

A Case Study in Programming Language Design

Project Part 1

(Dissertation for MSc in Software Engineering)

Dean Wakerley, BSc(Hons)(Open) MIAP MACM MIEEE

University of Bradford

Supervisor: Dr. D.R.W. Holton

Copyright © Dean Wakerley, May 2004

Contents

1.	Introduction	1
1.1	Impetus	1
1.2	Aims & Objectives	1
1.3	Scope	1
2.	Background Theory	3
2.1	What is a programming language?	3
2.1.1	Generations of Programming Language	3
2.1.2	Computational Paradigms	3
2.2	Formal Linguistics	3
2.2.1	Language	3
2.2.2	Programming Language	4
2.2.3	Translation	4
2.2.4	Morphology	4
2.2.5	Syntax	5
2.2.6	Semantics	5
2.2.7	Pragmatics	5
2.3	Programming Language Design	6
2.3.1	What makes a programming language successful?	6
2.3.1.1	Programming Language Design Principles	6
2.3.1.2	External Success Factors	8
3.	Critique of Programming Languages (Sample)	9
3.1	Flow Control	9
3.1.1	Guarded/Selective Statements	9
3.1.1.1	If Statements	9
4.	Status of Case Study	13
4.1	Work completed	13
4.2	Work-in-progress	13
4.3	Future Work	13
4.3.1	Timetable	13
4.4	Lessons learned	13
A.	Critique of Programming Languages (WIP)	15
A.1	Lexical Structure	15
A.1.1	Identifiers	15
A.1.2	Keywords	16
A.1.3	Literals	16
A.1.3.1	Numbers	17
A.1.3.1.1	Naturals	17
A.1.3.1.2	Reals	17
A.2	Modularity	17
A.2.1	Programs	17
A.2.2	Modules	18
A.2.3	Separate Compilation	18
A.2.3.1	Ada	18
A.3	Flow Control	19
A.3.1	Guarded/Selective Statements	19
A.3.1.1	If Statements	19
A.3.1.2	Multiple Selection Statements	22

A.3.2	Iterative Statements	25
A.3.2.1	Generalized Loop Statements	25
A.3.2.1.1	Pre-test Loop Statements	25
A.3.2.1.2	Post-test Loop Statements	26
A.3.2.1.3	Unconditional Loop Statements	27
A.3.2.1.4	Loop Control Statements (Mid-test).	28
A.3.2.2	Specialised Loop Statements	30
A.3.3	Goto Statements	32
A.4	Bibliographic Notes	32
	References.	35

1 Introduction

This is an intermediate report of a case study in programming language design. The ultimate goal of the case study is to create a programming language.

The aims of this report are to introduce the case study and review the theoretical background relating to programming language design. It starts with the impetus behind the case study, followed by the aims and objective.

The chapter on background theory reviews what programming languages are, their history and computational paradigms. It examines formal linguistics and the different branches of study. Programming languages are placed into this framework. Finally, programming language design is explored including the theoretical principles of language design and the success factors.

The Chapter “3 Critique of Programming Languages (Sample)” on page 9 is a fragment of the complete critique on programming language constructs. The complete work-in-progress can be found in the appendix.

The Chapter “4 Status of Case Study” on page 13 explains the work completed, the current progress and the work yet to be performed.

1.1 Impetus

During the 1980’s I was successful in writing extensions to the Minimal BASIC language, although no attempt was made to publish. Later I designed extensions to Pascal language, including object-oriented features. These are still present in a unnamed commercial product.

During my fourteens years working a software developer using Pascal and other languages I became aware of deficiencies and in some cases severe limitations with the facilities of these languages. This provided the impetus for this case study.

1.2 Aims & Objectives

As stated, the ultimate goal of the case study is to create a programming language. The first step is to review the background theory relating to programming language design (Chapter “2 Background Theory” on page 3).

After this, defining the aims of the language will establish the scope, direction and criteria for the new language. It is anticipated that the language will have be a general-purpose language covering the uses that C, Pascal, and Java are used for today.

Having established the aims of the language it is necessary to critically survey and analyse language constructs of existing languages. This survey should be conducted by focusing on each language construct, concurrently “cherry-picking” or creating new constructs for inclusion in the new language. Any problem areas identified should be noted for future study at a later date.

Finally, the syntax of the new language definition can be presented formally using EBNF (see ISO14977: 1996).

1.3 Scope

The time scale and manpower of a typical language design project should be noted. (Louden 1993: 38-39) identifies two languages taking four years and another taking two years to design, each project having a team of designer. Therefore, time is an important constraint that must be taken into consideration.

Whilst it is desirable to have a working compiler or interpreter, especially for testing the implementability of the language, compiler design & implementation is beyond the scope of this project.

Also, the time constraint must limit the breadth of the new language and the depth of background study permitted. This is unfortunate but the resulting language may be developed further after this case study is complete.

2 Background Theory

2.1 What is a programming language?

A programming language is a notational system for describing computation in machine-readable and human-readable form. (Louden 1993: 3). This is a nice and simple definition. As with most simple definitions it is the tip of the iceberg.

2.1.1 Generations of Programming Language

Programming languages started life as **assembly language** after von Neumann came up with the idea of the central processing unit architecture (*ibid.*: 2) we still have today. Assembly language had symbols and **mnemonics** that had a one-to-one relationship with the processor's instruction set (*ibid.*). This led to the development of languages with a higher-level of abstraction.

The different stages of language development can be classified into generations. Assembly languages are first generation, unstructured high-level languages are second generation, current imperative languages are third generation and fifth generation languages include Prolog and SETL (*ibid.*: 40-41). Fourth generation languages are really special purpose languages such as SQL (*ibid.*), because of this I prefer not consider these true programming languages but also because they do not exhibit the characteristics of the other generations. e.g. They are not compiled and lack general-purpose language constructs: loops, decision statement, variables.

2.1.2 Computational Paradigms

The **von Neumann** computational model is characterised by: sequential execution of instructions, the use of variables and assignment to change the variable's value (Louden 1993: 11). A programming language conforming to this model is called an **imperative** language. Most current languages are imperative.

There are at least two other paradigms: **functional** (also known as **applicative**) and **declarative** (also known as **logic**) (*ibid.*: 12, Scott 2000: 6). Functional paradigm does away with the use of variables (*ibid.*). Declarative paradigm is based on symbolic logic (*ibid.*: 13).

2.2 Formal Linguistics

The previous simple definition of what a programming language is may be adequate for a novice or a practitioner but to someone intending to study programming language design or to design a programming language it is necessary to see and understand the rest of the "iceberg".

Programming 'language design is one of the most difficult and poorly understood areas of computer science' (Louden 1993: 47). Thus, to understand programming language we must first understand what language is.

2.2.1 Language

What is language? What is a language? These seem simple questions to answer, however if we explore linguistics, we find that this is not the case. In 'The nature of language' (Graddol *et al.* 1994: Chap.1), we are given '... anyone who can speak a language knows something about what a language is. But it cannot be assumed that those who know how to use language can readily tell us what language is.' (*ibid.*: 1). This leads to the central problem of describing language.

To describe language we need to 'discover, describe and analyse the underlying principles which govern its functionality. The result of such an effort is a **theory** or **model** of language. That is, an understanding of how language "works"' (*ibid.*).

'Language is a complex system of communication ...' (*ibid.*: 13) and when studying anything that is complex it is appropriate to divide the subject into smaller parts or "divide and conquer"

as the common expression goes. There are many authors with different theories, as a result there are different divisions. The most common is to divide language into **sound**, **grammar** and **meaning**. **Pragmatics** is another division that is concerned with the study of language in its communicative context (*ibid.*: 16).

