UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT

Design of the Programming Language ForceOne

Andrew K. Wright

Research Report
CS-87-10

February 1987

Design of the Programming Language ForceOne

by

Andrew K. Wright

Department of Computer Science
University of Waterloo

February 1987

© Andrew K. Wright 1986, 1987

Abstract

The prototype programming language **ForceOne** is a general-purpose highly-expressive compiled procedural language. It provides a few orthogonal primitives, and its expressive power is derived from combinations of these primitives. **ForceOne** unifies the syntax for array element selection, procedure argument binding, and record element selection. The actions of binding arguments to a parameter and calling a procedure are separated. Overloaded identifiers, implicit conversions (coercions), and polymorphic procedures may be defined by the programmer. Visibility of identifiers is controlled both by block structure and by the location of objects within a hierarchical file system. Source files are separately compiled, and the compiler automatically determines which source files have been changed or have been affected by a change since the last time the program was compiled. The **ForceOne** language is defined and a prototype compiler for the language is described briefly.

Table of Contents

List of Figures

Chapter 3

Chapter 4

Chapter 1
Introduction

Why?

1. Why Programming Languages?

Computers are directed in a deterministic sequential manner according to an *algorithm*. Algorithms are expressed in *programming languages*. Unfortunately, most computers take direction in only one language: *machine language*. A *program* written in machine language is simply a sequence of bits. Such programs are very difficult for humans to read, write, or modify. For this reason, we invent programming languages in order to express algorithms in a manner more understandable to humans.

2. Why New Programming Languages?

The world abounds with many different programming languages all invented for the purpose of expressing algorithms. Each of these languages is capable of expressing the same algorithms; this can be proven by implementing any one language in terms of any other. Intuitively though, it seems that an expression of an algorithm in one language may be *better* than its expression in another. If it is generally true that the expressions of most algorithms in one language are better than in another, we say that the first language has greater *expressive power*. We continue to invent new programming languages in an attempt to provide greater expressive power so that the expressions of our algorithms will be easier to write, easier to read, and easier to modify.

3. Why Compiled Programming Languages?

In order to execute an algorithm expressed in a programming language on a computer, we must map that expression into the machine language for that computer. There are two approaches taken to performing this mapping: *interpretation* and *compilation*. An interpreter reads the source program and performs machine instructions to correspond to the logical operations of the source program. Translation is performed every time the program is executed. A compiler reads the source program and generates a machine language program which implements the same algorithm expressed in the source program. This machine language program is then executed directly by the computer. Interpretation does not involve any intermediate steps between the writing of the program and its execution; compilation places some restrictions on the design of the programming language but allows for much faster execution of programs. Compilation also allows for the detection of some errors before the program is run or distributed.

4. Why ForceOne?

ForceOne is intended to be a general purpose compiled programming language with greater expressive power than contemporary compiled languages. The expressive power of **ForceOne** is derived by combining primitives from a small set of very general orthogonal primitives. **ForceOne** is perhaps best viewed as a language *kernel*; it is sufficiently extensible that many features typically built into other languages are instead implemented by **ForceOne** source code.

Some of the more novel features of **ForceOne** are user defined types, overloaded identifiers, user defined automatic conversions (coercions), parameterized objects, and polymorphic procedures. **ForceOne** programs may be written in many source files which are arranged as a tree structure within the file system. Visibility of identifiers is controlled implicitly by the structure of the source file tree. The **ForceOne** prototype compiler allows separate compilation on a source file basis, with the order of compilation determined automatically by the compiler.

The philosophy behind programming in **ForceOne** is that programs should be written as a set of small independently compilable modules. The smaller the units of compilation, the less work must be done to recompile the program in the presence of minor changes. Furthermore, if we assume that modules will always be small, we can build a more efficient compiler. The prototype **ForceOne** compiler builds a parse tree in memory for the entire module being compiled. Although this compiler cannot compile large modules, it is much faster than compilers implemented as several programs passing information from one to another, such as the C compiler on Unix. This approach is especially viable for personal computers which may have a megabyte or more of memory but slow disks.

ForceOne has been designed with the aim of generality. Exceptions and special rules have been avoided wherever possible, and the syntax is lightweight. A prototype compiler for **ForceOne** has been implemented. The code generation phase of the compiler is not complete, however virtually all the features of **ForceOne** have been implemented up to the point of generating complete decorated parse trees.

This thesis primarily describes the **ForceOne** language, and not its implementation. It is hoped that this definition of **ForceOne** will provide a solid foundation upon which a truly useful production language can be built (**ForceTwo?**).

Background

ForceOne evolved from several predecessor languages. The first and most influential of these was L, presented in [Cor 81, Cor 83]. This language embodied the essentials of the type structure and separate compilation mechanisms of **ForceOne**. Leclerc's language *L* [Lec 84] was developed from Cormack's L although it did not contain many of the more powerful features of the type structure. M [Jud 85] was also developed from Cormack's L, and contributed to the design of records. Finally, the file structure is enhanced from Thoth [Car 79].

ForceOne was developed in parallel with the language Zephyr [Cor 85]. Zephyr was designed with the intent of becoming a production language. **ForceOne**, on the other hand, was being designed while a prototype **ForceOne** compiler was being constructed; **ForceOne** stopped evolving sooner than Zephyr did in order that the implementation could proceed. Two reports [Cor 86a, Cor 87] and [Cor 86b] examine in greater detail the **ForceOne** type structure, the philosophy behind its design, and some problems which it easily solves.

The rest of this thesis is organized into four chapters. Chapter two gives a brief overview of the novel features in **ForceOne** and compares them with similar features in other languages. Chapter three is a reference manual for **ForceOne**, and discusses its novel features as well as its unexceptional features in greater detail. Chapter four describes the implementation of a prototype compiler for **ForceOne**, and presents the most important algorithms used in the compiler. Finally Chapter five discusses some problems that have been discovered with **ForceOne**, presents some solutions, and suggests additional useful features. This chapter also elaborates on the derivation of **ForceOne** from its predecessor languages. The appendices contain information useful when learning to program in **ForceOne**, including a complete grammar for the language.

Chapter 2

The ForceOne Solution Comparison and Summary

This chapter discusses the presentation of four major language features in existing programming languages. While some languages present good implementations of one or two of these features, only **ForceOne** presents satisfactory implementations of all four features. A brief introduction to **ForceOne** is then given, and the presentation of these features in **ForceOne** is outlined.

The four features which will be discussed are overloading, coercions, parametric polymorphism, and modularity. Overloading is the assignment of two or more meanings to a single identifier or operator, and the selection of a particular meaning based upon information from context. Coercions are conversions of an object from one type to another which are not specified by the programmer but are determined by the compiler from context. Parametric polymorphism is defined by Cardelli [Car 85] to mean the parameterization of an object by a parameter which may be of many types. Modularity is the ability to encapsulate related entities into a *module* which provides an external interface to these entities but may restrict access to their internal structure. An *abstract data type* is a form of module which allows no access to the internal structure of the module. These features are all desirable in a programming language because they all serve to increase the expressiveness of the language and to decrease the complexity of programs written in the language. Overloading aids in conservation of names; coercions increase both writability and readability; polymorphism aids the creation of libraries of general purpose re-usable software; and modularity enhances the modifiability of software by reducing the extent to which a change to one module can affect the rest of the program.

Existing Languages

1. Overloading

PL/1 [PL1 76] is one of the oldest languages providing user-defined overloading. In PL/1, a `generic` statement groups together a number of procedures under a common name. When the generic name is used, the appropriate definition is selected according to the types of the actual parameters. These types are not matched against the formal parameter types of the procedure definition, but are matched instead against abstract type specifications in the `generic` statement. A disadvantage of the `generic` statement in PL/1 is that all of the overloaded functions must be specified in one common place; there is no method to add another definition elsewhere.

