Of the defined architectures so far, only the x86l has 64-bit registers to receive a 64-bit constant.

```
`cap,env,lang,arch,gen ~  (

   instruction ( load 'a with 'b )
    with (( a ? reg )&&( b # 4 )) over (
   for x86l is (  );
   for x86m is (  );
   for armm is (  ));

   instruction ( load 'a with 'b )
    with (( a ? reg )&&( b # 2 )) over (
   for x86l is (  );
   for x86m is (  );
   for x86s is (  );
   for armm is (  ));

   instruction ( load 'a with 'b )
    with (( a ? reg )&&( b # 1 )) over (
   for x86l is (  );
   for x86m is (  );
   for x86s is (  );
   for armm is (  ));

 $ );
```

1.5.13 Compare Register Values

Goal — Define Compare Register Values. unmet

Routines need to compare things.

```
`cap,env,lang,arch,gen ~  (

   instruction ( compare 'a with 'b )
    with (( a ? reg )&&( b # reg )) over (
   for x86l is (   );
   for x86m is (  );
   for armm is (  ));
 $ );
```

1.5.14 Conditional Relative Branch

Goal — Define Conditional Branch Instructions. unmet

Needed for if-then routines.

```
`cap,env,lang,arch,gen ~  (

   instruction ( branch to 'a )
    with ( a ? bin ) over (
   for x86l is (  );
   for x86m is (  );
   for x86s is (  );
   for armm is (  ));
 $ );
```

1.6 The Application Binary Interface

Goal — Declare a CAP ABI. partial

Each of the previous assembly language definitions encode one machine language instruction or the equivalent of one, but one instruction does not a (useful) program make. The next step up in expressiveness and organizational complexity of programming is the combining of instructions into routines and subroutines.

routine, subroutine

A routine is a collection of computational steps (e.g. machine instructions.) In a subroutine the programmer has extracted commonly used sequences of instructions and implemented them in separate routines that may be 'called' where they otherwise might have been duplicated. This early innovation in computer science[16] drastically reduces the size of the resulting program and makes changing and improving the software easier as an error in a routine need only be modified in one place.

application binary interface, ABI

Subroutines require a convention, adhered to by the caller(s) and the subroutine that is called. Agreed upon are the usage of registers (which ones hold the parameters, and which will contain the result? which registers may the subroutine over-write and which must it preserve?) and the arrangement of items on the stack. Such a convention for a particular architecture is its Application Binary Interface, or ABI.

The ABI will be developed in the following sub-sections:

1.6.1 *	Register Assignments.	36 — 4/4
1.6.2 *	Calling Convention.	37 — 2/2
1.6.3 *	Construction Stack Frame.	38 — 2/2
1.5.4 *	Construction Routines.	38 — 2/2

Sub-projects (sub-sections) for Section 1.5 The Application Binary Interface.

1.6.1 Register Assignments

Goal — Declare Register Assignments. 01/19/2009 ✓

Processors differ in how the capacity and number of their registers and in how the registers may be used. Some machine language instructions may only work with specific registers. Some registers are dedicated to providing hardware support for features such as a stack or the fast processing of interrupts.

In addition to hardware restrictions, programmers may wish to reserve certain registers for certain purposes. Having one register that always contains the result of the previous calculation makes programming easier. Each architecture will have its own register usage. Register usage in cap for several architectures are as follows:

The registers in these architectures vary in size. In all of them a register is reserved as the stack pointer; the ARM also exposes the program counter as a register and maintains a link register. Each reserves several registers for arguments to subroutines. The contents of those registers may also be over-

[16] Wilkes, M. V.; Wheeler, D. J., Gill, S. (1951). Preparation of Programs for an Electronic Digital Computer. Addison-Wesley

written and used by the called routine. The result of a computation should be returned in a register used for accumulating results. Certain registers are saved by the called routine and returned unchanged to the calling routine – these are indicated by an asterisk in the above chart.

1.6.2 Calling Convention

Goal — Describe Calling Convention. 01/19/2009 ✓

Two conventions required for subroutines are the calling convention and the stack layout. The calling convention includes which registers are saved by the caller and which are saved (or left unmodified) by the callee. A stack (or a more general heap) is required because a processor has only so many registers and yet subroutines may call other subroutines an indefinite number of times. The stack layout includes such elements as the return address for the subroutine, the previous frame pointer and arguments that could not fit in registers.

The arrangement of values on the stack for each architecture is as follows:

64-bit Position	32-bit Position	16-bit Position	Contents	Frame
(f) + 8n + 16	(f) + 4n + 8	(f) + 2n + 4	argument # n	
			...	Previous
(f) + 16	(f) + 8	(f) + 4	argument # 0	
(f) + 8	(f) + 4	(f) + 2	success address	
(f)	(f)	(f)	success frame	Current
(f) - 8	(f) - 4	(f) - 2	fail address	
(f) - 16	(f) - 8	(f) - 4	fail frame	
(f) - 24	(f) - 12	(f) - 6	Local # 0	
			...	
(f) - 8n - 24	(f) - 4n - 12	(f) - 2n - 6	Local # n	
			...	
(sp)	(sp)	(sp)	top of stack	

```
`cap,env,lang,arch,gen ~  (

     ()
     lang,arch  /t/( "arg0"; "suca"; "sucf"; "faila"; "failf"; "loc0" )(
     x86l     |t|(  0x10 ;  0X08 ;  0x00 ;   0xF? ;   0XF? ;  0xF?  );
     x86m     |t|(  0x08 ;  0X04 ;  0x00 ;   0xF? ;   0XF? ;  0xF?  );
     x86s     |t|(  0x04 ;  0X02 ;  0x00 ;   0xF? ;   0XF? ;  0xF?  );
     armm      |t|(  0x08 ;  0X04 ;  0x00 ;   0xF? ;   0XF? ;  0xF?  );
     ppcl     |t|(  0x?? ;  0X?? ;  0x?? ;   0x?? ;   0X?? ;  0x??  );
     ppcm     |t|(  0x?? ;  0X?? ;  0x?? ;   0x?? ;   0X?? ;  0x??  );
     spkl    |t|(  0x?? ;  0X?? ;  0x?? ;   0x?? ;   0X?? ;  0x??  );
     mmix       |t|(  0x?? ;  0X?? ;  0x?? ;   0x?? ;   0X?? ;  0x?? ));
  $ );
```

1.6.3 Construction Stack Frame

Goal — Define the Construction Stack Frame. unmet

The current stack frame contains local variables need by construction routines:

#	Local Variable in Current Frame
0	This Definition AST Reference
1	This Definition AST End + 1
2	Parent Definition AST Reference
3	Previous Definition AST Reference
4	This Definition Value End + 1

```
`cap,env,lang,arch,gen ~  (

    ()
    lang,arch  /t/( "thisd"; "thise"; "pard"; "pred" )(
    x86l     |t|(  0xF? ;   0xF? ;   0XF? ;  0xF?  );
    x86m     |t|(  0xF? ;   0xF? ;   0XF? ;  0xF?  );
    x86s     |t|(  0xF? ;   0xF? ;   0XF? ;  0xF?  );
    armm      |t|(  0xF? ;   0xF? ;   0XF? ;  0xF?  );
    ppcl     |t|(  0xF? ;   0xF? ;   0XF? ;  0xF?  );
    ppcm     |t|(  0xF? ;   0xF? ;   0XF? ;  0xF?  );
    spkl    |t|(  0xF? ;   0xF? ;   0XF? ;  0xF?  );
    mmix       |t|(  0xF? ;   0xF? ;   0XF? ;  0xF?  ));
  $ );
```

construction routines execute in the context of the construction stack frame. stack frame

1.6.4 Construction Routines

Goal — Define Construction Routines. unmet

The 'load from memory' instruction is all the system needs to define the first 'inherited' routine in xgenc. By convention in our system on the x86l when building a definition the next free value-space location is kept in the first local variable in the current stack frame (at an offset to the EBP of -8.) The following definition merely retrieves that value. stack frame

a type for values in the accumulator:

```
    `lang,indp,mch,typ~([:([accval])~]; $);

  ((( a ? cell )&&( b ? accval )) ; bin )
  lang,arch  /t/(  'a <= 'b                                  )(
    <:: <<< arch,cpus,_r >>> ,exo
      |t|(( secr <-- mem ref via framep ofset by thise;
            mem ref via secr <-- a in accr;
            secr <-- secr + 1;
            mem ref via framep offset by thise <-- secr ));  ::>);
```

Get distance:

```
   `lang,indp,mch,typ~([:([binref])~]; $);

((( a ? binref )&&( b ? binref )) ; bin )
lang,arch  /t/( offset 'a to 'b                              )(
  <:: <<< arch,cpus,_r >>> ,exo
     |t|((                                               )); ::>);
```

Conjunction, addition, multiplication, left shift:

```
   `lang,indp,mch,typ~([:([accmop])~]; $);
   `lang,indp,mch~([^ accmops :~ $]; $);

( accmop )
lang,indp,mch  /t/(  "&&"; "+"; "-"; "*"; "<<" )(
accmops        |t|(   && ; +  ;  - ;  * ;  <<  ));


((( a ? accval )&&( b ? accmop )&&( c ? accval )) ; accval; bin )
lang,arch  /t/( 'a 'b 'c                                  )(
  <::  <<< arch,cpus,_r >>> ,exo
     |t|((       c;
                 stack <-- acc;
                 a;
                 tmp <-- stack;
                 acc <-- acc b tmp ];
                                         )); ::>);
```

Get Location:

```
((( a ? binref )) ; bin )
lang,arch  /t/( addr of 'a                                 )(
  <:: <<< arch,cpus,_r >>> ,exo
     |t|((                                               )); ::>);
```

check size:

```
((( a ? binref )&&( n ? cell )) ; bin )
lang,arch  /t/( 'a # 'n                                   )(
  <:: <<< arch,cpus,_r >>> ,exo
     |t|((                                               )); ::>);
```

type check:

```
  ((( a ? binref )&&( b ? cell )) ; bin )
  lang,arch  /t/( 'a ? 'b                              )(
    <:: <<< arch,cpus,_r >>> ,exo
       |t|((                                             )); ::>);
```

The above code uses an accumulator. accumulator

The value-space address of parameters of the current definition can be found offset -12 of EBP or less by 16 * (parameter number in order - 1).

These xgenc routines depend on being executed in the stack frame context of a procedure that has established local variables four bytes each in size in the following order: start of value-space for currently-being-defined item, next-free-location in value-space of currently-being-defined item, a list of starts and one-past-ends and the types and flags (total 16 bytes each) of the value spaces of the parameters of the currently-being-defined item. The values for these local variables will be initialized before the xgenc routines are executed.

1.7 Functions

Goal — Implement Functions. partial

Most systems implementors prefer to code at a higher level of abstraction than assembly language. Also it is common to create an architecture-independent base from which to implement most of a system, leaving only a small set of low-level features to be (re-)implemented in assembly for each architecture the system is ported to. abstraction, functions

1.6a Provide an x86l base for architecture-independent function definition.
— not yet demonstrated

1.6b Provide an ARM base for architecture-independent function definition.
— not yet demonstrated

Goals, continued.

```
`cap,env,lang,indp ~  (
      [ base :~ $ ];
  $ );
```

`cap` requires architecture-dependent definitions to support the architecture-independent definitions.

```
`cap,env,lang,arch,x86l ~  (
      [ base :~ $ ];
  $ );
```

The architecture independent base has a core language definition, common libraries, and repositories of optional libraries.

```
`cap,env,lang,indp,base ~  (
      [ core :~ $ ];
  $ );

`cap,env,lang,indp,base,core ~  (
      ^cap,env,lang,arch,base;
  $ );
```

Control flow can be built from assembly language and a convention for representing true or false as the result of a calculation.

```
`cap,env,lang,arch,base ~  (
       [ true : '() ([^ bool :~ acvl ]) :bin~ zero eax ];
       [ false : '() bool :bin~ zero eax ; eax <-- eax - 0x01 ];
      ?[ if 'a then 'b else 'c
           : 'a bool
           :''b bin
           :''c bin
           : '() 'b
           : '() 'c
           : bin ~
         !a ;
         test eax ;
         jne e ;
         [ d :~ !b ;
            jmpr f ];
         [ e :~ !c ];
         [ f ] ];
      [ if 'a then 'b :~ if a then b else ([]) ];
  $ );
```

The question mark and exclamation marks create a lexical shelter for the sub-declarations of this routine (i.e. e and f) from the items referenced (a, b, and c) in order to preclude inadvertent variable capture.

Also note the forward references to d and e (they are referred to before they are defined. The parser has to perform some tricks to allow this.

Control flow in computer languages often use binary operations on true or false values.

```
`cap,env,lang,indp,base,core ~  (
      [ not 'a    : 'a bool:'() bool:bin~
                     if a then false else true ];
      [ 'a and 'b : 'a bool:'b bool:'() bool:bin~
                     if a then b else false ];
      [ 'a or 'b  : 'a bool:'b bool:'() bool:bin~
                     if a then true else b ];
      [ 'a xor 'b : 'a bool:'b bool:'() bool:bin~
                     if a then (not b) else b ];
  $ );
```

One useful feature of most modern computer languages is the function call, wherein the machine code for a routine is constructed in memory just once, saving space and allowing for recursion. 'callers' of the function push their parameters on the stack and then perform a transfer of execution to the location of that routine. That routine returns execution back to whichever caller when it is done.

```
`cap,env,lang,arch,base ~  (
      ?[ makeclosurefor 'a :bin~
       ];
      ?[ makeconparamframefor 'a :bin~
       ];
      ?[ removeconparamframe :bin~
       ];
      ?[ restorestack :bin~
       ];
      ?[ end :bin~ ret 0x00
       ];
  $ );
```

With the basic machinery we can build a function call.

```
`cap,env,lang,arch,base ~  (

      ?[ function 'a 'b 'c :bin~
          @[ d :bin~ { makeconparamframefor !a }  ;
                  !c ;
                  end ;
                  { removeconparamframe } ] ;
          [{^} {a} : {b} :bin~
            makeclosurefor !a ;
            call ;
            ? @ d ;
            restorestack
          ]
       ];

      ?[ local 'a 'b :bin~

       ];


      ?[ end with 'a :bin~
            restorestack;
            makeclosurefor !a ;
            jumptoclosuredbinstart ;
       ];

  $ );
```

Once we have functions we can use them.

```
`cap,env,lang,indp ~  (
      [ libs :~ $ ];
      [ sntx :~ $ ];
  $ );


`cap,env,lang,indp,libs ~  (
      [ math :~ $ ];
      [ grph :~ $ ];
      [ text :~ $ ];
      [ inet :~ $ ];
  $ );


`cap,env,lang,indp,libs,math ~  (

       function (fibb 'n) ('n num) (
          if ( n <= 2 )
          then 1
          else ( ((fibb (n-1))-(fibb(n-2)) )
       );


       function ( f 'a 'b 'n ) ('a num : 'b num : 'n num) (
          if ( n <= 1 )
          then ( end with b )
          else ( end with ( f b (a+b) (n-1) ));
        );

       function ( fib 'n ) ('n num) ( f 0 1 n )

       function ( charcount 's ) ('s string : '() num ) (
         local ( 'c num : 'd string ) (


         )
       );


  $ );
```

1.8 Objects

Goal — Implement Objects. unmet

objects Many systems also implement objects.

With the basic machinery we can build an object.

```
`cap,env,lang,arch,base ~  (

     `lang,indp,mch,typ~([:([obj])~]; $);
     `lang,indp,mch,typ~([:([rec])~]; $);

      ?[ object 'a 'b is 'c :obj~
      ?[ part 'd 'f is 'g :obj~

       ];
       ];

      ?[ object 'a from 'e 'c :bin~

       ];

  $ );
```

inheritance Object systems often make use of inheritance.

Chapter 2

The Syntax of Syntax

Machine Code

Once upon a time programmers crafted their programs by wiring circuits to perform logic operations. This was laborious. The stored program computer[1] made it easier to reconfigure a machine for a new computation–and introduced a machine code. Thus was introduced the new tedium of machine code programming wherein the binary numbers corresponding to machine operations are calculated by hand.

