PEG-based transformer provides front-, middle- and back-end stages in a simple compiler

Ian Piumarta
Viewpoints Research Institute
ian@vpri.org

Abstract

Traditional compiler generators target a single stage within the compilation process. Each generator typically uses a dedicated specification language and the generated code often imposes restrictions on the program representation accepted as input or generated as output. This makes compilers larger and more complex than they need to be. We present a simple compiler that unifies specification and implementation of all its stages, using PEG-based transformations on a single, versatile representation. The resulting compiler is small, easy to understand, and highly suited to implementing its own implementation language.

1. Introduction

This paper describes an experiment in compiler construction that uses a single parser to implement each of the three stages of compilation: source parsing to create an AST, intermediate code generation from the AST, and machine code generation from the intermediate representation.[1]

Compilers are often broken into three (or more) stages: front-end parsing, one or more middle-end analysis/optimisation steps, and back-end code generation. This separation helps assure simple and understandable transformations between adjacent stages, that are easy to construct, debug and maintain.

The front-end parser converts program text into a structured form. This parser is typically generated automatically from a grammar using tools like Yacc [5] or ANTLR [8]. Analysis and optimisation is often performed by tree rewriting—pattern matching to identify a particular subtree and replacing it with a "better" subtree. The tree rewriter can also be generated from a grammar that describes output structure (trees) generated for particular statements of an input language (patterns in a tree). Code generators can be implemented with grammar-driven bottom-up tree rewriting, a well-known example being the BURG [3] family of code generators. Each stage usually uses a specialised grammar and parser/generator generator.

Our experiment applies a single parsing mechanism, driven by a uniform grammar, to all three compilation stages. Section 2 describes the architecture of the compilation chain. Section 3 illustrates the compilation chain using a complete example. Section 4 discusses the performance of the compiler. Section 5 presents related work and Section 6 concludes.

[1] 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 and do not necessarily reflect the views of the NSF.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

S3 2010 September 27, Tokyo, Japan
Copyright © 2010 ACM . . . $10.00

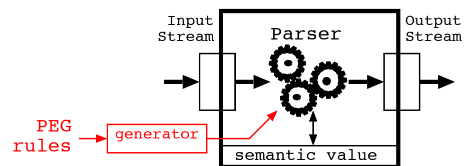

Figure 1. A parser has an input stream, a set of rules (generated from an extended PEG) that recognise input structure and generate output structures, an output stream to collect generated output, and a current result (semantic value from the most recent expression) that can be read and written within rules.

2. Architecture

A single data structure and parsing mechanism is used throughout the compilation chain, unifying the structure of the system. For structure we chose (Lisp-like) lists as they are easy to generate/manipulate and can represent many data structures: tables, streams, trees, and executable code (s-expressions)—i.e., programs at all levels of abstraction as well as the algorithms that operate on them. For parsing (pattern matching) we chose parsing expression grammars (PEGs) [2] as they are predictable, easy to write, and trivially converted into recursive-descent parsers. Simple extensions let them recognise structure within, and then generate, arbitrary lists of objects [1]. They can implement lexical as well as hierarchical syntax within the same grammar, eliminating the need for a separate tokeniser before syntax analysis proper.

Each stage in the compilation chain is a parser executing a set of PEG rules. Parsers have an input stream and an output stream (Figure 1). The input stream is read, patterns of objects within it recognised according to the parser's *matching expressions*, and *output expressions* executed to write objects on the output stream.

Each PEG rule is a function operating on a parser and a stream. The function returns true or false, indicating whether it recognised the input at the head of the stream. If matched, that input is (usually) consumed. If not, the stream position remains unchanged after the rule fails. Each parsing expression has a semantic value, which is stored as the parser's "current result." Expressions are provided for storing the current result in rule-local variables, whose scope is the current activation of the rule.

Objects are either atoms or lists. Atoms include integers (characters in text), strings (treated as a single object) and symbols (interned strings). Lists represent sequences of objects and tree-like structures.