Sound is further divided into **phonetics** and **phonology**. Grammar is divided into **morphology** (word structure) and **syntax** (sentence structure). **Semantics** is the study of meaning.

Noam Chomsky developed much of the early theory of formal languages (Scott 2000: 87) and he uses the term ‘grammar’ to refer to the entire system of organization of language (Graddol *et al.*: 66).

2.2.2 Programming Language

The Chomsky formalism is responsible for the term **programming language**, because programming languages exhibit a structure similar to languages (Wirth 1990: 8).

Programming languages were not designed to be spoken. The words of a programming language are either borrowed from a language, usually English or are abstract creations, such as **fi** in Algol-68 for bracketing the **if** statement. The words borrowed from spoken languages may be spoken with the same pronunciation from their borrowed language, however the created words may be difficult or impossible to pronounce.

Because programming languages are not designed for verbal communication the study of sound is irrelevant, but morphology, syntax, semantics and pragmatics are.

‘Chomsky’s early work attempted to formalize our innate linguistic knowledge in terms of rules’ (Graddol *et al.* 1994:86), and in the 1950s Chomsky developed the idea of context-free grammars (Louden 1993: 63).

These context-free grammars are used to describe the formal morphology and syntax of programming languages. Since the definition of Algol-60 the custom has been to define the syntax of a programming language formally. Algol-60 was defined with a notation known as Backus-Naur Form (BNF) (ISO14977:1996: v). BS ISO/IEC 14977:1996 defines an international standard form of Extended BNF. Unfortunately most programming language’s grammar are defined using their own variation of BNF or EBNF.

2.2.3 Translation

Whilst computer programs are written in programming language, computers however only execute instructions from their instruction set defined by the designers of the CPU. Source code must be converted into instructions the CPU understands. This conversion is performed by a translator. Translators come in two flavours: compilers and interpreters. An interpreter immediately executes a program from the source code. A compiler creates executable code or object code from the source code. Object code is an intermediate form which is linked or bind into an executable file. Compiler may even create an intermediate form such as assembly language or other language source code that is passed to another compiler to create the executable file. Perl is a scripting language that is actually compiled on-the-fly, blurring the distinction between compilers and interpreters.

The translation process consists of lexical analysis, syntax analysis, semantic analysis and code generation.

Lexical analysis is performed by a lexical analyser or scanner. Syntax analysis is performed by the syntax analyser or parser.

2.2.4 Morphology

Morphology is the study of **intra**-word structure. This is most relevant to inflected languages, such as the Romance languages, where the ending of words convey the person (1st, 2nd or 3rd),

number (singular or plural) and tense (past, present, future et al.). This can be seen in plethora of verb table books that exist for Spanish, French et al. It is less relevant for English where inflections are not as abundant.

Morphemes are the smallest units of analysis of sentence and word structure (Graddol *et al.* 1994:70). **Lexemes** are the basic unit of a language; a word in the most abstract sense.

Consider these words: jumped, unfair, singing, loves. The morphemes are: jump, -ed, un-, fair, sing, -ing, love, -s. The lexemes are: JUMP, FAIR, SING, LOVE.

Morphology, morpheme and lexeme are not terms usually found within the study of programming languages, however, (Louden 1993: 17) identifies **lexical structure** being 'similar to spelling in a natural language' and words in programming languages are called **tokens**. A token is a sequence of characters, these can be letters, digits or any other characters from a character set, this means the token is equivalent to morphemes, lexemes and even punctuation of natural languages.

The source code of a program is analysed by the lexical analyser, which breaks it up into tokens which are passed to the syntax analyser for processing.

2.2.5 Syntax

Where morphology studied intra-word structure, syntax is the study of **inter-word** structure or 'sentence structure' (Graddol *et al.* 1994:13). (Mak 1991: 62) states syntax is 'the set of "grammar rules" that specify how statements and expressions are correctly written' in a language.

(OCD 1995: 1414) states a **syntagma** is a word or phrase forming a syntactic unit. A sentence is one or more syntagma.

Thus, the syntax of a programming language is the rules pertaining to a syntagma or a sentence (a sequence of tokens).

2.2.6 Semantics

(Graddol *et al.* 1994:16) defines semantics as the 'branch of linguistics concerned with how **meaning** is expressed in language' and states the 'analysis of meaning has proved to be one of the most difficult and elusive tasks in linguistic description'.

Programming languages tend to have a more concrete definition and less ambiguous than natural language because they are designed by a small number of people (in comparison to the people involved in the development of natural languages) and in relatively short time (as opposed to the evolution of natural languages through the centuries) and the standards organizations usually become involved with popular languages.

Ambiguities do still arise, this is why Ada compilers are required to pass rigorous tests before being allowed to be called an Ada compiler.

There has been much work within the field of programming language semantics. For example, natural, denotational and axiomatic semantics.

2.2.7 Pragmatics

The study of language in its communicative context (Graddol *et al.* 1994:16). However, (Scott 2000: xviii) suggests programming language pragmatics is "how programming languages work" or language implementation. Donald Knuth has 'suggested that programming be regarded as the art of telling another human being what one wants the computer to do' (Knuth 1984, cited by Scott 2000: 5).

Traditionally languages have only been used for communication between people. However, research into voice recognition, optical character recognition and handwriting recognition has created **person-to-machine** communication.

Programming languages have always had the two communication contexts: person-to-machine and **person-to-person**.

Person-to-person communications can be seen as using programming languages to express algorithms in the same way mathematics and formal specification languages are used.

Results of person-to-machine communication can be seen as the results of the translation process.

2.3 Programming Language Design

Programming ‘language design is one of the most difficult and poorly understood areas of computer science’ (Louden 1993: 47). This did not stop the ‘explosion in programming languages’ (*ibid.*: 47-48) in the 1960s.

Over the last five decades many programming languages were created and many have disappeared. A body of knowledge as been accrued from these experiences. This body of knowledge encompasses the theory of language design and the external issues.

2.3.1 What makes a programming language successful?

The success of a programming language depends on many factors. Before setting out on designing a programming language it would be prudent to establish why some languages are successful and others not.

Naturally there are many technical aspects to programming language design that effect the success of a language. There are also external factors that have an impact on the success of a language, such as cultural, commercial and practicalities.

2.3.1.1 Programming Language Design Principles

In the beginning computers were very slow and the only design criterion for a programming language was **efficiency of execution** (or **performance**) (Louden 1993: 49).

Early software was written in machine code, soon assembly language was introduced where the machine code is represented by mnemonics and symbols (*ibid.*: 33). Assembly language is referred to as “low-level” languages. This “level” is the level of **abstraction** of the actions of computers a language purports. Languages with higher levels of abstraction were developed, FORTRAN being the first (*ibid.*) providing abstract notation instead of the specific machine instruction codes. Abstraction provides a means of representing data such as types, variables and constants; and control with loop and decision statements etc., thereby reducing the complexity of code. Good notation allows the programmer to concentrate on more advanced problems (Whitehead, cited by Louden 1993: 24).

Abstracting away from the underlying machine creates **machine independence** (or **portability**) because a program can be compiled for different machine architecture without any modification.

The whole reason for a high-level of abstraction is to make reading and writing programs easier. **Writability** is the quality of a language that enables a programmer to write clear, correct, elegant, concise and maintainable code with speed and ease (*ibid.*: 49).

Expressiveness (or **expressive power**) of a language is closely related to writability, conciseness and the power and **generality** of the abstraction mechanisms (*ibid.*: 51, Scott 2000: 3).

