Compiler Construction
Outline of Lectures
§1 — 107

by Mayer Goldberg

February 3, 2009

Abstract

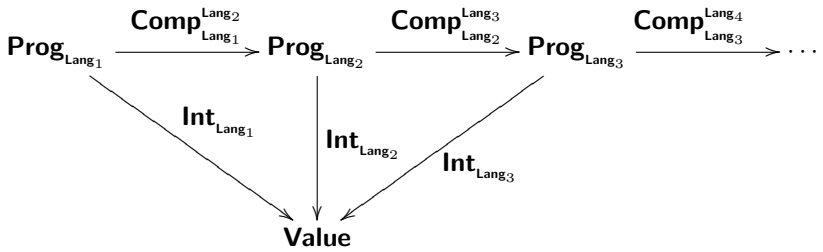
The following presentation outlines the lectures for the senior course *Introduction to Compiler Construction* given in the Fall of 2009, at the Department of Computer Science at Ben Gurion University. Compiling these notes takes up a great deal of my time, and though I try to keep them in sych with the lectures, this is a failing proposition as they are *always* behind. Still, they will be updated frequently throughout the semester, and should help you study the material at home. *They are not a substitute for attendence, and I make no claims that they cover everything that was discussed in class or all the material for the final exam.* If you notice any errors or omissions, please let me know and I will update the document.

This document pre-supposes that you are familiar with the programming language ML

Contents

1 Introduction to Compiler Construction (§1–41)

1. The basic tools of the software engineer are the *interpreter* and the *compiler*.
2. The interpreter *computes* the value of expressions or *performs* commands.
3. The compiler *translates* expressions and commands from one language to another.
4. Expressions have *values*. Statements and commands *perform actions* that *effect* the *environment*.
5. In this sense, an environment is anything that is observable: It includes global variables, IO devices, etc. This is a very broad notion of an environment, and contrasts with what you've studied in PPL, where the environment was a particular data structure that associates variables and values. In order not to confuse these two notions, we will use the term *observable environment* to refer to anything the program might effect.
6. Effecting the environment is what is known as a *side effect*. Side effects include changing the value at a location, printing output to the screen, reading data from a file or network connection, deleting files, creating tables in a database, etc.
7. When side effects are a part of a language, the language is said to be *imperative*. When side effects are not a part of the language, the language is said to be *functional*. Languages that discourage the use of side effects are called *quasi-functional* or *semi-functional* programming languages.
8. Scheme and ML are examples of quasi-functional programming languages. Good programming in these languages restricts the use of side effects and avoids it when possible. Programs in these languages consist mainly of expressions, and executing these programs amounts to evaluating these expressions.
9. Pascal, C, Java are examples of imperative programming languages. Good programmers in these languages do not avoid the use of side effects, however the quality of their design is related to how well they manage to decouple and encapsulate side effects.
10. Let $\mathsf{Prog}_{\mathsf{Lang}_j}$ be a program in some language j (e.g., Pascal or C), $\mathsf{Int}_{\mathsf{Lang}_j}$ be an interpreter from the language j to the space of values, $\mathsf{Comp}^{\mathsf{Lang}_k}_{\mathsf{Lang}_j}$ be the compiler that translates programs from language j to language k, then a diagram that relates the interpreter and the compiler in functional languages is as follows:



Another way of stating this diagram would be to say that for any languages j, k the following holds true:

$$\mathsf{Int}_{\mathsf{Lang}_j}[\![\mathsf{Prog}_{\mathsf{Lang}_j}]\!] \quad = \quad \mathsf{Int}_{\mathsf{Lang}_k}[\![\mathsf{Comp}^{\mathsf{Lang}_k}_{\mathsf{Lang}_j}[\![\mathsf{Prog}_{\mathsf{Lang}_j}]\!]]\!]$$

11. The diagram in § 10 depicts many things:
    (a) The value of a program is invariant of the language in which it is implemented, or into which it is translated.
        - Likewise, we say that compilation does not change the value (or behaviour) of the program, just the language in which it is written.
    (b) This invariance is what is meant by saying that the compiler is correct.

(c) Interpreters are *logically* required for computation; The compiler is optional: An interpreter is the only way in the diagram to go from source program to its value.

(d) A translated program can be translated again, into another language, and can be translated yet further, through more than two languages...