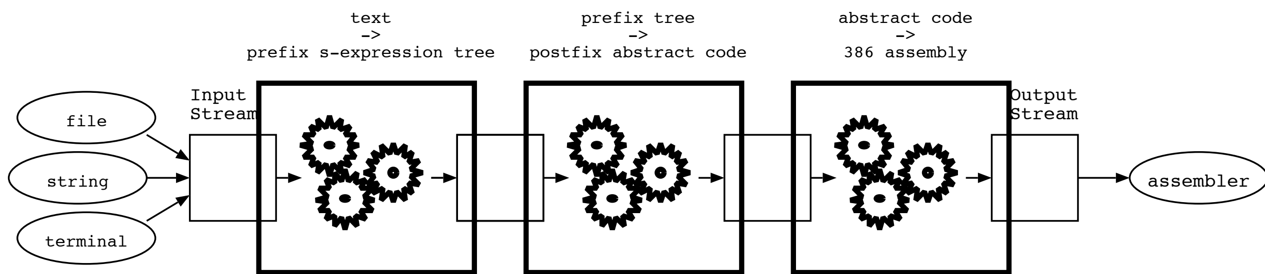

Figure 2. The front end converts a source program (list of characters) into an abstract syntax tree. The middle end converts the three into a program (list of instructions and operands) for a stack-oriented abstract machine. The back end converts the abstract machine code into concrete assembly language for the Intel 386. The same grammar and parser implementation is used in the specification and implementation of each stage.

Streams can be shared, the output stream from one stage being the input stream to the next. The first input stream in the chain converts a sequence of characters (in a file) into a stream of character objects for the front-end stage. The final output stream, following the back-end stage, converts a list of character objects into a sequence of characters (in a file) for assembly. Since streams (lists) contain objects, and characters and lists are kinds of object, the recognition and generation mechanisms work equally on textual input/output and tree-based intermediate representations. The compiler is thus a pipeline of stages, each transforming homogeneous data from a higher to a lower level of abstraction according to a set of PEG rules.

3. The compilation chain by example

Our compilation chain converts a Lisp-like source text into machine code for the Intel 386 processor (Figure 2). Every transformation (and its associated PEG rule) involved in the compilation of the small benchmark program, shown below, will be described.

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

(print (nfibs 32))
```

3.1 Front end: text to AST

This stage converts the text above into the corresponding tree structure consisting of nested lists, symbols and integers. The two "top-level" expressions appear as two consecutive list objects in the output stream.

The s-expression input language is a direct textual encoding of the corresponding AST, making the front-end trivial.[2] We include this stage to illustrate the parsing expressions that generate results (semantic values). (More substantive front ends would produce similar ASTs; one example is the block-structured language Forall, described by Yamamiya and Ohshima [12]).

The start rule of the front-end parser is invoked to generate an object each time the next stage underflows its input stream. Each object is an entire "top-level" s-expression.[3]

```
start    = sexpr
```

[2] The printed representation of the AST is identical to the source text and is omitted for brevity.

[3] Table 2 in Appendix A explains the matching expressions of our PEG.

An `sexpr` is made from whitespace (ignored) followed by either an atom (symbol or number) or list (zero or more s-expressions surrounded by parentheses). The rule for whitespace is given the name "_" to be suggestive of a blank space.

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

The parsing result from each of these rules is the object that they recognised. For example, the Kleene star in the `list` rule repeatedly invokes `sexpr` to read an expression from the input, until `sexpr` fails. The individual results from each successful match are assembled into a (possibly empty) list and stored in the parser's current result. The next expression `:l` saves that result in a local variable `l`. The final expression `->:l` restores the parser's current result from the local variable `l`. This is necessary because the intervening match of whitespace and closing parenthesis would destroy the desired result of the overall rule (i.e., the list object generated implicitly by `sexpr*`).

Symbols are made from a letter followed by any number of letters or digits. Letters (characters within identifiers) will be any alphabetic or punctuation character. A number is made from one or more digits. Digits are the usual 0 through 9.

