Chains of meaning in the STEPS system

Ian Piumarta

This material is based upon work supported in part by the National Science Foundation under Grant No. 0639876. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

VPRI Memo M-2009-011

Viewpoints Research Institute, 1209 Grand Central Avenue, Glendale, CA 91201 t: (818) 332-3001 f: (818) 244-9761

Chains of meaning in the STEPS system

ian@vpri.org

2009−10−21

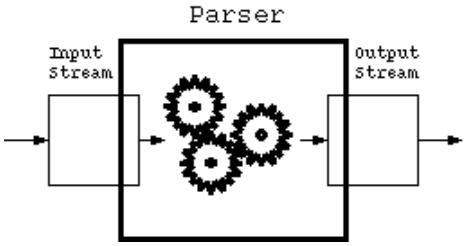
Contents:

1 Introduction

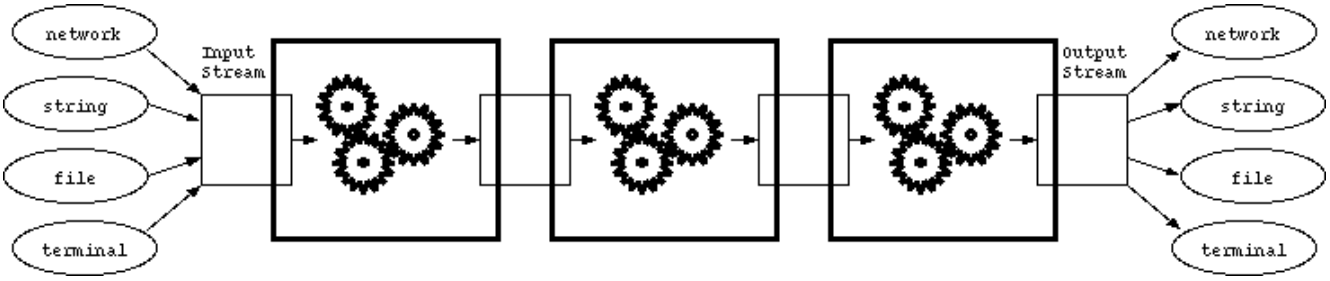
This memo illustrates the idea of a STEPS "Chain of Meaning" (CoM) using a complete example that converts a textual representation of abstract syntax trees into executable native code for the Intel 386. Simplicity and clarity at each stage in the chain are the primary goals of this example CoM.

2 Parsers, streams, pipelines and objects

Parsers operate on streams of objects. Each parser has an input stream and an output stream. The input stream is read, patterns of objects within it recognised, and corresponding actions performed. Actions can write objects to the parser's output stream.



Any stream can be used as input for a parser. A single object stream can be the output stream for one parser and the input stream for another. This creates pipelines of stages transformations, each stage being driven by a particular parser. The input end of a pipeline can be fed from a file input stream (converting the contents into a stream of integers) and the output end can be connected to file output stream (converting a sequence of objects back into a textual or binary form).

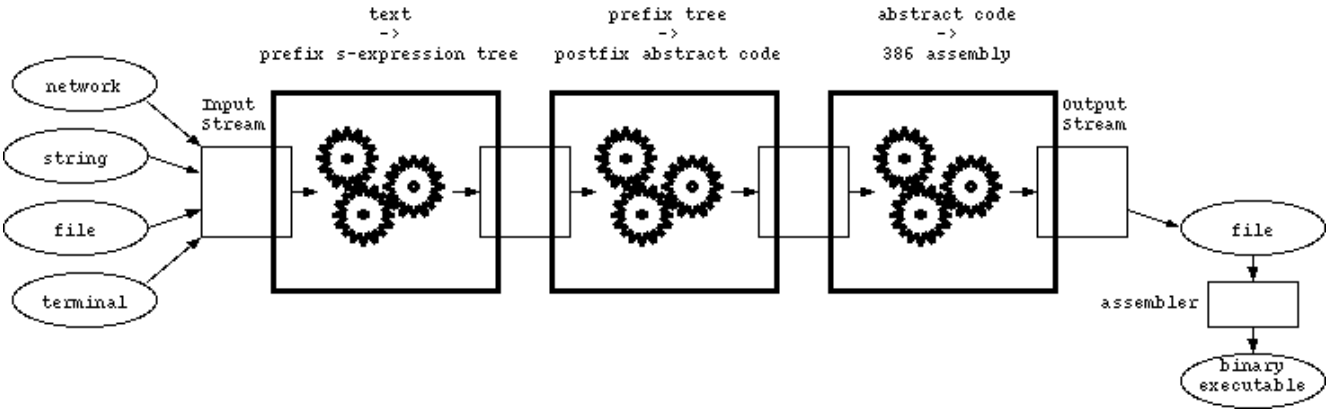


It should be clear that a Chain of Meaning is just a pipeline with higher-level abstractions entering on the left and lower-level ones exiting on the right.