Programs are read much more than they are written (Barnes 1998: 11). **Machine-readability** is the quality that allows a machine to read a language for translation. **Human-readability** is the quality that allows a programmer to read and understand code with speed, ease and accuracy (Louden 1993: 49).

Maintainability is closely related to writability, readability, complexity control, modularity and reusability. It is the quality that relates to how well software can be maintained and evolved.

Traditionally, maintenance is a phase in the software development life cycle (SDLC) where bugs are fixed and minor modifications are made. Today software evolves through many versions (e.g MS-Windows, Office). Maintenance and refactoring as become much more important. Refactoring is the process of rewriting code with the purpose of reducing entropy (complexity within a system) and so **controlling complexity**.

Modularity is the abstraction mechanisms that enables large programs to be divided into modules permitting separate compilation (Burns & Wellings 2001: 78) so large teams can write modules concurrently without interference.

Modularity permits the writing of libraries that can be reused. **Reusability** is therefore the quality that enables reuse of tried and tested libraries.

Object-orientation introduced the *module as a type* allowing components to be developed and reused. Modules and classes provide **encapsulation** and **information hiding**. (Burns & Wellings 2001: 73-74).

Relating to the human-readability of a language is the **aesthetic** quality (look and feel) and the **consistency**. This quality is particularly difficult to get right because aesthetic quality is intangible and exists only in the mind. This quality is closely related to personal preference.

Insulation is a negative quality related to abstraction and optimization. Where abstraction abstracts away from the machine, insulation is the distance between the language and the machine. A good language will have a high-abstraction and low-insulation, where each language construct will map to the smallest and fastest machine code.

Generality is the quality that avoids special-cases within the language, (Louden 1993: 52-53) provide examples of non-generality. **Orthogonality** is the quality where language constructs should not behave differently in different contexts, (*ibid.*: 53) provide examples of non-orthogonality. **Uniformity** is the quality that focuses on consistency of appearance and behaviour, (*ibid.*: 54) provide examples of non-uniformity.

Simplicity is a quality deceptively difficult to attain. ‘Everything should be made as simple possible, but not simpler’ (Einstein, *cited by ibid.*: 55). To attain simplicity in a language one must walk a fine line between simplicity and over-simplicity. Pascal and BASIC have too much simplicity and so lack some features making the programmers job more difficult.

Preciseness (or definiteness), reliability and security are closely related. **Preciseness** is the quality of how precise the language definition is. This as a knock-on effect to the reliability of programs and translators (*ibid.*: 56).

Reliability is the assurance that a program will not behave in unexpected or disastrous ways during execution. Exceptions (in C++) are an example of un-reliability since they allow programmers to ignore or be unaware of possible error conditions; the exception would propagate out of the program, imagine this occurring with flight-control software in an aeroplane.

Security is the principle that promotes a language design that minimises programming errors and maximises the translators ability to detect them. This is an example of bad security in C:

```
void f(void)
{
    char s[10];
    strcpy(s, "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
}
```

The code is legal and correct but on execution will corrupt the data held in the stack. This is a buffer overrun error and is how malicious users on the internet can gain control and access to computers. (Bates 2004) further examines buffer overruns. A good language design would prevent this type of error.

2.3.1.2 External Success Factors

‘Fortran owes much of its success to **extremely good compilers**’ that produce very fast code although this is partly due to language design (Scott 2000: 4).

BASIC has been very successful, few can deny this. The 1980’s was the era of the home computer. Almost every home computer came with BASIC in ROM making BASIC **freely available**. This was due to **ease of implementation**, the compactness of the interpreter and the minimal memory required for programs. BASIC being designed for beginners brought programming to the masses because it was **easy to learn and use** (*ibid.*). Pascal was also easy to learn and was distributed freely to universities for teaching to students.

A more worrying aspect to the success of BASIC is the **commercial exploitation** by the likes of Microsoft. I believe BASIC had a place up to the early 1990’s, from the late 1990’s the continued use of BASIC became an impediment to software engineering principles and software quality.

‘Ada owes its life to the US Department of Defense’ (Scott 2000: 4). They discovered they were using many programming languages that caused problems with maintenance, skills and hence, cost. Thus, they decided to **sponsor** the development of a new language (Ada).

During the approach of the millennium, the “millennium bug” highlighted the fact that legacy software is still in use, written in languages such as COBOL. A **large installed base** of software written in legacy languages may have no impetus or business cost justification to redevelop the software in a newer language (*ibid.*).

Some languages remain in use because of **personal preference**. ‘Forth has a small but dedicated following’ (*ibid.*).

Many languages are successful because they are designed for a **specific application domain**. C is considered a good language for development of operating systems, it has been successful in development of application software where it is less suitable. SQL is an example where specialising in communication between client software and a database server has been a success. Today the suitability of a language for a particular application domain appears to be of little importance. For example, I have attempted to write programs in C to access SQL databases, but the string facilities were and still are woefully inadequate. In the end I used Perl which has very good string facilities.

3 Critique of Programming Languages (Sample)

This chapter includes only the critique on the **if** statement. It is included in the main body of this report as a sample of the material in the appendix and indication of the depth of study.

Like the case study, the critique is a work-in-progress and so is incomplete. The current status of the critique and future direction can be found in “4 Status of Case Study” on page 13.

The full current text can be found in “Appendix A Critique of Programming Languages (WIP)” on page 15 including the **bibliographic reference information**.

3.1 Flow Control

3.1.1 Guarded/Selective Statements

Selection statements provide the facility to make decisions. The most general selection statement is the **if...then...else** statement and most languages provide some variation.

3.1.1.1 If Statements

The basic form of the **if** statement was introduced in Algol-60 and inherited in Pascal, C, C++ and Java.

Algol-60, Algol-W, Pascal

if statement = "if", condition, "then", statement, ["else", statement];

C, C++, Java

if statement = "if", "(", expression, ")", statement, ["else", statement];

This gives the Algol-60/Pascal form of the **if** statement.

```
if a>b then
  proc(a)
else
  proc(b);
```

The C variety is effectively the same but differs stylistically. Remember, in C the semicolon is the statement terminator but in the Algol-60/Pascal, it is the statement separator, hence the omission in the above code.

```
if(a>b)
  proc(a);
else
  proc(b);
```

The **else** clause is optional, and can be omitted.

```
if a>b then
  proc(a);
```

Both the **then** and **else** clauses are defined to contain only a single statement. The single statement can of course be a block.

```
if a>b then
  begin
    proc(a);
    proc(b)
  end
else
  proc(c);
```

There is a well-known ambiguity with this form of the **if** statement called the *dangling-else*.

```

if b1 then
    if b2 then
        proc1
    else
        proc2;

```

The ambiguity comes from deciding which **if** statement the **else** clause belongs. Algol-60 has a rule that disallows an **if** statement to follow the **then** keyword (**begin-end** block may be used to hide an **if** statement). Pascal, C and its derivatives resolve this ambiguity by connecting the **else** clause with the nearest **if** statement (last-in, first-out just like a stack).

Modern languages are usually designed taking this problem into account, although Bjarne Stroustrup (C++ designer) and Sun (Java designers) appear to have been on a desert island for the last four decades.