Assembly Language

A mere year later[2] an enterprising chap got a computer to assemble machine code for him. The simple assembly language he created mapped letters of the alphabet to machine operations. Subsequent assemblers allow a bit more flexibility but maintain the direct correspondence of assembly statements to machine instructions.

Low Level Language

The programmer attempting to express formulas, procedures and algorithms directly in assembly language soon found him or her self saddled with unweildy programs that were difficult to debug, integrate, and reason about due to the low conceptual and semantic level of the assembly languages. To make matters worse, the programs were seldom portable from one machine architecture to another.

[1]Williams, Kilburn (1948). Electronic Digital Computers. Nature, Vol 162, p. 487

[2]Campbell-Kelly (1998). Programming the EDSAC. Annals of the History of Computing, IEEE, Vol. 20 Issue 4 p. 46-67

High Level Language

To compensate, software engineers built compilers[3] and interpreters[4] to further abstract away from the machine and to allow for portable programs that would enable the programmer to move beyond simple instructions to higher level concepts such as procedures, functions, objects, modules, first-class and higher-order 'things' of every kind.[5]

Domain Specific Language

Not only did the variety of general purpose languages multiply as new machines and new techniques of program organization arrived, many programmers found that designing a specific language[6] for a problem or domain made their task easier.

Computer scientists learned (and are still learning) quite a bit from all those languages about how to design, specify, and implement a computer language. This chapter will use just a tiny fraction of the accumulated knowledge in order to bootstrap a general computer language implementation system.

2.1 Automatically generate language compilers from BNF-style syntax + semantics. 1/20/2013 ✓

Goals, continued.

In order to accomplish the goal, this chapter is organized as follows:

Section:		Page — req
2	(Introduction)	59 — 1/1
2.1 *	Syntax and the Backus-Naur Form.	61 — 0/0
2.2 *	Common ibnf Rules.	62 — 1/1
2.3 *	A Calculator Example.	65 — 1/1
2.4 *	Declaring ibnf in ibnf/six.	69 — 1/1
2.5 *	Implementing six in ibnf/six.	75 — 1/1
2.6 *	A toy compiler in ibnf/six	81 — 1/1
2.6 *	The i Language in ibnf/six.	72 — 0/1
2.6 *	Other Programming Languages.	80 — 0/9

Sub-projects (sections) for Chapter 2: The Syntax of Syntax.

[3] J.W. Backus, H. Herrick and I. Ziller. (1954) Preliminary Report : Specifications for the IBM Mathematical FORmula TRANSlating System, FORTRAN. Programming Research Group, Applied Science Division, International Business Machines Corporation

[4] J. McCarthy. (1959) Recursive Functions of Symbolic Expressions and Their Computation by Machine. Memo 8, Artificial Intelligence Project, RLE and MIT Computation Center

[5] Many of which we introduced in the previous chapter.

[6] J. Bentley. (1986) Programming Pearls: Little Languages. Communications of the ACM, Vol. 29, No. 8, pp. 711-721

Just as with the previous chapter, this is a literate program[7] wherein regular descriptive text (such as this paragraph) is mixed with computer evaluated program text in a `monospace font`.

2.1 Syntax and the Backus-Naur Form

Early on language designers realized they needed a formal way of describing what was a validly declared (as opposed to correct or bug[8]-free) program in a particular language and what was not.

John Backus (and Peter Naur) gave us that formal method of describing[9] computer language syntax. The following example shows how Backus-Naur Form describes a syntax with simple textual grammar made of lowercase letters, numbers and keyboard-accessible symbols:

BNF, Syntax

```
<us phone number> ::= ( <three digits> ) <three digits> - <four digits>
<three digits>    ::=  <digit> <digit> <digit>
<four digits>     ::=  <digit> <digit> <digit> <digit>
<digit>           ::==  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

`(111)222-3333` satisfies the above grammer, while `(44)5555-666` and `(777) 888 - 9999` (note the spaces) do not.

This chapter will use a variant of BNF called `ibnf`, formally defined in section 2.4.

In the over-arching project of which this chapter is a part, syntax definitions (starting with an ibnf description of commonly-used character sequences) will be placed in the `indp` branch of the tree:[10]

```
`cap,env,lang,indp ~ ( [ sntx :~  ( [ common-chr :~( [ ibnf :~ $ ]; $ )]; $ )]; $);
```

[7]Donald E. Knuth, Literate Programming, Stanford, California: Center for the Study of Language and Information, 1992, CSLI Lecture Notes, no. 27

[8]Hopper, (1947) U.S. Naval Historical Center Online Library Photograph NH 96566-KN

[9]J. W. Backus, (1959) The syntax and semantics of the proposed international algebraic language of the Zuerich ACM-GRAMM conference, ICIP Paris

[10]This line of i code declares a location where our syntax definitions may reside. See chapter 1 for an introduction to i code.

2.2 Common ibnf Rules

rules

An **ibnf** syntax consists of a sequence of rules. Many complete syntax definitions (often for different languages) may share a common set of rules. These rules can be collected and defined just once, then composed in a modular[11] manner. This section will define some commonly-used character sequences applicable to many grammars.

2.2 Collect common ibnf rules.	12/18/2010 ✓

Goals, continued.

The program text in this section, highligted in blue, is declared to be stored contiguously as `sntx,common-chr,ibnf` but when extracted from this document may named `common-chr.ibnf`.

digits, alternatives rule

Common definitions such as identifying digits in **ibnf** are substantially the same as they would be in BNF, with alternatives separated by a vertical bar (`'|'`) and with the `'?'` indicating that the this is an alternatives rule, in which the semantic result is the semantic result of the successful alternative. The semantic result of matching a character is the character itself.

```
dgt ? '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' ;
```

uppercase

The uppercase alphabet is declared the same way.

```
upr ? 'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|
      'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z' ;
```

lowercase

Likewise lowercase letters in the English alphabet:

```
lwr ? 'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|
      'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z' ;
```

alphabetic

Alphabetics are uppercase or lowercase.

[11]D.L. Parnas, (1972). On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM Vol. 15, No. 12 pp.1053 - 1058

```
alp   ? upr | lwr ;
```

Alphanumerics are uppercase, lowercase, or digits.

alphanumeric

```
aln   ? upr | lwr | dgt ;
```

A hex digit is a letter A to F or a digit.

hex digit

```
hex   ? dgt |'A'|'B'|'C'|'D'|'E'|'F'|'a'|'b'|'c'|'d'|'e'|'f' ;
```

There is a category for conventional character symbols.

character symbols

```
smb ? '-'|'_'|'+'|'='|'`'|'~'|'!'|'@'|'#'|'$'|'%'|'^'|'&'|
       '|'|'/'|':'|';'|'*'|'('|')'|'['|']'|'{'|'}'|','|'.'|'<'|'>'|'?' ;
```

Three specific symbols are always preceded by a backslash when included in typical strings. Two more may be. Each specific symbol is described as the accumulation of the rule for matching a backslash and for matching that symbol. Accumulation rules are identified by a forward slash.

escaped symbols, accumulation rule

```
sps ? bsl | btk | bqt | bnl | btb ;
bsl / '\\' '\\' ;
btk / '\\' '\'' ;
bqt / '\\' '\"' ;
bnl / '\\' 'n' ;
btb / '\\' 't' ;
```

Whitespace characters need definition too. The code in this chapter implements a scannerless parser[12] and whitespace is explicitly coded in ibnf grammars.

whitespace, scannerless parser

```
wsc  ? ' ' | '\t' | '\n' ;
```

Whitespace is often made of spaces in sequence. The period in this next

optional rule, recursive rule, sequences

[12]Daniel J. Salomon and Gordon V. Cormack. Scannerless NSLR(1) parsing of programming languages. SIGPLAN 89, pp 170-178. ACM Press, 1989.

rule indicates an optional item in the sequence, and in this case the item refered to is the rule itself and so the rule is recursive[13], allowing a sequence of one or more spaces to be met by the rule.

```
s  /  sp .s ;
```

tabs

Spaces can be made of space characters or tab characters.

```
sp   ? ' ' | '\t' ;
```

characters in strings

A character suitable for inclusion in a string is a number, a lowercase letter, an uppercase letter, a common symbol, or a special symbol, or whitespace.

```
sch ? dgt | upr | lwr | smb | wsc | sps ;
```

strings of characters

Strings in ibnf are made of one or more suitible characters.

```
chs / sch .chs ;
```

positive integers

Positive integers are made from one or more digits in sequence. In `ibnf` the period indicates an optional element of the sequence, and rules may be recursive.

```
pnt / dgt .pnt ;
```

alphanumeric symbols

Alphanumeric 'symbols' are made of uppercase, lowercase, and numeric characters.

```
als / aln .als ;
```

[13]Wirth, Niklaus (1976). Algorithms + Data Structures = Programs. Prentice-Hall p. 126.

2.3 A Calculator Example

A calculator that can compute simple math expressions will have a syntax for those expressions. This calculator example therefore contains additional syntax definitions.

2.2 Implement a Calculator Example.	1/28/2013 ✓

Goals, continued.

The calculator example syntax is defined in `sntx,calc-example,ibnf` and is exported as `calc-example.ibnf`):.

```
`cap,env,lang,indp,sntx ~ ( [ calc-example :~( [ ibnf :~ $ ];
                                              [ six-py  :~ $ ];
                                              [ input  :~ $ ]; $ )]; $ );
```

In this calculator example the syntax is a sequence of newline-terminated lines:

```
    syntax  = line .s '\n' .syntax ;
```

Semantic actions

But calculator does not just parse expressions... It must also return results. Returning the results of what was expressed in the parsed syntax make up the semantic actions[14], of the calculator.

In this chapter another set of definitions declare what semantic action is associated with the succesful parsing of a syntax item. In this calculator example the semantics are defined in `sntx,calc-example,six-py` and are exported as `calc-example.six-py`). For convenience the semantic actions are shown in a different color. Here is the semantic declaration for the above syntax rule:

```
    syntax  ^ .1.1 + .4.1
```

The above semantic action for the syntax rule merely says that the semantic results of the first component of the syntax rule (e.g. the result of the line rule) is concatenated with the fourth component of the syntax rule (e.g. the result of

[14]Dijkstra, Edsger W. (1976.) The characterization of semantics (A Discipline of Programming, Chapter 3) Prentice-Hall.

recursively applying the syntax rule.)

For the syntax of a line, one may include a variable assignment but will include a negatable expression:

```
    line  = .var nexpr ;
```

```
    line  : registers[ .1.1.1 ] = int( .2.1 )
          |  --> ` .2.1 `
```

For the semantics of a line, a register will be assigned the value of the negatable expression and then a textual string containing the value of the negatable expression will be returned to be accumulated by the syntax rule.

Implementation Language

The semantic expressions for these syntax rules are a pidgin[15] of 'semantics for ibnf expressions' mixed with the native syntax of another programming language used for implementation such as (in this case) python. In the above examples the references (e.g. .1.1 or .4.1) are references to sematic products of the 'six' parsing engine, while the funcitons, assignments, and operations (e.g. int(), =, +) are borrowed from the implementation language.

Parsing Engine, Syntax Tree

The ibnf/six system is a regularly constructed parsing engine that consumes a source text under the direction of an ibnf grammer text and produces a result. Each time a syntax rule successfully parses all of its sub rules, the collection of the sub rule results is passed as a tuple to the semantic action of the rule. The result of the semantic action is returned from the rule to be accumulated in a tuple at the next level up. The natural intermediate representation of this behaivor is a syntax tree.

For convenience there are three modes of semantic production, each seen once above. The caret indicates that a referenced item or the result of an operation (which may be on referenced items) should be returned. The colon indicates that a computation will occur before a result is returned. A sequence of lines with vertical bars declare quoted text to be returned, with back-ticks unescaping expressions embedded in the quoted text.

Continuing the example:

negatable

A negatable expression is one which may be negated, and the (optional) negation is identified with a '-' sign.

[15]Holm, John (2000), An Introduction to Pidgins and Creoles, Cambridge Univ. Press.

```
nexpr   = .s .neg expr ;
neg     ? '-' ;
```

```
nexpr   ^ (str(0 - int(.3.1)) if(len(.2) > 1 and .2.1 == "-") else .3.1 )
```

A variable assignment is just a lowercase letter followed by the equals sign:

```
var     = .s lwr .s '=' ;
```

```
var     ^ .2
```

An expression may be an addition, a subtraction, an multiplication, a division, or an item all by itself.

expression

```
expr    ? addexpr | subexpr | mulexpr | divexpr | itmexpr ;
```

The addition expression (with optional white space) is two items joined by a '+' sign. Likewise the subtraction, multiplication, and division.

addition
subtraction
multiplication
division

```
addexpr = .s itm .s '+' .s itm ;
subexpr = .s itm .s '-' .s itm ;
mulexpr = .s itm .s '*' .s itm ;
divexpr = .s itm .s '/' .s itm ;
```

```
addexpr ^ str(int(.2.1) + int(.6.1))
subexpr ^ str(int(.2.1) - int(.6.1))
mulexpr ^ str(int(.2.1) * int(.6.1))
divexpr ^ str(int(.2.1) / int(.6.1))
```

The item expression is just an item. An item may be a variable identifier, a positive integer or a parenthetical negatable expression.

item expression,
item,
variable

```
itmexpr = .s itm  ;
itm     ? vbl | pnt | parens ;
```

```
itmexpr ^ .2.1
```

variable

A variable is just a lowercase letter, but when the sytnax is matched in an expression the semantic action is to retrieve the previous value recorded for that variable.

```
vbl     = lwr ;
```

```
vbl     ^ str( registers[ .1.1 ] )
```

parentheses

Parentheses surround a negatable expression. Nesting is thus allowed.

```
parens  = .s '(' nexpr .s ')' ;
```

```
parens ^ .3.1
```

A calculator needs input:

```
a=1+34
b=a + 40
c = 56-6
 d = b - c
x = 10 / 2
v = x - 12
y = - ( 5 - 3 )
z = -80
e = (3 + 5 ) / ( 2 + 2 )
f = - 1
```

To build the example, first run the semantic rule parser on the calculator example semantic rules:

```
python sixparser.py calc-example.six-py blank.txt calc-example.smtx
```

Combine the grammars and then run the syntax rule parser on the calculator example syntax rules and include the generated semantics as a parameter:

```
cat ./common-chr.ibnf ./calc-example.ibnf > ./calc-example-full.ibnf
python ibnfmeta.py calc-example-full.ibnf calc-example.smtx calc-example.py
```

Run the calculator on the input and view the output:

```
python calc-example.py calc-example.input blank.txt calc.out
cat calc.out

 --> 35
 --> 75
 --> 50
 --> 25
 --> 5
 --> -7
 --> -2
 --> -80
 --> 2
 --> -1
```

2.4 Declaring ibnf in ibnf/six

compiler-compiler

Annotating rules with productions (as shown in the calculator example) enables the system to automatically construct an interpreter or even a compiler for input texts written in the described syntax. When the described syntax is a system for describing syntax (e.g. `ibnf.ibnf`) and when the semantics are those of semantics-generation (as will be seen in `ibnf.six-py`, `six.ibnf` and in `six.six-py`) then the result is a parser-generator or a compiler-compiler[16].

meta-compiler

If the compiler-compiler thus described is sufficiently versatile to reproduce its own executable when provided with its own syntax and semantics definitions then the result is a metacompiler[17]. `ibnf/six` is a metacompiler.

2.3 Declare ibnf in ibnf/six.	1/15/2013 ✓

Goals, continued.

`ibnf` will use the common definitions introduced in the first section and introduces more defined in `sntx,ibnf,ibnf` and exported as `ibnf.ibnf`. The semantic rules for parsing ibnf are defined in `sntx,ibnf,six-py` and are exported as `ibnf.six-py`.