Objects are either atomic or structured. Atomic objects include integers (text input stream, binary output stream), strings (treated as a single object), symbols (interned strings). Structured objects are ordered collections of objects.

Parsers can recognise patterns in both atomic and structured objects, and can generate both structured and atomic objects on their output stream. Structures are parsed as if their contents exist in their own local stream, with an "end of stream" condition at the end of the structure.

The remainder of this document describes a CoM for a simple s-expression language. Its CoM looks like this:



Note:

> *No explanation of the grammar of parsing expressions is given during the presentation. The appendices contain an overview of the expressions used to define the parsers in this document.*

3 A simple s-expression language

The goal is to create a CoM that can convert a text representation into an executable representation. For simplicity the input will be s-expressions (no irrelevant surface syntax) and the output will be 386 assembly language (no irrelevant complexity due to the construction of binary instructions). A small example s-expression program, shown below, will illustrate the CoM converting text to executable code.

```
(define nfibs
  (lambda (n)
    (if (< n 2)
      1
      (+ 1 (+ (nfibs (- n 1)) (nfibs (- n 2)))))))

(print (nfibs 32))
```

4 Stage one: file to text stream

The first CoM stage converts text from a file or string into a sequence of integer objects. It is of little interest and is omitted here.

5 Stage two: text to s-expression trees

The second CoM stage recognises s-expressions within the input text stream and creates corresponding tree-structured representations. Presented with the example program shown above, this stage should produce the following two structures on its output stream:

```
(print (nfibs 32))
(define nfibs (lambda (n) (if (< n 2) 1 (+ 1 (+ (nfibs (- n 1)) (nfibs (- n 2)))))))
```

The first rule tells the stage to look for the text of an entire s-expression on the input whenever the output stage needs to be refilled.

```
start    = sexpr
```

The next few rules deal with whitespace and comments. A blank is a space, tab or newline character. Comments run from a semicolon to the end of the line. Individual tokens are separated by any number of blanks or comments; the corresponding rule is given the very short name "_" because it is used frequently.

```
blank    = [ \t\n\r]
comment  = ";" (!eol .)*
eol      = ("\n" "\r"*) | ("\r" "\n"*)
_        = (blank | comment)*
```

Next come the lexical elements of the language. Letters (characters within identifiers) will be any alphabetic or punctuation character. Digits are the usual 0 through 9. Symbols are made of a letter followed by letters and digits. A number is made from a sequence of digits.

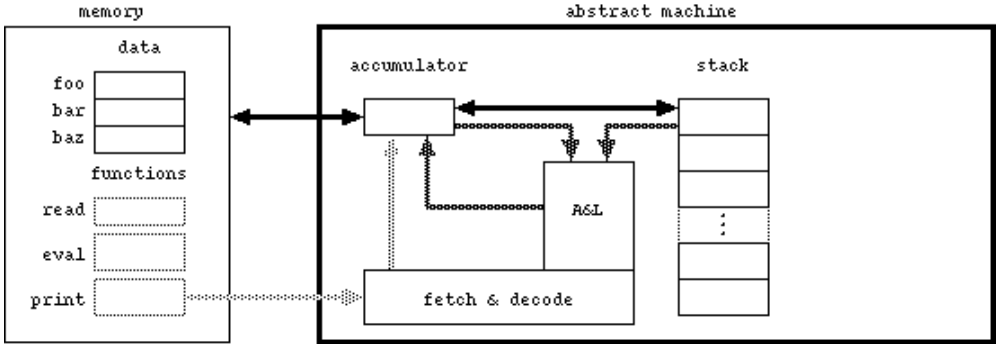
```
letter   = [-+!\$\%&*./:<=>?@A-Z\\^_a-z|~]
digit    = [0-9]

symbol   = ( letter (letter | digit)* ) $$ :s _ -> :s
number   = digit+ $#10 :n _ -> :n
```

An s-expressions is either an atom (symbol or number) or list, containing zero or more s-expressions surrounded by parentheses.

```
sexpr    = _ (atom | list)
atom     = symbol | number
list     = "(" sexpr* :l _ ")" -> :l
```

6 Stage three: prefix tree to postfix abstract instructions

Tree-structured s-expressions are converted to a linear, postfix form that is easy to execute on a stack machine. The machine has an accumulator, a stack, and a memory containing named storage locations (global variables). Numbered labels identify the locations of function entry points and internal branch destinations.