Algol-68

if statement = "if", if clause, {"elif", if clause}, ["else", statement], "fi";
 if clause = boolean expression, "then", statement sequence;

Modula-2, Oberon

if statement = "IF", if clause, {"ELSIF", if clause}, ["ELSE", statement sequence], "END";
 if clause = boolean expression, "THEN", statement sequence;

Ada 95

if statement = "if", if clause, {"elsif", if clause}, ["else", statement sequence], "end", "if";
 if clause = boolean expression, "then", statement sequence;

Algol-68 addresses the *dangling-else* problem by introducing the **fi** bracketing keyword. This indicates the end of the **if** statement.

```

if b1 then
    if b2 then
        proc1;
    fi
    else
        proc2;
    fi

if b1 then
    if b2 then
        proc1;
    else
        proc2;
    fi
fi

```

It is clear in both the above code snippets which **if** statement the **else** clause belongs. The introduction of a bracketing keyword removes the necessity for using a **begin-end** block to allow multiple statements. This is the approach adopted in the Modula family (including Oberon) and Ada. Modula with the **end** keyword and Ada with the **end if** keywords.

Algol-68 also introduced the **elif** clause. The Modula family and Ada use the **elsif** keyword. This allows a list of conditions to be checked and the first to be true is executed, the rest ignored.

```

if b1 then
    procla;
    proclb;

```

```

elsif b2 then
  proc2a;
  proc2b;
elsif b3 then
  proc3a;
  proc3b;
else
  proc4a;
  proc4b;
end if;

```

Modula in using the **end** keyword cause some ambiguity because other statement also use the **end** keyword for bracketing.

Perl

if statement =	if multiple if singleton;
if multiple =	"if" "unless", if guard, {"elsif", if guard}, ["else", if block];
if singleton =	statement, "if" "unless", expression;
if guard =	("", expression, ""), if block;
if block =	{"", statement sequence, "}"

Perl has greater expressive power by providing the most variety of **if** statements. The standard form unlike C requires braces avoiding the *dangling-else* problem and bracketing multiple lines.

```

if(b1)
{
  proc1;
}
elsif(b2)
{
  proc2a;
  proc2b;
}
elsif(b3)
{
  proc3;
}
else
{
  proc4;
}

```

Perl adopts a single statement form of the **if** statement. I am familiar with the same form available in a variety of BASIC on a PDP-10/11 at Wakefield District College, 1986/7.

```

proc if b1;

```

Here we have a tail condition. The statement **proc** is executed only if the expression **b1** is true. This form is claimed to provide greater **readability**. Note also the absence of the parentheses around the expression.

Perl is unique in that it also provides a “if not” statement by introducing the **unless** keyword. This is also on the grounds of readability.

```

proc unless b1;

```

Is semantically equivalent to

```

proc if !b1;

```

The exclamation mark is the logical negation (not) operator.

Note that the multiple statement form uses the **elsif** and **else** keywords: there is no **unless** version.

```
unless (b1)
{
    proc1;
}
elsif (b2)
{
    proc2a;
    proc2b;
}
elsif (b3)
{
    proc3;
}
else
{
    proc4;
}
```

The **elsif** expression is not negated.

4 Status of Case Study

4.1 Work completed

The background theory is presented in this report, see Chapter “2 Background Theory” on page 3.

4.2 Work-in-progress

The survey and critique of programming languages is incomplete and it is envisaged will take a different form in the final dissertation (see “4.4 Lessons learned” on page 13 below).

I have explored various language constructs and syntax for later inclusion.

4.3 Future Work

The aims of the language needs to be established, relating to the theoretical background. This is the next task.

The current languages need to be analysed (the critique) and the new language designed. These two tasks are entwined and so will be performed concurrently. The result will be a single narrative explaining the new language constructs in the context of equivalents in other languages and the relevant theory/research.

The language will be finalised by collecting the complete syntax in EBNF.

4.3.1 Timetable

The deadline for submission of the final dissertation is the beginning of September. I plan to complete the design of the language (writing up will be part of the design process) by the end of July. This gives August for proofreading and finalisation.

4.4 Lessons learned

I intended to survey and critique programming languages by focusing on all language constructs to the depth seen in “A.3.1 Guarded/Selective Statements” on page 19 and “A.3.2 Iterative Statements” on page 25. This material would serve as background material for the new design. However, the time required is simply not available and the amount of material resulting could be made into a book of 1000-2000 pages. This is inappropriate for this case study and after consideration a new incremental method shall be used. Each language construct shall be focused on whilst designing the new language. The survey and design shall thus be merged into a narrative.

Appendix A Critique of Programming Languages (WIP)

(Work-in-progress). The chapter is intended to be an in-depth critique of programming languages. Critically analysing existing programming languages by focusing on language constructs.

A.1 Lexical Structure

Most programming languages conform to a **context-free grammar** and **free-format**. A context-free grammar is where the semantics of the program do not alter the valid syntax, these can usually be formally defined using BNF. A free-format language is where the format of the source code is insignificant. A **Fixed-format** language unlike a free-format language, the format of the source code text affects the way tokens are recognized.

Python and occam2 are fixed-format languages, but the indentation corresponds to blocks in other languages.

```
IF
  B1
    A1
  B2
    A2
  B3
    A3
```

This **if** statement of occam2 demonstrates that the indentation is syntactically significant. The boolean conditions are indented and the statements that are executed on that condition are indented further.

A lexical analyser or scanner reads the source code and collects sequences of characters into tokens, these are passed to the parser for processing.

Tokens can be divided into several categories: **identifiers**, **keywords** (or reserved words), **literals**, **symbols**, **white space** and **comments**.

White space and comments can be considered **pseudo-tokens** in that they separate adjacent tokens. In languages where indentation is significant they may be considered a true token. In C comments are actually considered white space.

A.1.1 Identifiers

Identifiers are names that are assigned to variables, constants, types and procedures etcetera. Identifiers usually consist of letters and digits, some languages allow the underscore. Digits are not permitted as the first character.

Some languages regard case to be distinct (C, C++, Java, Modula-2) in identifiers and keywords whilst others do not (Pascal, Ada).

```
C
letter = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|
         "N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"|
```

```

"a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|
"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z";
digit =
underscore =
identifier = letter | underscore, {letter | digit | underscore};

```

Pascal

```

(* In Pascal case is not distinct *)
letter =
"A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|
"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z";
digit =
"0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
identifier = letter, {letter | digit};

```

Dialects of Fortran prior to Fortran-90 use a fixed-format, where different columns within a line reserved for different purpose.

There are two issues that relate to identifiers: the allowed length of identifiers and the number of significant characters.

These may seem like trivial issues but these are the issues that can have disastrous consequences.

Many languages place no restriction on the length of identifiers but consider only a fixed number of characters to be significant. Consider two compilers (maybe they are on different platforms), the first is significant to 32 characters and the second only has 8 significant characters. Software is developed to control the launch of a spacecraft into space with the first compiler. It contains among others, two variables `VerticalThrust` and `VerticalInertia`. `VerticalThrust` is the downward force the engines are currently applying and `VerticalInertia` is the inertia of the spacecraft that is opposing the engines. This software works correct but when compiled with the second compiler result in a catastrophic problem. Because the second compiler only has 8 significant characters it assumes these two variables are the same, resulting in a semantic error in the executable code.

This is demonstrative of the problem with significant characters. Had the second compiler issued an “identifier too long” error, it would have been possible to change the program.

In Ada 95 all characters are significant and there is no limit on the number of characters. The line length may have a limit, but must be at least 200 characters. This means the minimum length of an identifier must be 200 also.

A.1.2 Keywords

Keywords or reserved words are predefined identifiers that a language defines as part of its syntax. These words specify statements or declarations. Examples include **if**, **then**, **procedure** etc. Identifiers are not allowed to be defined using the same characters as a keyword.

