# PART I

# MILESTONES

In keeping with the problem-based approach of the text, the text of the milestones is presented first. Chapter 2 presents the specifications of the project. Milestone I asks you to learn a completely new (for most readers) language called Forth. Forth has an interesting history and is currently used to support Adobe's Portable Document Format® (pdf) processing. Forth is the target language for the compiler you will develop. The material in Part II is provided to explain how to develop such a compiler.

Milestones II and III are the first two components for the compiler. The purpose of these two is to read the users' programs, with two results: (1) making a determination as to whether the program is syntactically correct and (2) producing an intermediate, in-memory (deep structure) version of the program.

Milestone IV inspects the deep structure version of a program and enforces the typing rules.

Milestone V produces code for programs that are only constants. While not very interesting in the scheme of things, this is a mid-point check that the compiler is actually capable of producing code.

Milestone VI is perhaps the most difficult of the milestones in terms of implementation. The good news is that once the scope and local variable issue are worked out, the remaining milestones are much easier.

Milestone VII produces user-defined functions. Recursion by itself is not that difficult; the unprepared student, however, often has issues with recursion that must be worked out.

Milestone VIII tackles the issues of nonatomic data definitions. While strings were implemented in Milestone V, the design of strings was under control of the designer. Complex data types take that control from the compiler and place it with the programmer.

Milestone IX is actually an anticlimax: objects are a combination of Milestones VI and VIII. Be sure, however, to read Chapter 3.

# 3

# GENERAL INFORMATION

This chapter provides standing instructions concerning the course. Most importantly, the grading scheme is explained.

## 3.1 GENERAL INFORMATION ON MILESTONES

### 3.1.1 Contract Grading

Contract grading is a simple concept: the exact criteria for a certain level of grade is spelled out beforehand. This course has milestones, participation, and a final exam. For a one-semester course, the contract matrix is shown in Figure 3.1. This contract scheme is dictated by the highest milestone completed. That is, if a student completes Milestone VI, attends every class, and gets 100 percent on the final, the student still can make no better than a C. Notice that there are no in-term exams; the milestones are those exams.

The milestones must be completed in order. The final grade on the milestones for final grade computation is predicated on the highest milestone completed. The milestone grading scheme is the following:

- The milestone materials will be handed in electronically. The materials must contain a makefile with four targets: `clean`, `compile`, `stutest.out`, `proftest.out`. A skeleton makefile is provided as an appendix. To receive any credit at all for the milestone, the targets `clean`, `compile`, and `stutest.out` must run to completion with no unplanned errors. If these three targets complete, the student receives a provisional score of 50.
- The `stutest.out` target is used to demonstrate the student devised tests. The score is the perceived completeness of coverage. The score on the coverage ranges from 0 to 25 points. The input must be in `stutest.in`. The `proftest.out` target may be used; if the student has modifed the makefile and the `proftest.out` does not work, the student can receive a maximum of 12.5 points.

| Grading Contract | | | |
|---|---|---|---|
| *Letter Grade* | *Milestone*[a] | *Participation* | *Final Exam* |
| A | Milestone VII | 90% | 90%[b] |
| B | Milestone VI | 80% | 80% |
| C | Milestone V | 70% | 70% |
| D | Milestone IV | 60% | 60% |

[a] Passing score + milestone report.
[b] Exemption possible if done on time.

**Figure 3.1 Grading Example. Passing Grades Are Determined by Milestone**

- Each milestone is accompanied by a written report based on the `Personal Design and Development Software Process`. Each student must hand in the required PDSP0 forms (plus a `postmortem`). The postmortem report must address two distinct issues: (1) what was learned by the student in designing and implementing the milestone and (2) what was learned by the student about her or his software process. The postmortem is graded on a 25-point scale. *The written report is not handed in electronically, but on paper to facilitate comments on the report.*

## 3.2 MILESTONE REPORT

The milestone project report form is used in the reporting of personal design and software process information outlined in Chapter 11. Each milestone will have slightly different documentation requirements, outlined for each milestone, because we are developing a process. In general, though, the milestone project report will contain three types of information: design documents, project management information, and metacognitive reflective write-up.

### 3.2.1 Design Information

The purpose of the design information is to convince the instructor that the student wasn't just lucky. By this, I mean that the student designed the milestone solution using some rational approach to solving the problem. Because the work breakdown structure (WBS) is an immediate requirement, at a minimum the student's report should include a sequence of WBS sheets. As these sheets are in Microsoft® Excel, there is no particular burden to upgrading them; however, the student needs to show the progression of the design.

### 3.2.2 Software Project Management

The second set of data is that pertaining to the management of the software development process using the spreadsheets provided and described in Chapter 11.

### 3.2.3 Metacognitive Reflection

When we set about assessing the level of thinking that a student is using, we first think about *knowledge*. However, there is more to learning than knowledge—there's general cognitive use. The levels of application are (lowest to highest) comprehension, application, analysis, synthesis, and evaluation. These levels are collectively known as "Bloom's Taxonomy" (Bloom 1956). Higher-order thinking skills refer to the three highest levels; metacognition refers to higher-order thinking that involves active control over the cognitive processes engaged in learning.

Metacognition plays an important role in learning. Activities such as planning how to approach a given task, monitoring comprehension, and evaluating progress toward the completion of a task are metacognitive in nature. Because metacognition plays a critical role in successful learning, learn how to use metacognition in your professional life. While it sounds difficult, metacognition is just "thinking about thinking."

In the milestone report, the student describes the processes that were used in developing the milestone. This includes saying what the student did right and what the student did wrong, or at least could have done better. If the student feels that he or she did something innovative, then the student should analyze that activity in the report. During grading, I look for the student's discussion of "what I learned in this milestone."

## 3.3 GENERAL RESOURCES

- Makefile prototype as in Figure 3.2
- ANS Forth Reference Manual, provided with text
- Gforth Implementation Manual, provided with text
- Information about designing large projects (see Chapter 8 on design)

## 3.4 MAKEFILE PROTOTYPE

The makefile serves two very important purposes. For the student, it guarantees that only the latest versions of the code are included in any test. It also plays an interface role between the student and the grader: the grader does not have to know the inner details of the program. I assume the

```
# Created by D. E. Stevenson, 29 Jan 02
# Modification History
#    Stevenson, 17 Sep 02, added multi-
ple tests to studtest.out
#-----------------------
#INSTRUCTIONS
# Quoted strings must to altered by the student
CCC = "compilername"
CCFLAGS = "compiler flags"
OBJECTS = "list of .o files"
SOURCES = "list of source files, including input tests"
RUNFLAGS = "information for runtime flags"
#########################################################
#        Warning
# The spaces bfore the commands are not spaces;
# it is a tab. At prompt, type "man make"
# for more information.
#########################################################
clean:
 rm -f ".o/class files" core "executables" "outputs"
ls

compiler: clean $(OBJECTS)
$(CCC) $(CCFLAGS) -o compiler $(OBJECTS)

# You may need to specify target/source
information not in the default rules.

stutest.out: compiler
#
# Here is an example of how multifiles can be run.
# Suppose you have several input files, in1, ...
#  and the compiler produces out1, ...
cat in1
-compiler $(RUNFLAGS) in1
cat out1
# the minus sign tells make to ignore the return code.
cat in2
-compiler $(RUNFLAGS) in2
cat out2
# This is the proftest entry. must be there.
proftest.out: compiler
cat$(PROFTEST)
compiler $(PROFTEST)
cat proftest.out
```

**Figure 3.2 Makefile Prototype**

student has learned how to make a `Makefile` but I prefer to hand out one prototype.

The makefile has four targets:

**`clean:`** The `clean` target removes all work files as well as all previous outputs. The clean ends with listing the directory.

**`compile:`** This target compiles the milestone code and produces a runnable version of the project to date.

**`stutest.out:`** The `stutest.out` target runs the code with the student-prepared input. The student must print out the input test file(s)—there could be several—and the output produced. There should be the same number of output files as input files and each must be clearly labeled.

**`proftest.out:`** This target allows the grader to test special cases without having to change the student's makefile.

# Milestone I

# LEARNING A NEW LANGUAGE, Gforth

**Reference Materials**

ANSI/IEEE X3.215-199x. Draft Proposed American National Standard for Information — Programming Languages — Forth. X3J14 dpANS-6—June 30, 1993.

**Online Resources**

www.gnu.org: GNU Foundation. *Gforth Reference Manual*
www.Forth.org: a wealth of tutorials

## MILESTONE REQUIREMENTS: INTRODUCTION TO Gforth

The output of our compiler will be the input to a "machine." In C, the compiler generates an object file, the familiar `.o` file, that is then linked into the `a.out` file. To actually run the program you must still cause the operating system to load the `a.out` file into memory for the machine to execute.

For our project, we will use the program Gforth. Gforth is almost completely the opposite of Lisp. The syntactic format for Gforth is in *Polish Postfix* or, more commonly, *postorder*. Although Lisp has some syntax, Gforth only has spelling rules. As an example, a 1 + 2 expression in C would be entered as [+ 1 2] in SOL and 1 2 + in Gforth.

**SPECIAL INSTRUCTIONS.** Submit your answers in a file named "stutest.in." This makes the filenames consistent across milestones.

Before attempting to do the milestone assignment, read and do the cases contained in the sections entitled "Introduction to Gforth," "Programming in Gforth," and "Introduction to Milestone Cases."

## OBJECTIVES

**Objective 1:** To introduce you to Gforth. Gforth will be the machine that we code to. Typically, students are inexperienced in assembler-level coding. However, to understand compilers and interpreters one must understand how to talk to the underlying machine.
**Objective 2:** To emphasize the crucial role of generalized trees and generalized postorder traversal.
**Objective 3:** To get you to formulate a generalized expression tree data structure.

## PROFESSIONAL METHODS AND VALUES

The professional will learn to use many programming languages and paradigms throughout her or his professional career. The hallmark of the professional in this venue is the ability to quickly master a new programming language or paradigm and to relate the new to the old.

## ASSIGNMENT

The exercises below are simple, "Hello World"-type exercises. These short program segments or small programs are designed to help you to understand how to learn a new language: you just sit down and try some standard exercises that you already know must work.

## PERFORMANCE OBJECTIVES

In this milestone, you have several clear performance objectives.

1. Either learn to run Gforth on a departmental machine or how to install and use Gforth on your own machine.
2. Learn the simple programming style of Gforth.
3. Translate an infix style expression to an expression tree.
4. Do a postorder traversal of the expression tree to generate the Gforth input. The output of this step is Gforth code.
5. Produce running Gforth code that evaluates the programs equivalent to exercises. The output here is the running of the Gforth code.

## MILESTONE REPORT

Your milestone report will include versions of items 3 and 4 above. Submit the Gforth input file and makefile electronically. The grader will generate the output file by running Gforth. The standard report format is followed for the paper version of the report.

Your milestone report must include a data structure for an n-ary tree and a pseudo-code recursive algorithm to translate an arbitrary instance of these trees into postorder.

## FORTH EXERCISES

The required material for this milestone consists of two sections. One section is an exercise to translate C code snippets into Gforth code. The second section is an exercise in Gforth to implement string support needed in Milestone V; experience shows that this string exercise is a valuable learning tool.

1. Pseudo-C exercise. The following are statements in "pseudo-C." Write Gforth programs that will compute the same thing the C program would using the statements in Figure I.1
2. Gforth strings. Gforth's strings are even more primitive than those of C because the fundamental convention of the *delimited* string is absent. Recall that in C, a "string" is a one-dimensional `char` array that is terminated by the null value. The delimited approach is only one such convention: we could include the length of the string at the beginning for what is termed a *counted* string.

## CASE 1. STRING REPRESENTATION

Develop *your* representation for strings. The choice between delimited and counted should be made based on the trade-off between representation size and ease of operation. This decision is primarily based on *concatenation*. For this milestone, implement concatenation in your choice of representation.

## INTRODUCTION TO Gforth

Charles Moore created Forth in the 1960s and 1970s to give computers real-time control over astronomical equipment (see Moore 1980). Functions in Forth are called "words." The programmer uses Forth's built-in words to create new ones and store them in Forth's "dictionary." In a Forth program, words pass information to one another by placing data onto (and removing data from) a "stack." Using a stack in this way (Forth's unique contribution to the world of programming languages) enables Forth applications to run quickly and efficiently. The use of stacks in this way is familiar to users of scientific hand calculators such as the Texas Instruments TI-86.

| *Problem* | *Statement* |
|---|---|
| 1 | `printf("Hello World\n");` |
| 2 | `10 + 7 - 3*5/12` |
| 3 | `10.0 + 7.0 - 3.0*5.0/12.0` |
| 4 | `10.0e0 + 7.0e0 - 3.0e0*5.0e0/12.0e0` |
| 5 | `10 + 7.0e0 - 3.0e0*5/12` |
| 6 | `y = 10;`<br>`x = 7.0e0;`<br>`y + x - 3.0e0*5/12` |
| 7 | `if 5 < 3 then 7 else 2` |
| 8 | `if 5 > 3 then 7 else 2` |
| 9 | `for ( i = 0; i <= 5; i++ )`<br>`printf( "%d ",i);` |
| 10 | `double convertint(int x){ return ((double)x);}`<br>`convertint(7)` |
| 11 | `int fibonacci(int i) {`<br>`if (i < 0 ) abort();`<br>`else if (i == 0 ) return 1;`<br>`else if (i == 1 ) return 1;`<br>`else return fibonacci(i-1) + fibonacci(i-2);`<br>`}` |

**Figure I.1 Milestone I Elements**

The Forth Interest Group* and the Institute for Applied Forth Research† help promote the language. Two books by Brodie (1981, 1984) are perhaps the best-known introductions to Forth. Perhaps the most famous application is as Adobe's PDF® (portable document format).

Forth went through several standarization cycles. The final cycle ended in 1994. The Proposed Standard is available online from several vendors.* There are many implementations of Forth available; the Forth Interest Group lists 23 commercial vendors as of April 2005. Several implementations are open source. The implementation discussed here is available from the GNU Project server at www.gnu.org.