Literal values can be loaded into the accumulator. Values can be moved between the accumulator and the top of the stack, and moved between the accumulator and a named memory location. Operators use the accumulator as their first operand and take their other (if any) from the top of the stack. Results are left in the accumulator.

This stage of the CoM converts the two tree-structured objects produced by stage two into the following sequence of objects which form an equivalent abstract machine program.

```
(label 3
 enter
 load-long 2 save load-arg 0 less branch-false 1
 load-long 1 branch 2
 label 1
 load-long 2 save load-arg 0 sub save load-var nfibs call 1 save
```

```
load-long 1 save load-arg 0 sub save load-var nfibs call 1 add save
load-long 1 add
label 2
leave
main
long nfibs load-label 3 store-var nfibs
load-long 32 save load-var nfibs call 1 save load-var print call 1
exit)
```

The operation of each instruction is as follows.

instruction	***operation***
`label` *integer*	define the location of a numbered label
`long` *symbol*	create a named memory location (global variable)
`label` *symbol*	create a named memory location (global variable)
`load-long` *integer*	place a literal integer in the accumulator
`load-var` *symbol*	copy the value stored in the named memory location to the accumulator
`load-label` *symbol*	copy the address of the named memory location to the accumulator
`load-arg` *integer*	copy the value stored in the numbered argument to the accumulator
`save`	push the value in the accumulator onto the stack
`add`	pop the top of the stack and add it to the accumulator
`sub`	pop the top of the stack and subtract it from the accumulator
`less`	pop the top of the stack and compare it with the accumulator; set the accumulator to 1 if it was less than the stack item, zero otherwise
`store-var`	copy the value in the accumulator into the named memory location
`call` *integer*	call the address in the accumulator as a function with the given number of actual arguments
`enter`	create a new function activation record in the stack
`leave`	return from the most recent function activation
`branch` *integer*	transfer control to the numbered label
`branch-false` *integer*	transfer control to the numbered label if the accumulator is zero

The first rule in the parser tells the stage to look for an object encoding an `expr`ession on the input stream whenever the output stream needs to be refilled.

```
start = expr
```

The next three rules are for convenience: `long` and `name` recognise and yield an integer or symbol object; `arity` yields a count of the number items left in the current stream.

```
long    = &`long?   .
name    = &`symbol? .
arity   = .*:x                          ->`(list-length x)
```

Two kinds of lists occur within s-expressions: actual `args` to function calls and formal `params` in function definitions.

Each actual argument to a function call is an expression that must be saved on the stack as soon as it is available. Inserting a `save` instruction after each argument expression accomplishes this.

```
args    = expr:e args:a                 -> (::a ::e save)
        | expr:e                        -> (::e save)
        |                               -> ()
```

Each formal parameter is declared (to a supporting function `arg-name`) to differentiate it from a global variable during the compilation of a function body. The rule is written so that parameters are declared from right to left.

```
params  = ( name:h params:t | name:h )     ->`(arg-name h)
```

An expression is one of the objects or structures produced as output by stage one.

Integer literals are trivial and symbols name either a global or local (function parameter) variable. They appear verbatim in the tree and are converted into a corresponding `load` instruction. The support function `is-arg` differentiates between a local and global name.

```
expr    =    long:x                     -> (load-long :x)
        |    name:x &`(is-arg x):n       -> (load-arg  :n)
        |    name:x                      -> (load-var  :x)
```

Three binary operators are used in the example program. All operators are "applied" like functions and so appear inside a nested structure. The corresponding rule must match the start of this nested structure before checking for the operator. Operands are "evaluated" right to left, and the second must be saved before the first overwrites it. When both operands are available the instruction corresponding to the operator is emitted.

```
        | '( '< expr:x expr:y )          -> (::y save ::x less)
        | '( '+ expr:x expr:y )          -> (::y save ::x add)
        | '( '- expr:x expr:y )          -> (::y save ::x sub)
```

Three "special forms" must be dealt with before function calls.