12. The microprocessor is an implementation, in hardware (or microcode), of an interpreter for the programs written in the language defined by the microarchitecture. Running a native-code executable means running a program written in some microarchitecture on a hardware implementation of an interpreter for this microarchitecture.

13. If we want to create a similar diagram for imperative languages, we need to change the domain and range of the interpreter for the imperative programming language, to include the observable environment, in the sense of § 5, and return a new observable environment (that will contain the effects of any changes resulted by the execution of the program).

14. **Example:** Suppose we wish to model side effects to variables, a character input stream, and a character output stream. The observable environment will be an ordered triple $\langle env, InputStream, OutputStream\rangle$. The interpreter would be a function mapping statements[1] and such an ordered triple into a new ordered triple:

$$\mathbf{Int}_{\mathsf{Imp}}[\![\mathbf{Prog}_{\mathsf{Imp}}, \langle env, InputStream, OutputStream\rangle]\!] \quad = \quad \left\langle env', InputStream', OutputStream'\right\rangle$$

Presumably, after $\mathbf{Int}_{\mathsf{Imp}}$ has executed $\mathbf{Prog}_{\mathsf{Imp}}$, the values of some variables are now different, some input has been read, and some output has been printed, so $\left\langle env', InputStream', OutputStream'\right\rangle$ will be different from $\langle env, InputStream, OutputStream\rangle$, and this is precisely what we mean by a side-effect.

The equation that models the relationship between a compiler from $\mathbf{Lang}_{\mathsf{Imp}_j}$ to $\mathbf{Lang}_{\mathsf{Imp}_k}$, and the two respective interpreters for these languages $\mathbf{Int}_{\mathsf{Imp}_j}, \mathbf{Int}_{\mathsf{Imp}_k}$, would be given by:

$$\begin{aligned} &\mathbf{Int}_{\mathsf{Imp}_j}[\![\mathbf{Prog}_{\mathsf{Imp}_j}, \langle env, InputStream, OutputStream\rangle]\!] \quad = \\ \mathbf{Int}_{\mathsf{Imp}_k}[\![\underbrace{\mathbf{Comp}^{\mathsf{Imp}_k}_{\mathsf{Imp}_j}[\![\mathbf{Prog}_{\mathsf{Imp}_j}]\!]}_{\in \mathbf{Lang}_{\mathsf{Imp}_k}}, &\langle env, InputStream, OutputStream\rangle]\!] \end{aligned}$$

and the diagram would be constructed accordingly.

15. Each additional side effect we might wish to observe will require us to modify the observable environment accordingly. For example, if our imperative language includes pointers to RAM, and arrays of locations in RAM, then our observable environment will have to model such RAM. To do so, we can replace our ordered triple with an ordered quadruple (4-tuple) that includes a vector of numbers or characters. The equation defining the interpreter will have to change accordingly, to reflect the fact that our programming language can modify locations in memory, and consequently our diagram will have to change as well.

    One moral of this example is that functional languages are simpler to model.

16. **Exercise:** Can you find an example where it might be useful to chain compilers, i.e., to translate the translation of a computer program?

17. The diagram says nothing about the source and target languages of the compiler, and indeed, the target language need not be assembly language or machine code. Indeed, all combinations are logically possible: it is possible to translate between two high-level languages, between a high-level and low-level language, between a low-level and a high-level language, or between two low-level languages.

18. **Exercise:** Can you find an example where it might be useful to translate between two low-level languages?

[1] Remember: Expressions *have values*, whereas statements *effect an environment*.

19. A common misconception is to think that compilation is about generating executables. In fact, generating an executable is a matter of *packaging* and not of compilation. Even interpreted code can be packaged into an executable: Just create an executable from the interpreter and its hard-coded input. To drive the point even further, consider that many zip archiving programs will let you turn a zip archive into a standalone executable that when executed will unpack into directories and files. This has nothing to do with translation, and is done by packaging the unzip program with the zip archive, as *hard-coded data* written directly into the executable file. On the other hand, it is possible to compile to languages other than to machine code, and the compile can generate text files, or various "fast load" files with some internal format of their own.

20. Most of the time, compilation is like explaining something in simpler terms: Complex, rich, expressive forms are translated into simpler forms, and in many cases, the language of these simpler forms is either assembly or machine language. This is why most people think of compilers as having something to do with executables.

21. It is possible, however, to tranlsate programs from low-level languages to higher level languages. This kind of compilation is called *decompilation*. Decompilation is a form of compilation that employs special analyses that attempt to reconstruct high-level forms out of low-level forms.