```
`cap,env,lang,indp,sntx ~ ( [ ibnf :~( [ ibnf :~ $ ];
                                       [ six-py  :~ $ ]; $ )]; $ );
```

[16] Stephen C. Johnson. YACC: Yet Another Compiler-Compiler. Unix Programmer's Manual Vol 2b, 1979.

[17] Schorre (1964). A Syntax-Oriented Complier Writing Language. Proceedings of the 1964 19th ACM National Conference, ACM Press, New York, NY, 41.301-41.3011

PEG, packrat

The `ibnf/six` system implements a parsing expression grammer, or PEG[18]. A PEG imposes an order of selection on what might otherwise be an ambiguous choice in a BNF-style grammer– in a PEG the first option is selected when there are two possible matches. In addition the `ibnf/six` system implements a packrat[19]. parser, using memoization of previously visited states as an optimization for efficiency and for handling recursive rules.

The result of processing the syntax and semantics rules in this section will be an `ibnf` syntax parser, which is one half of the `ibnf/six` system. Crafting a `six` syntax parser with its own semantic rules will be the subject of the section following this one.

To begin with, a syntax is made of rules.

```
syntax = rules ;
```

```
syntax   ^ prologue + .1.1 + semantics + epilogue
```

The semantic action of the syntax rule is to produce a program of four parts: a prologue that sets up the parsing engine initial conditions, a set of function definitions (each corresponding to a defined rule in the syntax being parsed,) the semantic actions for the syntax being parsed and an epilogue that performs the final actions of the parsing engine before it exits. In this implementation of `ibnf/six` the result is a python program.

```
rules = rule .rules ;
```

```
rules   ^ .1.1 + .2.1
```

The function definitions that result from successfuly parsed rules are simply concatenated into a sequence of function definitions.

A rule may incorporate other rules, may be a choice amongst alternatives or may just be a blank line:

[18]Ford, Bryan (2004). Parsing Expression Grammars: A Recognition Based Syntactic Foundation. Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.
[19]Warth, Douglass, Millstein (2008). Packrat Parsers Can Support Left Recursion. ACM SIGPLAN 2008 Workshop on Partial Evaluation and Program Manipulation

```
rule ? incorp | altern | blankline ;
```

Blank lines have optional whitespace and end in a newline character. Blank lines provide nothing to the parsing program that is being assembled.

```
blankline = .s '\n' ;
```

```
blankline  ^ ""
```

Each rule type has a name and a body. The alternating rule is signified by a question mark.

```
altern =  .s name .s '?' albody  .s ';' .s '\n'  ;
```

The semantic action of matching an 'altern' rule is to compose a new function that takes an input text and indicates a match was found if that input text starts with one of the alternatives listed in the body of the 'altern' rule.

```
altern  |def `.2.1`_p( s, c):
        |  if been("`.2.1`",s): return was( c, "`.2.1`",s)
        |  else:
        |    mark("`.2.1`",s,(F,T,s,0,c,("",""))));met = F
        |`#.5#`
        |    if not met:
        |      return mark("`.2.1`",s,(F,T,s,0,c,("","")))
        |    else:
        |      return mark("`.2.1`",s,(met,mem,s,tl,tc,ta))
.albody ^ #.1.2#  + #.1.3.1#
.cmatch ^ #.1#
.cm     |    if not met: (met,mem,ts,tl,tc,ta)=cm('`.1`',s,c)
.btb    |    if not met: (met,mem,ts,tl,tc,ta)=cm('`.1`',s,c)
.bnl    |    if not met: (met,mem,ts,tl,tc,ta)=cm('`.1`',s,c)
.name   |    if not met: (met,mem,ts,tl,tc,ta)=`.1`_p(s,c)
```

The first thing the produced function does is check to see if the same position in the source text has already been evaluated for matching that syntax rule. If so the function returns the success or failure previously determined.

If that location in the source text has not been evaluated for matching the syntax rule before, the generated function marks the location as failing the match and sets a flag (for having met at least one rule) to False.

In the process of creating the generated function, the 'altern' rule next recurses on the 5th component of the rule, which is the 'albody'. Upon recursion one of six things will be included in the genrated function: a call to another named rule, a call to matching a newline, a call to matching a tab, a call to matching a character, a call to a superior rule to matching characters, or the results of composing recursive calls to the 'albody' rule.

Finally in the genrated function if nothing was matched then failure is returned, otherwise success is both memoized and returned.

The incorporating rule is signified by either a forwards slash or an equals sign:

```
incorp =  .s name .s iflag inbody  .s ';' .s '\n'  ;
iflag  ? '/' | '=' ;
```

The semantic action of matching an 'incorp' rule is to compose a new function that takes an input text and indicates if matches were found sequentially in the input text for all of the items listed in the body of the 'incorp' rule.

```
incorp  :smfnc="_s(a,m,s,e,c,n): return(T,T,s,e-s,c,(n,fi[s:e]))"
        |def `.2.1`_p( s, c):
        |  if been("`.2.1`",s): return was( c, "`.2.1`",s)
        |  else:
        |    mark("`.2.1`",s,(F,T,s,0,c,("","")))
        |    ok=True; ts=s; tl=0; a={0: ("","")}
        |    mem={0:True}; tc=c; n=0
        |`#.5#`
        |    if ok:
        |      rv=`.2.1`_s(a,andmemo(mem),s,ts+tl,tc,"`.2.1`")
        |      return mark("`.2.1`",s,rv)
        |    return mark("`.2.1`",s,(F,T,s,0,c,("","")))
        |`("def "+.2.1+smfnc if .4.1 == "/" else "")`
.inbody |    if ok:
        |      n=n+1; ( `(  "n" if .1.2.0 == "pnit" else "" )`ok,mem[n],ts,tl,tc,a[n])=\
        |      `#.1.2.1 if .1.2.0=="pnit" else .1.2#`
        |`#.1.3#`
.pnit   ^ "n"
.cmatch ^ #.1#
.name   ^ .1 +"_p ( (ts+tl), tc)"
.cm     ^ "cm(\'" + .1 + "\',(ts+tl), tc)"
.bsl    ^ " cm(chr(92) ,(ts+tl), tc)"
.btk    ^ " cm(chr(39) ,(ts+tl), tc)"
.bqt    ^ " cm(chr(34) ,(ts+tl), tc)"
.bnl    ^ " cm(chr(10) ,(ts+tl), tc)"
```

As with the previous rule, the first thing the produced function does is check to see if the same position in the source text has already been evaluated for matching that syntax rule. If so the function returns the success or failure previously determined.

If this is a new evaluation, the generated function marks the location as failing the match and sets a flag (for not having failed a rule yet) to True.

In the process of creating the generated function, the 'incorp' rule next recurses on the 5th component of the rule, which is the 'inbody'. Upon recursion one of nine things will be included in the genrated function: a call to another named rule, a call to matching a newline, a call to matching a tab, a call to matching a single quote, a call to matching a backslash, a call to matching a character, a call to a superior rule to matching characters, a call to matching a period (optional) named item, or the results of composing recursive calls to the 'inbody' rule.

Finally in the genrated function if everything was matched then the semantic function is called, after which success is both memoized and returned. Otherwise in the generated function failure is memoized and returned.

If the rule being generated was marked with a slash then the incorp rule will generate an additional semantic function for the rule that simply returns the extent of the source text matched as the result of that rule's semantic action.

A name is made up of one or more lowercase letters. The semantic action of matching the name rule is to return the letters making up the name.

```
name / lwr .name ;
```

Within the rule `albody` rule-named-items are separated by the vertical bar symbol. More alternatives may be found on the next line if the last thing on the albody line is a vertical bar symbol. For a semantic action, the albody rule returns the complete tuple of values available to it. The almore semantic action however is to return the fourth item in the tuple of values available to it (e.g. the result of its 'albody'.

```
albody = .s nit .almore ;
almore = .s '|' .alnewline albody ;
alnewline / .s '\n' ;
```

```
albody  ^ ..
almore  ^ .4
```

Within the rule `inbody` rule there may be multiple possibly period prefixed named items. Like albody, inbody passes upwards everything it gets.

```
inbody = .s onit .inbody ;
onit ? pnit | nit ;
```

```
inbody ^ ..
```

pnits are:

```
pnit = '.' nit ;
```

```
pnit ^ .2
```

A nit is a name or a character match.

```
nit ? name | cmatch ;
```

A character match is a suitable character (or escaped character) as defined in common-chr surrounded by single quotes.

```
cmatch = '\'' sch '\'' ;
```

```
cmatch ^ .2
```

To build the ibnf parser, first run the semantic rule parser on the ibnf semantic rules:

```
# python sixparser.py ibnf.six-py blank.txt ibnf-smtx.py
```

Combine the grammars and then run the syntax rule parser on the ibnf syntax rules and include the generated semantics as a parameter:

```
# cat ./common-chr.ibnf ./ibnf.ibnf > ./ibnf-full.ibnf
# python ibnfmeta.py ibnf-full.ibnf ibnf-smtx.py ibnf.py
```

Run the generated parser on the very input that produced it, then do it again:

```
# python ibnf.py ibnf-full.ibnf ibnf-smtx.py ibnfmeta2.py
# python ibnfmeta2.py ibnf-full.ibnf ibnf-smtx.py ibnfmeta3.py
```

2.5 Implementing six in ibnf/six

The previous section implemented a parser of syntax rules that allowed for the inclusion of externally generated semantic productions for those rules. This section implements a parser and generator for those semantic rules. The separation of syntax from semantics allows syntax to be re-used and semantics to be re-targeted.

2.4 Declare the syntax and semantics of six. 1/20/2013 ✓

Goals, continued.

`six` will use the common definitions introduced in the first section and introduces more defined in `sntx,six,ibnf` and exported as `six.ibnf`. The semantic rules for parsing six are defined in `sntx,six,six-py` and are exported as `six.six-py`.

```
`cap,env,lang,indp,sntx ~ ( [ six :~( [ ibnf :~ $ ];
                                        [ six-py  :~ $ ]; $ )]; $ );
```

The result of processing the rules of sytax for six will be an six parser.

```
syntax = srules end ;
```

```
syntax ^ .1.1 + .2.1
```

The six parser consumes semantic rules and (as its own semantic function) emits the results of processing those rules into semantic functions, followed by some predefined values that may be used by semantic functions.

The srules rule is defined recursively:

```
srules = srule .srules ;
```

```
srules ^ .1.1 + .2.1
```

The values of the rules (as semantic functions for inclusion in an ibnf parser) are returned concatenated together by the six parser.

A srule is a blank line or a basic semantic definition.

```
srule ? blankline | base ;
```

Blank lines have optional whitespace and end in a newline character.

```
blankline = .s '\n' ;
```

```
blankline  ^   ""
```

A basic semantic definition has a name followed by some optional setup code, a body of program code that is produced when the syntax for that named syntax component is matched by the parser, and then an optional sequence of recursive definitions that may be refrenced by that body of code.

```
base   =  .s name .setup body .recr;
body   ?  qlineset | cline  ;
setup  = .s ':' .s code .s '\n' .setup ;
recr   = .s '.' name .rsetup body .recr ;
rsetup = .s ':' .s rcode .s '\n' .rsetup ;
rcode  = ritm .rcode ;


ritm ?     string |  rcr | lwr | dpathw | dhas | pnt | '>' | '<' |
       '{' | '}' | ':' | '%' | '(' | ',' | ')' | '_' | '[' | ']' |
                   ';' | '+' | '-' | '*' | '/' | '=' | '!' | ' ' ;
```

The body of program text may be a set of 'quoted' text lines to be deposited as the resulting function's return value or it may be a singe code line, and the result of executing that code line is deposited as the resulting function's return value.

The semantic implementation of the basic semantic rule declares a semantic function which conditionally includes setup code for a rule being parsed by that rule, will include 'body' code for returning a result for a rule being parsed by that rule, and which conditionaly is followed by a recursive function definition for a semantic rule being parsed by that rule:

```
base  :rbody="  o = \"\"\n  rx="+.2.1+"_r\n  if a != \"\":\n"+.5.1+"  return (o)\n"
      |def `.2.1`_s(a,m,s,e,c,n):
      |`("  rx="+.2.1+"_r "  if .5.1 != "" else "")`
      |`("  "+.3.1 if .3.1 != "" else "")`
      |  return (T,T,s,e-s,c,( "`.2.1`", `.4.1` ))
      |`("def "+.2.1+"_r(a,m,s,e,c,n):\n"+rbody if .5.1 != "" else "")`

setup  ^ ( ( .4.1 + "\n  " + .7.1) if .7.1 != "" else .4.1 )

rsetup ^ ( ( "      "+.4.1 + "      " + .7.1+"\n") if .7.1 != "" else "      "+.4.1+"\n" )

recr   ^ "    if a[0] ==\"" + .3.1 + "\":\n" + .4.1 + "      o=" + .5.1 +"\n"+ .6.1
```

A code line starts with a caret:

```
cline    =  .s '^' .s code .s '\n' ;
```

```
cline    ^ .4.1
```

A set of quoted lines have a vertical bar at the start of each:

```
qlineset =  qlines ;
qlines   =  .s qlsep qline .qlines ;
qlsep    =  '|' ;
qline    =  qchs '\n' ;
```

```
qlineset ^ "\"" + .1.1 + "\""
qlines   ^ .2.1 + .3.1 + .4.1
qlsep    ^ "\\n\" + \\\n\""
qline    ^ .1.1
```

A code line may be escaped with back-tics. The result of code within the back-tics is inserted in-line with the quoted text:

```
qchs   = .qch .qchs ;
qch    ? aln | qq | qt | qs | qsmb | ' ' | qcode  ;
qq     = '\"' ;
qt     = '\'' ;
qs     = '\\' ;


qcode = '`' .s code .s '`' ;
```

```
qchs   ^ .1.1 + .2.1
qq     ^    "\\\""
qt     ^    "\\\'"
qs     ^    "\\\\"
qcode  ^ "\" + " + .3.1 + " + \""
```

A name is composed of lowercase letters:

```
name / lwr .name ;
```

Quoted symbols may not include the back-tick, because that is used to escape the quoted text:

```
qsmb   ? '-'|'_'|'+'|'='|'~'|'!'|'@'|'#'|'$'|'%'|'^'|'&'|'!'|'|'|'/'|
           ':'|';'|'*'|'('|')'|'['|']'|'{'|'}'|','|'.'|'<'|'>'|'?' ;
```

Strings may be interrupted by newlines and continued on the next line after a continuation character:

```
string  =  '\"' .strcs '\"' ;
strcs   =  sch .strcs ;
```

```
string  ^  "\"" + .2.1 + "\""
strcs   ^  .1.1 + .2.1
```

six-py is a pidgin of six syntax for data reference paths and recursive calls, and of python code. The pidgin is constructed of code item(s) and may span multiple lines when the newline is followed by a code continuation character:

```
code = citm .code ;
citm ? string | cnl | rcr | lwr | upr | dpathw | dhas | pnt | '>' | '<' |
        '{' | '}' | ':' | '%' | '(' | ',' | ')' | '_' | '[' | ']' |
                    ';' | '+' | '-' | '*' | '/' | '=' | '!' | ' ' ;
cnl = '\n' .s '^' ;
```

```
code     ^    .1.1  + .2.1
cnl      ^    "\\\n"
```

Data reference paths are turned into python array dereferences:

```
dpathw   =    dpath ;
dpath    =    '.' pnt .dpath ;
dhas     =    '.' '.' ;
```

```
dpathw   ^    "a" + .1.1
dpath    ^    "[" + .2.1 + "]" + .3.1
dhas     ^    "a"
```

Recursive calls are made between hash marks:

```
rcr   ?    rca | rcb ;
rca   =    '#' .s name .s ':' .s code .s '#' ;
rcb   =    '#' .s code .s '#' ;
```

```
rca   ^    .3.1 +"_s(" + .7.1 + ",m,s,e,c,n)[5][1]"
rcb   ^    "rx(" + .3.1 + ",m,s,e,c,n)"
```

The parsing engine assembled by the ibnf/six system requires some predefined functionality, which is encapsulatd in six in definitions for a prologue and an epilogue and which is placed in the semantic output after all of the semantic rules. The syntax rule for 'end' optionally matches whitespace, which will always succeed.