In Algol 68 [Wij 76], operators may be defined by the user and may be overloaded. Selection is done by matching formal and actual types. Operators, unlike procedures, are not first-class in Algol 68; they may not be assigned or passed as parameters.

In Ada [Ada 83], operators, functions, and enumeration literals may be overloaded by the user. The types of the parameters and the desired result type are used in selection. There is no way of preventing functions that have the same name from overloading one another. Furthermore, if two names would normally overload but cannot be distinguished because their parameter types are identical, the more recent definition hides the earlier one.

C++ [Str 86] allows user-defined overloaded operators and functions. Overloaded operators are resolved by one bottom-up pass over the expression, so the result type may not influence the selection. Overloading of parameterless objects is not possible in C++ for this reason.

Clu [Lis 77] is a fairly recent language based on the *class* concept of Simula 67 [Dah 70]. In Simula 67, a class describes a set of values and a set of operations. Each instance of the class has its own value, and its own set of operations. While Simula 67 does not allow user-defined overloading, Clu allows definitions of a function from different `clusters` (classes) to overload. When this function is applied the first parameter determines the `cluster` to which the function belongs.

2. Coercions

PL/1 does not allow user-defined coercions, but it does have a wide variety of builtin coercions which may be applied transitively: almost any type can be converted to any other type. The conversion rules in PL/1 often yield unfortunate results. For example, the integer `123` when converted to `CHAR(3)` is first converted to `'  123'` and then to `'  '`.

C [Ker 78], Pascal [Pas 80], and many other contemporary languages provide a fixed set of builtin coercions. However, none of these languages take the type of the eventual result into account when selecting coercions for an expression. It has become accepted by the computing community that the statement

```
float x;
x = 1 / 3;
```

assigns to `x` the value `0`, where a novice would quite justifiably assume that `x` should receive the value `0.333...`

C++ allows user-defined coercions. At each step in a bottom-up pass over the expression tree, a particular definition of an overloaded operator will be selected if no coercions are required to match its arguments. If no such definition exists, there must be only one definition that can be matched by applying coercions to some of its arguments. Suppose we introduce the new type `complex` along with the constant `i` and the overloaded operator + which accepts two arguments of type `complex` and yields a result of type `complex`. We create two coercions which convert from `real` to `complex` and from `complex` to `real`. The expression

```
complex c;
c = 1.0 + i;
```

cannot be resolved by C++ since there are two choices available: either `1.0` can be coerced to `complex` and the `complex` version of + used, or `i` can be coerced to `real`, the `real` version of + used, and the result coerced to `complex`.

3. Parametric Polymorphism

Traditionally, parametric polymorphism has been achieved either through loopholes in type checking or by the use of macro processors. C allows the definition of polymorphic functions simply by not checking that types match across function calls; it is up to the programmer to ensure he is doing something sensible. The function `printf` is the best example of this. `printf` accepts one or more arguments of any type; their types are encoded as a string which is passed as the first argument.

Algol 68 provides a more controlled approach known as a type `union`. Functions that operate on `union` types are in a sense polymorphic, though they must enumerate all possible types on which they operate. Algol 68 provides a `uniting` coercion that allows actual parameters to match the more general formal parameters. When the `union` type is used, a form of case statement called a `conformity clause` is used to determine the actual type of the parameter. Algol W [Sit 72] has unions only for `reference` types. EL1 [Weg 74] is a language that allows type manipulation and operations to be specified on objects whose type is known only at run time. However, these objects behave like unions: the set of possible types must be known in advance, and polymorphic operations are akin to case statements.

Simula 67 provides a polymorphic facility via class hierarchies. A general class may be defined that has a number of operations defined on it. These operations then automatically apply to any subclasses that are derived from this class. The superclass may also specify `virtual functions` that must be defined in the subclass. The Simula 67 class structure requires that all classes (and hence functions) be organized into a strict hierarchy, which severely restricts the nature of polymorphism that can be expressed in this way. Some languages (Taxis [Myl 80], for example) have *multiple inheritance* which allows a subclass to be derived from more than one superclass, ameliorating somewhat the restrictive nature of class hierarchies.

Clu reinforces the Simula 67 notion that a class contains the definitions of all permissible functions. However, inheritance and subclasses are abandoned in Clu in favour of parameterized `generic clusters`. A `generic cluster` is a template that describes an infinite set of clusters. A `generic cluster` cannot be used directly; a specific member of the set must first be *instantiated* by the programmer. Upon instantiation, constant parameter values are substituted and a new `cluster` is created. Ada provides polymorphic functions by special `generic` program units that resemble `generic clusters`.

Russell [Don 85] is a language that uses full run-time typing and type inference to provide polymorphism and considerable expressive power: types are first-class values that are computed by procedures and may be stored in variables. The notion that a type is the set of all permissible operations is essential to Russell – the run-time representation of a type is a list of procedures and a reference to a procedure is

an index into this list. A recent implementation of Russell [Boe 86] attempts to verify statically the type consistency of the program. The language Poly [Mat 85], based on Russell, has sacrificed some expressive power to simplify the language and its implementation; the essential philosophy is unchanged.

ML [Mil 78, Mil 85] is a functional language which does not have types or declarations in the traditional sense. An inference algorithm is used to infer the usage of functions, and to ensure that all uses of a function are consistent. Two consequences of this approach are that functions cannot be overloaded, and that only first order polymorphism can be expressed. A first order polymorphic object is an object with a parameter which may be matched by an infinite number of argument types. A second order polymorphic object is an object with a parameter whose argument must match a first order polymorphic object.

4. Modularity

Early languages provide only a weak form of modularity. Simula 67 allows objects and functions to be grouped together into classes, however no mechanism is available to prevent access by functions outside the class to the internal details of the class. More recently the need for the ability to hide information has been recognized, giving rise to the concept of an *abstract data type*. An abstract data type consists of a type plus a set of operations available on objects of that type. The internal details of the type and its operations are not available outside the abstract data type.

Modula 2 [Wir 82] is a Pascal-like language which allows objects to be encapsulated within a `module`. The types and operations to be visible outside the module must be listed in an `export list`. The types and operations of other modules used within a module must be listed in an `import list`. This redundant specification of the visibility of names means much writing for the programmer, and tends to encourage the use of standard import lists which simply import everything under the sun. Ada has `packages` which are similar to Modula 2's `modules` although the mechanism is somewhat less verbose.

Each object in **ForceOne** has a type which describes the set of values it may take on. The functions which operate on objects of a particular type are not considered to be part of that type. Types are not first-class values; they are manipulated only at compile time.

A type declaration introduces a new type which is implemented as some base type, but is nevertheless unique. The type declaration also defines two functions, `retype` and `detype`, which convert from the base type to the new type and from the new type to the base type respectively. These functions are overloaded. By restricting access to `retype` and `detype`, the programmer can make the new type opaque. By providing automatic conversions, the programmer can make the new type essentially interchangeable with the base type.

Primitive types in **ForceOne** include `int`, `real`, `bool`, `char`, and `void`. `ref` types describe storage that holds a value of a particular type, as in Algol 68 [Wij 76], e.g. `ref int`. `routine` types describe executable procedures that yield a particular type as a result, e.g. `routine bool`. `record` types contain fields of various types; declaration of a `record` type also defines functions to access these fields. Types may be parameterized. A typical parameterized type might be

```
[ real, int ] routine real
```

This type is the **ForceOne** equivalent of a Pascal procedure. Arrays are constructed by parameterizing a `ref` type.

```
[ 1..10 ] ref int       \ an array of 10 int's
[ 1..4, 1..4 ] ref real \ an array of 16 real's
```

1. Overloading in ForceOne

ForceOne provides both hiding and overloading of identifiers under control of the programmer. Any identifier will overload earlier definitions of the same identifier if its declaration is prefixed with the keyword `overload`. If `overload` is not present, the declaration hides all earlier declarations. For example,

```
overload i: ref int
```

declares `i` as an `int` variable which overloads all previous definitions of `i` whose declarations were also preceded by `overload`. Any identifiers except those representing types may be overloaded; all that is required is that their types be different. In resolving overloaded identifiers, the result type of the expression may affect the selection as in Ada.