22. **Exercise:** Can you think of circumstances under which decompilation might prove useful?

23. When the target of the compiler is a microarchitecture that is implemented in hardware, the compiler is said to be a *native-code compiler*.

24. When the target of the compiler is a microarchitecture that has no hardware implementation, the compiler is said to be a *byte-code compiler*, to compile to *byte code,* or to a *virtual machine*.

25. **Exercise:** What might be the advantages and disadvantages of native-code compilers over byte-code compilers?

26. Another misconception about compilers is that compilers are batch-oriented, while interpreters are interactive. Batch-oriented means that the process of evaluating or executing the code is done at a separate time, from the perspective of the user, than the process of creating the code. Interactive means that the user interacts with the underlying system, issuing commands to be executed or entering expressions to be evaluated, and that the system responds immediately.

    The association of compilation with batch processing is a misconception grounded in historical practices. It is true that the earliest compilers were batch-processed, but this is true for most of the earlier computing of the 1950's and 1960's. It is also true that some of the first interpreted languages were interactive — BASIC and APL come to mind here.

    What is an interactive compiler like? Typically, interactive compilers generate code directly to memory (RAM), and execute that code on the fly. There are implementations of Scheme, LISP, Perl 6, and other languages that offer on-the-fly, in-memory, interactive compilers.

    What is a batch interpreter like? Typically, batch interpreters are used for scripting: REXX, Perl 5, TCL, and other scripting languages are typically interpreted.

27. While interpreters are a logical necessity, we may wonder why use compilers? One reason is convenience. It is easier, quicker and less prone to bugs to program in a higher-level language. But is convenience all there is to compilation? The next items should give us the vocabulary to express the significance of compilers in a more precise way.

28. *Early binding, late binding.* Early binding means evaluating expressions and binding their values to identifiers as early as possible. Late binding means evaluating expressions and binding their values to identifiers as late as possible.

29. Efficient computation implies computing things as early as logically possible. Flexible computation implies computing things as late as logically possible. Programming in languages that encourage early binding (FORTH, assembly, etc) encourages efficiency at the expense of generality, abstraction, and

coupling, while slowing down the programmer. Programming in languages that encourage late binding (Smalltalk, Self, Ruby, etc) encourages the use of programming abstraction (including design patterns), code re-use, modularity, and decoupling, while allowing programmers to be very productive. The ideal programming language would allow the programmer to enjoy all the benefits of late binding, while still giving the compiler enough information to generate code that binds early, i.e., enable aggressive compiler optimizations.

30. A compiler optimization is a code transformation, a way of changing the code, so that it can be evaluated (executed?) more efficiently. The gains in efficiency come when the compiler finds opportunities to pre-compute an expression, i.e., when the compiler can introduce early binding into the code.

31. The highest-level optimizations are source-to-source transformations. Intermediate-level optimizations are code transformations that take place between intermediate languages. The lowest-level transformations take place at the level of assembly instructions.

32. In translating code from a source language, though progressively lower-level intermediate languages, the compiler can identify more situations in which earlier-binding, or pre-computing can be introduced. In doing so, the compiler reduces the generality, modularity, level of abstraction, and level of decoupling in the code, and generates efficient assembly language. The programmer can then modify their high-level, abstract, modular, decoupled source code, and the compiler will re-translate it afresh.

33. Getting back to our original question: Why is the compiler necessary? The answer is that the compiler is as necessary as its ability to produce efficient target code. The compiler is what affords the programmer the luxury of programming in a high-level, paradigm-rich programming language, that

    - lets the programmer churn out more lines of more significant code
    - makes programming clearer, less prone to errors, easier to understand, modify, restructure and re-use
    - allows the programmer to ignore, at least to some extent, the efficiency issues of the code the programmer is writing

34. The search for candidates for optimizations is pursued in the most opportunistic way. Compiler writers program the compiler to recognize numerous situations where optimizations are possible, and the compiler simply checks for these situations one-by-one, with the aim of performing as many optimizations as it can. It is in the area of optimizations (and the analysis on which they are based) that compilers distinguish themselves. In fact, as we shall see later on in the course, it is quite trivial to write a naïve compiler, that performs just a few optimizations. But once the basic compiler is up and running, the real challenge becomes to add more and more optimizations. With the right implementation language and compiler-writing tools, it can take several days to write a usable compiler for a not-too-large programming language; But it takes many man-years to create a mature, optimizing compiler.