Keywords usually consist of letters only. Some languages require keywords to be lower case (C) while others purchase (Modula-2) and others do not care (Pascal).

A.1.3 Literals

Literals are explicit constants. Literals can be divide into several categories: **numbers**, **characters** and **strings**. Some languages (Pascal, Modula-2, Oberon) do not distinguish between characters and strings.

Pascal

```

unsigned constant = unsigned number | character string | constant identifier | "nil";

```

A.1.3.1 Numbers

Numbers can be divide into **naturals** and **reals**.

Pascal

unsigned number = unsigned integer | unsigned real;

A.1.3.1.1 Naturals

C

digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
 identifier = letter | underscore, {letter | digit | underscore};

Pascal

digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
 digit sequence = digit, {digit};
 unsigned integer = digit sequence;

A.1.3.1.2 Reals

Pascal

digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9";
 sign = "+" | "-";
 digit sequence = digit, {digit};
 fractional part = digit sequence;
 scale factor = [sign], digit sequence;
 unsigned real = digit sequence, [".", fractional part], ["e", scale factor];

A.2 Modularity

Originally computers had little memory this meant that programs had to be small enough to fit into this memory. Software would be broken down into programs that could be run separately.

Early compilers consisted of separate programs that performed only a single stage of the compilation process. The output of one program being the input into later stages. (Wirth 1996: 2) identifies 4 to 6 stages were typical but reached 70 stages with PL/1.

Today memory is in abundance and much larger software systems are in existence. This gradual increase in available memory resulted in programming languages introducing language features that allowed modularity of software.

A.2.1 Programs

UCSD Pascal, Borland's Object Pascal

program = "program", identifier, ["(", parameters, ")"], ";", block, ".";

Modula-2

program = "MODULE", identifier, [interrupt protection], ";" program body, "END",
 identifier, ".";
 program body = import list, declarations, "BEGIN", body, ["FINALLY", body];

A.2.2 Modules

A.2.3 Separate Compilation

A.2.3.1 Ada

The main program consists of a separate file with a procedure or function.

```
procedure scanner is
begin
  ...
end parser;
```

A package consists of a specification and an optional body. The specification and body must be in separate files.

The specification declares the interface of the package, that is the subprograms, variables and types etc. that are exported.

```
package parser is
  ... exported items
end parser;
```

The body contains the implementation of the package. If the package specification does not declare any subprograms there may not be a body for the package.

```
package body parser is
  ... non exported items
end parser;
```

The items within the body are hidden from outside the package unless they are declared in the specification. Any subprogram declared in the specification must be implemented in the body.

Subprograms and package bodies may have subprograms in separate files.

```
package body stack is
  procedure push(x: integer) is separate;
  function pop return integer is separate;
begin
  ...
end procl;
```

The above package expects the two subprograms to be in separate files.

```
procedure procl is
  procedure sepl is separate;
begin
  ...
  sepl;
  ...
end procl;
```

This subprogram contains a nested subprogram that is in a separate file.

```
separate (procl)
procedure sepl is
  ...
end sepl;
```

This shows a subprogram implemented in a separate file.

A.3 Flow Control

A.3.1 Guarded/Selective Statements

Selection statements provide the facility to make decisions. The most general selection statement is the **if...then...else** statement and most languages provide some variation.

A.3.1.1 If Statements

The basic form of the **if** statement was introduced in Algol-60 and inherited in Pascal, C, C++ and Java.

Algol-60, Algol-W, Pascal

if statement = "if", condition, "then", statement, ["else", statement];

C, C++, Java

if statement = "if", "(", expression, ")", statement, ["else", statement];

This gives the Algol-60/Pascal form of the **if** statement.

```
if a>b then
  proc(a)
else
  proc(b);
```

The C variety is effectively the same but differs stylistically. Remember, in C the semicolon is the statement terminator but in the Algol-60/Pascal, it is the statement separator, hence the omission in the above code.

```
if(a>b)
  proc(a);
else
  proc(b);
```

The **else** clause is optional, and can be omitted.

```
if a>b then
  proc(a);
```

Both the **then** and **else** clauses are defined to contain only a single statement. The single statement can of course be a block.

```
if a>b then
  begin
    proc(a);
    proc(b)
  end
else
  proc(c);
```

There is a well-known ambiguity with this form of the **if** statement called the *dangling-else*.

```
if b1 then
  if b2 then
    proc1
  else
    proc2;
```

The ambiguity comes from deciding which **if** statement the **else** clause belongs. Algol-60 has a rule that disallows an **if** statement to follow the **then** keyword (**begin-end** block may be

used to hide an **if** statement). Pascal, C and its derivatives prevent this ambiguity by connecting the **else** clause with the nearest **if** statement (last-in, first-out just like a stack).

Modern languages are usually designed taking this problem into account, although Bjarne Stroustrup and Sun appear to have been on a desert island for the last four decades.

Algol-68

if statement = "if", if clause, {"elif", if clause}, ["else", statement], "fi";
 if clause = boolean expression, "then", statement sequence;

Modula-2, Oberon

if statement = "IF", if clause, {"ELSIF", if clause}, ["ELSE", statement sequence], "END";
 if clause = boolean expression, "THEN", statement sequence;

Ada 95

if statement = "if", if clause, {"elsif", if clause}, ["else", statement sequence], "end", "if";
 if clause = boolean expression, "then", statement sequence;

Algol-68 addresses the *dangling-else* problem by introducing the **fi** bracketing keyword. This indicates the end of the **if** statement.

```

if b1 then
    if b2 then
        proc1;
    fi
    else
        proc2;
    fi

if b1 then
    if b2 then
        proc1;
    else
        proc2;
    fi
fi

```

It is clear in both the above code snippets which **if** statement the **else** clause belongs. The introduction of a bracketing keyword removes the necessity for using a **begin-end** block to allow multiple statements. This is the approach adopted in the Modula family (including Oberon) and Ada. Modula with the **end** keyword and Ada with the **end if** keywords.

Algol-68 also introduced the **elif** clause. The Modula family and Ada use the **elsif** keyword. This allows a list of conditions to be checked and the first to be true is executed, the rest ignored.

```

if b1 then
    proc1a;
    proc1b;
elsif b2 then
    proc2a;
    proc2b;
elsif b3 then
    proc3a;
    proc3b;
else
    proc4a;
    proc4b;
end if;

```

Modula in using the **end** keyword cause some ambiguity because other statement also use the **end** keyword for bracketing.

Perl

if statement =	if multiple if singleton;
if multiple =	"if" "unless", if guard, {"elsif", if guard}, ["else", if block];
if singleton =	statement, "if" "unless", expression;
if guard =	("", expression, ""), if block;
if block =	"{" , statement sequence, "}";

Perl has greater expressive power by providing the most variety of selection statements. The standard form unlike C requires braces avoiding the *dangling-else* problem and bracketing multiple lines.

```

if (b1)
{
    proc1;
}
elsif (b2)
{
    proc2a;
    proc2b;
}
elsif (b3)
{
    proc3;
}
else
{
    proc4;
}

```

Perl adopts a single statement form of the **if** statement. I am familiar with the same form available in a variety of BASIC on a PDP-10/11 at Wakefield District College, 1986/7.

```
proc if b1;
```

Here we have a tail condition. The statement **proc** is executed only if the expression **b1** is true. This form is claimed to provide greater readability. Note also the absence of the parentheses around the expression.

Perl is unique in that it also provides a “if not” statement by introducing the **unless** keyword. This is also on the grounds of readability.

```
proc unless b1;
```

Is semantically equivalent to

```
proc if !b1;
```

The exclamation mark is the logical negation (not) operator.