2. Coercions in ForceOne

ForceOne provides two kinds of user-defined coercions, called *widenings* and *narrowings*. A narrowing is intended to be a coercion which loses information in coercing to a narrower type; a widening preserves information. The **ForceOne** compiler uses several rules to select among overloaded identifiers and coercions in an expression. The rules are set up so as to try to preserve the most information by doing a calculation in the widest of its operand and result types. For example, if the environment defines

```
'/': [  int,  int ] int
'/': [ real, real ] real
widen  int_to_real: [  int ] real
narrow real_to_int: [ real ]  int
```

which are the natural / operations and coercions for the types `int` and `real`, then the expression

```
x: ref real
x := 1 / 3
```

yields the value 0.333... Both `1` and `3` are widened to `real`, and the `real` version of / is used. Similarly the following example assigns to `i` the value 1.

```
i: ref int
i := 1 / 3 + 0.7
```

3. Parametric Polymorphism in ForceOne

ForceOne provides parametric polymorphism through a mechanism called *query* parameters. In a formal parameter list, all or part of the type of a parameter may be replaced with ? *identifier*, as in the following specification of the routine `poly`.

```
poly: [ a: ?t ] routine int
```

`poly` accepts only one argument but has two parameters, one of which is implicit. The value of the argument is bound to `a`, and the type of the argument is bound to `t`. Within `poly`, `t` is considered a new type, distinct from all others in the program. Typical calls to `poly` might be

```
i: ref int
poly[ i ]
poly[ 1 + 2.0 ]
```

An argument of any type may be passed to `poly`; the effect and implementation is identical to coding `poly` as

```
poly2: [ t: type, a: t ] int
```

and passing the type explicitly as follows.

```
i: ref int
poly2[ int, i ]
poly2[ real, 1 + 2.0 ]
```

Another mechanism known as *automatic* parameters allows the programmer to specify requirements for the calling environment of a procedure. Parameters following | | in a parameter list are automatic parameters. When a parameter is specified as automatic, the user provides no corresponding argument; the compiler will search the symbol table at the call site to find an identifier of the appropriate type to be passed as the automatic argument. Automatic parameters are generally used in conjunction with query parameters in order to allow a routine to perform operations upon its polymorphic parameters. For instance, the following specification of ** allows a user to call ** in the usual infix manner, but ** requires that whenever it is called a * must exist which ** uses to implement its operation.

```
'**': [ a: ?t, b: int || '*' : [ t, t ] t ] t
```

* is passed implicitly, as shown in the following call to **.

```
'*': [ real, real ] real == ...
i: ref int
x: ref real
x := 2.3 ** i
```

4. Modularity in ForceOne

Modularity in **ForceOne** programs is inherent in the structure of the source code. Every source file is a module. Source files are arranged in a tree, with visibility between source files controlled by their position in the tree. Identifiers are neither exported nor imported; they are declared once and their visibility is determined by the source file in which they are declared.

Each source file declares a set of identifiers which are called its *outermost* declarations. All outermost declarations of the program have the same duration, but they have different visibility. Program code outside the subtree rooted at a given node cannot reference identifiers declared in descendants of that node. Each node in the source file tree is an abstract data type; any types or operations it declares are available to other source files, but any internal details declared in its descendants are not available outside the subtree. Program code inside the subtree can reference identifiers declared in any of the subtree's ancestors and their siblings. Visibility toward the leaves of the tree is limited to one level; visibility toward the root is not restricted.

This visibility structure not only encourages the use of abstract data types, it also encourages a disciplined source organization. A programmer reading someone else's program will more easily be able to find the declarations of objects since their locations within source files is not haphazard.

Chapter 3

Language Reference Manual

This chapter is a reference manual for the **ForceOne** programming language. The intent of this chapter is to provide a concrete definition of **ForceOne** upon which future languages in this class may be based.

Grammar Description

Backus Naur Form is used to describe the language grammar. The syntax used is as follows.

A	$\equiv$	B	A is a B
	$\equiv$	C	or a C
		$\left[A \right]$	A is optional
		$\left[A \right]^*$	A repeated zero or more times
		$\left[A \right]^+$	A repeated one or more times
		$A \mid B$	A or B

There are several symbols used to represent classes of characters when defining the lexical structure of the language.

alpha	`a` through `z`, `A` through `Z`
digit	`0` through `9`
src_char	all printable ascii characters except \

Lexical Elements

A **ForceOne** *source file* is a stream of printable ascii characters terminated by an end-of-file. The smallest indivisible units within a **ForceOne** program that have an associated meaning are known as *tokens*. Some **ForceOne** tokens follow.

```
sail   +   {   ==   "hi there"   54   23.0
```

Tokens may be placed without regard for column position with any amount of whitespace between them. However, a token may not span more than one source line, and some tokens must be separated by whitespace from some others when they are to appear adjacently; for example, the tokens 3 and 4. *Comments* may appear anywhere in **ForceOne** source code so long as they do not break up tokens, and are delineated by a backslash (\) character and the end of the source line.

```
a := b + c                  \ This is a comment!
```

1. Identifiers

Identifiers are the means by which objects are tagged and referenced. **ForceOne** has two methods of representing an *identifier*. The first method is by just giving the identifier's name, but there are some restrictions on the characters that may be used to form the name.

$$identifier \equiv alpha \left[alpha \mid digit \mid \_ \right]^*$$

The name must begin with an upper or lower case alphabetic character. Subsequent characters may be alphabetics, digits, or underscores (_). There is no restriction on the length of the identifier. There are 18 reserved keywords which may not be used as identifiers. A list of these reserved keywords is given in Appendix A.

```
sailboard   ForceOne   number_5   i
```

The second method of representing an identifier is by enclosing the identifier's name in grave accent (`) characters.

$$identifier \equiv \text{`} \left[ src\_char \mid escape\_seq \right]^* \text{`}$$

With this form, any character may appear anywhere in the identifier, except the grave accent character. Furthermore, an escape sequence signalled by the \ character may be used to enter non-printable characters or grave accents in the identifier. A list of recognized escape sequences is given in Appendix B. The only character which is invalid in such an identifier is the ascii null (\$00) character.

```
`Honolulu, U.S.A.`   `+`   `\$1Bhi there\n`
```

The grave accent characters are considered part of the identifier for purposes of comparison.

The first form of identifier is used most commonly. The second form is needed when defining a builtin operator, but may be used anywhere.