`(define` *`name value`*`)` creates a global variable by reserving a data memory location wide enough to store a value and named according to the variable, then storing the result of evaluating the initialiser expression into the location.

```
        | '( 'define name:n expr:e )     -> (long :n ::e store-var :n)
```

`(lambda (`*`args...`*`)` *`expr...`*`)` creates a function value. The sequence of instructions corresponding to the expressions in the body of the function are created, delimited by `enter` and `leave` instructions (function prologue and epilogue, respectively). This sequence

must not be placed in the program at the point it occurs. It is saved for out-of-line compilation by the support function `save-lambda` which returns a unique label identifying the entry point. The address of this label is the "value" of the lambda expression, and a load of its address is compiled in-line in the code in place of the entire lambda expression.

```
| '( 'lambda '(params) expr*:b )   -> (enter :::b leave):l
                                   ->`(save-lambda l):n
                                   -> (load-label :n)
```

(`if` *condition consequent alternate*) evaluates the consequent if the condition is true, the alternate if not. Two labels are required, for the branch from the condition to the alternate clause and from the end of the consequent clause to the end of the entire expression. The labels are generated as unique integers by the support function `new-label`.

```
| '( 'if expr:t expr:x expr:y )    ->`(new-label):a ->`(new-label):b
                                   -> (          ::t branch-false :a
                                                 ::x branch :b
                                        label :a ::y
                                        label :b )
```

Function calls are a sequence whose first expression yields a function address to be called, with the remaining expressions in the structure being the actual arguments. The `call` instruction is told the number of actual arguments so that it can clean up the stack after the call returns.

```
| '( expr:f &arity:n args:a )      -> (::a ::f call :n)
```

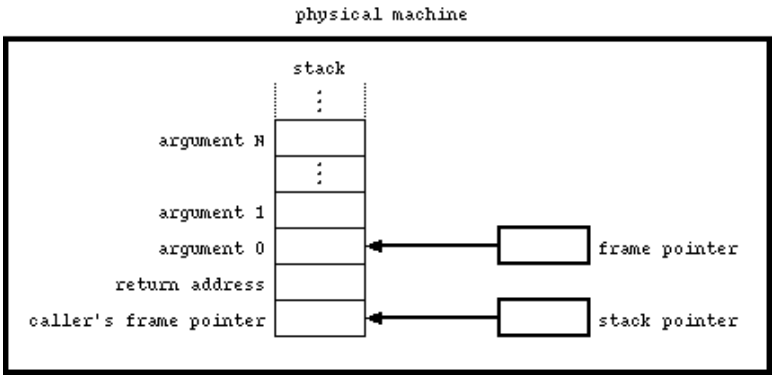
Anything else that occurs in a the post-fix abstract code is an error.

```
|    .:x                           ->`(error "unrecognised expression: " x)
```

7 Stage four: abstract instructions to Intel 386 assembler

For execution on real hardware, the abstraction is extended with a stack pointer (identifying the topmost item on the stack) and a frame pointer (identifying the start of the current function activation record).

Function arguments are passed on the stack. Each function activation saves the caller's frame pointer and return address in the stack, loads the frame pointer with the address of the first actual argument (numbered 0) and loads the stack pointer with the address of the saved return address (which is now the topmost item on the stack). The stack grows downward, towards lower memory addresses.



Machine registers are reserved for the accumulator, stack pointer and frame pointer.

register	***assignment***
`%eax`	accumulator
`%ebx`	stack pointer
`%esi`	frame pointer

The s-expression

```
(print (nfibs 32))
```

which stage 2 has transformed into an abstract program

```
( load-long 32 save load-var nfibs call 1 save load-var print call 1 )
```

will be transformed into native code

```
movl    $32, %eax        ; load-long 32
subl    $4, %ebx         ; save
movl    %eax, (%ebx)
movl    _V_nfibs, %eax   ; load-var nfibs
call    *%eax            ; call 1
addl    $4, %ebx
subl    $4, %ebx         ; save
movl    %eax, (%ebx)
movl    _V_print, %eax   ; load-var print
call    *%eax            ; call 1
addl    $4, %ebx
```

by this stage.