A Google search of the Internet returns many tutorials on Forth. Many of these tutorials are for a specific implementation although almost any such tutorial presents a general introduction. Any Forth tutorial after the 1994 Standard should contain information consistent across the spectrum. The Gforth manual accompanying the implementation contains an excellent tutorial.

* http://www.forth.org

† dec.bournemouth.ac.uk/forth/rfc/inst.html

* http://www.taygeta.com/forth/dpans.htm being just one.

## PROGRAMMING IN Gforth

The reader is certainly familiar with expressing arithmetic calculations in a language like C or Java. For example, consider the following:

```
1 + 2 * 3
```

which is evaluated to 7. Such a presentation is called *infix notation*. But this is not how such a calculation is expressed to the machine.

True assembler languages reflect the operational nature of the underlying machine's machine language. The distinction between *machine* language and *assembler* language is primarily the medium of presentation: machine language is electronic and assembler is character. Assemblers were developed because it was too difficult and error-prone to use machine language for large problems. Assemblers generally include features that make programming easier for the human but which must be translated (usually in a transparent manner) to machine information.

A generic picture of an assembler statement is the following:

$$\texttt{operation operand}_1\texttt{, ..., operand}_n$$

The `operation` could be a primitive data operation, a primitive control operation, or the name of a programmer-defined function. The operands could be primitive data constants or expressions relating to variables or addresses. Presented this way, we call such a notation *prefix notation*. Operations can be categorized by how many operands can be mentioned in a statement; that is, how large $n$ is in the schema above. The above $1 + 2 * 3$ calculation might be denoted in assembler as below:

```
load r1, 2
mult r1, 3
add  r1, 1
```

which computes 7, leaving the result in a place called `r1`.

Forth uses no addresses directly but relies on a stack to hold all values. Such a system results from a *postfix notation* often used in scientific handheld calculators. The postfix notation for our calculation might be

$$1\ 2\ 3\ \textit{mult}\ \textit{plus}$$

leaving the 7 on the top of the stack. Postfix-based processors are very fast as they bypass much of the fetch-execute cycle. Notice that no intermediate assignments are required to store temporary values in this postfix execution style.

## INTRODUCTION TO MILESTONE CASES

The goal of this milestone is to master some of the fundamentals of the Gforth. The fundamental processes to master in this milestone are

1. Take representative programming expressions in the C family and put them into a tree.
2. Process the tree in a postfix fashion to design the Gforth program.
3. Translate the design into Gforth.
4. Write the Gforth program.
5. Test the Gforth program.

## CASE 2. ELEMENTARY ARITHMETIC STATEMENTS

The first case study touches on some issues that you are already familiar with from your data structures class. Compiling is based on three basic ideas that you learned in algebra, although you were not taught with these words:

1. Every arithmetic expression can be put into a tree.
2. Every tree can be traversed in postorder.
3. The list generated by the postorder traversal can be evaluated to produce a value without intermediate storage.

For each of the expressions in Figure I.1 below,

1. Draw the tree that represents the correct precedence.
2. From the tree, generate a list by a postorder traversal.
3. From the postorder traversal list, generate the value.
4. From the postorder traversal list, write the Forth program, run it, and compare the value computed by a hand calculator.

### Assignment

Part 1. Evaluate the arithmetic statements in Figure I.2. **Caution:** these statements are presented as someone might write them in a mathematics class and not in a computer science class. This introduces the concept of *context* or *frame of reference.* In particular, carefully consider the values of the last two expressions before you compute.
Part 2. Run the resulting code in Gforth.
Part 3. On one page only, write the algorithm that takes any *infix* statement into a *postfix* statement.

(1) $1$
(2) $1 + 2$
(3) $1 + 2 \times 3$
(4) $-1 + 2 \times 3$
(5) $-1 + 2 \times 3^2$
(6) $1 + 2 \times (-3)^3$
(7) $-1 + 2 \times 3.0^{4.7}/6$
(8) $2^{-3^3}$
(9) $2^{-5}$
(10) $\max\{1, 2, 3, 4, 5\}$

**Figure I.2 Case 1 Problems**

## CASE 3. ROUND 2: MORE ARITHMETIC

The second case study touches on more complex issues in evaluation, but still working with just arithmetic expressions. We want to add variables. We still want

1. Every expression to be put into a tree.
2. Every tree traversed in postorder.
3. The list generated by the postorder traversal to be evaluated to produce a value without intermediate storage.
4. From the postorder traversal list, write the Forth program, run it, and compare the value computed above.

As with Case 1, the rules are

1. To take the mathematical expressions below and draw the tree that represents the correct precedence.
2. From the tree, generate a list of the leaves by a postorder traversal.
3. From the postorder traversal list, generate the Gforth input.
4. Test the resulting Gforth expression.

After you have done the exercises, update the algorithm you wrote for Case 1 to reflect what you have learned.

In Figure I.3 we want to consider sets of expressions, because to have variables, we need to have definitional expressions as well as evaluative expressions.

This is a significantly harder exercise than Case 1. Some of the issues are

1. How are you going to handle the semicolon?
2. How do you declare a variable in Gforth. The declaration sequence does not obey the strict postfix rule. Why?

$\{x = 1; x\}$
$\{z = 3; 1 + z\}$
$\{y = 10; -1 + y \times 10\}$
$\{y = 3.0; 1 + 2 \times y\}$
$\{x = 3; y = 7; -1 + x \times 3^y\}$
$\{q = 3; 1 + 2 \times (-z)^3\}$ This is correctly written
$\{e = 4.7; -1 + 2 \times 3.0^e/6\}$
$\{a = 3; b = -3^a; 2^b\}$

**Figure I.3 Sets of Expressions**

3. How do you assign a value to such a variable?
4. How do you retrieve the assigned value?
5. What happens if an undeclared variable is evaluated?

Run the resultant codes in Gforth.
Revise your algorithm produced in Case 1.

## CASE 4. `IF` AND `WHILE` STATEMENTS AND USER-DEFINED FUNCTIONS

Forth was designed as an interactive language and is probably quite different than what you're used to. Most importantly, Forth has two different input modes: interactive and compile. Words that are evaluated generally cannot be used in the compile setting and vice versa. This leads to the rather strange situation of having two `if` statements: one in the interactive mode and one in the compile mode. Each has its place, but we will concentrate on the compiled `if`.

Make up test cases to demonstrate the use of `if` and `while` statements. Be sure you understand the differences between the two modes. Run the test cases to demonstrate your understanding.

### Defining User Words

In Chapter 6 we introduce the concept of the metalanguage that enables us to talk *about* a language. We make use of that concept here. In Forth, words are defined in terms of a sequence of words and constants.

In order to discuss a language, for example Gforth, using another language, for example English, we must use the idea of metalanguage and object language. In our case, Gforth is the object language and English is the metalanguage. This gives rise to a convention concerning meta notation.

We will use the angle brackets around a symbol to indicate meta-names. In this case, ⟨*Statements*⟩ means "zero or more statements."

We use the term ⟨*ForthCompilableExpresssion*⟩ to mean any legitimate sequence of Forth words except a colon and a semicolon and ⟨*New Word*⟩ to indicate the word being defined. The syntax for a user-defined word in Forth is

: ⟨*NewWord*⟩ ⟨*ForthCompilableExpression*⟩ ;

Remember that you need blanks surrounding the colon (:) and the semicolon (;).

### `if` and `while`

Now we're able to use `if` and `while` statements as they normally appear in the C family. Using metavariables in the obvious way, we can define `if` and `while` as follows:

⟨*Condition*⟩ `if` ⟨*TrueComputation*⟩ `else` ⟨*FalseComputation*⟩ `endif`
⟨*Condition*⟩ `while` ⟨*Computation*⟩ `endwhile`

This case explores the two control structures, `if` and `while`. Here the postfix regime breaks down even further. We still can put the expressions into a tree, but we cannot evaluate everything: we can only evaluate the proper statements.

## CASE 5. COMPLEX DATA TYPES

Complex data types such as strings and arrays, C `structs`, and Java objects must be developed if the project is to be able to emulate the full range of data types that can be defined using the type systems of modern languages. In a one-semester course at the introductory level, it is not possible to develop a compiler that can reliably compile structures, objects, and the referencing environment.

However, it is possible to develop support for character strings. For many, this will be quite a challenge because it requires implementing several complex functions. One design approach is to use the CRUD (create, retrieve, update, delete) approach. This is a commonly used approach to database design. We return to this issue in Milestone V.

## CASE 6. FUNCTIONS

The third case study touches on the programming (rather than the mathematical) context issues in evaluation, but still working with just numbers. We need to add local variables and definitions. The goal remains the same: investigate the algorithmic issues in evaluation.

Below, we want to consider sets of expressions, because to have variables, we need to have definitional expressions as well as evaluative expressions.

```
define n! = if n <= 0 then 1 else n*(n-1)!
```

Here, the issue is much different than the straight arithmetic problems in Case 1 and Case 2. Now we have to make multiple copies and evaluate those copies. This process is called *unfolding*.

We have now seen an example of all the straight computational and evaluation issues.

Run your encoding for 5!.

# Milestone II

# LEXICAL STRUCTURE AND SCANNER

**Reference Materials**

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 4th Ed. Upper Saddle River, NJ: Prentice Hall. 1995. Section 3.3.

Kenneth C. Louden. *Programming Languages: Principles and Practice.* 2d Ed. Thompson, Brooks/Cole. 2003. Section 4.1

This milestone begins a sequence of four milestones (Milestone II–Milestone V) that use the same basic approach to design and implementation. In each of these milestones, we begin by using a formal language. The formal language is used to capture the solution of the milestone problem. Each formal language solution specifies an abstract program that can then be encoded into the implementation language.

The goal of this milestone is to design a *scanner*, the enforcer of the lexical definition of SOL. The purpose of a scanner is to read the individual characters from the input file and convert these characters in a token. *Tokens* are data structures that carry (1) information read by the scanner and (2) the terminal classification according to the grammar rules. The specification for the scanner comes from the *lexicon* as described by the system specification (Chapter 2).

This milestone also begins development of the *symbol table*. The symbol table is developed throughout the project.

## CASE 7. PROJECT MANAGEMENT

Before beginning this milestone, read Chapter 11. This chapter describes elementary project management processes for this project. The central focus should be the development of the WBS spreadsheet. Before attempting any

milestone, the WBS relevant to that milestone should be included in the project management portfolio.

In order to start at a common place, Figure II.1 contains a skeleton WBS. Using this WBS, develop your own based on the cases in each milestone.

## OBJECTIVES

**Objective 1:** To develop a formal definition of the scanner algorithm from the lexical structure for SOL. You will code the scanner from this formal definition.
**Objective 2:** To formalize the data structure used for the tokens.
**Objective 3:** To give you experience using a *state-transition diagram* as a design tool.
**Objective 4:** To give you experience in designing a program using a categorical style design.
**Objective 5:** To begin work on the symbol table for the project.
**Objective 6:** To give you practice in designing test files based on the formal definition and develop a test driver for the combined scanner-parser program.
**Objective 7:** To test the resulting program for correctness based on the formal definition.
**Objective 8:** To develop personal project management skills.

| *Task Number* | *Description* | *Inputs From* | *Outputs To* | *Planned Value* | |
|---|---|---|---|---|---|
| 1 | MAIN Function | | | Hours | Cum. Hours |
| 1.1 | Initialize | | | | |
| 1.2 | Open files | | | | |
| 1.3 | Call Parser | | | | |
| 1.4 | Call Type Checker (stub) | | | | |
| 1.5 | Call Semantics routines (stub) | | | | |
| 1.6 | Call Code Generator (stub) | | | | |
| 1.7 | Close files | | | | |
| 2 | Tree Data Structure | | | | |
| 3 | Token Data Structure | | | | |

**Figure II.1 Beginning Work Breakdown Structure (WBS) Based on Requirements**

## PROFESSIONAL METHODS AND VALUES

This milestone is the first of two that use formal language theory to specify the program to be written. In particular, scanners rely heavily on the *finite state machine* (FSM) formalism.

Because the formalism allows for complete specification of the lexicon, test case development is relatively easy. The grading points for this milestone are based on the fact that you should be able to test all cases.

## ASSIGNMENT

The purpose of this milestone is to develop a working scanner from the lexical definitions. Scanners are supported by over 50 years of development; these techniques were used by engineers in the 1960s to design circuits. Sections entitled "Scanner Design" and "Scanner Design Extended" provide details.

This is the proper time to start the symbol table development. The only requirement is that the lookup routine be an open list hash table: the lookup must work in $O(1)$. See the section entitled "Symbol Tables" for details.

## SPECIAL REPORTING REQUIREMENTS

Starting with this milestone, each milestone report will be accompanied by all design documentation along with your PDSP0, WBS, and time recording template (TRT) forms. The milestone report must include your FSM design, as well as your metacognitive reflection on what you did in the milestone and what you learned.

## ADDITIONAL INFORMATION

The lexical scanner rules are the same as C whenever a specific rule is not stated. For example:

1. Specialized words in SOL that are the same as C take on their specialized meaning. For example, the `if` and `while` have approximately the same semantics in SOL and in C.
2. Direct information on the format of integers has not been given, so C rules are used; the same is true for floating point and strings. **Caution:** Floating point numbers are somewhat difficult to deal with terminologically. SOL and C have the same view of floating point values; Gforth does not share that definition. In Gforth, 3.2 is treated as a `double` integer when 3.2*e* is treated as `floating`

`point`; SOL has no `double` but C's `double` is equivalent to Gforth's `floating point`. Confusing? No, the context keeps it straight.

## SCANNER DESIGN

The scanner reads in the program text and accumulates characters into words that are then converted into tokens. The scanner operates as a subroutine to the parser. The communication between the two is in terms of instances of tokens. The purpose of the token structure is to inform the parser of the terminal type of the token and the value associated with the token.