2. Constants

Since **ForceOne** is a type secure language, all *constants* have associated types. There are four different types of constants: *integer constants*, *real constants*, *character constants*, and *string constants*. Integer constants are further subdivided into *decimal constants* and *based constants*.

constant ≡ *decimal_const*

≡ *based_const*

≡ *real_const*

≡ *char_const*

≡ *string_const*

2.1. Integer Constants

An *integer constant* is an unsigned number with some maximum value fixed by the implementation. Integer constants have type `int`. Integer constants may be entered either in decimal or in a specified base from 2 to 36.

decimal_const ≡ *digit* [*digit* | _]*

based_const ≡ [*decimal_const*] $ [*alpha* | *digit* | _]+

A decimal constant simply consists of a string of one or more decimal digits. A based constant consists of an optional decimal constant immediately followed by a $ immediately followed by the constant to be specified. The decimal constant preceding the $ gives the base of the constant. If this decimal constant is omitted, base 16 is assumed. The alphabet is used to represent digit values from 10 to 35, with no distinction being made between upper and lower case. Underscores (_) may be inserted anywhere in an integer constant and have no effect on the value. The constants

```
1_234    $10_00    8$377
```

each have the following respective values in decimal.

```
1234    4096    255
```

2.2. Real Constants

A *real constant* contains either a dot (.) or a possibly signed exponent to differentiate it from a decimal constant. Real constants have type `real`.

real_const ≡ *decimal_const* . [*decimal_const*] [*exponent*]

≡ . *decimal_const* [*exponent*]

≡ *decimal_const exponent*

exponent ≡ e | E [+ | −] *decimal_const*

Like integer constants, real constants are unsigned and are subject to implementation restrictions upon their precision and range. Real constants may also contain _ characters. The following are all valid real constants.

```
123.4   12e-29   .124_5   654.   1e+10
```

2.3. Character Constants

A *character constant* is represented as a single character or escape sequence enclosed between single quotes ('). Character constants have type `char`.

char_const ≡ ' *src_char* | *escape_seq* '

There are 256 different characters, corresponding to the 256 possible ascii characters.

```
'a'   '$'   '\n'
```

2.4. String Constants

A *string constant* is represented as a sequence of zero or more characters or escape sequences surrounded by double quotes (").

string_const ≡ " [*src_char* | *escape_seq*]* "

A string constant has type `[ 1..n ] char` where `n` is the length of the string.

```
"hi there\n"   "Neil Pryde International"
```

3. Operators

There is a large set of builtin operators of various precedence levels and associativities which may be defined by the programmer. Each operator may be named for purposes of definition by using the second form of identifier. The identifier corresponding to a given operator is simply that operator surrounded by grave accent characters. Some of the operators follow.

```
+   *   >>   :=   =   &   --
```

Some of these operators are binary infix operators and some are unary prefix operators. The two operators + and − are both. A complete list of operators is given in Appendix C.

4. Punctuators

The only remaining tokens are punctuators which are required to parse the various forms of expressions. Some examples of punctuators follow.

```
{   }   {{   }}   ,   ;   ?   !
```

Expressions

The *expression* is the basic unit of program structure in a **ForceOne** program. **ForceOne** does not differentiate between statements and expressions as do many current programming languages. The analog of a sequence of statements is an *expression sequence*. A **ForceOne** source file consists simply of an expression sequence in which each expression is a declaration.

Expression sequences are formed by placing several expressions in sequence, with each expression optionally terminated by a semicolon (;).

$$expr\_sequence \equiv \left[expr \left[; \right] \right]^{+}$$

An example of an expression sequence follows.

```
f: type == int
i: ref f
i := 3 + 5
```

Semicolons are generally used to terminate an expression only when the place where the expression ends is not obvious. The only time a semicolon is actually necessary is in the sequence "`a; + b`". In this case, if the semicolon is omitted the sequence will be parsed as the single expression "`a + b`".

Some syntactic constructs require a list of expressions. An *expression list* is one or more expressions, each expression being separated by a comma (,).

$$expr\_list \equiv expr \left[, expr \right]^{*}$$

In an expression list the comma separates the expressions in the list. In an expression sequence the optional semicolon terminates each expression in the list.

```
a, b + c, 123, f[x]      \ expression list
a; b + c; 123; f[x];     \ expression sequence
```

The different kinds of expressions have different precedences and associativities in order to disambiguate otherwise ambiguous expressions. Parentheses may be used to force evaluation to occur in a different order.

```
a + b + c * d
```

Since * has a higher precedence than +, and since + is left associative, the above expression will be parsed as

```
( ( a + b ) + ( c * d ) )
```

unless parentheses are inserted to force a different evaluation order. In an expression list, expressions are evaluated from left to right across the list.

All expressions have a *type*, and all except those whose type is `void` are said to *yield* a *value*. The type of an expression is determined statically at compile time. The value yielded by an expression is usually determined at run time, however for certain expressions that always yield the same value this value may be computed at compile time. Since some expressions may have type `type`, and must therefore yield a type, the value yielded by these expressions is always determined at compile time.

expr ≡ *basic_expr*
≡ *type_expr*
≡ *declaration*
≡ *routine_text*
≡ *active_expr*
≡ *control_expr*
≡ *aggregate*
≡ *cast*
≡ *typemod_expr*
≡ *void_expr*
≡ *op_expr*
≡ (*expr*)

1. Type Expressions

Types are used in **ForceOne** to describe and to determine the properties of objects. Types have two attributes: size and alignment. The operations available on a type are not considered part of the type. Types are not first class objects of the language, and may therefore be thought of as meta-objects, although in some special situations they may be manipulated as if they were objects. Since the type of an object is determined statically, types are manipulated only at compile time, and all type expressions have constant value. A type expression is an expression built from *type generators* which yield type `type`, from identifiers of type `type`, and from *query declarations*. Query declarations are discussed later.

type_expr ≡ *ref_tgen*

≡ *routine_tgen*

≡ *param_tgen*

≡ *enum_tgen*

≡ *record_tgen*

≡ *type_tgen*

≡ *subrange_tgen*

≡ *query_decl*

≡ *identifier*

1.1. Type Generators

A type generator is a construct which yields a value of type `type`. **ForceOne** has seven forms of builtin type generators.

1.1.1. The Ref Type Generator

The *reference* type generator is used to generate a type which may refer to an object of a particular type.

ref_tgen ≡ `ref` *type_expr*

The value yielded by the type expression following the `ref` keyword indicates the type to which the reference may refer. The **ForceOne** reference type is the same as the Algol 68 reference type. A reference type is similar to a pointer in more conventional languages. Consider the following declaration:

```
i: ref int
```

This says that `i` can refer to an object of type `int`. This is analogous to the declaration

```
int i;
```

in the C language. The **ForceOne** declaration

```
ri: ref ref int
```

declares `i` to be, in C terminology, a pointer to an `int`.

1.1.2. The Routine Type Generator

The *routine* type generator is used to generate a routine type. An object of such a type must be called to yield a particular value. A routine is used to defer the evaluation of its enclosed expressions.

routine_tgen ≡ `routine` [*type_expr* | `void`]

The type expression following the `routine` keyword gives the type of the value which would be returned were a routine of this type called. If the keyword `void` is used instead, then the routine returns no result.

1.1.3. The Parameterization Type Generator

Unlike conventional programming languages which provide arrays of objects, **ForceOne** provides a much more general construct called *parameterization*. Except for two restrictions, all type expressions can be parameterized.

param_tgen ≡ `[` *expr_list* [`||` *expr_list*] `]` *type_expr*

The type expression following the] indicates the type being parameterized, and this type is called the *base type*. Each of the expressions in the expression lists indicates the type of a *parameter* of the parameterized type. Expressions preceding the || may be either all declarations or all type expressions. Any expressions following the || must be declarations. Declarations are of the form

identifier : *type expression*

If the || is present, the declarations following it are *automatic parameters* of the type which will be discussed later. A parameter may be used in the specification of the type of a parameter to its right and in the specification of the base type. The base type may not be the type `void`. The base type may not yet be the type `type` as user defined type generators are not yet supported.

```
[ r: real, i: int ] routine char
[ int, real ] int
```

The first example above is the analog of a procedure returning `char` whose parameters are of type `real` and `int`. The second example yields a type parameterized by two parameters: an `int` and a `real`. The type being parameterized is `int`. This is a good example of the power of the parameterization construct as in very few conventional languages can a comparable type be constructed. In Pascal terms this might be called an array of constants.