Stage four recognises each abstract instruction in a stream of instructions produced by the previous stage, emitting the corresponding assembly language as a side effect of recognition.

```
start =  insn*
```

Labels are numeric and must be given a prefix.

```
insn  = 'label .:l                    `"L\#l:"
```

Space for global variables is allocated in the data segment. Each global variable name is prefixed with `_V_` to lessen the danger of contention with externally-defined symbols.

```
| 'long .:n                    `"          .data"
                               `"_V_\$n:   .long 0"
                               `"          .text"
```

The `load` instructions copy their operand into the accumulator and `store` copies the accumulator into memory.

```
| 'load-long  .:l              `"          movl    $\#l, %eax"
| 'load-arg   .:n ->`(* 4 n):n `"          movl    (\#n)(%esi), %eax"
| 'load-var   .:n              `"          movl    _V_\$n, %eax"
| 'load-label .:n              `"          movl    $L\#n, %eax"
| 'store-var  .:n              `"          movl    %eax, _V_\$n"
```

The `save` instruction pushes the accumulator onto the stack.

```
| 'save                        `"          subl    $4, %ebx"
                               `"          movl    %eax, (%ebx)"
```

Arithmetic operators perform an operation between the stack and the accumulator, then pop the stack. Relational operators generate a concrete value (zero or non-zero) in the accumulator.

```
| 'add                         `"          addl    (%ebx), %eax"
                               `"          addl    $4, %ebx"
| 'sub                         `"          subl    (%ebx), %eax"
                               `"          addl    $4, %ebx"
| 'less                        `"          cmpl    (%ebx), %eax"
                               `"          setl    %al"
                               `"          movzbl  %al, %eax"
                               `"          addl    $4, %ebx"
```

Branches transfer control to a numbered label. Conditional branches test the accumulator for zero.

```
| 'branch .:l                  `"          jmp     L\#l"
| 'branch-false .:l            `"          cmpl    $0, %eax"
                               `"          je      L\#l"
```

Functions are applied by calling the computed destination address in the accumulator. Actual arguments are popped from the stack on return.

```
| 'call   .:n ->`(* 4 n):n     `"          call    *%eax"
                               `"          addl    $\#n, %ebx"
```

Function prologue retrieves the return address and pushes it onto the stack along with the caller's frame pointer. A new frame pointer is set up for the callee.

```
| 'enter                       `"          popl    %ecx"
                               `"          movl    %ecx, -4(%ebx)"
                               `"          movl    %esi, -8(%ebx)"
                               `"          movl    %ebx, %esi"
                               `"          subl    $8, %ebx"
```

Function epilogue undoes the prologue, popping the caller's frame pointer and return

address from the stack.

```
| 'leave                        `"          movl    %esi, %ebx"
                                `"          movl    -8(%ebx), %esi"
                                `"          pushl   -4(%ebx)"
                                `"          ret"
```

The program begins execution at the `main` instruction which tidies up the C stack and allocates a stack (using `malloc`) for the compiled program to use. The stack and frame pointers are initialised to point to the end of the stack memory. (The variable `_PREFIX` is defined to a string containing the prefix, if any, required for symbols in the C namespace on the target platform.)

```
| 'main                         `"          .globl  \${_PREFIX}main"
                                `"\${_PREFIX}main:"
                                `"          leal    4(%esp), %ecx"
                                `"          andl    $-16, %esp"
                                `"          pushl   -4(%ecx)"
                                `"          pushl   %ebp"
                                `"          movl    %esp, %ebp"
                                `"          pushl   %ecx"
                                `"          subl    $20, %esp"
                                `"          movl    $1024, (%esp)"
                                `"          call    \${_PREFIX}malloc"
                                `"          leal    1024(%eax), %esi"
                                `"          leal    -8(%esi), %ebx"
```

Execution finishes with an `exit` instruction, by performing a return from the C stack.

```
| 'exit                         `"          addl    $20, %esp"
                                `"          popl    %ecx"
                                `"          popl    %ebp"
                                `"          leal    -4(%ecx), %esp"
                                `"          movl    $0, %eax"
                                `"          ret"
```

The utility function `print` called from the example program is hand-written and placed at the end of the program.

```
                                `"print:    popl    -4(%ebx)"
                                `"          movl    %esi, -8(%ebx)"
                                `"          movl    $_S_fmti, (%esp)"
                                `"          movl    (%ebx), %eax"
                                `"          movl    %eax, 4(%esp)"
                                `"          call    \${_PREFIX}printf"
                                `"          movl    -8(%ebx), %esi"
                                `"          pushl   -4(%ebx)"
                                `"          ret"
                                `"          .data"
                                `"_V_print: .long   print"
                                `"_S_fmti:  .asciz  \"%d\\\""
                                `"          .text"
```