```
end   =  .s  ;
```

The prologue and epilogue are included as simple global variable assignments that may be referred to in other sematic productions (such as the production for 'syntax' in ibnf.)

```
end |prologue="""import sys
    |from binascii import *
    |fi = file(sys.argv[1]).read()
    |semantics = file(sys.argv[2]).read()
    |fo = open(sys.argv[3], "w+")
    |
    |h={}; registers={}; context={}; mseq=0; dseq=1; T=True; F=False
    |
    |def n2z( a ):
    |  return ( '0' if a=='' else a )
    |
    |def be2le( a ):
    |  return a[6:8]+a[4:6]+a[2:4]+a[0:2]
    |
    |def inclast( a ):
    |  return a.rpartition(".")[0]+"."+str(int(a.rpartition(".")[2])+1)
    |
    |def unesc( a ):
    |  return a.decode('string_escape')
    |
    |def mark( p, s, t ):
    |  ( v, m, ss, l, c, a ) = t
    |  if t[1]:  x = p +"-" + str(s);  h[x]=(v,m,l,a); return t
    |  else:
    |    if not t[0]:  x = p +"-" + str(s);  h[x]=(v,m,l,a); return t
    |  return t
    |
    |def been(p, s):
    | if h.has_key( p +"-" + str(s) ): return h[p +"-" + str(s)][1]
    | else:  return False
    |
    |def was(c,p,s): (v,m,l,a) = h[p+"-"+str(s)]; return (v,m,s,l,c,a)
    |
    |def cm( ch, s, c ):
    |  if s < len(fi):
    |    if fi[s] == ch:    return ( T, T, s, 1,  c, ( "cm", fi[s] ) )
    |  return ( False, True, s, 0, c, ( "cm", "") )
    |
    |def andmemo( m ):
    |  r = True
    |  for i in m:
    |    if not m[i]: r = False
    |  return r
    |
    |outdata = ""
    |
    |def output( s ):
    |  global outdata
    |  outdata = outdata + str(s)
    |
    |"""; epilogue="""
    |
    |(v,m,s,l,c,a) = syntax_p( 0, ({},'<1>','<0>') )
    |if v:
    |  print "Parsed "+a[0]+" OK"
    |else: print "Failed to Parse"
    |print >> fo, a[1]
    |fo.close()
    |"""
```

To build the six parser, run the semantic rule parser on the six semantic rules:

```
# python sixparser.py six.six-py blank.txt six-smtx.py
```

Combine the grammars and then run the syntax rule parser on the six syntax rules and include the generated semantics as a parameter:

```
# cat ./common-chr.ibnf ./six.ibnf > ./six-full.ibnf
# python ibnfmeta.py six-full.ibnf six-smtx.py sixmeta.py
```

Run the generated parser on the very input that produced it, then regenerate the semantics parser:

```
# python sixmeta.py six.six-py blank.txt six-smtx2.py
# python ibnfmeta.py six-full.ibnf six-smtx2.py sixmeta2.py
```

Rebuild the ibnf parser using the new semantic rule parser:

```
# python sixmeta2.py ibnf.six-py blank.txt ibnf-smtx.py
```

Combine the grammars and then run the syntax rule parser on the ibnf syntax rules and include the generated semantics as a parameter:

```
# cat ./common-chr.ibnf ./ibnf.ibnf > ./ibnf-full.ibnf
# python ibnfmeta.py ibnf-full.ibnf ibnf-smtx.py ibnf.py
```

Run the generated parser on the very input that produced it, then do it again:

```
# python ibnf.py ibnf-full.ibnf ibnf-smtx.py ibnfmeta2.py
# python ibnfmeta2.py ibnf-full.ibnf ibnf-smtx.py ibnfmeta3.py
```

2.6 A toy compiler in ibnf/six

The ibnf/six system can be used to create a compiler that produces machine code from source code in a defined language syntax.

2.6 Craft a toy compiler in ibnf/six. 2/8/2013 ✓

Goals, continued.

The toy syntax needs a place to reside.

```
`cap,env,lang,indp,sntx ~ ( [ toy :~( [ ibnf :~ $ ];
                                      [ six-py :~ $ ]; $ )]; $ );
```

The syntax of a toy program will consist of a series of statement sequence lines.

```
    syntax  =  program end ;
    program =  lines ;
    lines   = .seq .s '\n' .lines ;
    seq     = .s stmt .more ;
    more    = .s ';' seq ;
```

```
    syntax ^ header + code + literals

   program : # sizecode   : .1.1 #
           : # countvars  : .1.1 #
           : # makelit    : .1.1 #
           : # emitcode   : .1.1 #
           : print .1.1
           | ok

    lines  ^ ( "seq", .1.1 , .4.1 )
    seq    ^ ( "seq", .2.1 , .3.1 )
    more   ^ .3.1
```

The only statements for now are print, var, and exit.

```
    stmt    ?  lsprint | vsprint | vdef | exit ;
    lsprint   = 'p' 'r' 'i' 'n' 't' .s lstring ;
    vsprint   = 'p' 'r' 'i' 'n' 't' .s vname ;
    vdef     = 'v' 'a' 'r' s vname .s '=' .s varable ;
    vname   / lwr .vname ;
    exit    = 'e' 'x' 'i' 't' .s exitvalue ;
```

```
  lsprint  ^ ( "lsprint", .7.1 )
  vsprint  ^ ( "vsprint", .7.1 )
  vdef    ^ ( "vdef", .5.1, .9.1 )
  exit   ^ ( "exit",  .6.1 )
```

Only literal strings are printable right now, and one exits with an exit value.

```
  lstring = '\"' .strchs '\"' ;
  strchs  = strch .strchs ;
  strch   ? dgt | upr | lwr | smb | ' ' | '\t' | nsl | ntk | nqt | nnl | ntb ;
  nsl     = bsl ;
  ntk     = btk ;
  nqt     = bqt ;
  nnl     = bnl ;
  ntb     = btb ;
  varable   = '\"' .chs '\"' ;
  exitvalue / pnt ;
```

```
  strchs  ^ .1.1 + .2.1
  nsl     ^ "\\"
  ntk     ^ "\'"
  nqt     ^ "\""
  nnl     ^ "\n"
  ntb     ^ "\t"
  lstring  ^ ( "lstring", .2.1, str(s), str(len(.2.1)) )
  varable    ^ ( "lstring", .2.1, str(s), str(len(.2.1)) )
```

```
  end  = .s ;
```

Before we do anything else we need to know how big the executable portion of the program is:

```
sizecode : global lstart
         : lstart = int(# .. #)+232+4096+8
         | ok
 .seq    ^ str(0 + int(n2z(#.1#)) + int(n2z(#.2#)))
 .exit   ^ str(len(unhexlify(# exitx : 0 # )))
 .lsprint  ^ str(len(unhexlify( # lsprintx : (0,0) # )))
 .vdef   ^ str(len(unhexlify( # vdefx : (0,0,0) # )))
 .vsprint ^ str(len(unhexlify( # vsprintx : 0 # )))
```

Making the literal pool:

```
  makelit : global literals, llist, lend; llist = {}; lend = 0
          : literals = # .. #
          | ok
 .seq     ^ #.1# + #.2#
 .exit    ^ ""
 .lsprint   ^ #.1#
 .lstring  : global llist, lend; llist[.2] = lend; lend = lend + len(.1)
          ^ .1
 .vdef    ^ #.2#
```

Counting the local variables:

```
countvars : global vlist, vend; vlist = {}; vend = 0
          : literals = # .. #
          | ok
 .seq     ^ #.1# + #.2#
 .vdef    : global vlist, vend; vlist[.1] = vend; vend = vend + 1
          | ok
```

Emitting the code:

```
emitcode  : global lstart, code, vend
          : code = unhexlify(# mkloclx : vend #) + # .. #
          | ok
 .seq     : global lstart, llist
          ^  #.1# + #.2#
 .exit    ^ unhexlify(# exitx : int(.1) # )
 .lsprint ^ unhexlify( # lsprintx : ( int(.1.3),lstart+llist[.1.2]) # )
 .vdef    ^ unhexlify( # vdefx : (lstart+llist[.2.2],vlist[.1],int(.2.3)) # )
 .vsprint ^ unhexlify( # vsprintx : vlist[.1] # )
```

'Assembly language' for each language construct, starting with making space for local variables:

```
 mkloclx  ^ # decsp   : .. * 8  # +
          ^ # esptoebp: .. #
```

To make an 'exit' system call, put 1 in eax and push the exit value on the stack then call interrupt 0x80.

```
exitx    ^ # oneeax  : .. # +
         ^ # pushc   : ..  # +
         ^ # decsp   : 4  # +
         ^ # int     : 128  #
```

To make an 'write' system call with a string constant, put 4 in eax, push the length of the string, the location of the string and some padding on the stack, then call interrupt 0x80. When the system call returns, remove the values from the stack.

```
lsprintx ^ # pushc   : .0  # +
         ^ # pushc   : .1  # +
         ^ # pushc   : 1  # +
         ^ # toeax   : 4  # +
         ^ # decsp   : 4  # +
         ^ # int     : 128  # +
         ^ # incsp   : 16  #
```

To initialize a local variable to a string constant value, place the address of the string and the length of the string at the local variable's offset in the stack frame.

```
vdefx    ^ # toeax   : .0 # +
         ^ # ebpeax  :  .1 * 8 # +
         ^ # toeax   :  .2   # +
         ^ # ebpeax  : ( .1 * 8 ) + 4 #
```

To make an 'write' system call with a string variable, retrieve and push the length of the string, retrieve and push the location of the string, put 4 in eax and some padding on the stack, then call interrupt 0x80. When the system call returns, remove the values from the stack.

```
vsprintx ^ # eaxebp  : ( .. *8)+4 # +
         ^ # pusheax : .. # +
         ^ # eaxebp  : ( .. *8) # +
         ^ # pusheax : .. # +
         ^ # pushc   : 1  # +
         ^ # toeax   : 4  # +
         ^ # decsp   : 4  # +
         ^ # int     : 128  # +
         ^ # incsp   : 16  #
```

Machine code for the 'Assembly language':

```
oneeax   ^ "31c040"
esptoebp ^ "89e5"
incsp    ^ "81c4" + be2le("%08X" % (..))
decsp    ^ "81ec" + be2le("%08X" % (..))
toeax    ^ "b8"   + be2le("%08X" % (..))
eaxebp   ^ "8b45" + "%02X" % (..)
ebpeax   ^ "8945" + "%02X" % (..)
pushc    ^ "68"   + be2le("%08X" % (..))
pusheax  ^ "50"
int      ^ "cd"   + "%02X" % (..)
```

Building the mach-o program header:

```
end   : global header, code, literals
      : hdr =       unhexlify("cefaedfe07000000030000000200000002000000CC000000")
      : c1  =       unhexlify("00000000010000007c0000005f5f54455854000000000000")
      : c1  = c1  + unhexlify("00000000001000000010000000000000")
      : fsz =       unhexlify("00000000")
      : c2  =       unhexlify("07000000050000000100000000000000 5f5f7465787400000000")
      : c2  = c2  + unhexlify("0000000000005f5f5445585400000000000000000000E8100000")
      : bsz =       unhexlify("00000000")
      : c3  =       unhexlify("E800000002000000000000000000000000000000")
      : c3  = c3  + unhexlify("00000000000000000500000005000000")
      : c3  = c3  + unhexlify("01000000010000000000000000000000")
      : c3  = c3  + unhexlify("00000000000000000000000000000000")
      : c3  = c3  + unhexlify("00000000000000000000000000000000")
      : c3  = c3  + unhexlify("E8100000000000000000000000000000")
      : c3  = c3  + unhexlify("0000000000000000")
      : fsz = unhexlify( be2le( "%08X" % (232 + len(code) + len(literals)) ))
      : bsz = unhexlify( be2le( "%08X" % (len( code ) + len( literals )) ))
      : header = hdr + c1 + fsz + c2 + bsz + c3
      | ok
```

A toy compiler needs source code to compile:

```
print "This is a test of the emergency broadcast sytsem. This is only a test.\n"
var foo = "bar"
var baz = "quux"
print baz; print foo
print "This is another print statement.\n"
exit 42
```

Build the toy compiler semantics with sixparser then build the toy compiler with ibnfmeta:

```
python sixparser.py toy.six-py blank.txt toy-smtx.py
cat ./common-chr.ibnf ./toy.ibnf > ./toy-full.ibnf
python ibnfmeta.py toy-full.ibnf toy-smtx.py toy-compile.py
```

Compile the toy program source code with the toy compiler, then execute the toy executable:

```
python toy-compile.py test.toy blank.txt toy.out
chmod 755 toy.out
./toy.out
```

2.7 The i language in ibnf/six

ibnf and six can be used to create a parser of the i language and a constructor of things expressed in that language.

2.3 Declare i in ibnf.
— not yet demonstrated

Goals, continued.

i syntax and semantics need a place to reside.

```
`cap,env,lang,indp,sntx ~ ( [ i :~( [ ibnf :~ $ ];
                                    [ six-py :~ $ ];
                                    [ test :~ $ ]; $ )]; $ );
```

Something written in the i syntax (a.k.a i code) is at its simplest an i statement surrounded by optional whitespace.

The result of parsing the source text should be an image as defined by the source text. An image is created (once the icode is parsed) in several steps – 1) make an abstract syntax tree, 2) make a mosaic from the tree; 3) collapse the mosaic into an image.

```
syntax  = icode ;
icode   = w istmt w ;
```

```
syntax ^ image

icode  : global image; global st
       : st = # maketree   : ( .2.0, (0,".0",-1,-1,0), .2 ) #
       : ms = # makemosaic : st #; print "\n"; print st
       : print "\n"; print ( .2.0, (0,".0",-1,-1,0), .2 ); image = ""
       | ok
```

Optional white space is a common feature of a syntax.

```
w  = .wsc .w ;
```

```
w ^ .1
```

statement An i statement may be one of several things. The ibnf parser uses ordered choice to disabiguate alternatives. In order of priority they are a compound statement, a definition, an exansion, an import, a reference, or a marker, or a literal:

```
  istmt  ?   ltrl | cmpnd | defn | expand | import | refr | mrkr ;
```

literal A literal is a simple value such as a string of text or a numerical value.

```
  ltrl     ? istring | numval ;
```

string A string is delimited by quotation marks. When a string is found in the source text a string node is created in the tree which points to the un-escaped string value in the image.

```
  istring     = '\"' chs '\"' ;
```

```
istring ^ ( "string", unesc(.2.1) )
```

compound A compound statement is one or more statements (a phrase) in parentheses.

```
  cmpnd     = '(' w iphrase w ')' ;
```

```
cmpnd ^  .3.1
```

phrase A phrase in i is either several statements separated by semicolons or just a statement by itself.

```
  iphrase     = istmt w .iphrasec ;
  iphrasec    = ';' w iphrase ;
```

```
iphrase ^ ( "seq" , .1, .3  )
iphrasec ^ .3
```

i code may begin and end with parentheses, here's one to start things:

```
(
```

With the above ibnf and semantic definitions and assuming a closing parenthesis to match the opening one, the following is valid i code:

```
        "this is a test string
spanning\n three lines.";
```

Hex numbers, big endian or little endian, may be placed directly in the image resulting from the i code (Numbers that are just numbers are not included directly in the image but may be refered to or included as parameters to other definitions.) number

```
numval     ?  hexbnum | hexlnum | jnum ;
```

Just-numbers may be negative or positive.