Note that the multiple statement form uses the **elsif** and **else** keywords: there is no **unless** version.

```

unless (b1)
{
    proc1;
}
elsif (b2)
{

```

```

    proc2a;
    proc2b;
}
elsif (b3)
{
    proc3;
}
else
{
    proc4;
}

```

The **elsif** expression is not negated.

A.3.1.2 Multiple Selection Statements

The **if** statement provides selective execution based on a boolean expression limiting the selection to two options. Increasing to more than two states and we have the hideous **switch** statement of Algol-60 (not shown here).

BASIC

on statement = "on", integer expression, "go", "to", line number, {"", line number};

Algol-W

case statement = "case", integer expression, "of", "begin", statement, {"", statement}, "end";

BASIC and Algol-W provides statements that are equivalent to Algol-60's **switch** statement but they look completely different. BASIC takes the form of the **on...goto** statement.

```
on a go to 1000, 2000, 3000
```

The expression **a** is evaluated to an ordinal number. This number indexes into the list of line numbers. Executions jumps to the corresponding line number.

Algol-W takes the form of the **case** statement (not to be confused with the Pascal variety) attributed to C.A.R. Hoare.

```

case a of
begin
    c:=a-w;
    proc1;
    begin
        proc2a;
        proc2b;
    end;
    proc3;
end;

```

Like BASIC's **on...goto** statement, the expression **a** is evaluated to an ordinal number. This number indexes into the statements between the **begin-end** block and the corresponding statement only is executed. In the above code fragment the assignment will be executed if **a** is 1, **proc1** if **a** is 2, the **begin-end** block if **a** is 3 and **proc3** if **a** is 4.

Pascal improves over Algol-W's **case** statement.

Pascal

case statement = "case", expression, "of", case item clause, {"", case item clause}, "end";
 case item clause = constant, {"", constant}, ":", statement;

Pascal's **case** statement improves over Algol-W by explicitly specifying the constants to match and by allowing the expression to be any discrete type.

```

case a of
1:   c:=a-w;
2,3: proc1;
7,9: begin
      proc2a;
      proc2b;
    end;
4:   proc3
end;

```

The expression **a** is evaluated and the statement with corresponding guard constant is executed. The constants must be distinct. If the expression is evaluated to a value that is not present, it is an error. Many implementations of Pascal adopt the Extended Pascal or Modula-2 form.

Extended Pascal

case statement =	"case", expression, "of", case list clause, [case otherwise clause], "end";
case list clause =	case item clause, {";", case item clause};
case item clause =	case guard, {"", case guard}, ":", statement;
case guard =	constant expression, ["..", constant expression];
case otherwise clause =	"otherwise", statement sequence ;

Extended Pascal allows ranges to be specified and a clause to catch unlisted values. If the **otherwise** clause is present and the expression evaluates to an unlisted value its statements are executed. Without the **otherwise** clause being present an unlisted value will cause a run-time error.

```

case a of
1:   c:=a-w;
2,3: proc1;
7..19:
    begin
      proc2a;
      proc2b;
    end;
4:   proc3
otherwise
  procx;
  procy
end;

```

It is obvious that the **otherwise** clause has been added without wanting to alter the rest of the **case** statement given that it is the only guard that allows multiple statements.

Modula-2, Oberon

case statement =	"CASE", ordinal expression, "OF", case list clause, [case else clause], "END";
case list clause =	case item clause, {" " "!", case item clause};
case item clause =	case guard, {"", case guard}, ":", statement sequence;
case guard =	constant expression, ["..", constant expression];
case else clause =	"ELSE", statement sequence;

Modula-2 allows ranges to be specified. The **ELSE** clause allows unlisted values to be caught otherwise a run-time error occurs. To allow multiple statements to be present for all guarded statements a separator is introduced to separate each guard clause.

```

CASE a OF
1:
    c:=a-w;
    proc1
|2,3:
    proc1;
|7..19:
    proc2a;
    proc2b
|4:
    proc3
ELSE
    procx;
    procy
END;

```

Ada 95

case statement =	"case", expression, "is", case list clause, [case others clause], "end", "case";
case list clause =	case item clause, {" ", case item clause};
case item clause =	"when", case guard, {" ", case guard}, "= >", case statements;
case others clause =	"when", "others", "= >", case statements;
case guard =	expression discrete range;
case statements =	"null" statement sequence ;

Ada takes the **case** statement further. All possible values of the expression must be present or the **others** clause must be present otherwise a compilation error will occur if any values are omitted. The **others** clause must be the last clause within the **case** statement. The **null** statement is provided for the situation where no action is to be performed for any values.

```

case day is
    when tue..thu => pu:=2;
    when mon|fri => pu:=1;
    when others => null;
end case;

```

The guard expressions permit a variety of range specifications that the language contains.

C, C++, Java

switch statement =	"switch", "(", integer expression, ")", "{", {switch item clause}, "}";
switch item clause =	("case", integer constant) "default", ":", statement sequence;

C and its derivatives provide a variation of the **case** statement. This is the **switch** statement.

```

switch (a)
{
    case 1:
        func1 ();
        func1 ();
    case 2:
    case 3:
    case 4:
        func1 ();
        func1 ();
        break;
    case 7:
        func1 ();
        func1 ();
}

```

```

    break;
default:
    func1();
    func1();
}

```

The **break** keyword is required to exit the **switch** statement, if not present as in **case 1** in the above code fragment execution drops through to the following **case** clause. Only one **default** may be present but may be placed anywhere within the braces. If the **default** clause is not present and the expression evaluates to an unlisted value execution continues after the **switch** statement.

Perl

There is no formal case or switch statement.

Perl does not have a case or switch statement, but it is possible to create a similar effect from the standard language features.

```

SWITCH:
{
    if (/^abc/) { $abc = 1; last SWITCH; }
    if (/^def/) { $def = 1; last SWITCH; }
    if (/^xyz/) { $xyz = 1; last SWITCH; }
    $nothing = 1;
}

```

The above and below code fragments taken from Wall *et al* (2000:124-5) are equivalent. The expressions between the slashes are regular expressions that are tested against the value stored in the default variable `$_`. The **SWITCH:** word is actually a label given to the braced compound statement. The **last** keyword is used to exit from that block.

```

SWITCH:
{
    /^abc/ && do { $abc = 1; last SWITCH; }
    /^def/ && do { $def = 1; last SWITCH; }
    /^xyz/ && do { $xyz = 1; last SWITCH; }
    $nothing = 1;
}

```

A.3.2 Iterative Statements

Iteration is a mechanisms that allows statements to be performed repeatedly. Iterative statements come in generalized or specialized varieties.

A.3.2.1 Generalized Loop Statements

The generalized variety may have zero or more control expressions that are tested between iterations to decide whether to terminate or repeat. The generalized loops can be classified as pre-test, post-test and mid-test based on when the control expression is tested.

A.3.2.1.1 Pre-test Loop Statements

The pre-test generalized loop is commonly known as the **while** loop.

Algol-W, Pascal, Extended Pascal

while statement = "while", boolean expression, "do", statement;

C, C++, Java

while statement = "while", "(", expression, ")", statement;

Algol-68

while statement = "while", boolean expression, "do", statement sequence, "od";

Modula-2, Oberon

while statement = "WHILE", boolean expression, "DO", statement sequence, "END";

Ada 95

while statement = while loop | named while loop;

named while loop = identifier, ":", while loop, identifier;

while loop = "while", boolean expression, "loop", statement sequence, "end", "loop";

Perl

while statement = [identifier, ":", "while" | "until", "(", expression, ")", loop body;

loop body = body, ["continue", body];

body = "{", statement sequence, "}";

With the pre-test loop the control expression is tested before the first iteration. This means the body of the loop may not be executed.