Anything else appearing in the stream of instructions indicates an error in the implementation.

```
| .:x                           `(error "unrecognised instruction: " x)
```

```
)
```

When presented with the output from stage 3, this stage produces the following text on its output stream:

```
L3:
        popl    %ecx
        movl    %ecx, -4(%ebx)
        movl    %esi, -8(%ebx)
        movl    %ebx, %esi
        subl    $8, %ebx
        movl    $2, %eax
        subl    $4, %ebx
        movl    %eax, (%ebx)
        movl    (0)(%esi), %eax
        cmpl    (%ebx), %eax
        setl    %al
        movzbl  %al, %eax
        addl    $4, %ebx
        cmpl    $0, %eax
        je      L1
        movl    $1, %eax
        jmp     L2
L1:
        movl    $2, %eax
        subl    $4, %ebx
        movl    %eax, (%ebx)
        movl    (0)(%esi), %eax
        subl    (%ebx), %eax
        addl    $4, %ebx
        subl    $4, %ebx
        movl    %eax, (%ebx)
        movl    _V_nfibs, %eax
        call    *%eax
        addl    $4, %ebx
        subl    $4, %ebx
        movl    %eax, (%ebx)
        movl    $1, %eax
        subl    $4, %ebx
        movl    %eax, (%ebx)
        movl    (0)(%esi), %eax
        subl    (%ebx), %eax
        addl    $4, %ebx
        subl    $4, %ebx
        movl    %eax, (%ebx)
        movl    _V_nfibs, %eax
        call    *%eax
        addl    $4, %ebx
        addl    (%ebx), %eax
        addl    $4, %ebx
        subl    $4, %ebx
        movl    %eax, (%ebx)
        movl    $1, %eax
        addl    (%ebx), %eax
        addl    $4, %ebx
L2:
        movl    %esi, %ebx
        movl    -8(%ebx), %esi
        pushl   -4(%ebx)
        ret
```

```
          .globl  main
main:
          leal    4(%esp), %ecx
          andl    $-16, %esp
          pushl   -4(%ecx)
          pushl   %ebp
          movl    %esp, %ebp
          pushl   %ecx
          subl    $20, %esp
          movl    $1024, (%esp)
          call    malloc
          leal    1024(%eax), %esi
          leal    -8(%esi), %ebx
          .data
_V_nfibs: .long 0
          .text
          movl    $L3, %eax
          movl    %eax, _V_nfibs
          movl    $32, %eax
          subl    $4, %ebx
          movl    %eax, (%ebx)
          movl    _V_nfibs, %eax
          call    *%eax
          addl    $4, %ebx
          subl    $4, %ebx
          movl    %eax, (%ebx)
          movl    _V_print, %eax
          call    *%eax
          addl    $4, %ebx
          addl    $20, %esp
          popl    %ecx
          popl    %ebp
          leal    -4(%ecx), %esp
          movl    $0, %eax
          ret
print:    popl    -4(%ebx)
          movl    %esi, -8(%ebx)
          movl    $_S_fmti, (%esp)
          movl    (%ebx), %eax
          movl    %eax, 4(%esp)
          call    printf
          movl    -8(%ebx), %esi
          pushl   -4(%ebx)
          ret
          .data
_V_print: .long  print
_S_fmti:  .asciz  "%d\n"
          .text
```

8 Performance is not the goal

The above code runs at 70% the speed of the same program written in C and compiled with typical optimisation (`gcc-4.3 -O2`) on Intel Core and Core2 processors.

Several peephole optimisations are possible while converting the sequence of abstract instructions to concrete instructions. Their definitions are obvious and their impact on performance should be easily measurable.

Stages two, three and four contain less than 100 lines of "code" and are close to the simplest possible non-trivial CoM that transforms text input describing high-level semantics into machine code output. A complete, uncommented listing is given in the appendices.

Appendices

9 Parsing expression syntax

Note:

> *Only the subset of STEPS parsing expressions needed to understand this memo are explained here. Refer to the parser documentation for full details.*

Each rule has the form

```
name = e
```

where `e` is a parsing expression. An expression has two properties: success and a value. A rule fails (does not succeed) when the first expression in it fails; if all expressions succeed then the rule succeeds. Every expression can yield a value; if a rule succeeds then its value is that of the last expression "evaluated" within it. If an expression fails, its value is undefined.