Conventionally, the scanner is designed using FSM concepts. The work of the cases is to design that machine. The possible tokens are defined by the lexical rules of the language defined in Chapter 5.

## CASE 8. DESIGNING THE TOKENS

Read the specification documents and the parser and determine the various token classes that are required. Now develop a data structure for these tokens. Tokens are a data structure that contain both the "word" read from the input and an encoding of the terminal type. The word must be stored in its proper machine representation; that is, integers are not stored as strings of characters but in their 32-bit representation. **Requirement:** If you are using C, then you must define a *union;* in object-oriented systems you will need an *inheritance hierarchy.* You will develop some of the information for this case in Case 47.

Realize that the scanner is required to check constants for validity. As an example, a sequence of digits 20 digits long cannot be converted to internal representation in 32 bits. Therefore, this 20 digit number is illegal. Hence, the words are converted to their proper memory format for integers and floating point numbers.

### Scanner Design

#### *Finite State Automata*

The finite state automata (FSA) (sing. automaton) paradigm is the basis for the design of the scanner. The FSA paradigm as described in theory texts has two *registers* (or memory locations) that hold the *current character* and *current state.* The program for the machine can be displayed in several standard ways. The graphical representation is easier to modify during design. The implementation program that is the scanner is written using the graph as a guide.

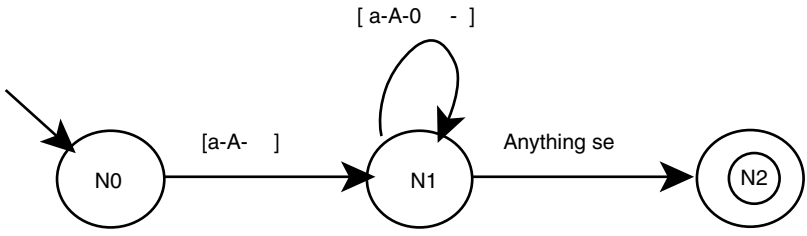


**Figure II.2 Finite State Machine, Graphic Presentation**

The graph in Figure II.2 represents a program that recognizes a subset of C variables. State *N*0 is the start state. If the machine is in the start state and it reads any upper- or lower-case alphabetic chararacter, the machine moves to state *N*1; the notation used in the diagram is consistent with Unix and editor notations used for *regular expressions*. As long as the machine is in state *N*1 and the current input is any alphabetic letter, a digit, or an underscore, the machine remains in state *N*1. The second arrow out of state *N*1 is the terminating condition. It should be clear with a moment's reflection that the self-loop on *N*1 is characteristic of a `while` statement. With that observation in mind, the C code in Figure II.3 can be used to recognize variable names. By convention, the empty string symbol $\epsilon$ means that no character is needed for this transition.

To understand the programs, we must first understand the machine execution cycle. This is actually not much different from the cycle used in a standard computer.

> **MC1:** The machine is in a state as indicated by the current state register. If that state is a valid state, then continue; otherwise go to step 5.
> **MC2:** Read the next character from the input. If there is no next character, halt and go to MC5.
> **MC3:** The program of the machine is a set of pairs *(state,character)*. Search the list of valid pairs for a match with the contents of the current state, current input registers.
> **MC4:** If there is a match, set the current state to the value associated with (current state, current input) and go to MC1. If there is no match, set the current state to an invalid value and go to MC5.
> **MC5:** The machine has a list of special state names called final states. If, upon halting, the current state value is in the final state list, then the input is accepted; otherwise, it is rejected.

The theory of FSA describes what inputs are acceptable to finite state automata. To develop the theory, consider a formalization by asking what

```
#include <ctype.h>
#include <stdio.h>

void recognize_name() {
   int current_input;
   enum state { N0, N1, N2 };
   enum state current_state = N0;
   current_input = getchar();
   if(current_state == N0 && isalpha(current_input))
     current_state = N1;
   current_input = getchar();
   while( current_state == N1 &&
          (isalpha(current_input) ||
           isdigit(current_input) ||
          current_input == '_')) {
     current_state = N1;
     current_input = getchar();
   };
   current_state = N2;
}
```

**Figure II.3 C Program for FSA Graph**

the various parts are. There are five:

1. The set of possible characters in the input (Programming-wise, this is the type of input and of the current input register.)
2. The set of possible state values in the current state register
3. The full set of pairings of *(state,character)*
4. The list of final states
5. The start state

Part of the difficulty of studying such abstractions is that the formulations can be quite different and yet the actual outcomes are the same. For example, this same set of concepts can be captured by *regular expressions, regular production systems,* and *state transition graphs*. The regular expressions used in editors like *vi* and pattern matching programs like *grep* are implemented by an automaton based on the above definition.

For design purposes, the preferred graphical representation is that of a *state-transition graph (ST graph)*. A state-transition graph is a representation of a function. In its simplest incarnation, an ST graph has circles

representing states and arrows indicating transitions on inputs. Each of the circles contains a state name that is used in the current state. The tail of the arrow represents a current state, the character represents current input, and the head of the arrow names the new state.

ST graphs are representations of functions. Recall that a function has a domain and a range. In the ST graph, the domain and range are circles and we draw an arrow from the domain to the range. The arrow is weighted by information relating to the function. The ST graph is directed from the domain to the range. It is customary in finite state automata work to have an arrow with no domain as the start state and double circles as final states. If every arrow emitting from the domain circle is unique, then the transition is *deterministic*; on the other hand, if two arrows weighted the same go to different ranges, then the transition is *non-deterministic*. It is possible to write a program that emulates either a deterministic or non-deterministic automaton; for this milestone, let's think deterministically.

The weight of the arc captures the functioning of the transition. Since FSA only need to indicate which character is involved, the graphs in texts are very simple.

## CASE 9. DEVELOPMENT OF FINITE STATE AUTOMATA DEFINITIONS

Develop a finite state automaton for each class of lexical input specifications.

### *Finite State Machines*

The technical difference between an *automaton* and a *machine* is that an automaton only gives an accept-reject response, whereas a machine does transformations.

Case 9 only develops the accept-reject portion of the scanner. The scanner must produce a token as output and, therefore, we need to think about the saving of characters, possibly converting some (like integers or special characters in strings) etc. We denote this by the weight. Figure II.4 illustrates the concept. That figure is just Figure II.2 with the function to be performed *after* the pattern is matched. Finite state machine diagrams are great design tools.

## CASE 10. FINITE STATE MACHINE DESIGN

Modify your output of Case 9 to reflect the processing to save and convert characters to form tokens.

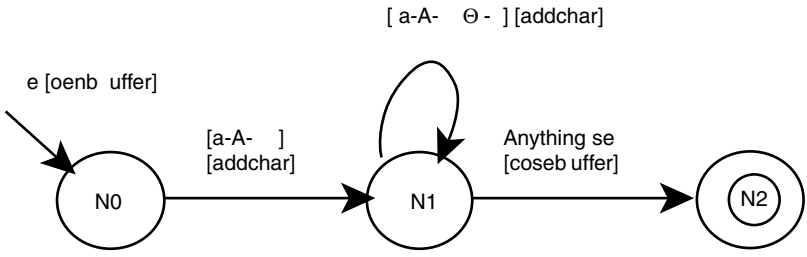


**Figure II.4 Finite State Machine, Graphic Presentation**

### Guidance

The purpose of all these exercises is to show you a professional method often used in hardware, protocol, and input design.

## SCANNER DESIGN EXTENDED

We now have a first design of our scanner. We need to now consider whether or not the design is worth anything. We will do this with two exercises.

### The State Exercise

Unfortunately for those who need to use the FSA and FSM ST graphs to design things, the graphs as the theoreticians have them are not completely useful. For any graph of any complexity, we need to ask, "What is the semantics of this graph?" That is, what does this graph actually do?

In theoretical studies, the states are treated as a more or less meaningless alphabet: a set of letters, say. That's actually not very interesting. What is interesting is that the states are predicates. *Predicates* are statements in which something (the *predicate*) is affirmed or denied of something (the *subject*), the relation between them being expressed by the *copula*. The extended definition means the form of thought or mental process expressed by this, more strictly called a *judgment* (*Oxford English Dictionary*). Interestingly enough, judgments can always be displayed as a directed acyclic graph (DAG).

## CASE 11. REASONING ABOUT DESIGN

For each of your ST graphs, annotate the nodes with predicates concerning the computation. Write down a list of test cases that can be used to show that your ST graph is correct. Using the predicates, choose representative

test cases and interpret the ST graph and show that this test case is correct. Keep it Simple!

## WORK BREAKDOWN STRUCTURE

We have designed the scanner and we have seen that it can be used to design tests for the complete scanner. But this is only part of the milestone because we must translate the design into a program in your favorite programming language. To this point, we have been focusing on logical sequence of flow; but there are other issues, such as

1. Opening and closing input files
2. Using error protocols
3. Saving the input string for the token
4. Creating new instances of a token

This is where the WBS exercise is crucial.

In the WBS form (see Chapter 11), the critical exercise is to name every possible task that must be accomplished for the scanner to function. The more complete the decomposition (breakdown), the better. Ultimately, we must be able to estimate the number of lines of code that we will have to write. The hours needed to code a function is the number of lines of code divided by 2.5.

## CASE 12. SCANNER WORK BREAKDOWN STRUCTURE

Produce a work breakdown structure for the scanner. Keep a record of how much time (earned value) you actually spend. Turn the work breakdown structure in with the milestone report.

## SYMBOL TABLES

Symbol tables are the heart of the information management functions in a compiler. At their core, symbol tables are database management systems. We will address the design of symbol tables in this manner.

### Database Management 101

A database is a collection of data facts stored in a computer in a systematic way so that a program can consult the database to answer questions, called queries. Therefore, there are three fundamental issues in the design:

1. What are the queries that the database is meant to provide information about?
2. How is the data retrieved by a query?
3. What is the actual data in the database and how is it turned into information?

A typical database has a basic layout as shown in Figure II.5.

Many different design disciplines have been proposed over the years based on underlying data organization (trees or directed graphs) or query formation, such as the relational approach. These disciplines are often suggested by the principal uses of the database system, the so-called "enterprise" model that must satisfy a divergent user population. We don't have that problem: the compiler is the sole user, so we can design the system to be responsive to a limited set of data and queries.

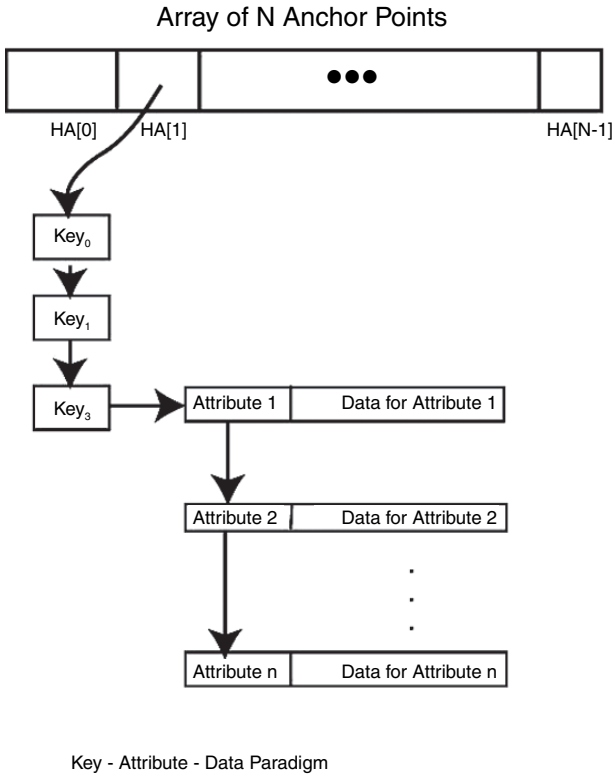


**Figure II.5 Simple Database Configuration**

### Initial Analysis

One approach to designing a symbol table is to simply ask, "What are the entities that should have data in the symbol table?" An approach to that is to look at the classes of symbols in the lexicon. We basically have symbols that are (1) constants (predefined with a fixed value), (2) tokens that have no semantic import, like '[', and (3) names. Since (2) represents information strictly between the scanner and parser, let's ignore them for the time being.

Constants and names, then, are problematic. What is even more problematic is the fact that until we have specific queries we don't know what data we need. This argues strongly for a very flexible approach.

On the other hand, the basic look-up requirement is clear: we need to be able to look up the entry associated with a "name" (string of characters). This is a data structures issue. The requirement here is for very fast look-up.

### Second-Order Analysis of Symbols

This is a good point at which to review the organization of a natural language dictionary, such as the *Merriam-Webster Unabridged* or the *Oxford English* dictionaries. For example, look up "lexicon" in the *Oxford English Dictionary*, second edition online. If you do, you will find that there are three major definitions, some accompanied by minor definitions. What we learn from this is that the dictionary (symbol table) must be aware of the *context* in which the query is being asked.

As a second experiment, look up the word "party." We find that there are four major definitions, one for different *parts of speech*. What is the equivalent concept in programming languages to "part of speech?" Parts of speech tell us the *use* of a word. Now we have something to go on: the basic organization in the symbol table is that there is one class (structure) of data for every possible use a symbol can have in a program.

From these two examples, we can conclude that we should take an *entity-property-value* approach to the design of the symbol tables. The individual entities are either constants or names. For every possible use that an entity can have, there should be a data structure that holds the unique values associated with the entity for this property.

The symbol table will be developed throughout the project.

## CASE 13. SYMBOL TABLE DESIGN

The generic symbol table has three basic considerations: (1) a hash function to convert *keys* to *integers*; (2) a collision maintenance system; and (3) a

variety of data structures, one for each use. A given key may have more than one data structure associated with it.

Design and implement a symbol table system for this milestone. The symbol table for this milestone must identify *keywords* such as `let` as keywords.

# Milestone III

# PARSING

**Reference Material**

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 4th Ed. Upper Saddle River, NJ: Prentice Hall. 1995. Section 3.4.

Kenneth C. Louden. *Programming Languages: Principles and Practice*. 2d Ed. Belmont, CA: Thompson, Brooks/Cole. 2003. Chapter 4.