```
letter   = [-+!\$\%&*./:<=>?@A-Z\\^_a-z|~]
digit    = [0-9]
symbol   = ( letter (letter | digit)* ) $$
number   = digit+ $#10
```

The first two rules above are regular expressions that succeed if the next character on the input stream matches a character in the given character class. The matched character is stored in the parser as the current result. In `symbol`, the final expression `$$` converts the current result (a list of letter or digit characters) into a symbol object. Similarly, `$#10` converts the current result (a list of digit characters) into an integer, base 10.

The last 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. Tokens are separated by the whitespace rule "_", matching any number of blanks or comments.

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

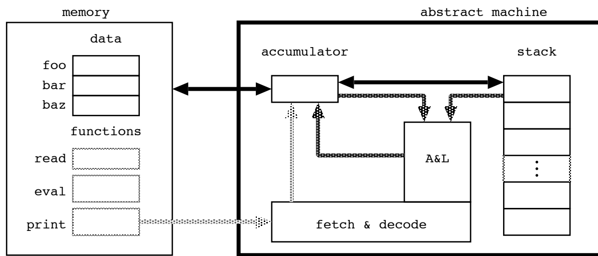

Figure 3. Abstract code runs on a stack machine with an accumulator (first operand and result for all operations, where relevant), a stack (second operand, saved intermediate results, and function arguments), and a store holding both variables (named) and code (with numbered labels).

3.2 Middle end: AST to abstract machine

The two AST objects produced by the front end are transformed into the following abstract machine program, represented as a list of symbols and integers.

```
(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 abstract machine (Figure 3) has an accumulator, a stack, and a memory containing named storage locations (global variables). Numbered labels identify the locations of function entry points and 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. The operation of each instruction is summarised in Table 1.

This form is convenient for the generation of a concrete machine code. (It could also be efficiently interpreted or stored for just-in-time, or similarly deferred, code generation.)

One s-expression at a time will be converted into abstract machine form, so the start rule in the middle-end parser matches an `expression`.

```
start = expr
```

The next three rules are for convenience: `long` and `name` match (using auxiliary predicates defined outside the grammar), consume and return an integer or symbol object; `arity` counts the number items left in the current stream.

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

Two kinds of list occur within s-expressions: actual `args` to function calls and formal `params` in function definitions. Each actual argument is an expression whose runtime value must be saved on the stack as soon as it is available. Inserting a `save` instruction after each argument expression accomplishes this.[4]

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

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

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

An expression can be any object or AST structure generated by the front end. 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 auxiliary predicate `is-arg?`, defined outside the grammar, differentiates between local and global names according to previous `arg-name` declarations.)

```
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. 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 runtime value of the second must be saved on the stack before the first overwrites it. When both operands are available the instruction corresponding to the operator is emitted (popping the runtime stack).

```
  | ’( ’< 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” are dealt with before function calls.

`(define` *`name value`*`)` creates a global variable by reserving a memory location labelled *name* wide enough to store the *value*, then generates the instructions to evaluate the initialiser and store its value into the location.

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

`(lambda (`*`args`*`...)` *`expr`*`...)` creates a function value—an address that can be called at runtime to execute the *expr*s. The sequence of instructions corresponding to the expressions in the body of the function are generated, delimited by `enter` and `leave` instructions (function prologue and epilogue, respectively). This sequence is not placed in the program at the point it occurs but rather saved for out-of-line compilation by the auxiliary function `save-lambda` which returns a unique label identifying the first instruction in the prologue. The address of this label is the runtime value of the entire lambda expression and is compiled in-line as a literal in place of the entire lambda expression.

[4] The output “template” for each rule is a list object in which substitutions can be made by the operators `:`, `::` and `:::` (see Table 3 in Appendix A).

instruction	*operation*
`label` *integer*	define the location of a numbered label
`long` *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` *integer*	copy the address of a numbered label 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 a numbered label
`branch-false` *integer*	transfer control to a numbered label if the accumulator is zero

Table 1. The abstract instructions needed to implement the example program.

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

`(if` *`condition consequent alternate`*`)` evaluates *consequent* if the *condition* is true, *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 auxiliary 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 evaluates to a function address to be called, with the remaining expressions in the structure being the actual arguments passed to it on the runtime stack. The `call` instruction is told the number of actual arguments so that it can clean up the stack after the called function returns.

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

Any other object appearing in the abstract machine code indicates an implementation error in the front end.

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

3.3 Back end: abstract machine to i386 assembly language

The runtime execution model corresponds directly to the abstract machine extended with an explicit stack pointer (identifying the topmost item on the stack) and frame pointer (identifying the start of the current function activation record in the stack). The runtime model and physical register assignments are shown in Figure 4.

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.