There are no restrictions on what types may be used for parameters. It is quite feasible to declare an infinite size array; its implementation will depend upon the method of declaration. Also, a parameterized routine need not be thought of as something special; it can be considered to be an array of unparameterized routines. Again, whether this is truly the case depends upon the method of declaration. Certainly there are some types which cannot be represented; if the user really wishes an infinite number of `int` variables declared with a storage declaration he is out of luck.

1.1.4. The Enumeration Type Generator

The *enumeration* type generator is used to declare a list of names to be values of some new type. The syntax for the enumeration type generator is

enum_tgen ≡ `enumerate` [*ident_list*]

ident_list ≡ *identifier* [, *identifier*]*

A new type is created which is the type yielded by the type generator, and each identifier is declared as

being a value of that type. It is important to realize that the enumeration type generator implicitly creates a new type, unlike the other type generators referenced earlier. Because of this,

```
i: ref int
j: ref int
a: enumerate[ aa ]
b: enumerate[ aa ]
```

`i` and `j` have the same type, buf `a` and `b` do not. This ensures that two enumerated types declared in different places in a program using the same names for values will not be the same type.

1.1.5. The Record Type Generator

The *record* type generator yields a new type composed of several *field types* and declares names for these field types. A new type is generated, and each of the field names is declared to be a type parameterized by this new type and with base type given in the field declaration.

record_tgen ≡ `record [` *field_list* [`|` *field_list*]* `]`

field_list ≡ [*field*]+

field ≡ `[` *field_list* [`|` *field_list*]* `]`

≡ *storage_decl* [`;`]

The definition of the record type generator given above is best described by some examples.

```
record [ a: ref int; b: ref real ]
```

This example yields a record type containing two fields.

```
record [ a: ref int | b: ref real ]
```

This example yields a record type containing either a `ref int` or a `ref real`.

```
record [
    a: ref int;
|
    [
        b: ref char;
        c: ref real;
    ]
]
```

This example yields a record type containing either a `ref int` or a `ref char` and a `ref real`.

When required the compiler generates an invisible *tag field* which is also a field of the record. This field contains one bit for each field of the record. When a variable is initially created of some record type, each of these tag bits will be false. When a field is assigned to, the tag bit for that field is set true and the tag bits for every other field which cannot exist at the same time as the field being assigned are set false. When a field is referenced, the tag bit for that field is checked to ensure that it is true. In this way records in **ForceOne** are kept type secure without requiring that the entire variant of a record be assigned to in one operation. There is a mechanism available called *query* which allows the programmer

to test the tag bit of a particular field.

Like enumeration, the record type generator implicitly generates a new type so that two records which happen to look the same will not be the same type.

1.1.6. The Type Type Generator

A new type may be explicitly introduced to the program by the use of the *type* type generator.

type_tgen ≡ **type**

This type generator yields the value **type**. This type generator may only be used on the left hand side of the == in an equivalence declaration or in a storage declaration which is a formal parameter. When it is used in a formal parameter it indicates that the formal parameter is a parameter of type **type**; such parameters will be discussed later.

1.1.7. The Subrange Type Generator

The *subrange* type generator generates the type `int` but also attaches some attribute information to the type. This information is not used in determining type equality; the type generated with the subrange type generator is exactly the type `int`.

subrange_tgen ≡ *expr* .. *expr*

When values are bound or assigned to an object of this type, the compiler will ensure that the value lies within the values specified. Note that the expressions representing the bounds are computed at run time each time the type expression is reached during execution. Therefore, in the routine

```
{
    i: ref 1..n
}
```

the variable `i` may have different bounds for each execution if the value of `n` changes.

1.2. Builtin Types

There are several builtin types defined in the library. They are

```
int
real
char
bool == enumerate[ false, true ]
```

The first three are the types of the various typed constants. The last, `bool`, is the type required by an *if expression*.

2. Basic Expressions

A *basic expression* is simply an identifier or a constant.

basic_expr ≡ *identifier*

≡ *constant*

3. Declarations

Declarations are the means by which new identifiers are introduced to the program. Except for the 18 reserved keywords listed in Appendix A, all identifiers, whether declared by the user's program or by a library, are created by declarations. Declarations may only appear as part of an expression sequence, or as part of an expression list which is used by the parameterization type generator. New identifiers introduced by declarations may hide existing definitions of the same name or may overload those definitions. There are three kinds of declarations: *equivalence declarations*, *storage declarations*, and *query declarations*.

declaration ≡ *equiv_decl*

≡ *storage_decl*

type_expr ≡ *query_decl*

3.1. Equivalence Declarations

An equivalence declaration introduces a new identifier, assigns it a type, and binds a value to it when the declaration is executed at run time. An equivalence declaration has the following syntax.

equiv_decl ≡ [*decl_type*] *identifier* : *type_expr* == *expr*

decl_type ≡ `overload` | `widen` | `narrow`

The identifier appearing on the left hand side of the colon is the name of the identifier being defined. The type expression appearing on the right hand side of the colon yields the type of the identifier. The expression appearing on the right hand side of the == operator yields the value of the identifier. Equivalence declarations always yield type `void`.

```
loop[] {
        \ j does not exist here
    j: int == n + 1
        \ j now exists
    n := n + 1
}
```

This example shows an equivalence declaration. The identifier `j` is created with type `int` and bound to the value `n + 1`. Since the computation of the value is done at run time, and since `n` has a different value during each execution of the loop, a different value will be bound to `j` in each execution of the loop.

One rather important instance of an equivalence expression occurs when the type expression is simply `type`. The identifier being declared is therefore a type, so this kind of equivalence declaration is called a type declaration. In this case, the value to be bound to the identifier is actually computed and bound at compile time.

```
a: type == int
b: type == [ 1..10 ] real
```

In the example above, `a` is declared to be a type whose value is `int`. `b` is declared to be a type whose value is a parameterized type. Except when the value expression is a record or enumeration type generator, a type declaration implicitly generates a new type distinct from all others in the program which becomes the value of the identifier being declared. This is discussed later in the section *Type Equivalence*. Type declarations have an important restriction: the declaration type of a type declaration cannot be `overload`, `widen`, or `narrow`.

When the type of the identifier being declared is parameterized, there is an additional rule which applies to an equivalence declaration. If the parameters of the type are named, the expression on the right of the == is a generic instance of the base type. The names of the parameters are visible on the right side of the ==. Otherwise, the expression after the == must have the same type as the whole parameterized type.

```
a: [ i: 1..5 ] int == i + 1
b: [ 1..5 ] int == {{ 2, 3, 4, 5, 6 }}
```

The first example uses a generic instance to define `a`. Using a generic instance has the effect of evaluating the right side for each possible parameter value and storing the results. The second example uses an *aggregate* to define `b`. The two declarations have equivalent effect.

There is another kind of equivalence declaration used to interface to other languages.

equiv_decl ≡ [*decl_type*] *identifier* : *type_expr*
== `external` *string_const*

The string following the `external` keyword is an external symbol manipulated by the host's linker. The purpose of this kind of equivalence declaration is to specify the type of procedures or variables declared in another language. In fact, in the prototype compiler all operations on builtin types, including :=, simply result in calls to assembly language routines which implement that function.

3.2. Storage Declarations

A storage declaration is similar to an equivalence declaration, except that the value of the identifier is bound to a storage location on the program stack. A storage declaration has the syntax

storage_decl ≡ [*decl_type*] *identifier* : *type_expr*

Since the identifier is being declared as a variable, the type of the identifer must be a reference or parameterized reference type. If the identifier is a parameterized type, the number of instances represented by the parameterization must be finite. Storage declarations yield type `void`.

```
i: ref int
j: ref record[ a: real, b: real ]
```

The first declaration in the example above introduces an `int` variable named `i`. The second declaration introduces an identifier `j` which is a record composed of two `real` fields.