This milestone completes the syntactic phase of the project. The parser directs the syntactic processing of the input and builds a version of the program in memory. The whole syntactic process fails if there is a grammar or lexical error.

In Milestone II we developed the program by using graphic definitions of finite state machines. The graphic representation of FSMs is not powerful enough to describe the processing of context-free languages. Instead, we will generalize another concept, that of *production systems*.

## OBJECTIVES

**Objective 1:** To produce the parser and continue the symbol table for the project

**Objective 2:** To use the data structure developed for the tokens

**Objective 3:** To give you experience using the production system concept as a design tool

**Objective 4:** To give you experience in designing a program using a categorical style design

**Objective 5:** To give you practice in designing test files based on the formal definitions

**Objective 6:** To develop personal project management skills

## PROFESSIONAL METHODS AND VALUES

This milestone requires careful planning in both the design and testing phases. We will use a formal method. Parsing has perhaps the strongest theoretical grounding of any subject in computer science. We review this material in the sections entitled "Grammars and Things Grammatical," and "Parser Design II," on the Chomsky Hierarchy.

## ASSIGNMENT

The purpose of this milestone is to develop a working parser utilizing the scanner and symbol table. Performance objectives are to

1. Develop a formal definition of the recursive descent parsing algorithm from the grammatical structure for SOL
2. Develop a parser program from a formal definition
3. Develop a data structure for the abstract syntax graphs, also known as abstract syntax trees
4. Develop a test driver for the developed program
5. Test the resulting program for correctness based on the formal definition

The milestone report should concentrate on the development of the program and what you learned about formal production system design concepts. It must include a copy of your formal design documents.

### Detailed Information

You are to develop the following:

- A scanner that reads characters from a file or standard input. The file names are taken from the command line. The task of the scanner is to classify the terminal symbols as specified by the parser's grammar.
- A parser is a program that gets its input from the scanner. The parser has two tasks:
  1. The parser verifies that the input meets the rules of the grammar.
  2. The parser produces a tree that encodes the input.
- The main program ("driver") sets up the input files by reading the command line flags. You should consider using flags to communicate debugging options. Any flag that does not start with the customary hyphen ('−') or a plus ('+') is taken as a file to be processed. This subsystem is the input user interface and could have multiple

input files. **Note:** A file can have multiple expressions. The loop outlined above must be continued until all files are processed. The process loop is as follows:

1. Set up the file to be processed so the scanner can read it.
2. Call the parser to parse the input and to receive the tree produced by the parser.
3. Echo print the tree to verify the input is correct. The printer program should be a separate, stand-alone function that can be used throughout the compiler's development.

### Testing Requirements

The trees will be used eventually to fulfill future milestones. For this project, you are to demonstrate that your parser can (1) properly process correct inputs and (2) reject incorrect inputs. You want to keep the print subroutine as a debugging aid for later in the project.

### The Project Grammar

The project grammar is given in the specifications.

## CASE 14. DO WE HAVE TO WORK HARD?

One possible approach to the parser is to implement—as directed below—the grammar in the specifications. However, careful thought (and maybe a little research) would indicate that there is a very simple underlying grammar that can be developed. This simpler grammar is a trade-off between what is *syntactic* and what is *semantic*.

If you take this much simpler route, then you must justify this in the milestone report. In particular, you must develop an argument as to how and why you will guarantee that a program that is not correct by the specification grammar will be stopped. **Hint:** Move some syntactic considerations to semantic considerations.

### Specific Grading Points

- Constraints
  - You are to implement the parser as a recursive descent parser.
  - The scanner must be a subroutine called by the parser.
  - You must not store the parentheses in the tree.
  - You may not cast to Object. Any variant record implementation must either be by inheritance for Java and C++ or unions in C.

  - You must be able to read any number of input expressions—properly bracketed lists—made up of an arbitrary number of subexpressions and properly constructed lexical elements.
  - You may not use counting of parentheses as the parser algorithm.
- Conditions. Recovery from input syntax errors is almost impossible. Throw an error and exit.

## TESTING REQUIREMENT

The program will be accompanied by test files. These test files are to be developed by you and should clearly demonstrate that the scanner and parser implement the complete specification. They should clearly show both correct processing of correct inputs and rejection of incorrect inputs.

## GRAMMARS AND THINGS GRAMMATICAL

We will use the vocabulary of *formal languages* to describe formal programming languages and their parsers.

For technical reasons, FSMs such as we used to design the scanner are not powerful enough to deal with context-free programming languages; this is because it is impossible to set a viable limit on the number of "parentheses" in advance. It is necessary to move to a more powerful model, *context-free grammars.* Specification of context-free grammars is more complicated than for FSMs, but the concepts are similar. We are unable to use a simple ST graph for context-free languages; this is due in part to the need to specify recursion.

A grammar $G$ is a four-tuple ($V$, $T$, $P$, $S$), where $V$ is a set of *categorical symbols;* $T$ is a set of *terminal symbols;* $P$ is a set of *grammar rules;* and $S$ is a member of $V$ known as the *sentential symbol.*

In order to have a definite example and to maintain continuity with the case studies, we consider the grammar for arithmetic expressions, known in the "compiler writing trade" as *Old Faithful*, because it is an easy, complete example of the issues.

### Basic Terminology

The motivation for formal languages is natural language, so the initial concepts are all borrowed from natural language. Grammars describe *sentences* made out of *words* or *symbols.* Both $V$ and $T$ are sets of such words or symbols.

A language would be all the sentences that are judged correct by the grammar: this is called *parsing.* Technically, we define the *Kleene closure*

| | | |
|---|---|---|
| N0 | → | a N1 |
| N0 | → | b N1 |
| . | . | . |
| . | . | . |
| . | . | . |
| N0 | → | Z N1 |

**Figure III.1 Regular Productions for Recognizing Names**

*operator* $A^*$ to mean zero or more occurrences of a string $A$. This notation is probably familiar to you from the use of regular expressions and as the "wildcard" operator in denoting file names. If $A$ is a set, then $A^*$ is every possible combination of any length. If $T$ is the set of all possible terminal symbols, then $T^*$ is the set of all possible strings that could appear to a language, but not all may be grammatically correct.

Which strings of $T^*$ are the "legitimate" ones? Only those that are allowed by the rules of $P$. So how does this particular description work?

## Productions

### *Regular Grammars First*

Productions are another way to describe a pattern. The productions are two words, for example, $Y$ and $Z$. Conventionally, a production in $P$ is denoted $Y \rightarrow Z$. Having already learned how to describe patterns with graphs (Milestone II) we can move from graphs to productions. In the graphs, we have nodes and arrows. Each node has a label and each arrow has a weight. We can collect all the node labels into a set; call this set the *nonterminals*. Likewise, we can collect the set of individual weights in the *terminal* set. A production is esentially a rule taking one nonterminal to another while recognizing input terminals.

As an example, consider the FSA developed to recognize names (Figure III.1). The start node is $N0$. We can traverse to $N1$ if the regular expression $[a - zA - z]$ is satisfied. In words, "Start at $N0$. If any alphabetic character is on the input, the next state is $N1$." The same concept is captured by the productions in Figure III.2.

## CASE 15. CONVERSION OF FINITE STATE AUTOMATON GRAPHS TO GRAMMARS

Complete the conversion of the "Name" graph from its graphic form to production rules.

$$E \rightarrow T + E$$

$$E \rightarrow T - E$$

$$E \rightarrow T$$

$$T \rightarrow F * T$$

$$T \rightarrow F / T$$

$$T \rightarrow F$$

$$F \rightarrow \text{any integer}$$

$$F \rightarrow (E)$$

**Figure III.2 Productions for Arithmetic Expressions without Variables**

Convert the floating point number definition from its FSM (not FSA) graphic form to a production form. **Hint:** Think of the FSA graph as an FSM graph that has no actions.

### *Context-Free Productions*

The regular language basis of the scanner is deceptive. Despite its apparent power, it has a flaw when it comes to describing languages such as SOL. Regular languages cannot balance parentheses because they have no memory. The strict form of the productions $N \rightarrow tN'$ with $N$ and $N'$ being nonterminals and $t$ being a terminal matches the fact that the FSA have only current input and current state memory locations. In order to recognize balanced parentheses, we need more memory; in fact, a simple stack will do. This is the model of *context-free languages.*

In order to deal with the parentheses issue, we need to loosen the rules on what constitutes a legitimate production. In order to have a definite example, we consider the definition of Old Faithful (Figure III.2), so named because it appears in most language texts; it captures the essence of arithmetic expressions.

To begin, we have two sets of symbols (just as in the above regular example): $V$ is the set of nonterminals and $T$ is the set of terminal symbols.

$$V = \{E, T, F\}$$

$$T = \{+, -, *, /, (, ), \text{any integer}\}$$

The members of a new set, call it $P$, are the productions. These rules describe an algorithm by which we can decide if a given sequence of symbols is grammatically correct.

**Note:** There is a contextual use of the symbol $T$ in two different ways: $T$ in the definition of grammars as the terminal set and $T$ in our example grammar (short for *term*). Such overloading of symbols is common; there are only so many letters. Therefore, you must learn to be cognizant of the context of symbols as well as their definition in context. Ambiguity should arise only when two contexts overlap.

## Derivations

A sentence is structured correctly if there is a *derivation* from the starting symbol to the sentence in question. Derivations work as shown in this example. Productions are *rewriting rules*. The algorithm works as follows:

> **RW1:** Write down the sentential symbol.
> **RW2:** For each word generated so far, determine which symbols are nonterminals and which are terminals. If the word has only terminal symbols, apply RW4. If the word has nonterminal symbols, choose the productions with that symbol as the left-hand side. If there are nonterminals, remove this word from consideration. Make as many copies of the word as there are productions and replace the nonterminal with the right-hand sides.
> **RW3:** Continue until there are no changes.
> **RW4:** If any word matches the input word, then accept the input word.

This certainly is not very efficient, nor easy to program for that matter, but we will address that later.

We use the '⇒' symbol to symbolize the statement "derives." That is, read "$A \Rightarrow B$" as "$A$ derives $B$."

Now for an example. Is "1+2*3" grammatically correct? See Figure III.3

After studying the rules we can find a direct path. See Figure III.4. So, yes: "1+2*3" is grammatically correct. But I made some lucky guesses along the way. But programs can't guess; they must be programmed.

Notice that the parse given in Figure III.5 is not the only one. For example, we could have done the following:

$$E \Rightarrow T + E$$

$$\Rightarrow T + F * T$$

$$\Rightarrow T + F * F$$

| *Word* | *Products* | *Comments* |
|---|---|---|
| $E$ | $T + E, T$ | Must start with $E$ |
| $T$ | $F * T, F$ | Choose any, expand |
| $F$ | $(E), i$ | Not worth expanding; can't be right |
| $F * T$ | $i * T, (E) + T$ | ditto |
| $T + E$ | $F * T + E, F + E$ | Should get us somewhere |
| $F * T + E$ | $i * T + E, (E) * T + E$ | Not worth expanding |
| $F + E$ | $1 + E, (E) + E$ | Second one not worth it |
| $1 + E$ | ... | This will eventually work |

**Figure III.3 Parsing $1 + 2 * 3$ by Simple Rules of Rewriting**

$$
\begin{aligned}
E &\Rightarrow T + E \\
&\Rightarrow F + E \\
&\Rightarrow 1 + E \\
&\Rightarrow 1 + T \\
&\Rightarrow 1 + F * T \\
&\Rightarrow 1 + 2 * T \\
&\Rightarrow 1 + 2 * f \\
&\Rightarrow 1 + 2 * 3
\end{aligned}
$$

**Figure III.4 Nondeterministic Parse of $1 + 2 * 3$**

| | |
|---|---|
| $E \rightarrow T$ | { $\$\$ = \$1 + \$3$} |
| $E \rightarrow T$ | { $\$\$ = \$1 + \$3$} |
| $E \rightarrow T$ | { $\$\$ = \$1$ |
| $T \rightarrow F * T$ | { $\$\$ = \$1 * \$3$} |
| $T \rightarrow F / T$ | { $\$\$ = \$1 * \$3$} |
| $T \rightarrow F$ | { $\$\$ = \$1$} |
| $F \rightarrow$ any integer | { $\$\$ = \$1$} |
| $F \rightarrow (E)$ | { $\$\$ = \$2$} |

**Figure III.5 Production System for Arithmetic Expressions without Variables**

$$\Rightarrow T + F * 3$$

$$\Rightarrow T + 2 * 3$$

$$\Rightarrow F + 2 * 3$$

$$\Rightarrow 1 + 2 * 3$$

In order to keep such extra derivations out of contention, we impose a rule. We call this the leftmost canonical derivation rule (LCDR): expand the leftmost nonterminal symbol first.

### *Production Systems*

Production systems are useful generalization of grammars. This generalization is also a generalization of FSAs to FSMs. Production systems are an obvious outgrowth of the transformations in the section "Regular Grammar First." The transformation method we used in that section can be applied to finite state *machine* graphs, leaving us with the schema

$$N \rightarrow V\{\text{actions}\}$$

where $N$ is a nonterminal and $V$ is any number of terminals and nonterminals (because of the form of context-free productions).

Production systems are used extensively in artificial intelligence applications. They are also the design concept in parser generating tools such as *LEX* and *YACC*. The basic idea is that pattern matching must terminate successfully before any execution of program segments can occur.

Instead of a long discussion, let's work with something we know: the simple arithmetic language defined in Figure III.3. This is the hand calculator problem from the famous "Dragon Book" (Aho and Ullman 1972). The way to read

$$E \rightarrow T + E \ \{\$1 + \$3\}$$

is as follows (Figure III.5):

1. The symbols preceded by a $ (dollar sign) are semantic in intent; think of the dollar sign as a special version of the letter 's'. $$ is the semantic content of the left-hand symbol $E$, while $\$n$ is the semantic value of the $n$th symbol on the right-hand side.
2. The program that implements the production first applies the production $E \rightarrow T + E$. Two outcomes are possible:

a. The production may fail. Since there may be many recursions, it is possible that the production does not complete. If this is the case, then nothing further other than error processing occurs.
b. If the production succeeds, the *action* segment (after translating the symbols preceded by $'s to memory locations) is then executed.