9.1 Primitive expressions

expression	***matches***	***value***
	nothing (always succeeds)	undefined
`.`	any object (fails at the end of the stream)	the object matched
`[A-Za-z]`	any letter (integer)	the letter matched
`"abc"`	a sequence of letters (integers)	the sequence matched
`( e )`	the expression `e`	the value of `e`
`' symbol`	a literal symbol object	the object matched
`' ( e )`	a structure whose contents match `e`	`e`
`-> o`	always	the value of the *output expression* `o`
`` ` "output string" ``	always	a sequence containing the generated output

9.2 Prefix operators

operator	*matches*	*value*
`& e`	`e`, without discarding the related input	`e`
`! e`	not `e`, without discarding the related input	undefined
`&`*`predicate`*	the current input object, iff *predicate* is true, without discarding the input	the input object matched

9.3 Postfix operators

operator	*matches*	*value*
`e *`	zero or more occurrences of `e`	a sequence of the values of each `e` matched
`e +`	one or more occurrences of `e`	a sequence of the values of each `e` matched
`e $$`	`e`	the symbol interned from the value of `e`
`e $#`*`base`*	`e`	the value of `e` converted to a number in the given base

9.4 Binary operators

operator	*matches*	*value*
`e1 e2`	`e1` and then `e2`	`e2`
`e1 | e2`	`e1` otherwise `e2`	the first `e` matched
`e1 :` *`name`*	`e1`	`e1` after storing it in *name*

9.5 Output expressions

Output expressions construct a new value while evaluating a rule. They all have the form

```
-> output-expression
```

where *output-expression* constructs an object as follows:

output expression	*value*
`symbol`	the given literal symbol

`( expression... )`	a sequence (structure) containing zero or more output *expression*s
`:name`	the object stored in the *name*d variable
`::name`	the objects in the sequence stored in the *name*d variable, spliced in-line into the enclosing output sequence
`:::name`	the objects in the sequences in the sequence stored in the *name*d variable, spliced in-line into the enclosing output sequence
`` `(host-expression) ``	the result of evaluating the *host-expression* (in the STEPS system this is a COLA "amino" expression)

Output expressions construct a new "current value" while scanning left-to-right through a rule. They do not write anything to the output stream (except for the value of the `start` rule, which is written implicitly to the output stream after a successful match).

Unstructured output (a sequence of integers) can be constructed from an output string expression. When not preceded by `->` the contents of the output string is written to the parser's output stream as a sequence of characters. Output strings have the form

```
` "output-characters"
```

where each *output-character* is as follows:

character	***value***
`\n`	a newline character
`\r`	a carriage-return character
`\t`	a tab character
`\"`	a double quote character
`\${name}`	the characters formed by converting the object stored in the *name*d variable to a string
`\#{name}`	the characters formed by converting the object stored in the *name*d variable to an integer
character	any other character is copied verbatim to the output

The braces around the variable *name*s can be omitted if that does not create any ambiguity (the *name* is followed by a non-identifier character). The following two output strings are equivalent.

```
`"L\#{number}:"
`"L\#number:"
```

10 Code listings

10.1 Stage two: s-expression text to prefix tree

```
blank    = [ \t\n\r]
comment  = ";" (!eol .)*
eol      = ("\n" "\r"*) | ("\r" "\n"*)
_        = (blank | comment)*

letter   = [-+!\$\%&*./:<=>?@A-Z\\^_a-z|~]
digit    = [0-9]

symbol   = ( letter (letter | digit)* ) $$ :s _ -> :s
number   = digit+ $#10 :n _ -> :n

sexpr    = _ (atom | list)
atom     = symbol | number
list     = "(" sexpr* :l _ ")" -> :l

start    = sexpr
```