3.3. Query Declarations

Query declarations are the means by which polymorphic routines are implemented. There are two flavours of query declaration: a query declaration of a type and a query declaration of a subrange bound.

$$\textit{query\_decl} \equiv \left[\ \texttt{?}\ \right]^{+}\ \textit{identifier}$$

Query declarations yield either type `type` or type `int` depending on where they are used. If the declaration appears on the left or right side of a subrange generator, the identifier is declared as an `int`; otherwise the identifier is declared as a type. The declaration

```
f: [ x: ?t ] routine int
```

declares `f` to be a routine which accepts one parameter of arbitrary type `t`. The type `t` is implicitly passed as a parameter to the routine. The number of `?` characters preceding the identifier determines the number of parameterization brackets outward the identifier is to be declared. Query declarations will be discussed in greater detail later on.

3.4. Overloading Declarations

As mentioned previously, a declaration may either hide earlier declarations of the same name or it may overload them. *Overloading* an identifier simply means giving it two different meanings. Which meaning is used depends upon the context. Most programming languages have overloading of some of their builtin operators. The + operator is very commonly overloaded to operate both on integer and on floating point operands. Ada allows the user to define overloaded procedures. The overloading supported by **ForceOne** is more general; any identifier may be overloaded by the user.

The declaration type determines whether the declaration overloads or hides earlier declarations. If none of the keywords `overload`, `widen`, or `narrow` appears before the identifier being declared in the declaration, then this declaration hides all earlier definitions. If, however, one of the keywords is present, then the declaration is an overloading declaration. The new definition is overloaded with all earlier definitions which are not hidden and which were declared by an overloading declaration.

```
         i: ref int     \ #1
         i: ref int     \ #2  this hides #1
overload i: ref int     \ #3  this hides #2
overload i: ref real    \ #4  this overloads with #3
```

Record fields and enumeration values are always considered to be overloading declarations, despite there being no `overload` keyword in front of each field declaration or enumeration identifier.

3.5. User Defined Coercions

If one of the keywords `widen` or `narrow` appears before the identifier then the declaration creates a *user defined coercion*. The type of the identifier must be a routine taking one parameter. This routine will be automatically called by the compiler if necessary in order to match a value to some context that requires a value of a different type. Both the parameter and the type returned by the routine may not be `ref` or `routine` types.

```
widen int_to_real: [ int ] routine real == ...
r: ref real
r := 2
```

In the example above, the coercion `int_to_real` is applied to `2` to yield a `real` value which can be assigned to `r`. User defined coercions are discussed at greater length in the section *Overload and Coercion Selection*.

4. Routine Texts

Routine texts are the means by which an algorithm is encapsulated into an object.

routine_text ≡ { *expr_sequence* }

A routine text may return a typed value. If there are no `return` expressions within the routine text, the value yielded by the last expression in the expression sequence is returned.

```
{ a := b;  2 }
```

The above routine text returns the value `2` which is of type `int`. The type of this routine text is therefore `routine int`. We can therefore say

```
rp: ref routine int
rp := { a := b;  2 }
```

which causes `rp` to refer to the object `{ a := b;  2 }`.

5. Active Expressions

There are four kinds of expressions in **ForceOne** which cause an action to be taken at run time. They are *calling, dereferencing, instance selection*, and *field query*.

active_expr ≡ *call*
≡ *deref*
≡ *instance_select*
≡ *field_query*

They correspond to the three type generators routine, reference, and parameterization. They may be thought of as stripping one level of type generator from the object to which they are applied.

5.1. Calling

Any object whose type is `routine` *something* may be called. A call is specified by

call ≡ `!` *expr*

The value yielded by a call is that returned by the called expression. For example, the expression

```
i := ! { a := b;  2 }
```

causes `b` to be assigned to `a` and the value 2 to be returned, which is then assigned to `i`.

Any number of calls may be automatically inserted by the compiler if doing so would cause the type of the value returned to match the desired type of the expression. For instance,

```
rp: routine int == { 2 }
i: ref int
i := rp
```

a call to `rp` will automatically be generated in the third line of this example, causing `i` to be assigned the value 2.

Routine texts are an exception to the rule of automatic call. A routine text which does not have type `void` will never be automatically called. Thus one can write

```
if[ true ] {
    a := b
    b := c
}
```

and the routine text will be automatically called since it has type `void`. However,

```
i: ref int
i := { 2 }
```

will cause an error because the expression `{ 2 }` has type `routine int` and the context requires an object of type `int`.

5.2. Dereferencing

Dereferencing is the operation of selecting the object referred to by some other object.

deref ≡ `@` *expr*

For example, in

```
i: ref int
i := 2
j: ref int
j := @i
```

the last expression causes `j` to receive the value 2.

Like calling, dereferencing may be automatically applied by the compiler if necessary. Thus the above example could have been written

```
i: ref int
i := 2
j: ref int
j := i
```

and a dereference would be automatically inserted.

5.3. Instance Selection

Instance selection is the mechanism by which arguments are supplied to a parameterized object to yield the base object.

instance_select ≡ *expr* [*expr_list*]

The expression must be an object or a reference to an object parameterized by as many actual parameters as there are elements of the expression list.

```
f: [ i: real ] routine real == {
    i + 0.1
}
b: ref real
b := !f[ 2.3 ]
```

In the example above, an instance of the parameterized routine `f` is selected, the routine is called, and the result `2.4` is assigned to `b`. Since calls may be automatically inserted, the `!` could have been omitted. In the following example an instance of a reference to a parameterized object is selected, causing the reference to be propagated across the parameterization.

```
vector: ref [ 0..9 ] char
vector[ 2 ] := 'a'
```

Note that selecting an instance of a parameterized routine is independent of calling the routine. We can write

```
ri: ref routine int
ri := f[ 3.4 ];
```

Using the definition of `f` from above, `ri` will now refer to a routine which when called will yield `3.5`.

An instance may be selected not only from a parameterized type, but also from a reference to a parameterized type. Thus selection applied to the type

```
ref [ int ] real
```

will yield an object of type

```
ref real
```

The reference is said to *propagate* across the parameterization. Selecting an instance from a reference to a parameterized type is similar to selecting an element of an array in a conventional language.

5.4. Field Query

Field Query is used to determine if a field of a record is currently valid.

field_query ≡ *expr* ? [*expr*]

The expression before the ? is the field name, and the expression in brackets is the record object in question. The field query expression will yield either `true` or `false` indicating whether or not the field is currently valid.

A cautionary note about field query is in order. A field query is intended only to allow a programmer to determine which fields in a record defined with |'s are valid. If the record contains no |'s then the compiler may perform optimization and leave out the tag field. If this is the case, a field query on such a field will always return `true`. For such records this eliminates the run time overhead of checking the tag bit on each reference to a field and of setting the tag bit whenever the field is assigned to.

6. Control Expressions

ForceOne provides four kinds of expressions for modifying the flow of control within a routine text. These expressions allow branching, multi-way branching, looping, and return from a routine text in a structured manner. There is no goto mechanism to allow uncontrolled transfers of control flow.

control_expr ≡ *if_expr*

≡ *select_expr*

≡ *loop_expr*

≡ *return_expr*

6.1. The If Expression

ForceOne has an *if expression* to allow the conditional execution of an expression.

if_expr ≡ `if` [*expr*] *expr* [`else` *expr*]

The expression inside the brackets is evaluated and must yield a result of type `bool`. If the result is `true`, the following expression is executed. If the result is `false` and the `else` part is present, then the expression following the `else` is executed.