The question should occur to you: Where is the data? The answer is, on the stack at the locations named by the $'s symbols. But the details are left to you.

## PARSER DESIGN USING PRODUCTION SYSTEMS

### ST Graphs to Production Systems

We could describe the FSA representation lexical rules by productions developed by the following rules:

- Name all the nodes of the graph. These names become the categorical symbols.
- For every arc, there is a *source node* and a *sink node.* The *terminal symbols* are the weights of the arcs.
- For every source node, for example $A$, and every sink node, for example $B$, write down a grammar rule of the form $A \rightarrow B$, where $t$ is the weight of the arc. If there is no such $t$, then discard the rule.

The second requirement is to move the functions from the ST graphs. The convention that has evolved is that we write the productions as above, then enclose the *semantic actions* in braces.

In effect, we have laid the foundations for a very important theorem in formal languages: every language that is specifiable by an ST graph is specifiable by a grammar. The content of the theorem is that many apparently different ways of specifying computations are in fact the same.

### Parse Trees

Remember the idea of a derivation:

$$\begin{aligned} E &\Rightarrow T + E \\ &\Rightarrow T + F * T \\ &\Rightarrow T + F * F \\ &\Rightarrow T + F * 3 \\ &\Rightarrow T + 2 * 3 \end{aligned}$$

$$\Rightarrow F + 2 * 3$$

$$\Rightarrow 1 + 2 * 3$$

The derivation induces a tree structure called a *parse tree*. When thought of as an automaton, the tree is the justification for accepting any sequence of tokens. Thought of as a machine, however, the tree is not the whole story.

The productions define a *function* that does two things: it recognizes syntactically correct inputs and it can compute with the inputs. Instead of the simple derivation above, we can think about tokens for the terminals and values for the categorical symbols. For this discussion to make sense, follow the LCDR. The LCDR requires that the parse be done by always expanding the leftmost categorical symbol. We do this in a top-down manner. Such a parser is called a *predictive parser* or, if it never has to backtrack, a *recursive descent parser*. Let's try this idea on $1 + 2 * 3$. In order to make it clear that this is a production system parse, we enclose the nonterminal symbols in angular braces $\langle\cdot\rangle$. The symbolic name is on the left and the semantic values are on the right. The symbol $\perp$, read "bottom," stands for the phrase "no semantic value defined." See Figure III.6. Line 3 contains the first terminal symbol that is not an operator. The relevant production is

$$F \rightarrow \text{any integer} I\{\$\$ = atoi(\$1)\}$$

In words, it says, "the next input is an integer. Recognize it. If the integer is recognized syntactically, then perform the function *atoi* on the token \$1. If the conversion works, assign the result to \$\$."

The parser returns to the previous level, so we have the $F$ replaced by $T$. The parser now predicts that the atom $\langle atom, + \rangle$ is next. When the parser sees the '+', it reads on.

Repeating the same reasoning, we end up with a final string of

$$\langle T, \langle int, 1\rangle\rangle\langle atom, +\rangle\langle F, \langle int, 2\rangle\rangle\langle atom, *\rangle\langle F, \langle int, 1\rangle\rangle$$

| | | | |
|---|---|---|---|
| 1 | $\langle E, \perp\rangle$ | | |
| 2 | $\langle E, \perp\rangle$ | $\Rightarrow$ | $\langle T, \perp\rangle\langle atom, +\rangle\langle E, \perp\rangle$ |
| 3 | | $\Rightarrow$ | $\langle F, \perp\rangle\langle atom, +\rangle\langle E, \perp\rangle$ |
| 4 | | $\Rightarrow$ | $\langle F, \langle int, 1\rangle\rangle\langle atom, +\rangle\langle E, \perp\rangle$ |
| 5 | | $\Rightarrow$ | $\langle T, \langle int, 1\rangle\rangle\langle atom, +\rangle\langle E, \perp\rangle$ |
| 6 | | $\Rightarrow$ | $\langle T, \langle int, 1\rangle\rangle\langle atom, +\rangle\langle T, \perp\rangle$ |
| 7 | | $\Rightarrow$ | $\langle T, \langle int, 1\rangle\rangle\langle atom, +\rangle\langle F, \perp\rangle\langle atom, *\rangle\langle E, \perp\rangle$ |
| 8 | | $\Rightarrow$ | .... |

**Figure III.6 Derivation Using Production System**

and we start returning. We return to a level that has the '∗' in it. The correct computation takes the 2 and 3 and multiplies them. So we get

$$\langle T, \langle int, 1\rangle\rangle\langle atom, +\rangle\langle T, \langle int, 6\rangle\rangle$$

We can now compute the last operation and get 7. So

$$\langle E, \langle int, 7\rangle\rangle$$

is the final value.

## PARSER DESIGN II

Work on the theory of computability shows that many different forms of algorithm presentation are equivalent in the sense that one form can be converted to another. This is a very powerful idea and we will approach this from a categorical point of view.

### Theoretical Underpinnings

The conversion (compilation if you will) we want to make is between the production system and a typical programming language. The background is based on work on the Chomsky Hierarchy. The Chomsky Hierarchy is a sub-theory of the theory of computation that indicates what abstract computing models are equivalent in the sense that programs in one model produce the same outputs as a particular algorithm specified in other models. In this case, we are interested in two models: production systems and recursive functions. All current programming languages are examples of recursive function systems. Therefore, we will make the task one of going from the formal definition of production systems to some sort of programming language.

We know that there are four parameters in the grammar: *N, T, G, S.* Consider now trying to write a program. What is being specified by *N, T, G,* and *S*? The key to Subtask 1 is to understand that the categorical symbols (symbols in *N*) are really the names for programs and that the programs implement the rules in *G*.

## CASE 16. CONVERTING THE *S* PRODUCTION TO CODE

Use pseudo-code to convert the *S* production of the grammar to a program.

## TESTING THE PARSER: CONTEXT-FREE LANGUAGE GENERATION

The possibility exists of writing a program to develop test inputs for the parser. The design of such a program indicates how we can think about these tests.

The term *language* in our context means a set of *words* that are comprised of only terminal symbols and those words must be ordered in the prescribed fashion. However, there are many intermediate forms derived during a parse. Since we're using a recursive descent parser with a programming discipline of writing a recursive function for each categorical symbol, the appearance of a categorical symbol within the words indicates where programs are called. The key to developing tests is that the parser must work from left to right.

For formality's sake, we can define the *length of a sentence* as the number of (terminal or categorical) symbols in the word. For example, the statement (1 + 2) has length 5. The key to testing, then, is to think of various ways you can generate words that are only comprised of terminal symbols in such a way that all the possible categorical symbols are also involved.

## CASE 17. STRUCTURAL INDUCTION OF WORDS

What is the shortest length terminal word? Show its derivation. What is the next shortest? Show its/their derivation.

Demonstrate a terminal word of length $n = 1, ..., 5$ or show that it is impossible to have a word of that particular length.

Outline how you could rewrite the parsing algorithm to a program that can generate the legal strings of any length. Generate five unique test strings.

We could use the input grammar specified in the specifications to read in the SOL statements. However, one reason to choose a language based on Cambridge Polish is that we can use a very simple input grammar and push many issues into the semantic phases. Close inspection of Cambridge Polish shows that it is just the language of balanced brackets with simple terminal words (names, numbers, strings, etc.) sprinkled throughout.

## CASE 18. DEVELOPING THE PARSING GRAMMAR

One grammar that recognizes balanced brackets is shown below. Let *S* be the *sentential symbol*.

$$
\begin{aligned}
S &\rightarrow [] \\
S &\rightarrow [S] \\
S &\rightarrow SS \\
S &\rightarrow Atom \\
Atom &\rightarrow integer \\
Atom &\rightarrow floating \\
Atom &\rightarrow string
\end{aligned}
$$

**Exercise.** Convince yourself that the grammar actually does what is purported.

**Exercise.** Modify the grammar to accept the lexical entities outlined in the SOL specification.

**Assignment.** Take the grammar and produce a production system that parses incoming tokens and builds a tree that represents the actual *parse tree.*

# Milestone IV

# TYPE CHECKING

**Reference Materials**

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 4th Ed. Upper Saddle River, NJ: Prentice Hall. 1995. Section 4.2; Chapters 5 and 6.

Kenneth C. Louden. *Programming Languages: Principles and Practice.* 2d Ed. Belmont, CA: Thompson, Brooks/Cole. 2003. Chapters 6 and 9.

## OBJECTIVES

**Objective 1:** To produce a static type checker for the language
**Objective 2:** To extend the symbol table to include type information
**Objective 3:** To give you experience using semantic formalisms in designing the type checker
**Objective 4:** To gain experience in modifying and enhancing your programs
**Objective 5:** To develop personal project management skills

## PROFESSIONAL METHODS AND VALUES

Type checking is the first step in semantic processing. We can again use a production system schema, but this time we will pattern match on node content. Designing the type checker requires that we enumerate all the possible legitimate patterns we might see; other patterns encountered constitute type errors.

## ASSIGNMENT

If we had only constants and primitive operations in SOL, then we could easily generate the Forth input by traversing the tree in postorder. However, this would not be very usable because we would not have

variables or user-defined functions. Once we introduce variables and functions, we introduce the need to know what types of values can be stored in variables and what types of values can be used with functions. These points introduce the need for types and type checking. If this were a language like C, then we would now have to do several things:

- Modify the scanner to accept new specially spelled lexemes
- Modify the parser to accept new syntax and construct new trees
- Modify the symbol table to develop new information

The first two are not necessary in SOL because the syntax of the language is so simple, but we will have to elaborate the symbol table information.

The type checker produces (possibly) new trees based on decisions made due to overloading of primitive operators and polymorphism in user-defined functions. This decision logically goes here since the type checker has the correct information in hand at the time of the decision.

## PERFORMANCE OBJECTIVES

1. To develop a formal definition of the type checker from the intended (naïve) semantics for SOL
2. To test the resulting program for correctness based on the formal definition

## MILESTONE REPORT

The milestone report should concentrate on the development of the program. It must include a copy of your formal design documents.

## WHAT IS A TYPE?

The importance of types was not appreciated in assembler or early programming languages. Starting in the late 1960s and early 1970s, programming languages included formal mechanisms for inventing user-defined types. ML was the first language to emphasize types and its initial proposal is attributed to Robin Milner. The extensive use of types is now a hallmark of functional languages, whereas the C-based languages are somewhere in-between.

The concept of types originated with Bertrand Russell (1908) as a way out of his famous paradox in set theory. Various uses of type theory have evolved since then, with its use in logic being the most like that of programming languages. A type $T$ is a triple $(T, F, R)$ where $T$ is a set of

objects, $F$ is a set of functions defined on $T$, and $R$ is a set of relations defined on $T$. In what is known as an "abuse of language," the name of the type $T$ is the same as the name of the set in the definition. This abuse is often confusing to people first learning programming.

We shall not be interested in the theory of how such types are developed and justified. However, since we are developing a functional language, we must understand how to tell whether or not an expression is well formed with respect to types. For example, we would normally say that an expression such as `5 + "abcd"` is incorrectly typed primarily because we cannot conceive of a sensible answer to the expression. When all is said and done, that is the purpose of types: to guarantee sensible outcomes to expressions.

Types come naturally from the problem that the program is supposed to solve. Types in programming are conceptually the same as *units* in science and engineering. A recent mistake by NASA illustrates the importance of each of these ideas.

> The *Mars Climate Orbiter* was destroyed when a navigation error caused the spacecraft to miss its intended 140–150 km altitude above Mars during orbit insertion, instead entering the Martian atmosphere at about 57 km. The spacecraft would have been destroyed by atmospheric stresses and friction at this low altitude. A review board found that some data was calculated on the ground in English units (pound-seconds) and reported that way to the navigation team, who were expecting the data in metric units (newton-seconds). *Wikipedia*.

In this case, *units* play the same role as *types*.

## REASONING ABOUT TYPES

As data structures become more complex, the type system becomes more important. Early programming languages had little or no support for types. Early Fortran and C compilers assumed that undeclared variables were integers and that undeclared functions returned integers. In fact, Fortran had a complicated system of default types based on the first letter of the name.

The programming language ML was originally developed in about 1973 at Edinburgh University for the research community as part of a theorem-proving package called LCF (Logic of Computable Functions). Syntactically, ML looks very much like an earlier language, Algol. The Algol language may have had more impact on programming languages than perhaps any other. Two versions of Algol came out close to one another, in 1958 and 1960. A third came out in 1968 that would be the last, but even Algol 68 (as it was known) had a lasting impact on Unix: the Bourne shell. ML demonstrated that a strongly typed programming language was practical and efficient.

## CASE 19. HISTORICALLY IMPORTANT LANGUAGES

Using the resources available to you, develop a short history of Algol and ML. Also, develop a list of areas these languages impacted and continue to impact today.

## DEVELOPING TYPE RULES

Like SOL, ML allows the programmer to define types that are combinations of primitive types, user-defined types, and type constructors. ML does not require the user to type declare user names because the compiler is able to infer the correct type—a good reason to use languages such as ML.

ML is a strongly typed language, meaning that the compiler type checks at *compile* time and can reliably infer correct types of undeclared names. This is decidedly different from languages such as Fortran and C that require the user to declare types for all names. Eventually, SOL will be strongly typed, but our project does not require us to develop the algorithm to produce the correct inferences. For this milestone, we will consider type coercion but not polymorphism. (Ah, but coercion is a type of polymorphism.)