```
jnum     ?  nnum | pnum ;
```

Hexadecimal numbers with a capital X have a node created for them in the tree and their values are stored in big-endian format in the image. big endian hex

```
hexbnum =  '0' 'X' hexdigitsb ;
```

```
hexbnum ^ unhexlify( .3.1 )
```

Hexadecimal numbers with a lowercase x have a node created for them in the tree and their values are stored in little-endian format in the image. little endian hex

```
hexlnum     = '0' 'x' hexdigitsl ;
```

```
hexlnum ^ unhexlify( .3.1 )
```

Hexadeximal digits include the regular digits and the letters A through F. hex digits

```
  hexdigitsb     =  hex hex .hexdigitsb ;
  hexdigitsl     =  hex hex .hexdigitsl ;
```

```
hexdigitsl ^ "" + .3.1 + .1.1 + .2.1
hexdigitsb ^ "" + .1.1 + .2.1 + .3.1
```

With the above ibnf and semantic definitions, the following two values are also valid i code (and they are equivalent):

```
0x41424344;
0X41424344;
```

A positive number may be floating point or an integer.

```
  pnum     ?  floatnum | integer ;
```

floating point

floating point numbers have a decimal point in them.

```
  floatnum     /  integer '.' dfract ;
```

decimal fraction

A decimal fraction is an integer after the dot.

```
  dfract     =  integer  ;
```

```
dfract ^ ""
```

integer

An integer is one or more digits.

```
  integer     /  dgt .integer ;
```

negative numbers

A negative number is preceeded by a minus sign.

```
  nnum     /  '-' pnum ;
```

With the above ibnf definitions, the following values are also valid:

```
3.14;
512 ;
-25;
-7.75;
```

With statements and phrases defined as they are above, statements in sequence and nested parentheses are implemented through rule recursion. Sequences like the following can be used:

```
(  3.14; "This is a string";
  512 ;
 ( -25 ; 0XFFFFFFFF);
  -7.75 );
```

A definition uses square brackets and has many optional parts: definition

```
  defn = .s .sq w '[' w .ddnm w ':' w .tsq w '~' w .istmt w ']' ;
```

```
  defn ^ ( .2, .6, .10, .14 )
```

```
def defn_s (a,m,s,e,c,n):
  (c0,c1,c2)=c; c0[c1]=a[14]
  if len(a[2][1])>0: return ( True, False, s, e-s, (c0,c2[0],c2[1]),(n, "" ))
  else:   return ( True, False, s, e-s, (c0,c2[0],c2[1]),(n,a[14] ))
```

An example:

```
(  [:~];
   [:~ "There are four" ];
   [:~ [:~ "anonymous definitions here." ]] );
```

Definition items... Definition items

```
  ddnm  =  dnm ;
```

```
  ddnm ^ .1.1
```

```
def ddnm_s (a,m,s,e,c,n):
 (c0,c1,c2)=c; return ( True, False, s, e-s, (c0,a[1][1],c2),(n,fi[s:e] ))
```

defname

The name of a definition is made from a sequence of labels that may indicate variables.

```
    dnm     /  w vlabel .dnm ;
```

vlabel

Variables in a name are indicated with a tick-mark. Otherwise they are just labels.

```
    vlabel     /  .'\'' w label ;
```

label

A label starts with an alphabetic character and may continue with alphabetic or numeric characters.

```
    label     /  alp  .als ;
```

And so definitions can be named:

```
(  [ foo :~ 99 ];
   [ 'x frob 'y :~ "Yowza!" ];
   [ nesting :~ [ definitions :~ 0X3233343536373839 ]] );
```

In the AST tuple, the values are: 0) nest depth, 1) image offset, 2) logical offset, 3) size in image

Preprocessing the AST:

```
maketree  ^ # .. #
.cmpnd    : z =  #( .2.1.0, ( .1.0 + 1 , .1.1 + ".0",-1,-1,0), .2.1 )#
          ^ z
.iphrase  : z =  #( .2.1.0, .1, .2.1 )#
          ^ z
.iphrasec ^  #( .2.1.0, .1, .2.1 )#
.seq      : y = #( .2.1.0, .1, .2.1 )#; z = #( .2.2.0, ( .1.0, inclast(.1.1),-1,-1,0 ), .2.2 )#
          ^ ( "seq", ( .1.0, .1.1,-1,-1,y[1][4]+(z[1][4] if z!="" else 0)),  y, z )
.istring  ^  #( .2.1.0, .1, .2.1 )#
.string   ^ ( "string", ( .1.0 , .1.1,-1,-1,len(.2.1)),  .2.1  )
.hexlnum  ^ ( "hexbin", ( .1.0 , .1.1,-1,-1,len(.2.1)),  .2.1  )
.hexbnum  ^ ( "hexbin", ( .1.0 , .1.1,-1,-1,len(.2.1)),  .2.1  )
.nnum     ^ ( "nnum",   .1,  .2.1  )
.floatnum ^ ( "floatnum",   .1,  .2.1  )
.integer  ^ ( "integer",   .1,  .2.1  )
.defn     ^ ( "defn",   .1,  .2.1.1.1,  #( .2.1.3.0, ( .1.0 + 1 , .1.1 + ".0",-1,-1,0) , .2.1.3 )# )
.mrkr     ^ ( "mrkr",   .1,  "$" )
.refr     ^ ( "refr",   .1,  .2.1 )
.expand   ^ ( "expand",   .1,  .2.1 )
```

Making the mosaic of images:

```
makemosaic: global image
          : # .. #
          | ok
.seq      : # .2 #; # .3 #
          | ok
.string   : print str(.1.2)+" "+str(.1.3)+" "+str(.1.4)+" "+.1.1+"      String  ! "+ .2
          | ok
.hexbin   : print str(.1.2)+" "+str(.1.3)+" "+str(.1.4)+" "+.1.1+"      Hexbin  ! "+ .2
          | ok
.nnum     : print str(.1.2)+" "+str(.1.3)+" "+str(.1.4)+" "+.1.1+"      Nnum    ! "+ .2
          | ok
.floatnum : print str(.1.2)+" "+str(.1.3)+" "+str(.1.4)+" "+.1.1+"      Floatnum! "+ .2
          | ok
.integer  : print str(.1.2)+" "+str(.1.3)+" "+str(.1.4)+" "+.1.1+"      Integer ! "+ .2
          | ok
.defn     : print str(.1.2)+" "+str(.1.3)+" "+str(.1.4)+" "+.1.1+"      Defn    !" + .2 ;  # .3 #
          | ok
.mrkr     : print str(.1.2)+" "+str(.1.3)+" "+str(.1.4)+" "+.1.1+"      Mrkr    ! "+ str(.2)
          | ok
.refr     : print str(.1.2)+" "+str(.1.3)+" "+str(.1.4)+" "+.1.1+"      Refr    ! "+ str(.2.1)
          | ok
.expand   : print str(.1.2)+" "+str(.1.3)+" "+str(.1.4)+" "+.1.1+"      Expand  ! "+ str(.2)
          | ok
```

space qualifier

A space qualifier for a definition may provide a path to a space definition or, absent the path, may simply indicate that the definition may be discontinuous with the previous or parent definition.

```
   sq / '@' w .path ;
```

The binary image produced by the following is 'includedincluded'.

```
(    [ a :~ "included"     ];
   @[ b :~ "not included" ];
    [ c :~ "included"     ] );
```

Once a thing is defined it can be referred to.

```
   refr     = path ;
```

```
 refr ^ ( "refr" , "not implemented" )
```

```
def refr_s (a,m,s,e,c,n):
  if c[0].has_key(fi[s:e]): return ( True, False, s, e-s, c,(n, c[0][fi[s:e]]))
  else: return (False, False, s, e-s, c,(n, fi[s:e]))
```

path

A path is a name or a sequence of names separated by commas.

```
   path     / iname .pathe ;
   pathe    / w ',' w path ;
```

name

A name consists of one or more atoms separated by white space.

```
   iname     / atom w .iname ;
```

atom

The atomic part of a name may be label, a string, a number, or a compound statement.

```
   atom     ?  cmpnd | label | ltrl ;
```

With the above syntax and semantics rules the following can be done:

```
(@[ X Y definition : ~ "second" ];
  [ X Z definition : ~ "first" ];
    0xaabbccdd;
    X Y definition );
```

In the i programming language, the dollar-sign is a marker for the lexical location that will accept future expansion. marker

```
  mrkr     = '$' ;
```

```
mrkr ^ ""
```

```
def mrkr_s (a,m,s,e,c,n):
  (c0,c1,c2)=c; global mseq ; mseq = mseq + 1
  nm='>'+str(c1)+'<'
  c0[nm]=(''); return ( True, False, s, e-s, (c0, c1, c2),(n, nm  ))
```

Expansion is indicated by the back-tick character followed by a path to a previously marked location, and then (following a tilde) a statement expands into the previously marked location. expand

```
  expand     = '`' w path w '~' w istmt ;
```

```
expand ^ ""
```

```
def expand_s (a,m,s,e,c,n):
  (c0,c1,c2)=c;  key=">"+a[3][1]+"<"
  print "exapnding "+key
  if c0.has_key(key):
    c0[key]=a[7];  return ( True, False, s, e-s, (c0,c1,c2),(n, '' ))
  else:
    print "expansion not found!"
    return ( False, False, s, e-s, c,(n, '' ))
```

The following i code shows reserving a spot for expansion and then expanding it later:

```
(  @[ a :~ $     ];
    [ b :~ "second" ];
     `a :~ "first"      );
```

import A lexical import of an item defined elsewhere in the abstract syntax tree is a path to that item prefaced by a caret.

```
    import     = '^' w path ;
```

```
def import_s (a,m,s,e,c,n):
  return  ( True, True, s, e-s, c,(n, a[3] ))
```

typeseq When there is more than one type the types are separated by colons.

```
    tsq     =  typeref ':' .tsq ;
```

```
def tsq_s (a,m,s,e,c,n): return ( True, True, s, e-s, c,(n, fi[s:e] ))
```

typeref A type reference...

```
    typeref     = refr ;
```

```
def typeref_s (a,m,s,e,c,n): return ( True, True, s, e-s, c,(n, anot( typ( deref (a[0] ))) ))

spcs = "

def build (v,m,s,l,c,a): return (rbuild (v,m,s,l,c,a,0))

def rbuild (v,m,s,l,c,a,d):
 print spcs[0:(d*2)] + a[0]+" ("+str(len(a)-1)+")"
 if (a[0]=='integer') or (a[0]=='floatnum') or (a[0]=='nnum') or (a[0]=='mrkr'):
   if a[0]=='mrkr':
#     o=str(c[0][a[1]])
     o = rbuild (v,m,s,l,c,c[0][a[1]],d+1)
   else:
     o=''
 else:
   o=''
   for i in range (len(a)):
    if (i > 0) :
     if type(a[i])==type(()):
       o = o + rbuild (v,m,s,l,c,a[i],d+1)
     if type(a[i])==type(""):
       o = o +a[i]
 return o
```

A close parentheses to end our example text:

```
" end of example text " )
```

2.8 Other Programming Languages

We can do syntaxes for other languages as well.

Landin, Peter J. 1966. The next 700 programming languages. Communications of the ACM 9(3):157166.

Reynolds, John C. 1972. Definitional interpreters for higher-order programming languages. In Proceedings of the ACM national conference, vol. 2, 717740. New York: ACM Press. Reprinted in Higher-Order and Symbolic Computation

11(4): 363397.

.

```
`cap,env,lang,indp,sntx ~  (
      [ c :~ $ ];
      [ apl :~ $ ];
      [ oz :~ $ ];
      [ occam :~ $ ];
      [ basic :~ $ ];
      [ haskell :~ $ ];
      [ python :~ $ ];
      [ smalltalk :~ $ ];
      [ lisp :~ $ ];
      [ java :~ $ ];
      [ perl :~ $ ];
      [ forth :~ $ ];
      [ pascal :~ $ ];
      [ fortran :~ $ ];
  $ );

`cap,env,lang,indp,sntx,oz ~  (
      [  decl :~ $ ];
      [  conc :~ $ ];
      [  lazy :~ $ ];
      [  mesg :~ $ ];
      [  objt :~ $ ];
      [  coco :~ $ ];
      [  rela :~ $ ];
      [  cnst :~ $ ];
  $ );
```

2.9 ibnf syntax bootstrapping

Here's the six-py for ibnf.

The syntax parsing code generated when applying ibnf rules to themselves is as follows:.

```
def syntax_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "syntax", prologue + a[1][1] + semantics + epilogue  ))

def rules_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "rules", a[1][1] + a[2][1]  ))

def blankline_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "blankline", ""  ))
```

The first rule composes our parsing program from a prologue containing program code setting up the execution environment, the result of parsing our syntax, program code implenenting the semantics of successfuly parsed syntax, and an epilogue of program code.

The next two rules accumulte rules and excluede blank lines.

The alternate rule is quite a bit more complex as in this implementation it produces the python code for parsing an alternating rule:

```
def altern_s(a,m,s,e,c,n):

  rx=altern_r ;  return (T,T,s,e-s,c,( "altern", "\n" + \
"def " + a[2][1] + "_p( s, c):\n" + \
"  if been(\"" + a[2][1] + "\",s): return was( c, \"" + a[2][1] + "\",s)\n" + \
"  else:\n" + \
"    mark(\"" + a[2][1] + "\",s,(F,T,s,0,c,(\"\",\"\")));met = F \n" + \
"" + rx(a[5],m,s,e,c,n) + " \n" + \
"    if not met:\n" + \
"      return mark(\"" + a[2][1] + "\",s,(F,T,s,0,c,(\"\",\"\")))\n" + \
"    else:\n" + \
"      return mark(\"" + a[2][1] + "\",s,(met,mem,s,tl,tc,ta)) " ))
def altern_r(a,m,s,e,c,n):
  o = ""
  rx=altern_r
  if a != "":
    if a[0] =="albody":
      o=rx(a[1][2],m,s,e,c,n)  + rx(a[1][3][1],m,s,e,c,n)
    if a[0] =="cmatch":
      o=rx(a[1],m,s,e,c,n)
    if a[0] =="cm":
      o="\n" + \
"    if not met: (met,mem,ts,tl,tc,ta)=cm(\'" + a[1] + "\',s,c)"
    if a[0] =="btb":
      o="\n" + \
"    if not met: (met,mem,ts,tl,tc,ta)=cm(\'" + a[1] + "\',s,c)"
    if a[0] =="bnl":
      o="\n" + \
"    if not met: (met,mem,ts,tl,tc,ta)=cm(\'" + a[1] + "\',s,c)"
    if a[0] =="name":
      o="\n" + \
"    if not met: (met,mem,ts,tl,tc,ta)=" + a[1] + "_p(s,c)"
  return (o)
```

Similarly the incorprating rule:

```
def incorp_s(a,m,s,e,c,n):
  smfnc="_s(a,m,s,e,c,n): return(T,T,s,e-s,c,(n,fi[s:e]))"
  rx=incorp_r ;  return (T,T,s,e-s,c,( "incorp", "\n" + \
"def " + a[2][1] + "_p( s, c):\n" + \
"  if been(\"" + a[2][1] + "\",s): return was( c, \"" + a[2][1] + "\",s)\n" + \
"  else:\n" + \
"    mark(\"" + a[2][1] + "\",s,(F,T,s,0,c,(\"\",\"\"))) \n" + \
"    ok=True; ts=s; tl=0; a={0: (\"\",\"\")}\n" + \
"    mem={0:True}; tc=c; n=0\n" + \
"" + rx(a[5],m,s,e,c,n) + " \n" + \
"    if ok:\n" + \
"      rv=" + a[2][1] + "_s(a,andmemo(mem),s,ts+tl,tc,\"" + a[2][1] + "\")\n" + \
"      return mark(\"" + a[2][1] + "\",s,rv)\n" + \
"    return mark(\"" + a[2][1] + "\",s,(F,T,s,0,c,(\"\",\"\")))\n" + \
"" + ("def "+a[2][1]+smfnc if a[4][1] == "/" else "") + "" ))
def incorp_r(a,m,s,e,c,n):
  o = ""
  rx=incorp_r
  if a != "":
    if a[0] =="inbody":
      o="\n" + \
"    if ok:\n" + \
"      n=n+1; ( " + (  "n" if a[1][2][0] == "pnit" else "" ) + "ok,mem[n],ts,tl,tc,a[n])=\\\n" + \
"      " + rx(a[1][2][1] if a[1][2][0]=="pnit" else a[1][2],m,s,e,c,n) + "\n" + \
"" + rx(a[1][3],m,s,e,c,n) + ""
    if a[0] =="pnit":
      o="n"
    if a[0] =="cmatch":
      o=rx(a[1],m,s,e,c,n)
    if a[0] =="name":
      o=a[1] +"_p ( (ts+tl), tc)"
    if a[0] =="cm":
      o="cm(\'" + a[1] + "\',(ts+tl), tc)"
    if a[0] =="bsl":
      o=" cm(chr(92) ,(ts+tl), tc)"
    if a[0] =="btk":
      o=" cm(chr(39) ,(ts+tl), tc)"
    if a[0] =="bqt":
      o=" cm(chr(34) ,(ts+tl), tc)"
    if a[0] =="bnl":
      o=" cm(chr(10) ,(ts+tl), tc)"
  return (o)
```

The remainder of the rule genrating semantics are simple:

```
def almore_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "almore", a[4] ))

def albody_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "albody", a ))

def inbody_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "inbody", a  ))

def pnit_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "pnit", a[2]  ))

def cmatch_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "cmatch", a[2]  ))
```

The above rules should be auto-generated.