35. The above discussion addressed compilation in the abstract. In practice, the architecture that has emerged over the years is that of a pipeline of algorithms the output of one serving as input to the next. The choice of algorithms, and of the stages of the pipeline, are defined by the computational complexity of the algorithms that are needed. For the rest of this course, we will assume, and constantly refer back to the pipeline of the compiler given in Figure 1.

36. We identify three main stages in the pipeline of the compiler: The *syntactic analysis*, the *semantic analysis*, and *code generation*. The compiler composes these states into a single software entity that reads in characters and outputs code in the target language, usually assembly language or machine code.

37. To conceptualize the operation of the compiler we can think of how a computer program might try to translate text written in some natural language, e.g., English, to another natural language.

38. The translator would read in characters from some input stream. This could be a text file, or a console, or a text area, etc. As the characters are read, the program would try to group characters into *lexically*

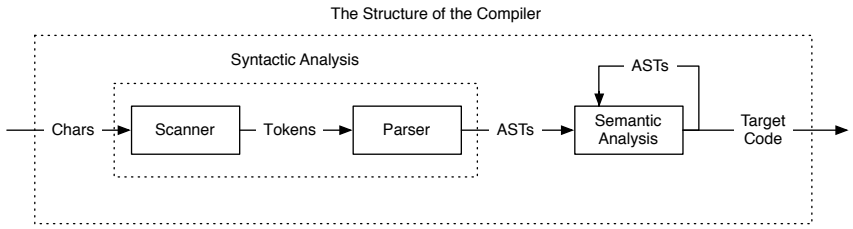


Figure 1: The Structure of the Compiler

significant units such as words, numbers, punctuation marks, etc. Such units are known as **tokens**. The phase that reads in characters and outputs tokens is called **lexical analysis**, and the software unit that would perform this analysis is called a **scanner**. The output of the scanner is a stream of lexically

39. The next phase in the translator would try to impose a grammatical structure upon the text: To divide the text into sentences (and paragraphs, and chapters, etc). This phase is called **parsing**, and the software unit that would perform this analysis is called a **parser**. The parser reads in a stream of tokens and outputs a stream of abstract structures that represent grammatical sentences. Such structures are known as **parse trees**, or **abstract syntax trees**.

40. Keep in mind that grammatically-correct sentences may be utterly meaningless. The American Linguist Noam Chomsky demonstrates this in his book *Syntactic Structures* (1957), using the now-famous, quaint construction *colourless, green ideas sleep furiously.*

    The task of ascertaining the meaning of grammatically-correct sentences is known as **semantic analysis**, and the corresponding tool is known as a **semantic analyser**. The results of the semantic analysis typically annotate the abstract syntax tree, and perhaps update some global tables as well. The semantic analyser can be said to map ASTs to annotated ASTs.

41. The output of the semantic analyser should be detailed enough to allow for immediate translation into the target language. In a compiler, this last phase is known as the **back end**.

2 Introduction to Scanning (§42–46)

42. The scanner is a function that maps a stream of characters to a stream of tokens. Tokens, as was stated earlier, in the context of natural languages, are the smallest, lexically-significant units in the language. Those would include *names* (of functions, variables, classes, etc), numbers (integers, floating point, etc), characters, punctuation marks, etc.

    The driving idea here is that identifying tokens should be *easy*. Just how easy? It should be possible to describe tokens using *regular expressions*. This is a *normative* idea, which means that it is meant to affect the way people define tokens. There is no "rule" to the effect that tokens must be describable using regular expressions, but there is something better: Readily-available scanner-generators will take descriptions of tokens, the regular expressions that define them, and the actions that should be performed upon reaching such a token, and generate a scanner automatically.

43. **Example:** What would it be like to have tokens that are not describable by regular expressions? Imagine a programming language in which legal variable names had to be of the form $a^n b^n \cdots$. In such a [silly] language, `aa`, `abc`, `aabbcc` would be legal variables, whereas `abba`, `acc`, etc would not.

44. In most programming languages, the following would be tokens: parenthesis/brackets/braces, numbers, characters, strings, booleans.

45. **Example:** An exception to § 44 are programming languages that allow for *nested strings*. The best-known example of such a language would probably be the Unix Desk Calculator (`dc`), though there are others.