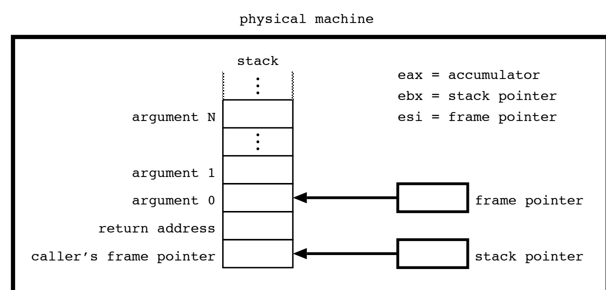

Figure 4. The physical execution model follows closely that of the abstract machine. Three physical registers are permanently assigned to the stack and frame pointers, and to the accumulator. The stack grows downward in an area of memory allocated explicitly during program initialisation.

The program expression `(print (nfibs 32))` was transformer into an AST by the front end, and into the abstract machine program

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

by the middle end. The back end transforms that abstract program into the following assembly language program.

```
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
```

The back end recognises a sequence of abstract machine instructions

```
start =  insn*
```

and emits corresponding assembly language on its output stream as a side effect of recognising each legal abstract instruction.

Program labels generated by the middle end are integers. Prefixing them with `L` keeps the assembler happy.[5]

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

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

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

The `load` instructions copy their operand into the accumulator; `store` copies the accumulator into a named memory location.

```
  | 'load-long  .:l `"movl    $\#l, %eax"
  | 'load-label .:n `"movl    $L\#n, %eax"
  | 'load-arg   .:n ->`(* 4 n):n
                    `"movl    (\#n)(%esi), %eax"
  | 'load-var   .:n `"movl    _V_\$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 top of stack and the accumulator, popping the stack. Relational operators generate an explicit Boolean 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 first test the accumulator for false (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 after the function returns.

```
  | '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 aligns the C stack pointer to a 16-byte boundary and allocates a 1024 byte runtime stack (using `malloc`) for the compiled program to use. The stack and frame pointers are initialised to point to the end of the allocated stack.

```
  | '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"
```

(The variable `_PREFIX` is defined to a string containing the prefix, if any, prepended to external symbols in the C namespace on the target platform. Note that each `pushl` implicitly subtracts 4 from the C stack pointer `%esp`. Subtracting 20 from it re-aligns it to a 16-byte boundary, as required by the C ABI.)

Program execution is terminated by the `exit` instruction, which performs a return (back to the operating system) from the C stack set up by `main`.

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

The helper function `print` called by the example program is hand-written for the purposes of this paper, and emitted at the end of the program as part of the final `exit` instruction.

```
           `"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"
```

[5] The ` expression and its string substitution operators (`\#` and `\$`) are explained in Table 3.

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

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

When presented with the output from the middle end, 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
```

4. Discussion

The generated code runs at 75% the speed of the same program written in C and compiled with typical optimisation (`gcc-4.3 -O2`) on Intel Core and Core2 processors. The executable code is 1.8 times larger than the C version.

Several peephole optimisations are possible between the middle- and back-end. Their definitions are obvious (from inspection of the generated abstract code for frequently-occurring sequences) and their impact on performance easily measurable.

Removing the occurrences of

```
addl $4, %ebx
subl $4, %ebx
```

increases performance to 80% and reduces size to 1.55 times optimised C. Making a special case of comparison with a literal integer followed by a conditional branch (to give