ML supports polymorphic functions and data types, meaning that you need only define a type or function once to use it for many different data or argument types, avoiding needless duplication of code. This gain in production is from the use of *type variables*; C++ templates are a poor substitute for type variables. C has no such capability: each such need requires that you write the code and that you name the code uniquely. Java's polymorphism would require many methods, each taking different data types and performing similar operations. In other words, C-type polymorphism (C, C++, Java, among others) provides no help in development, requiring the programmer to design, develop, and manage many different aspects that the compiler should be doing.

ML also provides support for abstract data types, allowing the programmer to create new data types as well as constructs that restrict access to objects of a certain type, so that all access must take place through a fixed set of operations. ML also allows for type variables, variables that take on types as their values. ML achieves this through the development of *modules* and *functors*.

The original development of ML was spurred by the need to have a complete type system that could infer correctly undeclared names. This type system is completely checked at compile time, guaranteeing that no type errors exist in the resultant code.

## CASE 20. ML TYPE SYSTEM

Research the ML type system using resources at your disposal. Report on three aspects of the system:

1. How does the compiler infer the correct type of an undeclared variable?
2. What are the strengths of the ML type system?
3. What are the weaknesses of the ML type system?

## DESIGN ISSUES

The first version of SOL must imitate the type checking used by C. Since C is so common, we leave it up to the designers to find the detailed rules. Detailed information on designing rule-based systems is given in Chapter 8.

The metalanguage for type checking is that of logic. Type checking is about *inference,* which is inherently rule-based. Modern expositions in logic use a *tree* presentation. The presentation of the rules follows a simple pattern. You know that logic revolves around the *if-then* schema: *if A then B*. This would be displayed as

$$\frac{A}{B} MP$$

In the context of type and type rules, an example would be for plus.

$$\frac{\text{integer} \quad \text{integer}}{\text{integer}} + \text{integer}$$

## DEVELOPING TYPE-CHECKING RULES

*Every* possible combination of types for any given operator must be considered. Every legitimate combination of types must have a rule and every illegitmate combination must have a rule describing the error action or correction.

A simple approach to analyzing and developing these rules is to develop a matrix of types. We have a very simple type system to consider initially: `bool, integer, float, string,` and `file`. Every unary operator must be considered with each type; every binary operator must be considered with each *pair* of operators.

## TYPE CONVERSION MATRIX

Develop a matrix of types as described above. In each element of the matrix, write the correct Forth conversion statement. For incompatible elements, write a Forth `throw` statement.

In SOL, the standard operators are overloaded, meaning they automatically convert differing types into a consistent—and defined—type, which is then operated on. This feature follows mathematical practice but has long been debated in programming languages. Pascal did not support automatic type conversion, called mixed-mode arithmetic at the time. SOL does support mixed-mode expressions (just like C); therefore, the design must include type checking and automatic conversion. Because of overloading, there are implicit rules for conversions. The design issue is to make these implicit rules explicit.

Standard typography in textbooks as described above does not have a standard symbology for this automatic conversion. Therefore, we'll just invent one. It seems appropriate to simply use functional notation. All the type names are lower case, so upper-case type names will be the coercion function. This doesn't solve the whole problem, since it must still be the case that some conversions are not allowed.

As an example,

$$\frac{\text{FLOAT(integer)} \quad \text{float}}{\text{float}} + \text{integer-float conversion}$$

Such a rule is interpreted as follows. An example to work from is `[+ 1 3.2]`. The implicit rule requires that the integer be converted to a float. We note that by having `FLOAT(integer)`.

An example of something that should not be allowed would be

$$\frac{\text{FLOAT(integer)} \quad \text{float}}{\text{float}} + \text{integer-float conversion}$$

## SPECIFICATION FOR THE WHOLE TYPE CHECKER

Because the SOL language is a functional language, virtually every construct returns a value: the `if` statement, for example. Therefore, the entire expression must be checked, starting with the `let` and the `type` statements. Did you notice that there is no `return` statement? Type checking SOL is far more complicated than checking simple arithmetic statements because the `if` statement can only return *one* type. Therefore, `[if` *`condition`* `1 2.0]` is improperly typed.

In order to design the whole type checker, you will have to resort to a design technique known as structural induction. Structural induction requires that we develop a tree of all the subexpressions that make up an expression. For example, what are all possible expressions that can start with a `let`? Here is a partial analysis.

1. The general form of the `let` statement is

   `[let [` *definiendum definiens* `] ... ]`

2. The definiendum can be one of two forms: a simple variable *name* or a *functional expression*.

   `[let [ [` a expresssion`] ... ] ]`
   `[let [ [ [`a *arguments*`]` expresssion`] ... ] ]`

There could be no, one, or many arguments, of course.

The point here is that you must develop a recursive function that can traverse any proper input: this is the reason to develop the printer program in the parser milestone. This printer program is the scheme for all the other milestone programs.

## CASE 22. FORM OF INPUT

It is simple to read the input if you do not try to put any semantic meaning onto the result. Such processing will not work for type checking because the type checking rules are driven by the form and content of the structures. In order to design the type checker, you must first describe the possible structures through structural induction.

Using the grammar from Milestone III, and using substitution, develop a design of the type checker, making each "nonterminal" a recursive function.

## THE SEARCH ALGORITHM

The algorithm for the type checker is more complicated than the parser. The parser is not comparing the structure to another structure, whereas the type checker is checking an expression against a rule. For example, consider the question of whether or not `[+ 1 2.5]` is correctly typed.

In order to discuss the problem, we first have to realize that there are different addition routines, one for each possible combination of allowed arguments. Say there are three allowed arguments: `bool`, `integer`, and `float`. This means there are nine possible combinations of arguments for '+'.

## CASE 23. SHOULD `BOOLS` BE ALLOWED?

While we may want to consider `bools` in the arithmetic expressions, should they be allowed? Mathematical convention would argue for inclusion.

| *Forth Word Name* | *Return Type* | *Argument 1* | *Argument 2* |
|---|---|---|---|
| +_int_int | int | int | int |
| +_int_float | float | int | float |
| +_float _int | float | float | float |
| +_float _float | float | float | float |

**Figure IV.1 Design Matrix for Addition**

Regardless of the outcome of the above case, there is more than one possible legitimate argument type situation. Suppose we decide that `bools` should not be used; then there are still four separate possibilities, given in Figure IV.1. We can take advantage of Forth's flexibility in word names by defining a unique operator for each pair. For example:

```
: +_int_float ( arg n -- ) ( float f -- f )
   s>d d>f f+
;
```

This is simple enough for each element in the type conversion matrix.

## UNIFICATION AND TYPE CHECKING ALGORITHM

Let's consider how the full type checking algorithm should proceed.

1. The type checking algorithm is entered with a tree with form

   $$[\ a_0\ a_1\ \ldots\ a_n\ ]$$

   Based on the Cambridge Polish scheme, $a_0$ must be a function and the $a_i$, $i = 1, \ldots, n$ can be either atoms or expressions.
2. Because $a_0$ must be predefined, it should be in the symbol table.
3. Suppose, for example, that $a_0$ is $+$. The symbol table entry must contain the list of possible types as outlined in Figure IV.1.
4. We need to search the list of possible types to find the (first) one that matches the arguments $a_1, \ldots, a_n$. In the example at hand, $n = 2$.
5. Recursively determine the type of $a_1, \ldots, a_n$. Match these types to the list of possible types.
6. When we find a match, we know both the return type and the operator that should be applied.
7. Modify the tree to replace $a_0$ with the correct operation.
8. Return the type.

This algorithm is a simplifed form of the *unification algorithm*, the fundamental evaluation algorithm used in Prolog. The above algorithm is much

simpler because we do not have type variables as do ML and the other functional languages.

## CASE 24. TYPE CHECKING AND THE SYMBOL TABLE

The type checker will rely on the symbol table to produce the list of possible functions that can be substituted for $a_0$. This case and the next should be completed together, because the two solutions must be able to work together.

For the symbol table, develop a data structure for the operators. Keep in mind that when we introduce user-defined functions, they will have the same capabilities as the primitive operators.

Along with the data structure, you must develop a process to load the symbol table initially. For hints, consider how the SOL type definition process works by consulting the specification.

## CASE 25. DEVELOPING THE TYPE CHECKING ALGORITHM

On approach to developing the unification algorithm, you need really only consider a few cases. One specific case is

```
[ +  1  2 ]
[ +  1  [ *  3  4 ] ]
[ +  [ *  3  4 ] 5 ]
```

Based on these cases, develop the search algorithm and complete the design of the symbol table.

# Milestone V

# ELEMENTARY COMPILING: CONSTANTS ONLY

**Reference Materials**

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 4th Ed. Upper Saddle River, NJ: Prentice Hall. 1995. Chapters 5 and 8.

Kenneth C. Louden. *Programming Languages: Principles and Practice*. 2d Ed. Belmont, CA: Thompson, Brooks/Cole. 2003. Chapter 8.

## OBJECTIVES

**Objective 1:** To produce output suitable for use by Forth
**Objective 2:** To compile constant only operations (*No program should contain variables for this milestone.*)
**Objective 3:** To give you experience using semantic formalisms in designing the code generator
**Objective 4:** To test the actual use of the parse tree
**Objective 5:** To gain experience in modifying and enhancing your programs
**Objective 6:** To develop personal project management skills

## PROFESSIONAL METHODS AND VALUES

The principal formal concept in this milestone is the use of *structural induction*, discussed in Chapter 6. As with the parser and type checker, the general principle is the production system schema: pattern match to guarantee consistency, then application of a transformation.

## ASSIGNMENT

We are now ready to produce a minimal compiler. We have developed the parse tree and we can print that tree; the outline of the printer routine is the basis for this milestone. We now finish the first development cycle by adding a code generator.

The type checker has constructed a tree that is type correct. In the previous milestone, the type checker chose the correct polymorphic routine name.

## PERFORMANCE OBJECTIVES

1. Develop a formal definition of the code generation algorithm from the intended (naïve) semantics for SOL based on the previously developed parser.
2. Test the resulting program for correctness based on the definition.

For this milestone, you generate Forth code and run that code to demonstrate the correctness of the result. Your tests must show that the operations correctly implemented based on their customary definitions in boolean, integer, floating point, string and file operations. A basic approach for testing these primitive operators is to choose *simple* values that you can calculate easily. It is not a valid test case if you can't predict the result ahead of time. Therefore, you should keep the following in mind: keep the test statements short.

For the most part, this milestone is a straightforward binding exercise: choosing which word to bind to the SOL operator of a given type. This is primarily an exercise in loading the symbol table with correct information.

The details of the primitive operators are given in Figures V.1, to V.5. The special values are listed at the bottom of Figure V.5.

| *Operator Name* | *Operator Symbol* | *Semantics* |
|---|---|---|
| | *Primitive Data* | |
| | *When no semantics are given, C is intended* | |
| | Boolean | |
| And | *A*&*B* | Same as && in C |
| Or | *A*\|\|*B* | |
| Not | *A* | |
| Implies | *A* => *B* | False when *A* is true and *B* is false |
| Biconditional | *A* <=> *B* Logical equals | |

**Figure V.1 Booleans**

| | *Integers* | |
|---|---|---|
| Plus | $A + B$ | |
| Minus | $A - B$ | |
| Unary Minus | $-A$ | |
| Times | $A * B$ | |
| Divide | $A / B$ | |
| Remainder | $A \% B$ | |
| Power | $A \wedge B$ | Integer A raised to an integer power B |
| Less | $A <$ | |
| Equal | $A = B$ | Same as C's === |
| Greater | $A > B$ | |
| Less than or equal to | $A \leq B$ | |
| Greater than or equal to | $A \geq B$ | |
| Not equal | $A = B$ | |

**Figure V.2 Integers**

| | *Floating Point* | |
|---|---|---|
| Plus | $A + B$ | |
| Minus | $A - B$ | |
| Unary Minus | $-A$ | |
| Times | $A * B$ | |
| Divide | $A / B$ | |
| Remainder | $A \% B$ | |
| Power | $A \wedge B$ | A float, B either float or integer |
| Less | $A <$ | |
| Equal | $A = B$ | Same as C's === |
| Greater | $A > B$ | |
| Less than or equal to | $A \leq B$ | |
| Greater than or equal to | $A \geq B$ | |
| Not equal | $A = B$ Sine | $\sin(A)$ |
| Cosine | $\cos(A)$ | |
| Tangent | $\tan(A)$ | |
| Exponential | $\exp(A)$ | $e^A$ |

**Figure V.3 Floats**

## STRING IMPLEMENTATION

One nontrivial issue is the full development of string support. The formal concept here is CRUD: creating, reading, updating, and destroying strings. Creating and destroying string constants introduces two storage

| | | |
|---|---|---|
| | *Strings* | |
| Concatenation<br>`strcat` in C | $A + B$ | |
| Insert | insert($A$, $B$, $C$)<br>Insert character $B$ at location $C$ in string $A$ | |
| Character of | charof($A$, $B$) | Return the $B$th character of string $A$ |
| Less | $A <$ | |
| Equal | $A = B$ | Same as C's === |
| Greater | $A > B$ | |
| Less than or equal to | $A \leq B$ | |
| Greater than or equal to | $A \geq B$ | |
| Not equal | $A = B$ | |

**Figure V.4 Strings**

| | | |
|---|---|---|
| | *Files* | |
| Open | open($A$) | $A$ is a path expression as a string<br>Returns a file value |
| Close | close($A$) | Closes open file $A$ |
| Test end of file | endoffile ($A$) | returns true is file $A$ is at end of file |
| Read boolean value | readbool($A$) | File $A$ is read for boolean |
| Write a boolean value | writebool($A$, $B$) | Write boolean value $B$ to file $A$ |
| Read integer value | readint($A$) | File $A$ is read for integer |
| Write a integer value | writebool($A$, $B$) | Write integer value $B$ to file $A$ |
| Read float value | readfloat ($A$) | File $A$ is read for float |
| Write a float value | writefloat ($A$, $B$) | Write float value $B$ to file $A$ |
| Read string value | readstring($A$) | File $A$ is read for string |
| Write a string value | writestring($A$, $B$) | Write string value $B$ to file $A$ |
| | *Special Values* | |
| Pi | Pi | Closest representable value of $\pi = 3.14\ldots$ |
| Standard in | stdin | |
| standard out | stdout | |

**Figure V.5 Files and Special Constants**

management issues: (1) obtaining and releasing memory and (2) knowing when it is safe to destroy the string.