46. **Example:** A feature less-baroque than nested strings would be nested comments. There are two kinds of comments: Froms some comment marker to the end of the line, and comments that have open-comment and close-comment markers. The first kind do not nest (obviously!), and the second kind may or may not nest, depending on the language definition. For languages with matching comment delimiters, a robust compiler would always support nested comments, at least with a compiler switch, because people use comments to "comment out code". Nested comments are not describable using regular expressions alone, and the natural way to support such comments is to use a counter to count the level of nesting. A DFA with such a counter would be equivalent to a stack automaton.

3 Introduction to Parsing (§47–47)

47.

4 The Language of S-Expressions (§48–48)

48.

5 The Reader (§49–49)

49.

6 The Tag Parser (§50–50)

50.

7 Macro Expansion (§51–79)

51. Programming languages contain many different syntactic constructs: Functions, applications, loops of various kinds, conditionals, etc. These forms are a part of the language supported by language processor (interpreter or compiler), and support for these forms means that they are recognised, and either interpreted or translated correctly. Different forms, however, are supported at different *levels*, and while this is an implementation issue, we will see later on that the way constructs are handled does effect the way the experience of the user.

52. Some language constructs are said to be *core forms*. This means that the language processor works with these forms from the very first stage, the scanner, and up to the very last stage, the code generator. Other forms are supported only *syntactically*, i.e., by the scanner, reader and parser. They are later translated into expressions involving core forms. Forms that are supported only syntactically are said to be *syntactic sugar*, and are considered as mere abbreviations expressions involving the core forms.

53. One example of syntactic sugar is how C handles strings and arrays. The subscript notation (`[ ... ]`) is only syntactic sugar for the corresponding [pointer-]arithmetic expression. For example, `"hello"[1]` is the same as `1["hello"]`, which in turn is the same as `*("hello" + 1)`, which is the character `'e'`.

54. **Exercise:** Write a brief C program to verify that `"hello"[1]`, `1["hello"]`, and `*("hello" + 1)` all evaluate to `'e'`.

55. Writers of language processors prefer to write language processors that support a small number of core forms, so that the language processor would be easier to write, maintain, validate, etc.

56. The disadvantages of supporting a small number of core forms, while expanding all the rest, are important:

    - While easier to compile and interpret, it is considerably more difficult to identify opportunities for opimization. The smaller the set of core forms, the more the intention of the programmer is lost in translation. While the semantics of the program remains the same, efficient translation will be harder.
    - The error messages the user receives for his/her code refer to the expanded forms, and not to the forms used by the user.
    - The debugger will only show the user the expanded form; Not what he/she wrote. So debugging is harder.

57. There is a natural tension between the drive "to focus on the essentials", and support only the smallest core forms, and the desire to create an efficient, user-friendly language processor.

58. **Exercise:** Write C macros that will be equivalent to the built in `for`, `while`, `do-while` forms in C, but that will expand into labels, `goto`-statements, and conditionals. This will show that the looping constructs in C need not be core forms.

59. **Exercise:** Concerning strings and vectors in C and Java, what information is available to either compilers? In what ways can the respective compilers use this information?

60. In the remainder of this section, we will examine various special forms in Scheme and study how to expand these into the core forms. We will also need to justify our choice of core forms, both on practical and theoretical grounds.

61. When we consider the Scheme language abstractly, we speak of the following syntactic forms: Constants, variables, conditionals, sequences, various forms of abstraction (`lambda`), application, assignment, definitions, conjunctions, disjunctions, and various forms for defining local state. This list covers a fairly large number of constructs. We will ignore all others. Anything that can be programmed at all can be programmed elegantly and efficiently with the above constructs. Later, if you take to Scheme and want to hone your skills as a Scheme programmer, you may study additional constructs on your own. Let us now examine each of these constructs, one-by-one, to make sure we understand what they stand for.

62. **Constants.** These include all forms of Scheme data, including numbers of various kinds (small int, bignum, fractions, floating point, complex, Gaussian complex), Booleans (`#f`, `#t`), characters, strings, pairs, the empty list, vectors, the *void* object, closures, continuations, IO ports, etc.

63. Noteworthy is the *void* object, not because of its great importance, but rather because one would not normally notice it passing by... To understand the *void* object, and the motivation for having it around, we must first discuss the *modus operandi* of Scheme.