```
movl 0(%esi), %eax
cmpl $2, %eax
jge  L1
```

just before `L1`) increases performance to 96% and reduces size to 1.48 of C.

More complex instruction selection schemes are possible within the framework described here, but are most effective when the output from stage 2 remains in a structured (tree-like) form. BURG-style bottom-up “tree covering” can be described easily with a PEG, with ordered choice guiding alternate selections rather than “cost” functions. Compilation performance can be good but at the cost of transparency and complexity. Backtracking must be limited, by memoising matches on shared substructure and by properly factoring common prefixes within alternate expressions (either manually by the compiler writer, or automatically by a more sophisticated analysis and translation of PEG into executable parser). A full discussion of the issues and approaches is beyond the scope of this paper.

Simple static type systems could be accommodated by keeping a type token with each expression in the structured form and then matching legal combinations of type tokens for each operator implemented by stage 2. The linear form of the program would make the corresponding machine type explicit in each abstract machine instruction, with a corresponding increase in the number of such instructions and the stage 3 expressions needed to generate native code for them. Support for extensible type systems (or those that involve inference beyond trivial checking and propagation of synthesised types from sub-expressions) involves behaviour that is more dynamic than (and less naturally expressed as) PEG-based matching and transformation, more closely resembling an evaluator.[6]

A more sophisticated runtime system could place additional demands on the compiler. Whether these demands exceed the capabilities of a transformational approach to compilation, and the extent/location of complexity that they introduce, depends largely on the design of the runtime system. Examples might include the use of inline caches to optimise dynamic binding, the use of read and/or write barriers for pointer manipulation, and the generation of maps of pointer locations in the stack to support a precise garbage collector. The first two are hardly more complicated to implement than the arithmetic operators described above, and the last places almost all additional complexity in auxiliary functions called by the transformation rules rather than in the rules themselves.

Simplicity and clarity, rather than performance, are the primary goal—to make the successive transformation of source to binary understandable at every stage. Less than 100 lines of PEG code and about 20 lines of auxiliary function code are needed to compile the example program presented here. In the author's opinion, any competent CS student should be able to understand the compilation chain in a matter of hours and add the two optimisations described above with less than one day of work. We consider this an excellent result to achieve 96% the speed of optimised C (for simple numerical programs), which is sufficiently fast for many purposes.

Practical application of PEG-based compilation chains would replace the front end parser with one recognising a different (probably non s-expression oriented) language generating ASTs similar to those described here. Abstract instructions would be added as required by the semantics of the source language, and their translations to low-level executable code added to the back end. For execution on other architectures, the output expressions of the back end can be made to generate instructions for a different processor architecture or byte codes for a high-performance virtual machine. Yamamiya and Ohshima [12] describe such specialised front and back ends, in their implementation of the Forall language compiled to ActionScript Byte Code running on the Adobe Virtual Machine 2.

The compilation chain is not particularly fast. Compiling the example program 100 times gives a time of 3.37 seconds for 1100 lines of source (and one AST per source line in our example program) transformed into 516 kilobytes of assembler source—or about 326 source lines per second. The PEG implementation itself is very simple (less than 150 lines of code) and could be sped up significantly by applying optimisations such as those used in Rats! [4]. The cost would be a significant increase in complexity in the overall implementation.

A complete listing of the three stages (without annotation) is given in Appendix B. An archive containing everything needed to generate and run the compiler described is available from: `http://piumarta.com/S3-2010/`.

[6] Of particular interest is the smallest static type system that permits all of the source language's runtime support to be expressed within the source language itself, creating a self-sustaining implementation. The design of a "minimal" environment supporting compilation chains similar to those described here, and the treatment of static types within it, may be the topic of a future paper.

5. Related work

Parsing expression grammars (PEGs) [2] are a development of regular expressions [10], adding predicates and named sub-rules, and replacing alternation with ordered choice. Schorre describes META [9] in which the basic operations of parsing expression grammars are compiled into a recursive-descent parser. Baker [1] extends META to parse arbitrary data structures. The technique we use to convert PEGs into parsers is effectively that described by Schorre and Baker. Another system for matching patterns of objects, based on some of the same ideas, is OMeta [11].

Front end parser generators have traditionally been bottom-up, table driven, and difficult to understand or debug; the classic example is Yacc [5]. They usually require a separate lexical analyser (tokeniser), Yacc often being paired with the regular-expression based Lex [7]. Alternatives such as ANTLR [8] have appeared recently that generate recursive-descent parsers for context-free languages in the form of a highly-readable program, and which combine lexical and syntax analysis in a single parser generated from a single grammar. (PEGs offer similar benefits, as well as natural mechanisms to handle context sensitivity.)