```
def build (v,m,s,l,c,a): return 'success'

T=True; F=False

prologue = """import sys
from binascii import *

fi = file(sys.argv[1]).read()
semantics = file(sys.argv[2]).read()
fo = open(sys.argv[3], "w+")

h={}; registers={}; context={}; mseq=0; dseq=1; T=True; F=False

def n2z( a ):
  return ( '0' if a=='' else a )

def be2le( a ):
  return a[6:8]+a[4:6]+a[2:4]+a[0:2]

def inclast( a ):
  return a.rpartition(".")[0]+"."+str(int(a.rpartition(".")[2])+1)

def unesc( a ):
  return a.decode('string_escape')

def mark( p, s, t ):
  ( v, m, ss, l, c, a ) = t
  if t[1]:  x = p +"-" + str(s);  h[x]=(v,m,l,a); return t
  else:
    if not t[0]:  x = p +"-" + str(s);  h[x]=(v,m,l,a); return t
  return t

def been(p, s):
 if h.has_key( p +"-" + str(s) ): return h[p +"-" + str(s)][1]
 else:  return False

def was(c,p,s): (v,m,l,a) = h[p+"-"+str(s)]; return (v,m,s,l,c,a)

def cm( ch, s, c ):
  if s < len(fi):
    if fi[s] == ch:    return ( T, T, s, 1,  c, ( "cm", fi[s] ) )
  return ( False, True, s, 0, c, ( "cm", "") )

def andmemo( m ):
  r = True
  for i in m:
    if not m[i]: r = False
  return r
```

TBD.

```
outdata = ""

def output( s ):
  global outdata
  outdata = outdata + str(s)
"""

epilogue = """
(v,m,s,l,c,a) = syntax_p( 0, ({},'<1>','<0>') )
if v:
  print "Parsed "+a[0]+" OK"
else: print "Failed to Parse"


print >> fo, a[1]

fo.close()"""

outdata = ""

def output( s ):
  global outdata
  outdata = outdata + str(s)
```

TBD.

Here is what the semantic transformation turns the sibnf semantics into:

```
def syntax_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "syntax", a[1][1] + a[2][1]  ))

def srules_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "srules", a[1][1] + a[2][1] ))

def blankline_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "blankline", "" ))

def base_s(a,m,s,e,c,n):

  rbody="  o = \"\"\n  rx="+a[2][1]+"_r\n  if a != \"\":\n"+a[5][1]+"  return (o)\n"
  return (T,T,s,e-s,c,( "base", "\n" + \
"def " + a[2][1] + "_s(a,m,s,e,c,n):\n" + \
"" + ("  rx="+a[2][1]+"_r "  if a[5][1] != "" else "") + "\n" + \
"" + ("  "+a[3][1] if a[3][1] != "" else "") + "\n" + \
"  return (T,T,s,e-s,c,( \"" + a[2][1] + "\", " + a[4][1] + " ))\n" + \
"" + ("def "+a[2][1]+"_r(a,m,s,e,c,n):\n"+rbody if a[5][1] != "" else "") + "" ))

def setup_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "setup", ( ( a[4][1] + "\n  " + a[7][1]) if a[7][1] != "" else a[4][1] ) ))

def rsetup_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "rsetup", ( ( "      " + a[4][1] + "\n      " + a[7][1]+"\n") if a[7][1] != "" else

def recr_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "recr", "    if a[0] ==\"" + a[3][1] + "\":\n" + a[4][1] + "      o=" + a[5][1] +"\

def cline_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "cline", a[4][1] ))

def qlineset_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "qlineset", "\"" + a[1][1] + "\"" ))

def qlines_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "qlines", a[2][1] + a[3][1] + a[4][1] ))

def qlsep_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "qlsep", "\\n\" + \\\n\"" ))

def qline_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "qline", a[1][1] ))

def qchs_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "qchs", a[1][1] + a[2][1] ))

def qq_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "qq", "\\\"" ))

def qt_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "qt", "\\\'" ))

def qs_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "qs", "\\\\" ))

def qcode_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "qcode", "\" + " + a[3][1] + " + \"" ))

def string_s(a,m,s,e,c,n):
  return (T,T,s,e-s,c,( "string", "\"" + a[2][1] + "\""   ))
```

And finally the epilogue:

```
def build (v,m,s,l,c,a): return 'success'

T=True; F=False

prologue = """import sys
from binascii import *

fi = file(sys.argv[1]).read()
semantics = file(sys.argv[2]).read()
fo = open(sys.argv[3], "w+")

h={}; registers={}; context={}; mseq=0; dseq=1; T=True; F=False

def n2z( a ):
  return ( '0' if a=='' else a )

def be2le( a ):
  return a[6:8]+a[4:6]+a[2:4]+a[0:2]

def inclast( a ):
  return a.rpartition(".")[0]+"."+str(int(a.rpartition(".")[2])+1)

def unesc( a ):
  return a.decode('string_escape')

def mark( p, s, t ):
  ( v, m, ss, l, c, a ) = t
  if t[1]:  x = p +"-" + str(s);  h[x]=(v,m,l,a); return t
  else:
    if not t[0]:  x = p +"-" + str(s);  h[x]=(v,m,l,a); return t
  return t

def been(p, s):
 if h.has_key( p +"-" + str(s) ): return h[p +"-" + str(s)][1]
 else:  return False

def was(c,p,s): (v,m,l,a) = h[p+"-"+str(s)]; return (v,m,s,l,c,a)

def cm( ch, s, c ):
  if s < len(fi):
    if fi[s] == ch:    return ( T, T, s, 1,  c, ( "cm", fi[s] ) )
  return ( False, True, s, 0, c, ( "cm", "") )

def andmemo( m ):
  r = True
  for i in m:
    if not m[i]: r = False
  return r

outdata = ""

def output( s ):
  global outdata
  outdata = outdata + str(s)

"""

epilogue = """

q3 = chr(34)+chr(34)+chr(34)
predefs = "prologue="+q3+prologue+q3+"; epilogue="+q3+epilogue+q3
```

TBD.

Chapter 3

Kernels and Runtimes

Programs expect support infrastructure to be in place.

kernel,
operating system

NoMMU.

Monolithic.

Microkernel.

3.1 Interrupts and Device Drivers

Timers and X and Y, oh my.

interrupt,
syscall

3.2 The scheduler

The kernel is expected to divide time among the programs that are ready to be executed.

scheduler,
time slice

3.3 Memory Management

mmu, paging

A key service of operating system kernels is the management of memory.

3.4 Purpose-specific kernels

embedded

.

Chapter 4

Bootstrapping Development

Goal — Bootstrap Development. 01/01/2009 ✓

While the cap system is intended to be self sufficient some other system was first required to interpret the source text to build it for the first time. This chapter describes the environment used to create an inital cap system and can be used to re-create it from just the source text (cap.i) if a similar Unix-like host system with some basic text processing tools (sed, awk, grep, etc.) is available.

The following sub-sections construct a bootstrapping environment for cap:

Section:		Page — ssp — req
4.1	Unpacking the source code.	78 — 1 — ✓
4.2	Producing a PDF.	79 — 1 — ✓
4.3	Building a Parser maker.	81 — 0/4 — 0/5
4.4	Constructing the System.	85 — 0/4 — 0/5
4.5	Uploading the Work in Progress.	86 — 0/4 — 0/0

Bootstrapping sections (tasks)

4.1 Unpacking the source code

Goal — Unpack Bootstrap Files. 12/12/2009 ✓

Those programs are, of course, included in the source text for cap as follows:

```
`cap,env ˜ ( [ bootstrap :˜ ( [ xta.sh           :˜ $ ];
                              [ xtf.sh           :˜ $ ];
                              [ i2l.sh           :˜ $ ];
                              [ cap.bookpre.txt  :˜ $ ];
                              [ pro.py           :˜ $ ];
                              [ i2py.sh          :˜ $ ];
                              [ s2py.sh          :˜ $ ];
                              [ epi.py           :˜ $ ];
                              [ cap.ibnfsmntx.sh :˜ $ ];
                              [ cap.mkinterp.sh  :˜ $ ];  $ ) ];  $ );
```

The first script uses the Unix shell and the second script to extract all of the bootstrapping files in this chapter as well as other places.

```
#!/bin/bash
# This is xta.sh
 ./xtf.sh "cap,env,bootstrap,xta.sh"                    xta.sh
 ./xtf.sh "cap,env,bootstrap,xtf.sh"                    xtf.sh
 ./xtf.sh "cap,env,bootstrap,i2l.sh"                    i2l.tmp
 sed -e 's/vbtm/verbatim/g' i2l.tmp >                   i2l.sh
 ./xtf.sh "cap,env,bootstrap,cap.bookpre.txt"           cap.bookpre.txt
 ./xtf.sh "cap,env,bootstrap,pro.py"                    pro.py
 ./xtf.sh "cap,env,bootstrap,i2py.sh"                   i2py.sh
 ./xtf.sh "cap,env,bootstrap,s2py.sh"                   s2py.sh
 ./xtf.sh "cap,env,bootstrap,epi.py"                    epi.py
 ./xtf.sh "cap,env,lang,indp,sntx,calc-example,ibnf"    calc-example.ibnf sntx
 ./xtf.sh "cap,env,lang,indp,sntx,calc-example,six-py"  calc-example.six-py
 ./xtf.sh "cap,env,bootstrap,smtx.py"          smtx.py
 ./xtf.sh "cap,env,lang,indp,sntx,calc-example,input"         calc-example.input
 ./xtf.sh "cap,env,bootstrap,cap.ibnfsmntx.sh" cap.ibnfsmntx.sh
 ./xtf.sh "cap,env,bootstrap,autosmtx.py"      autosmtx.py
 ./xtf.sh "cap,env,bootstrap,README"           README
 ./xtf.sh "cap,env,bootstrap,cap.mkinterp.sh"  cap.mkinterp.sh
 ./xtf.sh "cap,env,lang,indp,sntx,common-chr,ibnf" common-chr.ibnf
 ./xtf.sh "cap,env,lang,indp,sntx,i,ibnf"      i.ibnf
 ./xtf.sh "cap,env,lang,indp,sntx,i,smtx"      i.smtx
 ./xtf.sh "cap,env,lang,indp,sntx,i,six-py"    i.six-py
 ./xtf.sh "cap,env,lang,indp,sntx,i,test"      test.i
 ./xtf.sh "cap,env,lang,indp,sntx,ibnf,ibnf"   ibnf.ibnf
 ./xtf.sh "cap,env,lang,indp,sntx,ibnf,smtx"   ibnf.smtx
 ./xtf.sh "cap,env,lang,indp,sntx,ibnf,six-py" ibnf.six-py
 ./xtf.sh "cap,env,lang,indp,sntx,six,ibnf"    six.ibnf
 ./xtf.sh "cap,env,lang,indp,sntx,sibnf,smtx"  sibnf.smtx
 ./xtf.sh "cap,env,lang,indp,sntx,six,six-py"  six.six-py
 ./xtf.sh "cap,env,lang,indp,sntx,toy,ibnf"    toy.ibnf
 ./xtf.sh "cap,env,lang,indp,sntx,toy,test"    test.toy
 ./xtf.sh "cap,env,lang,indp,sntx,toy,six-py"  toy.six-py
 chmod 755 *.sh ; ./cap.mkinterp.sh
```

The above script works in tandem with the following script and these two must exist on the host operating system (along with the bash, sed, awk, and pdflatex utilites that the scripts depend on.) These two scripts are small enough to type by hand into a text editor if necessary.

```
#!/bin/bash
# This is xtf.sh

  echo "Extracting File" $1 " as " $2 from cap.i
  awk -v pats="$1 ~ " -v pat1e=":xcpt~ \"$1" -v pat2e=":xcps~ \"$1" -v pat3e=":xcpe~ \"$1" '
   BEGIN { pout = 0 }
   $0 ~ pats { pout = 1 }
   $0 ~ pat1e { pout = 0 }
   $0 ~ pat2e { pout = 0 }
   $0 ~ pat3e { pout = 0 }
   pout > 1 {print}
   pout > 0 { pout++ }
  ' cap.i > ./$2
```

To begin bootstrapping the development process from a Unix host system which includes bash, sed and awk, place the files cap.i, xta.sh and xtf.sh in a directory and then execute the xta.sh shell script.

4.2 Producing a PDF

Another bash script produces readable documentation from the cap source text in advance of the cap system being able to do so itself. In this way the system was developed using its own text markup and coding conventions.

The PDF produced by this method is not identical to but is indicative of what would be produced by the cap system and is sufficient for holistic development in absence of a native cap system.

The following pipeline of sed regexps incrementally transforms the i source text document markup strings into latex document markup and then calls pdf2latex to produce a PDF document.