64. Interacting with the Scheme system involves what is known as the *Read-Eval-Print-Loop* (REPL). Scheme is an interactive quasi-functional language, and this means that the user types expressions at the prompt, the expressions are then *read* in, *eval*uated, the value is *print*ed, and the system *loop*s back to the beginning, i.e., displays the promt, etc.

65. Being a quasi-functional language, Scheme supports expressions that have side effects. In traditional imperative languages such as C or Pascal, such imperative forms would be *statements*, i.e., syntactic constructs that denote commands or actions to be performed, *and that do not have any meaningful return values.* The syntax of Scheme does not include statements, but only expressions, i.e., syntactic constructs that do have values. Vulgarly put, every expression in Scheme has a value, even if because of its imperative nature, this value might not be of interest or use. This is where the *void* object comes in: The *void* object is used to denote the value of something which should not really have had a value. It is used to denote an unimportant value, a value *that should not be printed.*

When the value of an expression that has been typed at the prompt is the *void* object, the value is not printed. Instead, the Scheme prompt appears immediately following the given expression.

For example, defining expressions in Scheme should not print anything for a value. Since expressions must have a value, the value of a define expression is the *void* object, which is not printed. Consider the following interaction with Scheme:

```
> (define x 3)
> x
3
>
```

No value is printed for the expression that defines x. This is so because the value of the defining expression is the *void* object, and the REPL is written specifically not to print the *void* object. The value of the expression right under it *is* printed.

66. Another expression that returns the *void* object is the *if-then*-expression in Scheme, when the *test-expression* evaluates to `#f`. To understand this, consider the value of the *if-then-else*-expression in Scheme: `(if` ⟨*test*⟩ ⟨*thenExpr*⟩ ⟨*elseExpr*⟩ `)`. The value of this expression is either the value of ⟨*thenExpr*⟩ or the value of ⟨*elseExpr*⟩, depending on the value of ⟨*test*⟩. Now consider the *if-then*-expression, the structure of which is `(if` ⟨*test*⟩ ⟨*thenExpr*⟩`)`. What should be the value of this expression in case the value of ⟨*test*⟩ is `#f`? It turns out that the value is the *void* object. In fact, the expression `(if` ⟨*test*⟩ ⟨*thenExpr*⟩`)` can be thought of as syntacic sugar for `(if` ⟨*test*⟩ ⟨*thenExpr*⟩ ⟨*void* object⟩ `)`. So here we have our first example of syntactic sugar in Scheme!

67. The *void* object cannot be entered at the prompt, although we can write expressions the value of which is the *void* object. Some Scheme systems (e.g., Chez Scheme) support the `void` procedure, which takes no arguments and returns the void object. We can easily define the `void` procedure in other Scheme systems, as follows:

```
(define void
  (let ((v (if #f #f)))
    (lambda () v)))
```

68. If the *void* object is a part of some larger value, then it is printed. The specific printed representation is implementation-dependent, but a reasonable choice is `#<void>`. Consider the following Scheme code:

```
> (let ((v (if #f #f)))
    (list v v v))
(#<void> #<void> #<void>)
```

69. There are other expressions that return the *void* object in Scheme. Whenever you type an expression at the prompt, and immediately get a new prompt, without some printed return value in between, you know that the expression evaluated to the *void* object.

70. LISP, the linguistic ancestor of Scheme, does not have a void object. Instead, some semi-useful value is returned. The sarcastic tone in "semi-useful" is due to the awkwardness of using the return value in such situations. Here is a brief example: Assignment in Scheme reutrns the *void* object. Assignment in LISP returns the value assigned. The only reasonable use of this is to "see" the value assigned, when the assignment expression is at the prompt. However it would be awkward to use an assignment expression as part of a larger expression. This would make for highly idiomatic, otherwise unreadable code.

71. **Variables.**

72. **Conditionals.**

73. **Sequences.**

74. **Abstractions.**

75. **Application.**

76. **Assignment.**

77. **Definitions.**

78. **Conjunctions & disjunctions.**

79. **Forms for defining local state.**

8 Semantic Analysis (§80–87)

80. The semantic analysis phase attempts to understand various facets of the semantics of the program. This understanding is important for:

    - Finding program errors
    - Finding potential run-time problems
    - Optimizing the code
    - Generating code in the target language

81. The input to the SA phase is a stream of ASTs. The output of the SA phase is a stream of ASTs.

82. The knowledge gained during the SA phase is kept as **decorations** on the nodes of the ASTs, as well as in global data structures, such as symbol tables.