If no else part is present, or if the context requires a `void` result type, then the if expression has type `void`. If the else part is present, then the if expression may be used in-line.

```
a := if[ condition ] b else c
```

has the same effect as

```
if[ condition ]
    a := b
else
    a := c
```

in the absence of coercions.

`else` is right associative, so when an if expression is used as the body of another if expression, the `else` is associated with the closest `if`.

```
if[ a ]
    if[ b ]
        c
    else
        d
```

The above code fragment is exactly the same as the following fragment.

```
if[ a ]
  (
    if[ b ]
        c
    else
        d
  )
```

6.2. The Select Expression

The *select expression* is the mechanism provided by **ForceOne** for multi-way branching.

select_expr ≡ `select [` *expr* `]`

$$\left[\ \texttt{case [}\ \textit{expr\_list}\ \texttt{]}\ \textit{expr}\ \right]^{+}$$

$$\left[\ \texttt{else}\ \textit{expr}\ \right]$$

The control expression contained in brackets following the `select` keyword is evaluated, and a search is initiated over the expression lists following the `case` keywords for a matching value. A match is detected by invoking the routine

`'='`: [*type*, *type*] `routine bool`

with the control expression and the case expression as arguments. When `'='` returns `true`, the expression for that case is executed. If no match is found and an `else` part is present, the expression following the `else` keyword is executed. If no match is found and there is no `else` part, the `select` expression does nothing. The select expression itself has type `void`.

```
a: ref int
select[ @a ]
    case[ 1, 2 ]
        !f
    case[ x % 2 + 1 ]
        !g
    else
        !h
```

6.3. The Loop Expression

ForceOne provides a *loop expression* for iteration.

loop_expr ≡ `loop [` [*expr_list*] `]` *expr*

There are actually four different kinds of loops allowed, depending on the number of elements of the expression list. There may be one, two or three expressions in the expression list. All forms of the loop expression have type `void`.

If the expression list is not present, then the loop simply executes forever, or until a *return expression* causes termination of the containing routine.

If the expression list consists of one expression, then that expression must yield a value of type `bool`. Before each execution of the loop, this control expression is evaluated, and if it yields `false`, execution of the loop is terminated. This is the popular "while" loop.

If the expression list contains two expressions, the first is considered a control expression as above, and executed before each iteration of the loop. The second is an increment expression, and is executed after each iteration of the loop. Its value must be `void`.

If the expression list contains three expressions, the first is an initialization expression. It is executed just once before any executions of the loop are done. The second expression is the control expression, and the third is the increment expression. This is similar to the "for" loop in C.

```
loop[]
    !f                  \ loop calling f forever

loop[ a != 0 ] {
    !f
    a--                 \ loop while a != 0
}

loop[ a = 0, a-- ]
    !f                  \ same as above

loop[ a := 10, a != 0, a-- ]
    !f                  \ call f 10 times
```

6.4. The Return Expression

A *return expression* is used to cause termination of a named routine text without execution of its last expression.

return_expr ≡ `return [` *expr* `]`

A named routine text is a routine text which appears following the == in an equivalence declaration. When a return expression is executed, the expression inside the brackets is evaluated and its value becomes the return value for the routine.

```
a: routine int == {
    return[ 2 ]
    a := b
}
```

In the above example, the return expression causes a return from the routine `a`, returning the value 2 before the assignment takes place. The result type of the return expression itself is `void`.

Each return expression of a named routine must return the return type of the routine. When a return expression is used in a named routine, the last expression of the routine no longer yields a return value. The last expression will be expected to have type `void`. However, if no return expressions are used, the last expression of the named routine yields the routine's return value.

7. Aggregates

ForceOne provides *aggregates* to allow the user to supply a parameterized object to a context which requires one. An aggregate consists of a list of expressions, each expression yielding a value which becomes an instance of the parameterized type.

aggregate ≡ `{{` *expr_list* `}}`

The number of instances of the base type which are represented by the parameterized type must be finite. Therefore, objects parameterized by a `real` or a `type` may not be constructed with aggregates. The instances of the base type are listed in ascending order within each parameter, and from left to right across the parameter list.

```
rotate: [ 1..5 ] int == {{ 2, 3, 4, 5, 1 }}
firstrow: [ 1..2, 1..2 ] int == {{ 1, 1, 0, 0 }}
```

8. Casts

Since **ForceOne** supports overloading and user defined coercions, situations may occur in which an expression is ambiguous. A *cast* may be used to disambiguate such otherwise ambiguous expressions.

cast ≡ *type_expr* == *expr*

The type expression on the left side of the == gives the type of the expression on the right side of the ==. The cast yields the result yielded by the expression being casted. For example, with the declarations

```
overload vbl: ref int
overload vbl: ref real
overload fn: [ int  ] routine void == ...
overload fn: [ real ] routine void == ...
```

the following expression is ambiguous.

```
!fn[ @vbl ]
```

However, by applying a cast we can disambiguate this expression.

```
!fn[ real == @vbl ]
```

9. Typemod Expressions

A type declaration introduces a new type which is implemented as some other type, but is nevertheless a new type. A *typemod expression* is used to convert to or from the implementation type of the new type. The use of typemod expressions is restricted in order to support information hiding; where they may be used is discussed later in the section *Type Equivalence*. There are two forms of typemod expressions, one for converting an object of the new type to the implementation type, and one for converting an object of the implementation type to the new type.

typemod_expr ≡ *detype_expr*

≡ *retype_expr*

9.1. The Detype Expression

A *detype expression* is used to convert an object of some new type to its implementation type.

detype_expr ≡ `detype [` *expr* `]`

The value yielded by the detype expression is the same value as that yielded by the expression in brackets. The detype expression's type is the implementation type of the expression in brackets.

```
sail_area: type == int
my_sail: ref sail_area
i: ref int
i := 2 + detype[ @my_sail ]
```

When a type declaration is made, a special coercion is created which is simply a detype operation. Therefore it is not always necessary to explicitly detype an object; one level of detyping may be automatically applied by the compiler if necessary. The last expression in the above example could have been written as follows.

```
i := 2 + my_sail
```

9.2. The Retype Expression

A *retype expression* is used to convert an object of some implementation type into a new type built from that implementation type.

retype_expr ≡ `retype [` *expr* `]`

The retype expression is essentially an overloaded expression; its result value depends on the context in which it is used. Retyping is the inverse of detyping.

```
mod7: type == int
'+': [ i: mod7, j: mod7 ] routine mod7 == {
    retype[ (i + j) % 7 ]
}
```

In this case, `i` and `j` are detyped to `int` and the result of the addition is retyped to `mod7`.

10. Void Expressions

It may occasionally be desirable to have an expression which does nothing. In **ForceOne** such an expression is called a *void expression*.

void_expr ≡ `void`

This expression has type `void`, and yields no value.

11. Operator Expressions

ForceOne provides 33 *operator expressions* of varying precedences and associativities for use as short forms. Five of these expressions are unary expressions.

op_expr ≡ *expr binary_op expr*

≡ *unary_op expr*

A complete list of the available operators is given in Appendix C. The two expression forms

```
expr_1 b_op expr_2
u_op expr_1
```

are simply converted by the parser into the following forms.

```
'b_op' [ expr_1, expr_2 ]
'u_op' [ expr_1 ]
```

Program Structure

A **ForceOne** program is structured as a set of *source files* arranged in a tree. Each node in the tree contains source code and may have descendant nodes. This tree structure controls the visibility of objects declared at the outermost scope level of each source file. Objects declared at the outermost level of a source file have global duration; i.e. they are created at program invocation and destroyed at program termination. Within a source file, block structure controls the visibility and duration of declared objects. **ForceOne** has no explicit import or export mechanism; the tree structure of source files and block structure within source files are the only means available for controlling the visibility of identifiers.