```
#!/bin/bash
# This is i2l.sh which transforms i source text to latex format and then creates a pdf
cat cap.bookpre.txt cap.i cap.changelog.i | sed -e '
s/\[\:2fig......../\\begin\{figure\}\[ht\] /' | sed -e '
s/\[\:epdf . \"\(.*\)\.pdf\"..\"\(.*\)\" \]./\\includegraphics\[width=\2in\]\{\1\.pdf\} /' |sed -e '
s/.\[\:figr./\\hspace\{0.5cm\} /' | sed -e '
s/..\[\:figc..Caption...../\\end\{figure\} /'| sed -e '
s/\[\].\]\;//' | sed -e 's/\"\]\;//' | sed -e '
s/\[\:para.../         /' | sed -e '
s/\[\:.\[\^cap\^para\:capml....../         /' | sed -e '
s/\[\:codh./\\begin\{minipage\}\[h\]\{.4\\textwidth\} \\begin\{code\}\\normalsize\\begin\{vbtm\}/' | sed -e
s/\[\:code./\\noindent\\begin\{minipage\}\[h\]\{1.1\\textwidth\}  \\begin\{code\}\\normalsize\\begin\{vbtm\
s/\[\:skip.\]./\\smallskip/' | sed -e '
s/\".\[\:flush.\].\"/\\hspace\*\{\\fill\}/' | sed -e '
s/\[\:capt...\(.*\)\" \]\]\;/\\end\{vbtm\}\\end\{code} \\end\{minipage\}/' | sed -e '
s/\[\:xtnd.*\~./\\noindent\\begin\{minipage\}\[h\]\{1.1\\textwidth\} \\begin\{code\}\\normalsize\\begin\{vb
s/\[\:xtni.*\"/\\begin\{minipage\}\[h\]\{.9\\textwidth\} \\begin\{code\}\\normalsize\\begin\{vbtm\}/' | sed
s/\[\:xtnt.*\"/\\noindent\\begin\{minipage\}\[h\]\{1.0\\textwidth\} \\begin\{ptext\}\\normalsize\\begin\{vb
s/\[\:xtns.*\"/\\noindent\\begin\{minipage\}\[h\]\{1.0\\textwidth\} \\begin\{p2text\}\\normalsize\\begin\{v
s/\[\:xtne.*\"/\\begin\{minipage\}\[h\]\{.9\\textwidth\} \\begin\{example\}\\normalsize\\begin\{vbtm\}/' |
s/.*\[\:xcpt...\(.*\)\" \]\]\;/\\end\{vbtm\}\\end\{ptext\} \\end\{minipage\}/'| sed -e '
s/.*\[\:xcps...\(.*\)\" \]\]\;/\\end\{vbtm\}\\end\{p2text\} \\end\{minipage\}/'| sed -e '
s/.*\[\:xcpe...\(.*\)\" \]\]\;/\\end\{vbtm\}\\end\{example\} \\end\{minipage\}/'| sed -e '
s/.*\[\:xcpd...\(.*\)\" \]\]\;/\\end\{vbtm\}\\end\{code\} \\end\{minipage\}/' | sed '42,50d' | sed -e '
s/\[\:bleah...Give up converting now...\]\;/\\end\{document\}/' | sed -e '
s/.\]\;\[.\(.*\).\:sect../\\section\{\1\}/'| sed -e 's/.\]\;\[.\(.*\).:...cap.sect\:capml...../\\section\{\
s/.\[\:.\[.cap.prel\:capml...../\\section\*\{\}/' | sed -e '
s/.\[ .\(.*\). \:.\[.cap.chap\:capml.\]../\\chapter\{\1\}/' | sed -e '
s/.\].\;.\[ .\(.*\). \:chap../\\chapter\{\1\}/' | sed -e '
s/.\[\:prel../\\section\*\{\}/' | sed -e 's/\[\:bnot. .\(.*\). \] \;/\\framebox\{\\framebox\{\1\}\}/' | sed
s/\[\:mnot. .\(.*\). \] \;/\\marginpar\{\1\}/' | sed -e '
s/....mono..\(.*\). \] . ./\\texttt\{\1\}/' | sed -e '
s/....bold..\(.*\). \] . ./\\textbf\{\1\}/' | sed -e 's/....itlc..\(.*\). \] . ./\\emph\{\1\}/' | sed -e '
s/....ftnt..\(.*\). \] . ./\\footnote\{\1\}/' | sed -e '
s/.cap.doc . . \[req\:sdoc. . \]. . ..//' | sed -e '
s/\[\:stxt..cap.doc.req . ./\\fbox\{\\begin\{minipage\}\[h\]\{1.05\\textwidth\}/' | sed -e '
s/..\;\[\:scpt. .\(.*\). \]\]\;/\\end\{minipage\}\} \1/' | sed -e '
s/\[\:stxt..cap.doc.web . ./\\fbox\{\\begin\{minipage\}\[h\]\{1.05\\textwidth\}/' | sed -e '
s/..\;\[\:wend. .\(.*\). \]\]\;/\\end\{minipage\}\} \1/' | sed -e '
s/\\subsection*{\(.*\).}/\\subsection\*\{\1\}/' | sed -e '
s/\\subsection{\(.*\).}/\\subsection\{\1\}/' | sed -e '
s/\\subsection*{ }/\\subsection\*\{ \}/' | sed -e '
s/\\subsection*{ }/\\subsection\*\{ \}/' |  sed -e '
s/\[\:newp.\]./\\newpage /' > cap.latex; echo "\\end{document}" >> cap.latex; \
/usr/texbin/pdflatex cap.latex
```

i2l.sh needs a file with latex pre-configuration settings:

```
\documentclass{book}
\usepackage{amsfonts}
\usepackage{graphicx}
\usepackage{xcolor}
\usepackage{upquote}
\definecolor{mygreen}{rgb}{0.9,1,0.9}
\definecolor{mygrey}{rgb}{1,1,1}
\definecolor{myblue}{rgb}{0.9,0.9,1}
\definecolor{myred}{rgb}{1,0.9,0.9}
\makeatletter
\def\verbatim@font{\ttfamily\small}
\makeatother
\makeatletter\newenvironment{code}{%
   \noindent\begin{lrbox}{\@tempboxa}\begin{minipage}{\columnwidth}\setlength{%
   \parindent}{1em}}{\end{minipage}\end{lrbox}\colorbox{mygreen}{\usebox{\@tempboxa}}
}\makeatother
\makeatletter\newenvironment{ptext}{%
   \noindent\begin{lrbox}{\@tempboxa}\begin{minipage}{\columnwidth}\setlength{%
   \parindent}{1em}}{\end{minipage}\end{lrbox}\colorbox{myblue}{\usebox{\@tempboxa}}
}\makeatother
\makeatletter\newenvironment{p2text}{%
   \noindent\begin{lrbox}{\@tempboxa}\begin{minipage}{\columnwidth}\setlength{%
   \parindent}{1em}}{\end{minipage}\end{lrbox}\colorbox{myred}{\usebox{\@tempboxa}}
}\makeatother
\makeatletter\newenvironment{example}{%
   \noindent\begin{lrbox}{\@tempboxa}\begin{minipage}{\columnwidth}\setlength{%
   \parindent}{1em}}{\end{minipage}\end{lrbox}\colorbox{mygrey}{\usebox{\@tempboxa}}
}\makeatother
\setlength{\marginparwidth}{1.2in}
\setlength{\parskip}{0.5cm}
\setcounter{tocdepth}{0}
\let\oldmarginpar\marginpar
\renewcommand\marginpar[1]{\-\oldmarginpar[\raggedleft\footnotesize #1]%
{\raggedright\footnotesize #1}}
\begin{document}
\title{The Computer Applications Platform, 0th Edition}
\author{Copyright 2009-2012, Charles Perkins}
\date{December 2012}
\maketitle
\thispagestyle{empty}
\tableofcontents
```

4.3 Building a Parser Maker

The first step in bootstrapping is to build a scanner for the i syntax. A python program, in several parts, will serve the purpose. Another script will piece it together for us, and then call it:

```
#!/bin/bash
# This is cap.mkinterp.sh

#  cat ./common-chr.ibnf ./calc-example.ibnf > ./tmp.ibnf
#  cat ./common-chr.ibnf ./i.ibnf > ./itmp.ibnf
  cat ./ibnf.ibnf ./common-chr.ibnf > ./ibnftmp.ibnf
  cat ./six.ibnf ./common-chr.ibnf > ./sibnftmp.ibnf
#  grep ": " tmp.ibnf  > ./auto.smtx
#  grep ": " itmp.ibnf  > ./iauto.smtx
  grep ": " ibnftmp.ibnf  > ./ibnfauto.smtx
#  grep "/ " tmp.ibnf  >> ./auto.smtx
#  grep "/ " itmp.ibnf  >> ./iauto.smtx
  grep "/ " ibnftmp.ibnf  >> ./ibnfauto.smtx
#  python autosmtx.py auto.smtx auto.py
#  python autosmtx.py iauto.smtx iauto.py
  python autosmtx.py ibnfauto.smtx ibnfauto.py
  chmod 755 i2py.sh
#  ./i2py.sh tmp.ibnf tmp1.py
#  ./i2py.sh itmp.ibnf itmp1.py
  ./i2py.sh ibnftmp.ibnf ibnftmp1.py
  chmod 755 s2py.sh
  ./s2py.sh smtx.py tmp2.py
#  ./s2py.sh i.smtx itmp2.py
  ./s2py.sh ibnf.smtx ibnftmp2.py
#  cat ./pro.py ./tmp1.py ./tmp2.py ./auto.py epi.py | sed -e '/^$/d' > parser.py
#  cat ./pro.py ./itmp1.py ./itmp2.py ./iauto.py epi.py | sed -e '/^$/d' > iparser.py
  cat ./pro.py ./ibnftmp1.py ./ibnftmp2.py ./ibnfauto.py epi.py | sed -e '/^$/d' > ibnfparser.py
#  chmod 755 parser.py
#  chmod 755 iparser.py
  chmod 755 ibnfparser.py
```

The script continues:

```
  echo "Making the interpreter and then calling it."

  python parser.py calc-example.input blank.txt test.out; cat test.out
#  python iparser.py test.i iauto.smtx test.out
#  cat test.i

  echo "constructing ibnf meta compiler using sed hack compiler"
  python ibnfparser.py ibnftmp.ibnf ibnf.smtx ibnfmeta.py

  echo "calling the ibnf meta compiler on its own ibnf specification"
  python ibnfmeta.py ibnftmp.ibnf ibnf.smtx ibnfmeta2.py

  echo "calling the ibnf meta compiler on ibnf of sibnf"
  python ibnfmeta.py sibnftmp.ibnf sibnf.smtx sixparser.py

  echo "calling the sibnf meta compiler on sibnf of ibnf"
  python sixparser.py ibnf.six-py sibnf.smtx ibnfmeta.smtx

  echo "calling the sibnf meta compiler on sibnf of sibnf"
  python sixparser.py six.six-py sibnf.smtx sibnfmeta.smtx

  echo "Building the calculator Parser"
  cat ./common-chr.ibnf ./calc-example.ibnf > ./calc-example-full.ibnf
  python sixparser.py calc-example.six-py blank.txt calc-example.smtx
  python ibnfmeta.py calc-example-full.ibnf calc-example.smtx calc-example.py
  python calc-example.py calc-example.input blank.txt calc.out
  cat calc.out


  echo "Building the ibnf Parser"
  python sixparser.py ibnf.six-py blank.txt ibnf-smtx.py
  cat ./common-chr.ibnf ./ibnf.ibnf > ./ibnf-full.ibnf
  python ibnfmeta.py ibnf-full.ibnf ibnf-smtx.py ibnf.py
  python ibnf.py ibnf-full.ibnf ibnf-smtx.py ibnfmeta2.py

  echo "Building toy compiler"
  python sixparser.py toy.six-py blank.txt toy-smtx.py
  cat ./common-chr.ibnf ./toy.ibnf > ./toy-full.ibnf
  python ibnfmeta.py toy-full.ibnf toy-smtx.py toy-compile.py
  python toy-compile.py test.toy blank.txt toy.out
  chmod 755 toy.out
  ./toy.out

  echo "Building i compiler"
  python sixparser.py i.six-py blank.txt i-smtx.py
  cat ./common-chr.ibnf ./i.ibnf > ./i-full.ibnf
  python ibnfmeta.py i-full.ibnf i-smtx.py i-compile.py
  python i-compile.py test.i blank.txt i.out
  chmod 755 i.out
  ./i.out
```

A python script prologue sets up the envirnoment for our parser:

```
import sys
from binascii import *
fi = file(sys.argv[1]).read(); h = {}; T=True; F=False

def n2z( a ):
  return ( '0' if a=='' else a )

def be2le( a ):
  return a[6:8]+a[4:6]+a[2:4]+a[0:2]

def inclast( a ):
  return a.rpartition(".")[0]+"."+str(int(a.rpartition(".")[2])+1)

def unesc( a ):
  return a.decode('string_escape')

def mark( p, s, t ):
  ( v, m, ss, l, c, a ) = t
  if t[1]:  x = p +"-" + str(s);  h[x]=( v, m, l, a ); return t
  else:
    if not t[0]:  x = p +"-" + str(s);  h[x]=( v, m, l, a ); return t
  return t
def been(p, s):
 if h.has_key( p +"-" + str(s) ): return h[p +"-" + str(s)][1]
 else:  return False

def was(c, p, s):
 ( v, m, l, a ) = h[ p +"-" + str(s) ]
 return (v, m, s, l, c, a)

def cm( ch, s, c ):
  if s < len(fi):
    if fi[s] == ch:    return ( True, True, s, 1,  c, ( "cm", fi[s] ) )
  return ( False, True, s, 0, c, ( "cm", "") )

def andmemo( m ):
  r = True
  for i in m:
    if not m[i]: r = False
  return r

outdata = ""

def output( s ):
  global outdata
  outdata = outdata + str(s)


registers={}
context={}
mseq = 0
dseq = 1

semantics = file(sys.argv[2]).read();
fo = open(sys.argv[3], "w+")
```

We need to convert grammars of the following form:

```
dgt     ? '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' ;
pnt     : dgt .pnt ;
```

Into python functions like the following:

```
def dgt_p( s ):
  if been("dgt",s): return was("dgt",s)
  else:
    mark("dgt",s,(False,s,0,"")); met = False
    if not met: (met, ts, tl, ta) = cm( '0', s)
    if not met: (met, ts, tl, ta) = cm( '1', s)
    if not met: (met, ts, tl, ta) = cm( '2', s)
    if not met: (met, ts, tl, ta) = cm( '3', s)
    if not met: (met, ts, tl, ta) = cm( '4', s)
    if not met: (met, ts, tl, ta) = cm( '5', s)
    if not met: (met, ts, tl, ta) = cm( '6', s)
    if not met: (met, ts, tl, ta) = cm( '7', s)
    if not met: (met, ts, tl, ta) = cm( '8', s)
    if not met: (met, ts, tl, ta) = cm( '9', s)
    if not met: return mark("dgt",s,(False,s,0,""))
    else: return mark("dgt",s,(met, s, tl, ta))

def pnt_p( s ):
  if been("pnt",s): return was("pnt",s)
  else:
    mark("pnt",s,(False,s,0,"")); ok=True; ts=s; tl=0; a1=""; a2=""
    if ok: (ok, ts, tl, a1) =  dgt_p   ( ts+tl )
    if ok: (ack, ts, tl, a2) = pnt_p ( ts+tl )
    if ok: return mark("pnt",s,(True, s, ts+tl-s, pnt_s( a1+a2 )))
    return mark("imt",s,(False,s,0,""))
```

Another script will effect that transformation by sending the ibnf source text through a pipeline of sed regular expressions:

```
#!/bin/bash
# This is i2py.sh
  echo "Creating scanner for" $1 " as " $2
```

First rejoin lines broken at an alternative:

```
sed -e 's/: / /g' $1  |sed -e 's/ = / : /g' |sed -e 's/ \/ / : /g' | sed '/|$/N;s/\n *//'    | sed -e '
```

change = to : and append p to ibnf definitions and definition references:

```
/ [?:] /s/\([a-z]\) /\1_p /g'   |  sed -e "
```

Next create match-by-ascii-number functions for several special characters and a general character match function call for the rest:

```
s/'\\\\\\\\'/ cm(chr(92) ,s) /g"| sed -e "s/'\\\\''/ cm(chr(39) ,s) /g" | sed -e "
s/'\\\\\"'/ cm(chr(34) ,s) /g"  | sed -e "s/'\\\\t'/ cm(chr(9) ,s) /g"  | sed -e "
s/'\\\\n'/ cm(chr(10) ,s) /g"   | sed -e "s/'\(.\)'/ cm('\1' ,s) /g"| sed -e "
```

Next put spaces around character literal ibnf items:

```
s/'\(.\)'/ '\1' /g"        |  sed -e '
```

Turn ibnf alternatives into bash else-if clauses:

```
s/ | / ( s )insertnewline    if not met: (met, mem, ts, tl, tc, ta ) = /g' |  sed -e '
```

Turn ibnf definitions into bash functions:

```
s/^ *\([a-z]*_p\) *\([?:]\) *\(.*\); *$/def \1( s )\2\3}\2/' |  sed -e '
```

Prepend an if keyword for ibnf alternative bash functions:

```
s/( s )?/\( s ):insertnewline  if been("!1!",s): return was( c, "!2!",s)~:~/'  |  sed -e '
s/~:~/insertnewline  else:insertnewline~:~/'  |  sed -e '
s/~:~/    mark("!3!",s,(False,True,s,0,c,("",""))); met = Falseinsertnewline~:~/'  |  sed -e '
s/~:~/    if not met: (met, mem, ts, tl, tc, ta) = /'  |  sed -e '
```

Create ”if not met” statements for ibnf alternative bash functions:

```
s/}?/ ( s )insertnewline    if not met:  return mark("!4!",s,(False,True,s,0,c,("","")))}?/'  |
s/}?/insertnewline    else:        return mark("!5!",s,(met,mem,s,tl,tc,ta))/'   |  sed -e '
s/^def \(.*\)_p\(.*\)!1!\(.*\)!2!\(.*\)!3!\(.*\)!4!\(.*\)!5!\(.*\)$/def \1_p\2\1\3\1\4\1\5\1\6\
s/,s) *( s )/,s)/g'  |  sed -e '
s/_p ( s )/_p( s )/g'  |  sed -e '
s/_p  ( s )/_p( s )/g'  |  sed -e '
```

Put newlines in sequence definitions:

```
s/def \(.*\)_p( s ):\(.*\)}:/def \1_p( s ):!1!"\1"!2!"\1"!3!"\1"!4!\2!5!"\1"!6!\1!7!"\1"!8!/'   |  sed -e '
s/_p /_p ( ts+tl )insertnewline    !=!/g'        |  sed -e '
s/ ,s) / ,(ts+tl))insertnewline    !=!/g'        |  sed -e '
s/!1!/insertnewline  if been(/'        |  sed -e '
s/!2!/,s): return was( c, /'        |  sed -e '
s/!3!/,s)insertnewline  else:insertnewline    mark(/'        |  sed -e '
s/!4!/,s,(False,True,s,0,c,("",""))); ok=True; ts=s; tl=0; a={0: ("","")}; mem={0:True}; tc=c; n=0insertnew
s/!=! */!=!/g'        |  sed -e '
s/!=!!5!/if ok: return mark(/'        |  sed -e '
s/!6!/,s, /'        |  sed -e '
s/!7!/_s( a, andmemo(mem), s, ts+tl) )insertnewline    return mark(/'        |  sed -e '
s/!8!/,s,(False,True,s,0,c,("","")))/'        |  sed -e '
s/!=!\./if ok: n=n+1; (nok, mem[n],ts, tl,tc, a[n]) = /g'        |  sed -e '
s/!=!/if ok: n=n+1; (ok, mem[n], ts, tl, tc,a[n]) = /g'        |  sed -e "
```

Put semicolons after parsing calls and match calls in sequences then remove the colons:

```
#/):{/s/\(_p\)/\1 ; /g"      |  sed -e "
#/):{/s/\(_p\) ;  ( )/\1 ( )/g"      |  sed -e "
#/):{/s/\('.'\)/\1 ; /g"      |  sed -e '
#s/):{/) {/'                  |  sed -e '
s/passfail_p/passfail/'      |  awk '
```

Finally, place the newlines and write the file:

```
{ gsub ("insertnewline", "\n") }1' | sed -e '
s/def \(.*\)s ):/def \1s, c ):/g'  | sed -e '
s/    if ok: n=n+1;\(.*\))/    if ok: n=n+1;\1, tc)/g'  | sed -e '
s/    if ok: return\(.*\)) )/    if ok: return\1, tc) )/g'  | sed -e '
s/    if not met: (met\(.*\))/    if not met: (met\1, c)/g' | sed -e '
s/mark("\(.*\)",s, \(.*\)) )/mark("\1",s, \2,"\1") )/g'  | sed -e '
s/mark("\(.*\)",s,(False\(.*\)))/mark("\1",s,(False\2))/g'  > ./$2
```

Generate simple semantics:

```
import sys
from binascii import *

fi = open(sys.argv[1], "r")
fo = open(sys.argv[2], "w+")
line = fi.readline()
params = "_s(a,m,s,e,c,n):  return ( True, True, s, e-s, c,(n,"
tail = "]))"

while line:
 x = line.find(" /")
 if x > 0:
     print >> fo, "def " + line[0:x].strip() + params + "fi[s:e" + tail
 x = line.find(" =")
 if x > 0:
   y = line.count(":")
   if y == 1:
     t = line[x+2:].strip(); u = t[0:len(t)-1];  n=0; q=False
     for c in u:
       if c==':': v = n
       if (c!=' ') and not q: n = n + 1; q = True
       if (c==' ') and q: q = False
     print >> fo, "def " + line[0:x].strip() + params + "a["+ str(v) + tail
 line = fi.readline()
```

Another script will tranform grammer semantics into shell code operations:

```
#!/bin/bash
# This is s2py.sh

echo "Creating semantic actions for" $1 " as " $2
cp $1 $2
```

A parser implementation needs some basic functionality, and a command to kick it all off:

```


(v,m,s,l,c,a) = syntax_p( 0, ({},'<1>','<0>') )
if v:
  print "Parsed OK"
else: print "Failed to Parse"

print >> fo, a[1]

fo.close()
```

A calculator syntax will stand in for the i syntax to test things out:

The semantic actions for elements of a grammer are coded separately. The semantic actions for i.ibnf are as follows:

```
#!/bin/bash
# This is cap.ibnfsmntx.sh
```

We need a README file for upload of the source code to other repositories:

```
This is a README file for the Computer Applications Platform,
which is meant to be a self-sustaining computing environment
for commodity computers. The CAP system is intended to provide
features corresponding to compilers, operating systems, and
applications.

         This is revision 0 of the software,
          THE SYSTEM IS NOT YET FUNCTIONAL.

See cap.pdf for an exploration of the systems goals, status,
techniques, and implementation.

To bootstrap the system (as far as it is currently implemented)
in a Unix environment with bash, sed, awk, python, place the files
cap.i, xtf.sh, and xta.sh in a directory, make the
shell scripts executable, and then execute xta.sh

After bootstrapping the system, if you have latex installed on your
host system you can execute the i2l.sh script to produce the cap.pdf file.

The system is released under the MIT free software license:

Copyright (c) 2009-2011 Charles Perkins

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the
Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES
OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
OTHER DEALINGS IN THE SOFTWARE.
```

4.4 Constructing the System

Now the system executables can be created:

4.5 Uploading the Work in Progress

Another script uploads the source text and pdf file to a server:

Chapter 5

Documents

The system prints its own documentation and has text markup and domain specific languages for vector images and diagrams.

Chapter 6

User Interface

How the user interacts with the system.

Chapter 7

Networking

The whole is worth more than the sum of its parts.

Chapter 8

Applications for the Computer Platform

cap exists to get useful work done.

Chapter 9

Distributing the System

Few successful programming projects are the isolated product of a single individual. Frequently programming projects have web sites to coordinate the work of diverse contributors and to distribute the product of their efforts.

cap includes a template for project coordination web sites. Templates for other kinds of sites may be added later.

```
`cap,env,serv,web,template ~  (
   [ devprj 'a: a ? website ~ $ ];
  $ );
```

For deployment with actual content the template needs a site configuration.

```
`cap,env,serv,web,site ~  (
   [ caphub : website ~ $ ];
  $ );
```

The coordination web site template is composed of items that may be included in pages, and the pages themselves.

```
`cap,env,serv,web,template,(devprj 'a) ~  (
   [ items :~ $ ];
   [ pages :~ $ ];
  $ );
```

A vibrant development web site needs a welcoming overview home page, up-to-date news content, appropros helpful information, a place to discuss the project, a way to get the product, and a way to contribute enhancements The development project web site template includes six top-level pages, one for each of these areas.

```
`cap,env,serv,web,template,(devprj 'a),pages ~  (
   [ home       'b: b ? htdata ~ $ ];
   [ news       'b: b ? htdata ~ $ ];
   [ help       'b: b ? htdata ~ $ ];
   [ discuss    'b: b ? htdata ~ $ ];
   [ download   'b: b ? htdata ~ $ ];
   [ contribute 'b: b ? htdata ~ $ ];
  $ );
```

The pages have a common skeletal structure. In addition, Every page will have a banner at the top and a navigation menu to the left.

```
`cap,env,serv,web,template,(devprj 'a),items ~  (
   [ pageskel 'd :~ $ ];
   [ banner :~ a,bannertext ];
   [ navmenu :~ ( "<table>"; [ menuitems :~ $ ]; "</table>")];
  $ );
```

The text (and graphics) of the banner will be supplied when the template is applied to an actual project web site.

All of the pages have a banner at the top, a navigation menu to the left, and a main content area.

```
`cap,env,serv,web,template,(devprj 'a),items,(pageskel 'd) ~  (
   [ prologue :~ ("<html><head><title>"; d,title; "</title></head><body>") ];
   [ topsect :~ ( "<table><tr><td>"; banner; "</td></tr><tr><td><table><tr>")];
   [ leftsect :~ ( "<td>"; navmenu; "</td><td>") ];
   [ rightsect :~ d,content ];
   [ epilogue :~ ("</td></tr></table></td></tr></table></body><html>") ];
  $ );
```

The home page is first in the navigation menu.

```
`cap,env,serv,web,template,(devprj 'a),items,navmenu,menuitems ~  (
    "<tr><td><a href=\"/\">Home</a></td></tr>";
  $ );
```

In addition to the the navigation menu and the banner, the home page has its own title and contains static content.

```
`cap,env,serv,web,template,(devprj 'a),pages,(home 'b)~  (
   items,(pageskel [:~(
     [ title :~ (a,name; " home") ];
     [ content :~ a,hometext ] )]);
  $ );
```

News is similarly constructed.

```
`cap,env,serv,web,template,(devprj 'a),items,navmenu,menuitems ~  (
    "<tr><td><a href=\"/\">News</a></td></tr>";
  $ );

`cap,env,serv,web,template,(devprj 'a),pages,(news 'b)~  (
   items,(pageskel [:~(
     [ title :~ (a,name; " news" )];
     [ content :~ a,newstext ] )]);
  $ );
```

Same for Help.

```
`cap,env,serv,web,template,(devprj 'a),items,navmenu,menuitems ~  (
    "<tr><td><a href=\"/\">Help</a></td></tr>";
  $ );

`cap,env,serv,web,template,(devprj 'a),pages,(help 'b) ~  (
   items,(pageskel [:~(
     [ title :~ (a,name; " help" )];
     [ content :~ a,helptext ] )]);
  $ );

`cap,env,serv,web,template,(devprj 'a),pages,(discuss 'b)~  (
   items,(pageskel [:~(
     [ title :~ (a,name; " discuss" )];
     [ content :~ a,discusstext ] )]);
  $ );

`cap,env,serv,web,template,(devprj 'a),pages,(download 'b)~  (
   items,(pageskel [:~(
     [ title :~ (a,name; " download" )];
     [ content :~ a,downloadtext ] )]);
  $ );

`cap,env,serv,web,template,(devprj 'a),pages,(contribute 'b)~  (
   items,(pageskel [:~(
     [ title :~ (a,name; " contribute" )];
     [ content :~ a,contributetext ] )]);
  $ );

`cap,env,serv,web,site,caphub ~  (
   serv,web,template,(devprj [:~(
   [ name :~ "caphub" ];
   [ bannertext :~ "Internet Hub for the Computer Applications Platform" ];
   [ hometext :~ "Welcome to the Computer Applications Platform distribution and contribution w
   [ newstext :~ "Not much news yet." ];
   [ helptext :~ "Good luck!" ];
   [ discusstext :~ "The discussion boards are not yet open." ];
   [ downloadtext :~ "Download here." ];
   [ contributetext :~ "Upload here." ])]);
  $ );
```

Computer Applications Platform coordination web site

banner

The web page links the above items together.

Chapter 10

Interfacing to Legacy Data

A maximally useful system will interoperate well with existing, entrenched systems. For example, databases are a cornerstone of corporate information technology, and may not be trivially replaced but must instead be interfaced to. cap can be used to create a private intranet site which accesses legacy data using templates created in the previous chapter. A corporate employees benefits tracking intranet interface to SQL stored benefits data will provide an example.

```
`cap,env,serv,web,template ~  (
   [ intra 'a: a ? website ~ $ ];
  $ );
```

For deployment with actual content the template needs a site configuration:

```
`cap,env,serv,web,site ~  (
   [ bene : website ~ $ ];
  $ );
```

As with the development project web site, the benefits intranet site template is composed of items that may be included in pages, and the pages themselves.

```
`cap,env,serv,web,template,(intra 'a) ~  (
   [ items :~ $ ];
   [ pages :~ $ ];
  $ );
```

A corporate intranet site has a different focus than a public development project and the structure of the template reflects this. A default home page presents login and status information, and employee, supervisor, and administrator pages provide different views on the intranet information.

```
`cap,env,serv,web,template,(intra 'a),pages ~  (
   [ home      'b: b ? htdata ~ $ ];
   [ empview   'b: b ? htdata ~ $ ];
   [ superview 'b: b ? htdata ~ $ ];
   [ adminview 'b: b ? htdata ~ $ ];
  $ );
```

The pages have a common skeletal structure. In addition, Every page will have a banner at the top and a navigation menu to the left.

```
`cap,env,serv,web,template,(intra 'a),items ~  (
   [ pageskel 'pagedata :~ $ ];
   [ banner :~ a,bannertext ];
   [ navmenu :~ (  "<table>"; [ menuitems :~ $ ]; "</table>")];
  $ );
```

The text (and graphics) of the banner will be supplied when the template is applied to an actual project web site.

All of the pages have a banner at the top, a navigation menu to the left, and a main content area.

```
`cap,env,serv,web,template,(intra 'a),items,(pageskel 'pagedata) ~  (
   [ prologue :~ ("<html><head><title>"; pagedata,title; "</title></head><body>") ];
   [ topsect :~ ( "<table><tr><td>"; banner; "</td></tr><tr><td><table><tr>")];
   [ leftsect :~ ( "<td>"; navmenu; "</td><td>") ];
   [ rightsect :~ pagedata,content ];
   [ epilogue :~ ("</td></tr></table></td></tr></table></body><html>") ];
  $ );
```

The home page is first in the navigation menu.

```
`cap,env,serv,web,template,(intra 'a),items,navmenu,menuitems ~  (
    "<tr><td><a href=\"/\">Home</a></td></tr>"
  $ );
```

In addition to the the navigation menu and the banner, the home page has its own title and contains static content.

```
`cap,env,serv,web,template,(intra 'a),pages,(home 'b)~  (
   items,pageskel ([:~(
     [ title :~ (a,name; " home") ];
     [ content :~ a,hometext ] )]);
  $ );
```

Employee Data is similarly constructed.

```
`cap,env,serv,web,template,(intra 'a),items,navmenu,menuitems ~  (
    "<tr><td><a href=\"/\">Employee</a></td></tr>"
  $ );

`cap,env,serv,web,template,(intra 'a),pages,(empview 'b)~  (
   items,pageskel ([:~(
     [ title :~ (a,name; " employee" )];
     [ content :~ a,emptext ] )]);
  $ );
```

Same for Supervisors.

```
`cap,env,serv,web,template,(intra 'a),items,navmenu,menuitems ~  (
    "<tr><td><a href=\"/\">Supervisor</a></td></tr>"
  $ );

`cap,env,serv,web,template,(intra 'a),pages,(superview 'b)~  (
   items,pageskel ([:~(
     [ title :~ (a,name; " supervisor" )];
     [ content :~ a,supertext ] )]);
  $ );
```

Same for Administrators.

```
`cap,env,serv,web,template,(intra 'a),items,navmenu,menuitems ~  (
    "<tr><td><a href=\"/\">Administrator</a></td></tr>"
  $ );

`cap,env,serv,web,template,(intra 'a),pages,(adminview 'b)~  (
   items,pageskel ([:~(
     [ title :~ (a,name; " administrator" )];
     [ content :~ a,admintext ] )]);
  $ );
```

And the text of the pages must be provided:

```
`cap,env,serv,web,site,bene ~  (
   cap,env,serv,web,template,intra ([:~(
   [ name :~ "bene" ];
   [ bannertext :~ "Intranet site for corporate employee benefits tracking" ];
   [ hometext :~ "Welcome to the employee benefits web site." ];
   [ emptext :~ "Nothing for employees to see yet." ];
   [ supertext :~ "Nothing for supervisors to see yet." ];
   [ admintext :~ "Nothing for Administrators to see yet." ])]);
  $ );
```

Corporate intranet benefits tracking database.

banner

The web page links the above items together.

Chapter 11

Adapting to the world

Internationalization, Localization, etc.

Chapter 12

Appendix I – Goals

Claims and Fulfillment.