83. By the SA phase, the syntax is already completely verified, and is correct. Nor further syntactic isssues remain in the code.

84. Some of the questions that are addressed during the SA phase:

    - Whether all names of variable, function, method, datatype, record, class, etc, used in the code have been defined, and have the correct type/structure
    - Are all variables, functions, structures, modules, functors, etc, correctly typed
    - Are all expressions correctly typed
    - Are there memory leaks in the code
    - Are there array/string references that are outside the array/string
    - Does the code go into infinite loops
    - Does the code contain dubious constructions
    - Are there unreachable areas in the code (aka *dead code*)
    - What expressions are free of side effects
    - What expressions are repeated throughout the code (and can therefore be factored out)
    - ...

    We are barely scratching the surface with the above questions. Any serious compiler would be asking many more questions, and in fact, the semantic analysis phase is the most complex phase in the compiler. We just want to get a feel for what are semantic questions, and how can this information be used.

85. The above questions, as well as many other questions that would be asked during the SA phase of the compiler are *language-dependent*. Some programming languages give the compiler so little information, that many of these questions cannot be answered for programs in these languages. Specifically, languages that encourage early binding give more infromation than languages that encourage late binding. Programs in statically-typed programming languages give the compiler more information than programs in dynamically-typed programming languages.

86. **Example:** Consider the following equivalent expressions, `(+ 2 3 4)` in Scheme, and `2+3+4` in C.

    - The type correctness of the expression in Scheme will only be verified at run-time. The type correctness of the expression in C, however, will be known at compile-time.
    - The value of the Scheme expression cannot be known in compile time, since the value of `+` will only be known at run-time, and only then will we "know" that the code actually adds numbers. The value of the global `+` can be redefined any time prior to the evaluation of `(+ 2 3 4)`. On the other hand, the value of `2+3+4` in C is computed during compile-time, since the value of `+` cannot be overloaded or redefined.

87.

9 Nested Definitions (§88–88)

88.

10 Boxing Variables (§89–89)

89.

11 Lexical Environment (§90–92)

90. The lexical environment is a data structure that resides in the *heap*, is a part of a *closure*, and that stores the values of *idendifiers* that can be accessed from within the body of the closure.

91. Lexical scope is implemented via algorithms for accessing and extending the lexical environment. The static nature of lexical scope allow for very efficient access to the lexical environment.

92. The same lexical environment can be shared by more than a single closure. For example, the procedures `foo`, `goo`, and `boo` defined like thus

```
(let ((a 3) (b 5))
  (let ((foo (lambda () ... ))
        (goo (lambda (x) ... ))
        (boo (lambda (x y) ... )))
    ... ))
```

    All have the same lexical environment, which provides values for the bound variables `a` and `b`.

12 Lexical Addressing (§93–93)

93.

13 The Tail Call Optimization (§94–106)

94. Imagine you are using your web browser to read some web page, and that this page contains hyperlinks to other pages, as in Figure 2. As you are reading the page you would like to branch off and read the pages that are referenced by these hyperlinks. You do not want simply to transfer to other pages, but rather to have them open in new windows or tabs, so that you could read them, possibly open additional pages from within those pages, and when you read the end of a page, close its window or tab, and continue reading. Now suppose the last link on the page just so happens to be the last *thing* on the page. There is nothing to read further down. In this case, it would be inefficient to open a new window or tab, and upon closing it, to close the parent page as well. We can simply re-use the current