Grammar-based back end generators have traditionally also been bottom-up. IBURG [3] is a well known example that uses a cost-minimising tree-covering algorithm to rapidly find "optimal" concrete instructions by rewriting patterns within a tree of low-level abstract machine code from the leaves up. Our PEG-based approach could do similar analyses if the tree-structured representation of code is maintained until the back end. (Experiments by the author suggest that the performance of PEG-based tree-covering instruction selection can be very good, with a relatively small penalty due to backtracking, at the cost of significantly more complex back end grammars than those described here.)

6. Conclusion

Many compilers use a grammar-based parser, code generator, or both. None appear to use a single grammar-based generator to create parser, transformation/analysis on intermediate forms, and code generator.[7]

By adopting a single representation for all levels of abstraction and within all stages of compilation, we unify the structure of the compiler and the transformations that are applied to program representations at each stage. A single PEG-based recogniser was extended to match structures within the representations, and to generate new structures for subsequent stages of compilation. The resulting compiler is small, having only one "compiler generator" language throughout, and is easy to understand because of the obvious correlations between the output templates at any given stage and the input patterns matched by the next.

Small, simple, dynamically-typed, polymorphic languages are good candidates for the compilation chain described above. They are well suited for manipulating the kinds of atomic and list data needed to implement a PEG-based compilation chain. In other words, they can easily become self-describing and self-sustaining. (The work described in this paper was part of an investigation into building such languages to support the STEPS project [6].) Complications arise, however, when trying to implement primitive behaviour in such languages for two reasons. First, external calling conventions (to library or system routines) almost always depend on the declared type of each parameter which complicates code generation for "foreign" functions. Second, suitable coercions (guided by the declared parameter types) must be applied to outgoing arguments and incoming results.

[7] There may be some, but an hour online searching for "parser-based compilation" and similar terms produced no useful results.

Just-in-time (or other in-memory) compilation can be facilitated by extending the back end's output expressions with syntax to construct integer instructions from bit fields. Forward references complicate the process, and either a two-pass or back-patching back end, with appropriate label management, is needed.

References

[1] H. Baker. *Pragmatic parsing in Common Lisp*. ACM SIGPLAN Lisp Pointers, 4(2):3–15, April/June 1991. `portal.acm.org/citation.cfm?id=121984`

[2] B. Ford. *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation*. 31st ACM SIGPLAN Symposium on Principles of Programming Languages (POPL), January 2004, pp. 111–122. `pdos.csail.mit.edu/~baford/packrat/popl04/peg-popl04.pdf`

[3] C. W. Fraser, D. R. Hanson and T. A. Proebsting. *Engineering a Simple, Efficient Code Generator Generator*. ACM Letters on Programming Languages and Systems, 1(3):213–226, September 1992. `storage.webhop.net/documents/iburg.pdf`

[4] R. Grimm. *Better Extensibility through Modular Syntax* ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2006, pp. 38–51. `cs.nyu.edu/rgrimm/xtc/rats.html`

[5] S. C. Johnson. *Yacc: Yet Another Compiler Compiler*. Unix Programmer's Manual, Volume 2b, AT&T Bell Laboratories, 1979. `dinosaur.compilertools.net/yacc`

[6] A. Kay, D. Ingalls, Y. Ohshima, I. Piumarta and A. Raab. *Steps Toward The Reinvention of Programming*. NSF Project Proposal, granted on August 31st 2006. `http://www.vpri.org/pdf/rn2006002_nsfprop.pdf`

[7] M. E. Lesk. *Lex—a lexical analyser generator*. Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, NJ, 1975. `dinosaur.compilertools.net/lex/lex.ps`

[8] T. Parr. *The Definitive Antlr Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, May 2007. ISBN 978-0978739256 `www.antlr.org`

[9] D. V. Schorre. *META II: a syntax-oriented compiler writing language*. 19th National Conference of the ACM, August 1964. `portal.acm.org/citation.cfm?id=808896`

[10] K. Thompson. *Regular expression search algorithm*. Communications of the ACM, 11(6):419–422, 1968. `portal.acm.org/citation.cfm?id=363347.363387`

[11] A. Warth and I. Piumarta. *OMeta: an Object-Oriented Language for Pattern Matching*. Techincal Report TR–2007–003, Viewpoints Research Institute, Glendale, CA, 2007. `www.vpri.org/pdf/tr2007003_ometa.pdf`

[12] T. Yamamiya and Y. Ohshima. *Tamacola — A Meta Language Kit for the Web*. 2nd ACM SIGPLAN Workshop on Self-Sustaining Systems (S3 2010), August 2010. `tinlizzie.org/~takashi/tamacola.pdf`

A. Parsing expression syntax

Table 2 summarises the syntax of the extended parsing expression grammars used in the compiler. Each rule has the form "`name = e`" where `e` is a parsing expression. An expression has two results, success (whether the input was matched) and a semantic value. A rule fails (does not succeed) when the first expression in it fails; if all expressions succeed then the rule succeeds. Every expression yields 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.

Output expressions construct a new result value for the rule. They are summarised in Table 3. 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