The basic **while** loop is that present in the Pascal and C families.

```
while not eof(fin) do
  read_record;
```

This form permits only a single statement. A **begin..end** block must be used to have a sequence of statements. Algol-68 introduces the bracketing keyword **od** to include a sequence of statements. Modula-2 uses the bracketing keyword **END**.

```
WHILE NOT eof(fin) DO
  read_record;
  display_record
END;
```

Ada uses the bracketing keywords **end loop** and replaces the **do** keyword with **loop** to match the unconditional loop statement.

Perl has two varieties of the pre-test loop. The **while** loop and the **until** loop. The **until** loop is equivalent to the **while** statement but the control expression is negated.

```
until (b1)
{
  s1;
}
```

Although this is syntactic sugar, this form is used for better readability.

A.3.2.1.2 Post-test Loop Statements

The post-test loop as introduced in Pascal, is commonly known as the **repeat..until** loop.

Pascal, Extended Pascal

repeat statement = "repeat", statement sequence, "until", boolean expression;

Modula-2, Oberon

repeat statement = "REPEAT", statement sequence, "UNTIL", boolean expression;

C, C++, Java

do while statement = "do", statement, "while", "(", expression, ")";

This statement executes the body of the loop before testing the control expression. This ensures the body of the loop is executed at least once.

```
repeat
  readline;
  writeline
until b1;
```

As with the Perl pre-test **until** statement the control expression is negated with respect to the **while** statement.

Louden (1993) identifies the **repeat..until** statement has syntactic sugar because it can be expressed using the **while** statement. The above code fragment could be expressed like this:

```
readline;
writeline
while not b1 do
  begin
    readline;
    writeline
  end;
```

However the loop body is duplicated before the loop. If the loop body is modified during maintenance the modification must also be made to the body preceding the loop.

If we were only allowed one loop statement and had to choose between the **while** statement and the **repeat..until** statement, the latter appears to be the better choice. Consider the above code fragments expressing the **repeat..until** statement using the **while** statement. Now compare the earlier Pascal **while** statement code fragment expressed using **repeat..until**.

```
if not eof(fin) then
  repeat
    read_record;
  until not eof(fin);
```

The **if** statement is used to add a control expression prior to executing the loop body thus simulating a **while** statement. This version is more compact however the control expression is duplicated although the duplication is reduced.

Ada and Perl do not have a post-test loop statement, however the same effect can be created with the standard language features.

A.3.2.1.3 Unconditional Loop Statements

A more general form of loop statements is the unconditional loop.

Modula-2, Oberon

loop statement = "LOOP", statement sequence, "END";

Ada 95

loop statement = loop | named loop;

named loop = identifier, ":", loop, identifier;

loop = "loop", statement sequence, "end", "loop";

Modula-2 has the **LOOP** statement. This loop will never terminate.

```

LOOP
  s1;
  s2
END;
```

Not all languages contain unconditional **LOOP** statements but it can be expressed with the **while** statement by specifying **true** for the control expression. In Pascal for example we have:

```

while true do
  s1;
```

Most programs however are expected to terminate gracefully, this means some mechanism must be made available for exiting the loop.

A.3.2.1.4 Loop Control Statements (Mid-test)

Loop control combined with the unconditional loop gives us the mid-test loop.

C, C++, Java

break statement = "break";

Modula-2, Oberon

loop exit = "EXIT";

Ada 95

loop exit statement = "exit", [identifier], ["when", boolean expression];

Perl

last statement = "last", [identifier];

Modula-2 introduced the **EXIT** statement with the **LOOP** statement. This allowed the unconditional loop to terminate.

```

LOOP
  s1;
  IF b1 THEN EXIT END;
  s2
END;
```

Ada has the same **exit** statement as Modula-2 but recognizing that it will usually have a condition like the **IF** statement in the above code fragment, to guard it's execution. Therefore Ada's **exit** statement has a conditional clause.

```

loop
  s1;
  exit when b1;
  s2;
  if b2 then exit end if;
  s3;
  exit when b3;
```

```
s4;
end loop;
```

Any number of **exit** statements may be present.

Loop control statements are not restricted for use with the unconditional loop. C which does not have the unconditional loop statement but has the **break** statement.

```
while (b1)
{
    s1;
    if (b2)
        break;
    s2;
}
```

Sometimes it is necessary to exit nested loops or other nested statements. The **break** statement is also used to exit other statements, this presents problems because there is no way of specifying which statement to exit.

To overcome this problem Ada introduces named loops. This feature is very useful and very annoying when not present in a language.

```
outer:
while b1 loop
    while b2 loop
        ...
        exit outer when b3;
        ...
    end loop;
end loop outer;
```

Here the outer loop is named *outer*. The **exit** statement exits the *outer* loop. When a loop is named the identifier must also be present at the close of the statement. This permits the compiler to match the open and close of the statement. Perl also has named loops and named statements.

The **exit** statement is not the only loop control statement.

C, C++, Java

continue statement = "continue";

Perl

next statement = "next", [identifier];

redo statement = "redo", [identifier];

C has the **continue** statement. This statement skips to the end of the loop body where the loop continues with it's test of the control expression and iteration.

```
while (b1)
{
    s1;
    if (b2)
        continue;
    s2;
}
```

Perl has the **next** statement which is equivalent to C's **continue** statement. Perl also add the **redo** statement which jump to the start of the loop body, skipping the loop control test.

A.3.2.2 Specialised Loop Statements

The generalized loop statement provide all the facilities required for iteration. Specialized loop statements, commonly known as the **for** statement provide more specific looping.

Algol-W

for statement = "for", identifier, ":", integer expression, "step", integer expression, "until", integer expression, "do", statement;

Pascal

for statement = "for", identifier, ":", expression, direction, expression, "do", statement;
direction = "to" | "downto";

Extended Pascal

for statement = "for", identifier, iteration clause, "do", statement;
iteration clause = (":", expression, "to" | "downto", expression) | ("in", expression);

The **for** statement in Algol-60 included two varieties, a **while** and a **step-until** clause. Algol-W cleans up the **for** statement.

```
for i := 1 step 3 until 30 do
  writeln(i);
```

Pascal simplifies the **for** statement further.

```
for i := 1 to 10 do
  writeln(i);
```

The control variable *i* starts with the value 1, and with each iteration is incremented until it equals 10. The increment is limited to 1 or -1 by using the **downto** keyword in place of the **to** keyword. The control variable must be an ordinal type and the standard prefers it not to be modified within the loop body.

Extended Pascal introduces iteration over a set.

```
for i in e do
  writeln(i);
```

Modula-2, Oberon

for statement = "FOR", identifier, ":", range clause, [step clause], body clause;
range clause = ordinal expression, "TO", ordinal expression;
step clause = "BY", constant expression;
body clause = "DO", statement sequence, "END";

Whilst Pascal removed the step clause from the **for** statement Modula-2 re-introduced it in the form of the **BY** clause, thus the increments of 1 or -1 restriction is removed along with the direction keyword **downto**. If the **BY** clause is not the step will be assumed to be 1. Like the **WHILE** statement, the **FOR** statement is bracketed for statement sequences.

```
FOR i := 1 TO 30 BY 3 DO
  proc(i)
END;
```

As with Pascal, the control variable may not be modified within the loop body and it must be declared locally, preventing it from being a parameter, for example. The loop is not executed if the initial value is already beyond the final value in the direction of the step. The condition is actually a greater-than (or less-than if step is negative) check with the final value, so if it isn't a direct multiple of step plus the initial value the loop will still terminate.