### IF AND WHILE

This is the appropriate place to implement `if` and `while` statements. Admittedly, the testing is fairly trivial, except for the `while`; complete testing of the `while` must wait.

## MILESTONE REPORT

The milestone report should concentrate on the development of the program. It must include any design decisions that you make that are not also made in class; all such decisions should be documented by explaining why the decision was made.

# Milestone VI

# SCOPING AND LOCAL VARIABLES

**Reference Material**

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 4th Ed. Upper Saddle River, NJ: Prentice Hall. 1995. Chapters 6 and 7; Section 9.2.

Kenneth C. Louden. *Programming Languages: Principles and Practice*. 2d Ed. Belmont, CA: Thompson, Brooks/Cole. 2003. Chapter 8.

## VARIABLES AND LOCAL VALUES

### Objectives

**Objective 1:** To produce output suitable for use by Forth
**Objective 2:** To introduce user-defined names
**Objective 3:** To give you experience using semantic formalisms in designing the code generator
**Objective 4:** To gain experience in modifying and enhancing your programs
**Objective 8:** To develop personal project management skills

### Professional Methods and Values

Local names and values are a binding issue: Under what circumstances does the evaluation of a name result in a correct value? The design and implementation of local variables revolves around understanding how the execution sequence of a program is influenced by calls and returns.

### Assignment

We now have a compiler that can take in any arithmetic statement and can generate code for the two obvious commands: *if* and *while*. We now want to add commands that allow the programmer to have meaningful names. There are some issues:

1. Is the variable's scope *global* or *local*?
2. Is the variable's value *persistent* or *transient*?
3. How to reference the value of a local variable in the local activation.
4. How to reference the value of a local variable in a nonlocal activation.
5. How to reference the value of a global variable.
6. How to assign values to either global or local variables.

### Performance Objectives

1. Develop a formal definition of the code generation algorithm from the intended (naïve) semantics for SOL based on the previously developed integer code generator.
2. Test the resulting program for correctness based on the formal definition.

### Milestone Report

The milestone report should concentrate on the development of the program. It must include a copy of your formal design documents.

## SCOPING AND SOL

The concept of *scope* is intimately connected to the concept of *visibility of names* and with storage management. Storage management issues are discussed in detail in Chapter 10. Scope rules determine how long a name is bound to a value.

Specifically in SOL, there are several types of scoping environments inherited from most block-structured languages such as C. For this milestone, we have the following scope protocols:

1. File scope
2. Function header scope
3. Function body scope

The function header scope is discussed separately because it is a composite of file scope and body scope.

| *Scope Protocol* | *Variables* | *Functions* |
|---|---|---|
| File Scope | | |
| Header Scope | | |
| Body Scope | | |

**Figure VI.1 Analysis of Scope-Storage Interactions**

## CASE 26. BINDING TIMES AND SCOPE

Consideration of scope comes from considering the binding time of a name with its definition. In the simple programming language context, a name can only have one active binding at a time: otherwise, there is ambiguity. We should note that multiple inheritance in object-oriented languages such as C++ introduces ambiguity. Therefore, binding time requirements lead to the need for scoping. What properties should *scope* have?

The original idea of scope and binding time came from logic. The introduction of *quantifiers* by Pierce, Frege, and others in the 1880s introduced the problem of name-definition ambiguity. The basic issue in logic is the nesting and duration of definitions. Programming adds another dimension: storage. Whenever a name is defined, there is some consequence for memory management. Although individual routines are accounted for at compile time, variables have other issues, such as access and duration of a value. All nonpersistent, local storage must be reclaimed when the execution leaves the scope.

## CASE 27. SCOPES: FILLING IN THE DETAILS

For each of the scope protocols listed above, determine the consequences for memory management. Do this by displaying a table (as shown in Figure VI.1) that lists the rules for each possibility.

## COMPILING ISSUES FOR SOL

### Symbol Table Issues

Introduction of local variables and scope alters the way we think about the symbol table. Up to this point, we have treated the symbol table as a repository for system-defined names and symbols. It was used primarily in its lookup capacity. That no longer is possible.

To understand the issue, consider again the issues of scope; more to the point, nested scopes. A simple C program segment is shown in Figure VI.2: the two `i` definitions are completely legitimate because the left brace opens a new scope. The symbol table must account somehow for this nesting.

```
{
   int i = 5;
   int j = 7;
...
      {
         inti = 12;
         printf("%d", i + j);
      }
}
```

**Figure VI.2 Illustration of Scoping Issues**

## CASE 28. HANDLING NESTED SCOPES

Two different approaches come to mind on how to handle the nesting of scopes. One would be to have a separate symbol table for each scope. The second approach would be to make the symbol table more complicated by keeping track of the scopes through some sort of list mechanism. One such mechanism would be to assign a number to each different scope and use the number to disambiguate the definitions. This approach keeps a stack of numbers. When a scope opens, the next number is placed on the stack; when the scope closes the current number is popped and discarded. Every definition takes as its scope the number at the top of the stack.

Use each of these approaches in a design and determine the running time ("Big O") and the space requirements for each. From this, make a decision on which approach you will use in this milestone.

### Space Management Issues

SOL has an important problem for local variable compilation: the local variables "live" on the stack. However, Forth has several different stacks: the *argument stack* for integers and integer-like values (such as pointers) and a *floating point* stack. There are other stacks that are handy to know about, such as the return stack and the control stack. For this milestone the argument and floating stacks are the focus. To illustrate an issue, the routine with header

```
float foo(int a, float b)
```

must have its `a` argument on the argument stack and the `b` argument on the floating point stack; the return value must be pushed onto the floating point stack.

This problem is made worse by the fact that local variables are also on the stack. In fact, the local variables are defined early on in the computation, so the compiler must be able to determine where a given value is and how to move that value to the top of the stack, as well as assign the top of the stack to a variable.

In effect, the compiler must simulate the running of the program because the dynamic program will be moving values around on the stack in accordance with the instructions laid down by the compiler.

## CASE 29. GETTING TO THE DETAILS: SCOPE AND FORTH

Develop a *simple* SOL program that is comprised only of integer arguments and integer variables; three of each should be sufficient. Now write several arithmetic operations that compute values and store them in local variables. Write the equivalent Gforth program. Use this experience to develop the "simulator" needed to compile code.

The basic issue facing the compiler in this milestone, then, is the simulation of the stack. In order to simulate the stack, you must have a list of rules of the effect of each operation on the stack. This is actually a straightforward exercise: develop a table that lists the operations on the rows and stacks information across the columns.

## CASE 30. STACK EFFECTS

Develop a *stack effect table* using the following outline as a starting point:

| Operation | Input<br>Argument Floating | Output<br>Argument Floating |
|---|---|---|

Your design should designate operations for *all* defined operations, including those you may have introduced to simplify type checking.

Once the table is complete, study the resulting pattern of operations. There are relatively few unique patterns. A simple approach to the problem of looking up what pattern to use is to extend the information in the symbol table to include the stack information.

# Milestone VII

# USER-DEFINED FUNCTIONS

**Reference Materials**

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 4th Ed. Upper Saddle River, NJ: Prentice Hall. 1995. Chapter 9.
Kenneth C. Louden. *Programming Languages: Principles and Practice.* 2d Ed. Belmont, CA: Thompson, Brooks/Cole. 2003. Chapters 7 and 8.

## OBJECTIVES

**Objective 1:** To produce code generation for nonrecursive and recursive functions
**Objective 2:** To extend the symbol table to include function header information
**Objective 3:** To give you experience using semantic formalisms describing lambda expressions
**Objective 4:** To gain experience in modifying and enhancing your programs
**Objective 5:** To develop personal project management skills

The principles invoked in this milestone are the relationship of the λ-calculus and program evaluation.

## ASSIGNMENT

We now have a complete compiler, but not a very usable one because we do not have functions. You must now include the headers for functions, arguments, and return types. We use the specification for the syntax and use your understanding of programming to produce the semantics.

## PERFORMANCE OBJECTIVES

1. Develop a formal definition of the intended (naïve) semantics for SOL based on the previously developed assembler and your knowledge of programming.
2. Test the resulting program for correctness based on the formal definition.

## MILESTONE REPORT

The milestone report should concentrate on the development of the program. It must include a copy of your formal design documents.

## BINDING THE FUNCTION NAME TO THE FUNCTION

Chapter 9 describes the fundamental theoretical understanding of functions and function evaluation: the λ-calculus. Some languages, such as ML and OCAML, use the descriptive term *anonymous function* to describe the role of the λ-calculus: to describe function bodies. But how does one bind the λ-expression to a name?

This actually brings up a deeper issue: How do we think about binding any name to its definition? The basic approach is not unlike normal *assignment* in a program. For example, in the C statement `int a = 5;` we think of the location denoted by `a` as naming the location of integer 5. Does this work for functions? For example, how can one rename a function? How about

```
double(*newsin)(double) = sin;
```

If we then write `newsin(M_PI/4)` do we compute $\sin \pi/4$? Of course.

We're almost there. Suppose we don't want the value of `newsin` to be changeable. Again in C, the way to do that is to use `const` in the right place.

```
double(* const newsin)(double) = sin;
```

The purpose of this exercise is to demonstrate function names are nothing special, but before we finish this investigation, what is happening in terms of the λ-calculus? What is `sin` in terms of λ-calculus? Without investigating algorithms to compute the sine function, we can at least speculate that the name `sin` is a λ-expression of one argument. In other words, the assignment of `sin` to `newsin` is simply the assignment of the λ-expression.

The problem is type. Chapter 9 does not explore the issue of types, leaving it intuitive. The λ-calculus was introduced to explore issues in the natural numbers and integers. The development of the expanded theory came later but, for our purposes, can be seen in such languages as Lisp, ML, and OCAML. Types are bound to names using the colon ':'. Thus,

$$\lambda x : \text{integer}.\ \texttt{body} : \text{integer}$$

directs that the argument `x` is an integer and that the result also be an integer.

## RECURSIVE AND NONRECURSIVE FUNCTIONS

Early in the history of computer science, special status was accorded to recursive functions. Generally, recursive functions were treated as something to avoid, and often were required to have special syntax. C went a long way to change that since it has no "adornments" on the functions' header. We take this approach in SOL: all functions are by nature recursive.

This approach works fine for the naïve approach, but is not suitable to the theoretical underpinnings. In the theoretical literature, the following issue raises its head:

$$let\ fibonacci(n) = \lambda n.$$
$$if\ n < 2\ then\ 1\ else fibonacci(n-1) + fibonacci(n-2)$$

Notice that there are three instances of the word *fibonacci*, one on the left and two on the right. Question: How do we know that the one on the left is invoked on the right? Answer: Nothing stops us from the following:

$$let\ foobar(n) = \lambda n.$$
$$if\ n < 2\ then\ 1\ else fibonacci(n-1) + fibonacci(n-2)$$

in which case $foobar(2)$ is undefined. For this reason, languages based on the λ-calculus have a second `let` statement. This second statement is written `letrec`; the `letrec` guarantees that the recursive name is properly bound. In keeping with this convention, extend SOL to include `letrec`.

## IMPLEMENTATION ISSUES

Functions add three issues to the implementation: (1) passing of arguments, (2) the return of computed values, and (3) storage management. It should be clear that the stack is the communications medium.

### Parameter Passing and Value Returning

Because Gforth is a stack-based system, the development of protocols for parameters and returns is forced on us.

## CASE 31. PASSING PARAMETERS IN SOL

The protocol for passing arguments is not particularly intricate for a simple implementation such as ours. Because of the ability to type arguments, there should be no type mistakes. You will have to develop rules for arguments that must reside on different stacks, such as the *argument* and *float* stacks.

Write down the rules for passing arguments of the primitive types. One issue to address is the question of what the stack state(s) should be and when the final states are assumed.

Returning values is more complicated than passing because the *calling routine* must retrieve those values and dispose of them properly, leaving the *stacks* in their correct state. Potentially, there could be many return values for a given call. However, if you think about it carefully, you can see that returning values is not completely unlike passing arguments.

## CASE 32. RETURN VALUES

Modify the protocol for passing arguments to include the return protocol. For this initial compiler, assume that only one return value is possible.

What will you do if there is *no* return value?

### Storage Management Issues

Milestone VII provided the basic mechanisms for stack management *within* the body of a function. However, there is still a question of what should happen to the parameters themselves? The answer is clear: the basic concept of a function in our constructive world is that we replace the function and its parameters with the result. Therefore, the stack must be returned to the state it was in *before* the first parameters were pushed on *and then* the return value is placed on the correct stack.

The last issue in this regard is the assignment of the result once control has returned to the *calling routine*. There are several possibilities:

1. `[assign a [foo x]]`. In this case, `assign` naturally accounts for the extra stack location.
2. `[assign a [foo x]]` and `[foo x]` returns no value. In this case, we have a type error.
3. `[foo x]` returns a value but it is not assigned to any name. In this case the value must be `popped`; the *called* routine does not have any information about the use of the values.

# Milestone VIII

# USER-DEFINED COMPLEX DATA

**Reference Materials**

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 4th Ed. Upper Saddle River, NJ: Prentice Hall. 1995. Chapter 9.

Kenneth C. Louden. *Programming Languages: Principles and Practice.* 2d Ed. Belmont, CA: Thompson, Brooks/Cole. 2003. Chapters 7 and 8.