```
#include <stdio.h>

long nfibs(long n)
{
  return (n < 2)
    ? 1
    : 1 + nfibs(n - 1) + nfibs(n - 2);
}

int main(int argc, char **argv)
{
  printf("%ld\n", nfibs(32));
  return 0;
}
```

Figure 5. C version of the example program, used for benchmarking.

parser's output stream as a sequence of characters, possibly with substitutions from the contents of rule-local variables. The braces around variable *name*s can be omitted if no ambiguity arises.

The colon operators (`:`, `::` and `:::`) "flatten" nested list structures. An example might be useful. Consider

```
a = ((a b) (c d))
b = ((e f))
```

then

```
(   :a   :b) = ( ((a b) (c d)) ((e f)) )
(  ::a  ::b) = (  (a b) (c d)   (e f)  )
( :::a :::b) = (   a b   c d     e f   )
```

Output expression syntax is designed to keep declaration of intent in the same place as its use. While it might be a matter of personal preference, the author finds a few additional output operators in the PEG grammar preferable to the many auxiliary constructor functions that would have to be defined far from their single points of use.

B. Code listings

Figure 5 contains the equivalent C version of the example program, used for benchmarking. Figures 6, 7 and 8 contain complete listings for the PEG grammars that generate the front-, middle- and back-ends, respectively.

expression	*matches*	*semantic value (result of expression)*
	primitive values	
	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
‘ "*output string*"	always	a sequence containing the generated output
	prefix operators	
& e	e, without discarding the related input	e
! e	not e, without discarding the related input	undefined
-> e	interpret e as an *output expression* (see Table 3)	the value of e (see Table 3)
&‘*predicate*	the current input object, iff host langauge *predicate* is true, without discarding the related input	the input object matched
	postfix operators	
e ?	zero or one occurrences of e	a sequence of the value of any e matched
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 $$	the symbol interned from the value of e	
e $#*base*	the value of e converted to a number in the given base	
	binary operators	
e1 e2	e1 and then e2	e2
e1 | e2	e1 otherwise e2	the first e matched
e1 : *name*	e1	e1 after storing it in the *name*d variable

Table 2. Parsing expressions for recognising patterns in the data structures on the input stream.

expression	*value written to output stream*
	output expressions following a ->
symbol	the indicated literal symbol object
(*expression...*)	a sequence (structure) containing zero or more output *expressions*
:*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 “kernel” s-expression)
	character sequence (string) output expressions
‘"*characters...*"	a string containing each of the individual *characters* with substitutions as follows
	output character substitutions
\n \r \t	a newline, tab or carriage-return character (respectively)
\"	a double quote character
\\	a backslash 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

Table 3. Expressions for generating new data structures on the output stream. (Output expressions always “succeed.”)

```
start   = sexpr

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

symbol  = ( letter (letter | digit)* ) $$
number  = digit+ $#10

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

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

Figure 6. Grammar describing the transformation of text (a sequence of character objects) into an AST. Note that the output `list` structures are generated implicitly as a result of the Kleene star operator.

```
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
```

Figure 7. Grammar describing the transformation of ASTs into stack-oriented abstract machine instructions.

```
start   = insn*

insn    = 'label .:l                            `"L\#l:"
        | 'long .:n                             `"          .data"
                                                `"_V_\$n:   .long 0"
                                                `"          .text"
        | 'load-long  .:l                       `"          movl    $\#l, %eax"
        | 'load-label .:n                       `"          movl    $L\#n, %eax"
        | 'load-arg   .:n ->`(* 4 n):n          `"          movl    (\#n)(%esi), %eax"
        | 'load-var   .:n                       `"          movl    _V_\$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"
        | .:x                                   `(error "unrecognised instruction: " x)
```

Figure 8. Grammar describing the transformation of stack-oriented abstract code into concrete assembly language for the Intel 386.