Ada 95

for statement =	for loop named for loop;
named for loop =	identifier, ":", for loop, identifier;
for loop =	"for", identifier, "in", ["reverse"], discrete subtype definition, body;
body =	"loop", statement sequence, "end", "loop";

Ada 95's **for** statement is more restrictive and appears to be based on Pascal rather than on Algol-W or Modula-2.

```
for i in reverse 1..100 loop
    proc(i);
end loop;
```

The control variable is locally declared by the **for** statement and it is declared like a copy-in parameter: it is read-only. Therefore any statement attempting to modify it will cause a compilation error. The control variable will in turn be assigned in order a value from the range, the step being 1. The direction may be reversed by using the **reverse** keyword.

C, C++, Java

for statement =	"for", "(", initialization, ";", condition, ";", increment, ")", statement;
initialization =	expression;
condition =	boolean expression;
increment =	expression;

The C family's **for** statement is really a generalized loop statement.

```
for(int i=0, int j=0; i<50; i++, j*=3)
    printf("%i, %i\n", i, j);
```

The initialization clause beside assigning values to variables, may also declare them, their scope being restricted to the **for** statement. The condition is evaluated before each iteration and the increment clause is evaluated between each iteration.

Although C does not have an unconditional loop statement, the **for** statement is the preferred choice to implement one.

```
for(;;)
{
    printf("%i, %i\n", i, j);
    if(done)
        break;
}
```

Whereas the **while** statement requires a true condition to be present.

```
while(1)
{
    printf("%i, %i\n", i, j);
    if(done)
        break;
}
```

Perl

for statement =	identifier, ":", "for", "(", expression, ";", expression, ";", expression, ")", block;
foreach statement =	identifier, ":", "foreach", [identifier], "(", list, ")", block, ["continue", block];
block =	"{", statement sequence, "}";

Perl as usual adds it's own extras, including the **foreach** keyword, although a synonym for **for**. It can iterate through a list of items.

```
foreach &day (@days)
{
    print "$day\n";
}
```

Lists the days from the array @days.

It is possible to use other loop statement in place of the **for** statement, however it has the advantage of better readability when used appropriately.

A.3.3 Goto Statements

The **go to** or **goto** statement is a legacy from assembly language. In assembly language there are various form of jump instructions to control the execution flow and when high level languages were introduced it was a natural development to create a **goto** statement.

The result of this legacy was to create a controversy, that rage during the 1960s, with Dijkstra's (1968) "Goto Statements Considered Harmful" inflaming the argument.

The argument was resurrected in the 1980's by Rubin's (1987) article "'Goto Statement Considered Harmful' Considered Harmful". At the time I agreed with his argument. It's important to understand at the time, the home micro was at it's peak of popularity. Minimal BASIC was installed in read-only-memory as standard on all but a few makes. This was an unstructured form of BASIC and it was impossible to write programs without the **goto** statement. Other languages were little heard of and rarely available due to the size of compilers and the use of cassette tapes for storage.

Most languages place restrictions on the use of the **goto** statement or have eliminated it completely.

BASIC

goto statement = "go", "to", line number;

The BASIC **goto** statement jumps to line numbers.

```
goto 1000
```

Pascal

target = label, ":";
goto statement = "goto", label;

Ada 95

target = "<<", label, ">>";
goto statement = "goto", label;

Pascal places restrictions on the **goto** statement. It can jump out of a nested block but not into a block. It cannot jump out of a procedure and into another. This facilitates the exiting of nested loops for example. Ada 95 follows similar rules.

Loop control statement may be used to exit multiple nested block and so make the **goto** statement redundant.

A.4 Bibliographic Notes

Information on the definition of Pascal (ISO7185:1990), Extended Pascal (ISO10206:1991), Modula-2 (ISO10514-1:1996) (Holmes 1994), Occam (Burns & Wellings 2001), Ada 95

(Barnes 1998), Algol-60 (Naur 1963), Algol-W (Wirth & Hoare 1966), C (Schildt 2000), C++ (Stroustrup 2000), Java (Schildt 2001), Oberon-2 (Mössenböck & Wirth 1995), Perl (Wall *et al* 2000).

References

- ISO10206:1991, *Programming languages — Extended Pascal*, BS ISO/IEC 10206:1991, British Standards Institute (BSI)
- ISO7185:1990, *Programming languages — Pascal*, BS EN 27185:1992 ISO/IEC 7185:1990, British Standards Institute (BSI)
- ISO10514-1:1996, *Programming languages — Part 1: Modula-2, Base Language*, BS ISO/IEC 10514-1:1996, British Standards Institute (BSI)
- ISO14977:1996, *Syntactic Metalanguage — Extended BNF*, BS ISO/IEC 14977:1996, British Standards Institute (BSI)
- Barnes, J. (1998) *Programming in Ada 95*, 2nd Edition, Pearson Education Ltd.
- Bates, R., (2004) ‘Buffer Overrun Madness’, *ACM Queue*, 2(3) pp.12,14, May 2004
- Burns, A., Wellings, A. (2001) *Real-Time Systems and Programming Languages*, 3rd Edition, Pearson Education Ltd.
- Dijkstra, E.W., (1968) ‘Goto Statement Considered Harmful’, *Communications of the ACM*, 11(3) pp.147-148, March 1968
- Graddol, D., Cheshire, J., Swann, J. (1994) *Describing Language*, 2nd Edition, Open University Press
- Holmes, B.J. (1994) *Modula-2 Programming*, 2nd Edition, DP Publications Ltd.
- Louden, K., C. (1993) *Programming Languages: principles and practice*, PWS Publishing Company
- Mak, R. (1991) *Writing compilers and interpreters: an applied approach*, John Wiley & Sons
- Mössenböck, H., Wirth, N. (1995) *The Programming Language Oberon-2*, Institut für Computersysteme, ETH Zürich
- Naur, P. (ed.), (1963) ‘Revised Report on the algorithmic language Algol 60’, *Communications of the ACM*, 6(1) pp.1-17, January 1963
- OCD (1995), Thompson, D. (ed.) *The Concise Oxford Dictionary of Current English*, 9th Edition, Oxford University Press
- OED (1989), Simpson, J.A. & Weiner, E.S.C. *The Oxford English Dictionary*, Oxford University Press
- Rubin, F., (1987) “‘Goto Statement Considered Harmful’ Considered Harmful”, *Communications of the ACM*, 30(3) pp.195-196, March 1987
- Schildt, H., (2000) *C: The Complete Reference*, 4th edition, The McGraw-Hill Companies
- Schildt, H., (2001) *Java 2: The Complete Reference*, 4th edition, The McGraw-Hill Companies
- Scott, M., L., (2000) *Programming Languages Pragmatics*, Morgan Kaufmann Publishers
- Stroustrup, B. (2000) *The C++ Programming Language*, Special edition, Addison Wesley Longman
- Wall, L., Christiansen, T., Orwant, J. (2000) *Programming Perl*, 3rd edition, O’Reilly & Associates
- Wirth, N., Hoare, C.A.R. (1966) ‘A Contribution to the Development of ALGOL’, *Communications of the ACM*, 9(6) pp.413-432, June 1966
- Wirth, N. (1990) *The Programming Language Oberon*, Revision 1. 10. 90, Institut für Computersysteme, ETH Zürich
- Wirth, N. (1996) *Compiler Construction*, Addison Wesley Longman