## OBJECTIVES

**Objective 1:** To produce code generation for complex data types: arrays and structures

**Objective 2:** To extend the symbol table to include type information for complex data organizations

**Objective 3:** To give you experience using semantic formalisms describing data referencing operations

**Objective 4:** To gain experience in modifying and enhancing your programs

**Objective 5:** To develop personal project management skills

## PROFESSIONAL METHODS AND VALUES

The methods and values of this milestone harken back to the days when assembler languages were all we had. In assembler language, all processing is carried out by the programmer. Designing, implementing, and manipulating complex data structures at the assembler level are fraught with issues requiring complete concentration. Users of modern programming languages have extensive libraries for manipulating standard data structures (such as C's `stdlib`). However, such libraries cannot fulfill all contingencies; therefore, compilers must support user-defined complex structures.

## ASSIGNMENT

We now have a complete compiler, including functions, but we do not have facilities for user-defined complex data. We use the specification for the syntax and now use your understanding of programming to produce the semantics, primarily the semantics of pointers and pointer manipulation.

## PERFORMANCE OBJECTIVES

1. Develop a formal definition of the intended (naíve) semantics for SOL based on the previously developed assembler and your knowledge of programming.
2. Test the resulting program for correctness based on the formal definition.

## MILESTONE REPORT

The milestone report should concentrate on the development of the program. It must include a copy of your formal design documents.The metacognitive portion should address your understanding of the semantics of the constructs.

## FUNDAMENTAL CONCEPTS

Chapter 10 describes the fundamental operations for defining and accessing complex storage structures. The concepts are

1. The amount of storage required must be computed based on the sizing rules of the computer memory system in use.
2. Every allocated memory area has a base address.
3. Every reference is calculated from the base address with an offset.
4. The size of the referenced data is computed by the sizing rules.

In Chapter 10, we develop formulas to calculate the location of an element within an array. The ideas are encapsulated in the C program (shown in Figure VIII.3) with output shown in Figure VIII.1.

The same schema works for structures, except that the computation of the offset is more complicated. As an example, look at the C program shown in Figure VIII.2. From this, we can infer that the offsets computed and then added to the base pointer `(char*)(&blspace)` is the same as using the more familiar "dot" notation.

```
Element vec[0] has int offset 0
Element vec[0] has byte offset 0
Element vec[1] has int offset 1
Element vec[1] has byte offset 4
Element vec[2] has int offset 2
Element vec[2] has byte offset 8
Element vec[3] has int offset 3
Element vec[3] has byte offset 12
Element vec[4] has int offset 4
Element vec[4] has byte offset 16
```

**Figure VIII.1 Output of Program Processing a Five Element Integer Array.**

```
sizeof(char) 1
sizeof(int) 4
sizeof(int*) 4
sizeof(double) 8
sizeof(struct a1) 16
sizeof(union bl) 16
0 0 0 30ffffff
offset to d: 0
offset to i: 8
offset to c1: 12
```

**Figure VIII.2 Output of `struct` Text Program.**

## WRITING ARRAYS AND STRUCTURES IN SOL

The SOL syntax makes it easy to develop support for complex data. We can use the type checker to ensure we have the right number of indices as well as taking an interesting view of complex data. To do that, we must make arrays fully functional. First, recall the CRUD acronym for developing data.

### Array Development

1. *Creation*. The creation of an array requires (a) computing the size of the array as discussed in Chapter 10 and (b) allocating the space through the operating system. A straightforward function for this would be [`createarray` *type dimension*$_1$ ... *dimension*$_n$].
2. *Reading a value from an array*. We have illustrated the concepts of reading a value in both Chapter 10 and earlier in this milestone.

```
#include <stdio.h>
#include <assert.h>

struct a1 {
  double d;
  int i;
  char c1;
};

int vec[5];

int main(int argc, char* argv[] ) {
  int i;
  int p;
  int *pi, *p0;
  char *ci, *c0;
  for( i = 0; i < 5; i++ ) vec[i] = i;
  p0 = &vec[0];
  c0 = (char*)(&vec[0]);
  for( i = 0; i < 5; i++ ) {
    pi = &vec[i];
    ci = (char*)(&vec[i]);
    /*p = &((char*)vec[i])-&((char*)vec[0]);*/
    p = pi - p0;
    printf(" Element vec[%d] has int
offset %d\n",i,p);
    assert( vec[i] == *(p0+(pi-p0)));
    printf(" Element vec[%d] has byte
offset %d\n",i,(ci-c0));
    assert( vec[i] == *(int*)(c0+(ci-c0)));
  }
}

struct a1 {
 double d;
 int i;
 char c1;
};
```

**Figure VIII.3 Class Syntax.**

```
union bl {
 struct a1 ta1;
 int blast[sizeof(struct a1)/4];
};

int main(int argc, char* argv[] ) {
 union bl bl_space;
 int i;
 int p;
 printf( "sizeof(char)\t%d\n", sizeof(char));
 printf( "sizeof(int)\t%d\n", sizeof(int));
 printf( "sizeof(int*)\t%d\n", sizeof(int*));
 printf( "sizeof(double)\t%d\n", sizeof(double));
 printf( "sizeof(struct a1)\t%d\n",sizeof(struct a1));
 printf( "sizeof(union bl)\t%d\n", sizeof(union bl));

 for( i=0; i<4; i++) bl_space.blast[i]=-1;
 bl_space.ta1.d = 0.0;
 bl_space.ta1.i = 0;
 bl_space.ta1.c1 = '0';
 printf("%x %x %x %x\n", bl_space.blast[0],
 bl_space.blast[1], bl_space.blast[2],
 bl_space.blast[3]);
 p = (char*)(&bl_space.ta1.d)-(char*)(&bl_space.ta1);
 printf("offset to d: %d\n", p);
 p = (char*)(&bl_space.ta1.i)-(char*)(&bl_space.ta1);
 printf("offset to i: %d\n", p);
 p = (char*)(&bl_space.ta1.c1)-(char*)(&bl_space.ta1);
 printf("offset to c1: %d\n", p);
 assert( bl_space.ta1.d==(*((char*)&bl_space+pd)));
 assert( bl_space.ta1.i==(*((char*)&bl_space+pi)));
 assert( bl_space.ta1.c1==(*((char*)&bl_space+pc1)));
}
```

```
[structure  → structure-name
              optional type declaration
              optional let statements for variables
              ... ]
```

**Figure VIII.3 Class Syntax. (*Cont.*)**

Reading a value means (a) computing the offset, (b) computing the address from the base address, and (c) moving the data from the array. The following syntax suffices: [`getarray` *array dimension*$_1$ ... *dimension*$_n$].

3. *Writing a value into an array.* Writing ("putting") a value into a location of an array is almost identical to reading, except that the new value must be part of the function [`putarray` *array value dimension*$_1$ ... *dimension*$_n$].
4. *Destroying an array.* Since we use the operating system to allocate the space for the array, it is only natural to use the operating system to deallocate ("destroy") the data—something like [`destroyarray` *array*].

However, far more interesting is the question of *when* it is safe to destroy such a structure. Java has one answer: garbage collection. We do not pursue garbage collection here, but the serious reader should consider the subject (Jones and Lins 1996). For our application, we leave this decision to the programmer.

### Structure Development

The same line of reasoning and similar functions result from designing support functions for structures: `createstructure`, `readstructure`, `putstructure`, and `destroystructure`.

Forth has built in structure processing, so the translation of structure functions is less difficult than one might initially estimate. Once again, the issue is computing the offsets.

## CASE 33. Gforth STRUCTURES

Read the ANS specification for Forth. Using this as a guide, develop the semantics of SOL's support for structures.

### Defining Structures

Unlike arrays, structures, by definition, do not have a simple accessing method. Structures are ordered field definitions, *ordered* in the sense of *first*, *second*, ... . Referencing a data item by its ordinal position would be very error-prone in large structures. Therefore, the usual definition of structures is

1. Start a structure with a keyword. The keyword begins a definitional scope.
2. List the component parts of the structure.
3. Terminate the structure to end the definitional scope.

As an example, C structures look like

```
struct list { double real; struct list *next; }
```

The term `struct` is the keyword and the scope is indicated by the braces. Because structures can be self-referential, the name (`list`) is required in this case.

Creating structures instances is done with `createstructure`, which has the obvious form

```
[createstructure structure-name]
```

**Notice** that arrays and structures are types. The introduction of structures requires changes to the type checker to allow for new types. Languages in the ML family also allow for type variables, which greatly enhances the power of the type system.

## Reading and Writing Values

Structures also introduce name ambiguity issues, just as introducing local variables introduced ambiguity. In the case of local variables, the *last defined* rule is used. Such a simplistic rule will not work for structures. Looking at C, we know that if the variable `node` is defined to have type `struct list`, then the expression `node.next` references the second value.

In SOL, the "dot" would get lost next to the left bracket. A different tack is needed. It should be noticed that a structure reference to a field is very much like a function call and the type checker can be used to verify the relation of the field to a data value. Just as with arrays, we introduce two functions: `getstructure` and `putstructure`.

```
[getstructure {\em structure-instance}
                      {\em field name}]\\
[putstructure {\em structure-instance}
                      {\em field name} {\em value}]
```

## Destroying Allocated Structures

Structures that were `allocated` may be `freed` to reclaim space. There is no implied requirement for garbage collection in the SOL specification, although this is an important decision: to have or not to have garbage collection.

# Milestone IX

# CLASSES AND OBJECTS

**Reference Material**

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 4th Ed. Upper Saddle River, NJ: Prentice Hall. 1995. Chapter 9.
Kenneth C. Louden. *Programming Languages: Principles and Practice*. 2d Ed. Belmont, CA: Thompson, Brooks/Cole. 2003. Chapters 7 and 8.

## CLASSES AND CLASS

### Objectives

**Objective 1:** To produce code generation for user-defined classes
**Objective 2:** To extend the symbol table to include function header information
**Objective 3:** To give you experience using semantic formalisms used to design semantics
**Objective 4:** To gain experience in modifying and enhancing your programs
**Objective 5:** To develop personal project management skills

### Professional Methods and Values

The principles invoked in this milestone are those of the class-oriented paradigm: the encapsulation of names and inheritance.

## ASSIGNMENT

We now have a complete compiler with functions and complex data. All of the development so far makes it possible to implement classes.

## MILESTONE REPORT

The milestone report should concentrate on the development of the program. It must include a copy of your formal design documents.

## ENCAPSULATION AND INFORMATION HIDING

In software engineering, encapsulation means to enclose programming elements inside a larger, more abstract entity. Encapsulation is often taken as a synonym for *information hiding* and *separation of concerns*.

Information hiding refers to a design discipline that hides those design decisions that are most likely to change, protecting them should the program be changed. This protection of a design decision involves providing a stable interface that shields the remainder of the program from the implementation details. The most familiar form of encapsulation is seen in C++ and Java; however, polymorphism is a form of information hiding, also. Separation of concerns is the process of partitioning a program into distinct features. Typically, *concerns* are synonymous with *features* or *behaviors*.

Until the advent of structures, and especially structured programming, in programming languages, all names visible in a scope were visible everywhere in that scope. Structures altered that by making field names invisible within a scope and requiring a special dereferencing operator (the "dot") required. The next step was taken when classes *encapsulated* names, meaning that some names could only be referenced from *within* the class. In this sense, encapsulation is a form of information hiding because `private` names (they could be functions or variables) are a form of design decision that could change if the algorithms change. One step toward objects is easy—make the symbol table associated with a structure unusable *outside* the structure.

## INHERITANCE

In the programming language context, inheritance is a mechanism for forming new classes of data using previously defined classes and primitive data types. The "previously defined" classes are called *base classes* and the newly formed classes are called *derived classes*. Derived classes inherit fields (*attributes*) from the base classes. The attributes include data definitions, sometimes called state variables, and functional definitions, often referred to as methods. An unfortunate asymmetry exists in many object-oriented languages in that primitive data types such as C `ints` do not participate unchanged in inheritance. In these cases, the primitive type is "wrapped" with a class definition.

Inheritance is a method of providing context to state variables and methods. This context is always a rooted, directed acyclic graph (DAG). If only single inheritance is allowed, then the rooted DAG becomes a rooted tree. Examples are obviously Java (single) and C++ (multiple).

## POLYMORPHISM

A consequence of the inheritance hierarchy is that the same method name can occur in many different classes. If all these definitions were in the same visible definition frame, then there would be chaos; one would never know which definition would be used.

This sort of problem actually does exist: overloaded arithmetic operators. Overloading allows multiple functions taking different types to be defined with the same name; the compiler or interpreter "automagically" calls the right one. The "magic" part means that the compiler has a rule for choosing; unfortunately, overloading falls apart quickly. *Parametric polymorphism*, on the other hand, works by matching the function based directly on the types of the inputs. How this works requires an excursion into types.

The type checker, Milestone IV, is not very interesting as type systems go. In adding arrays and structures in Milestone VIII, we still had not deviated much from the C type system. We gain a hint of what is desired by looking at C++'s templates. Actually, the concept of parametric types is quite old, going back to the 1930s. The problem we want to solve notationally is illustrated by the concept of lists. In direct manipulation of lists, the actual internal structure of the elements is irrelevant because we do not—and do not want— to manipulate the information of the elements. Therefore, the actual type of the element is irrelevant and can be anything. This suggests that the type of the elements can be a variable.

This stronger (than C) type theory goes by several names: intuitionistic type theory, constructive type theory, Martin-Löf type theory, or just Type Theory (with capital letters). Type Theory is at once (1) a functional programming language, (2) a logic, and (3) a set theory.

The notation used in the literature is based on the λ-calculus, except the arguments *precede* the type name. In the theoretical literature, type variables are written as lower-case Greek letters. Using ML, `array` is a type; since we don't have Greek letters on English keyboards, arguments are names preceded by the reverse apostrophe. Here are some ML examples:

| | |
|---|---|
| `'a array` | a one-dimensional array |
| `('a,'b) array` | an array with a pair of types `a, b` as elements |
| `(int array) array` | an array with integer arrays as elements |

In ML, arrays do not carry their dimension as part of the type, which is determined at runtime. ML does not have a separate notation for array indexing, either. Despite this, ML arrays are useful data structures just as C.

The useful nature of type variables allows us to express data structure types algebraically, leading to work in category theory and algebraic specifications.