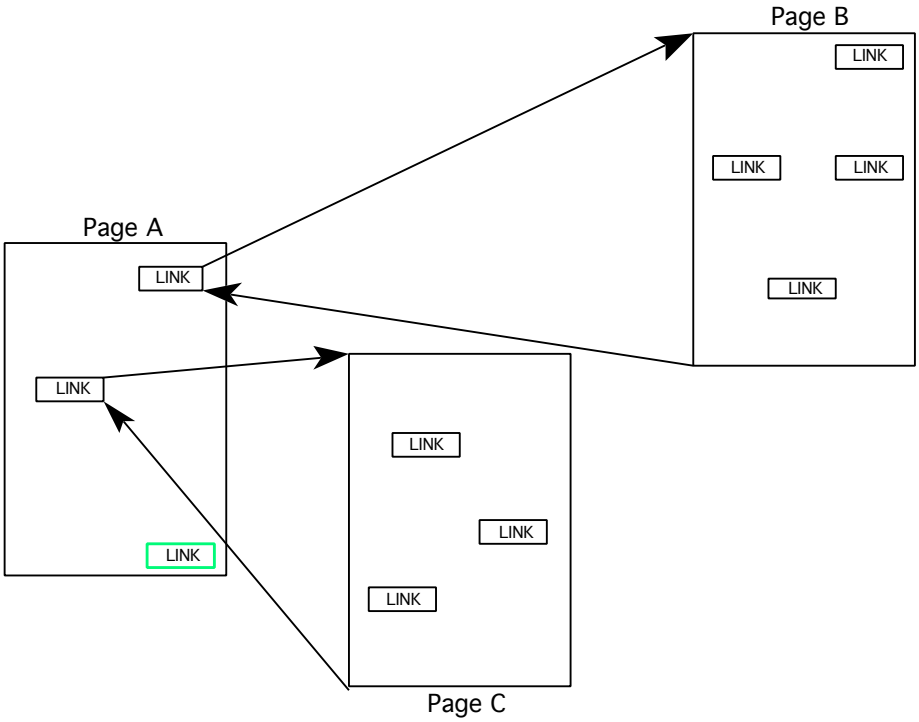


Figure 2: Hyperlinked Documents

window or tab by clicking on the link in such a way that the new page opens in the same window or tab as the current one. This would be the case when clicking on the last link of Page A (boxed in green). Welcome to the tail-call optimization.

95. A tail call is the last procedure call to be evaluated in a given frame. Ordinarily, procedure calls, or applications, set up an activation frame on the stack. The activation frame contains the values of the arguments passed on to the procedure, as well as a pointer to the lexical environment at the time the closure was created. The activation frame may also contain other information, such as the number of arguments that were passed to the procedure, local variables, etc. A tail call is a procedure call that happens just before a return from the current call and the deallocation of the current activation frame. The tail-call optimization re-uses the current activation frame, modifying it as necessary, in order to avoid setting up a new activation frame on the stack.

96. A **tail-recursive call** is a recursive call that is in tail position.

97. A recursive procedure in which every recursive call is in tail position is compiled, by a compiler that optimizes tail calls, into a loop.

98. Conversely, all looping constructs can be written as recursive functions in which every recursive call is in tail position.

99. **Example:** : Here is an implementation of a *while*-loop in Scheme:

```
(define while
  (lambda (test expr)
    (if (test)
      (begin
        (expr)
```

```
            (while test expr)))))
```

Notice that the recursive call to `while` is indeed in tail position. Here is some imperative code in C:

```
for (int i = 1; i < 5; ++i) {
  printf("Hello #%d\n");
}
```

The above code prints

```
Hello #1
Hello #2
Hello #3
Hello #4
```

Here is how the above code could be written using the above `while` procedure in Scheme:

```
(let ((i 1))
  (while (lambda () (< i 5))
    (lambda ()
      (display (format "Hello #~a~%" i))
      (set! i (+ i 1)))))
```

As you can see, the code is very similar (and not very Scheme-like!).

100. **Exercise:** Implement your own *for*-loop in Scheme, using a tail-recursive procedure. There are many ways of implementing *for*-loops; Think up some nice API, and implement it.

101. **Exercise:** Because Scheme is a quasi-functional programming language, procedures that do not return meaningful and useful values are not very interesting. Design and implement a *for*-loop in Scheme, using a tail-recursive procedure, in which the value of the body of the loop is used to construct a list. Think abot the kind of API that would be useful for such a loop.

102. **Example:** : The recursive call in this procedure is not in tail position:

```
fun length [] = 0
  | length (a :: s) = 1 + length s;
```

As you can see, after the recursive calls to `length`, the procedure adds 1 to the return value of the recursive call, so the call is not the last thing that happens.

103. **Exercise:** Rewrite the procedure `length` so that the recursive call is in tail position.

104. It is possible to construct recursive procedures in which only some of the recursive calls are in tail position. Here is our choice:

```
fun ackermann (0, b) = b + 1
  | ackermann (a, 0) = ackermann (a - 1, 1)
  | ackermann (a, b) =
    ackermann (a - 1, ackermann (a, b - 1))
```

105. **Exercise:** Underline in the above code for *Ackermann's function* the one recursive call that is not in tail position.

106.

14 Annotating Tail Calls (§107–107)

107.