10.2 Stage three: prefix tree to postfix abstract code

```
long    = &`long?   .
name    = &`symbol? .
arity   = .*:x                          ->`(list-length x)

args    = expr:e args:a                 -> (::a ::e save)
        | expr:e                        -> (::e save)
        |                               -> ()

params  = ( name:h params:t | name:h )  ->`(arg-name h)

expr    =    long:x                     -> (load-long :x)
        |    name:x &`(is-arg x):n       -> (load-arg  :n)
        |    name:x                     -> (load-var  :x)
        | '( '< expr:x expr:y )         -> (::y save ::x less)
        | '( '+ expr:x expr:y )         -> (::y save ::x add)
        | '( '- expr:x expr:y )         -> (::y save ::x sub)
        | '( 'define name:n expr:e )    -> (long :n ::e store-var :n)
        | '( 'lambda '(params) expr*:b ) -> (enter :::b leave):l
                                        ->`(save-lambda l):n
                                        -> (load-label :n)
        | '( 'if expr:t expr:x expr:y ) ->`(new-label):a ->`(new-label):b
                                        -> (          ::t branch-false :a
                                                      ::x branch :b
                                            label :a ::y
                                            label :b )
        | '( expr:f &arity:n args:a )   -> (::a ::f call :n)
        |    .:x                        ->`(error "unrecognised expression: " x)

start = expr
```

10.3 Stage four: postfix abstract code to Intel assembly

```
start =  insn*
```

```
insn  = 'label .:l                    `"L\#l:"
      | 'long .:n                     `"          .data"
                                      `"_V_\$n:   .long 0"
                                      `"          .text"
      | 'load-long  .:l               `"          movl    $\#l, %eax"
      | 'load-arg   .:n ->`(* 4 n):n  `"          movl    (\#n)(%esi), %eax"
      | 'load-var   .:n               `"          movl    _V_\$n, %eax"
      | 'load-label .:n               `"          movl    $L\#n, %eax"
      | 'store-var  .:n               `"          movl    %eax, _V_\$n"
      | 'save                         `"          subl    $4, %ebx"
                                      `"          movl    %eax, (%ebx)"
      | 'add                          `"          addl    (%ebx), %eax"
                                      `"          addl    $4, %ebx"
      | 'sub                          `"          subl    (%ebx), %eax"
                                      `"          addl    $4, %ebx"
      | 'less                         `"          cmpl    (%ebx), %eax"
                                      `"          setl    %al"
                                      `"          movzbl  %al, %eax"
                                      `"          addl    $4, %ebx"
      | 'branch .:l                   `"          jmp     L\#l"
      | 'branch-false .:l             `"          cmpl    $0, %eax"
                                      `"          je      L\#l"
      | 'call   .:n ->`(* 4 n):n      `"          call    *%eax"
                                      `"          addl    $\#n, %ebx"
      | 'enter                        `"          popl    %ecx"
                                      `"          movl    %ecx, -4(%ebx)"
                                      `"          movl    %esi, -8(%ebx)"
                                      `"          movl    %ebx, %esi"
                                      `"          subl    $8, %ebx"
      | 'leave                        `"          movl    %esi, %ebx"
                                      `"          movl    -8(%ebx), %esi"
                                      `"          pushl   -4(%ebx)"
                                      `"          ret"
      | 'main                         `"          .globl  \${_PREFIX}main"
                                      `"\${_PREFIX}main:"
                                      `"          leal    4(%esp), %ecx"
                                      `"          andl    $-16, %esp"
                                      `"          pushl   -4(%ecx)"
                                      `"          pushl   %ebp"
                                      `"          movl    %esp, %ebp"
                                      `"          pushl   %ecx"
                                      `"          subl    $20, %esp"
                                      `"          movl    $1024, (%esp)"
                                      `"          call    \${_PREFIX}malloc"
                                      `"          leal    1024(%eax), %esi"
                                      `"          leal    -8(%esi), %ebx"
      | 'exit                         `"          addl    $20, %esp"
                                      `"          popl    %ecx"
                                      `"          popl    %ebp"
                                      `"          leal    -4(%ecx), %esp"
                                      `"          movl    $0, %eax"
                                      `"          ret"
                                      `"print:    popl    -4(%ebx)"
                                      `"          movl    %esi, -8(%ebx)"
                                      `"          movl    $_S_fmti, (%esp)"
                                      `"          movl    (%ebx), %eax"
                                      `"          movl    %eax, 4(%esp)"
                                      `"          call    \${_PREFIX}printf"
                                      `"          movl    -8(%ebx), %esi"
                                      `"          pushl   -4(%ebx)"
                                      `"          ret"
```

```
                                    `"          .data"
                                    `"_V_print: .long   print"
                                    `"_S_fmti:  .asciz  \"%d\\""
                                    `"          .text"
        | .:x                       `(error "unrecognised instruction: " x)
        )
```