The root node of a **ForceOne** program must declare a routine named `main`. This routine is called by the library upon program invocation. When this routine returns, the library terminates the program.

Figure 3.1 is a simple **ForceOne** program which just prints "hi there" on the user's terminal. It consists of two source files: one called "hi_there" and one called "hi". The root source file "hi_there" declares the routine `main` and calls the routine `hi` to do the work.

```
\ Root file "hi_there"
main: routine void == {
    !hi
}

\ Source file "hi", a child of "hi_there"
hi: routine void == {
    put[ "hi there\n" ]
}
```

Figure 3.1: A simple program

When **ForceOne** is implemented on an operating system such as Unix which does not directly support the node structure of **ForceOne** source files, some mapping between the operating system's file system structure and the **ForceOne** node structure must be defined. The current version of the **ForceOne** compiler is designed to run on Unix like systems. A Unix directory is created for each **ForceOne** source file. The contents of the source file are stored in a Unix file named "src" in the corresponding Unix directory. Thus, the source code for the routine `main` in the above example is stored in "hi_there/src"; the source code for the routine `hi` is stored in the file "hi_there/hi/src".

1. Visibility of Identifiers

The visibility of identifiers is controlled both by the source file structure and by *scopes* delineated by open ({) and close (}) braces. Declarations outside of any braces are called *outermost* declarations, and have global duration and visibility controlled by the source file structure. All other declarations are called *local* declarations, and have duration and visibility determined by their containing scope.

Braces are used to enclose a scope. Identifiers declared within braces are created at their declaration, and destroyed when the } for their scope is encountered, hence declarations and executable statements can be intermixed. Scopes may be nested.

When discussing trees the following terminology is used. The children of a node are those nodes that are immediately connected to the node in question in the direction of the leaves of the tree. The parent of a node is that single node immediately connected to the node in question in the direction of the root. The descendants of a node are the node's children plus their children plus their children's children etc. The ancestors of a node are the node's parent plus its parent's parent etc.

The source file tree is viewed with the root at the top, and descendants of the root on a level below the root. The identifiers declared in outermost declarations visible to a node are:

1. identifiers declared in children of the node
2. identifiers visible to the node's parent

Restating these rules in a nonrecursive manner:

1. identifiers declared in children of the node
2. identifiers declared in siblings of the node and the node itself
3. identifiers defined in ancestors of the node and their siblings

There may be several outermost declarations within one source file. These declarations are considered to be unordered, as are the children nodes of a node. Declarations at a lower level in the tree may hide declarations of the same identifier at a higher level. Within the set of siblings of any node, all non-overloaded declarations must have unique identifiers. Also, if an overloaded declaration of an identifier is present within a set of siblings, then all declarations of that identifier within that set must be overloaded declarations.

Figure 3.2 shows a pictorial representation of the scopes of a typical program. Figure 3.3 shows some overloading declarations, hiding declarations, and references in a **ForceOne** program and how they are resolved. A declaration of the identifier `i` is represented as `i: ...;` a reference to the identifier `i` is represented as `@i`.

This tree structure lends itself naturally to the definition of abstract data types. An abstract data type (ADT) consists of an object type plus a set of operations available on objects of that type. The type and the external operations applicable to the type are defined within one node. Any algorithms needed to implement the ADT are defined in descendant nodes. Since nodes outside of the tree rooted at the ADT node cannot see declarations within the ADT tree, only the external interface to the ADT is visible.

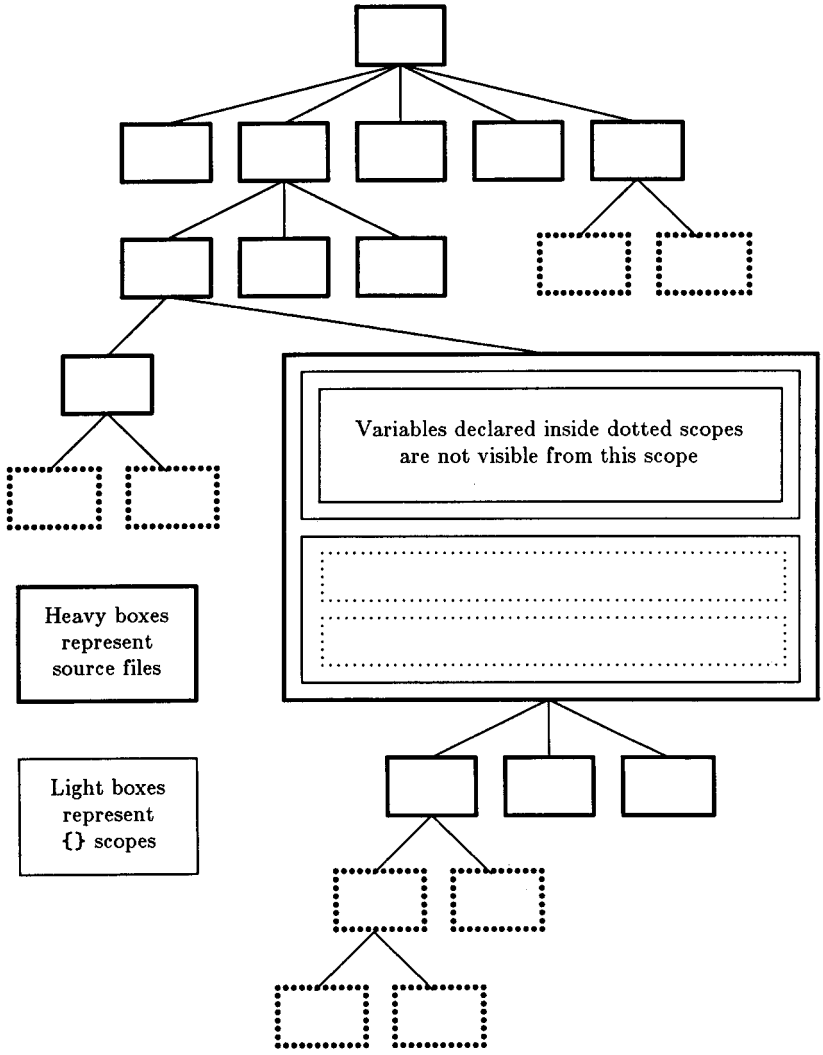

Figure 3.2: Visibility of Identifiers

```
\ Source file "main"
main: ...                   \ decl #1
i: ...                      \ decl #2


    \ Source file "main/m1"
    i: ...                  \ decl #3, hides #2
    {
        @i                  \ resolves to #5
        i: ...              \ decl #4, hides #3
        @i                  \ resolves to #4
    }


        \ Source file "main/m1/sm1"
        i: ...              \ decl #5, hides #3
        { @i }              \ resolves to #5


    \ Source file "main/m2"
    overload i: ...         \ decl #6, hides #2
    overload i: ...         \ decl #7, overloads #6
    {
        @i                  \ resolves to #6 or #7
        overload i: ...     \ decl #8, overloads #6 and #7
        i: ...              \ decl #9, hides #6, #7, and #8
    }


        \ Source file "main/m2/tm1"
        overload i: ...     \ decl #10, overloads #6 and #7
        { @i }              \ resolves to #6, #7, or #10
```

Figure 3.3: Resolution of Identifiers

Parameters of Type type

ForceOne allows the programmer to construct parameterized objects which accept parameters of type **type**. Since a **ForceOne** type does not include the operations available on that type, all the compiler has to pass to a parameterized routine is the type's size and alignment. The identifier naming the type parameter is declared as a new type. From that point on the new type functions just as any other type, although as yet it has no operations defined upon it.

```
alloc: [ t: type ] routine ref t == external "_malloc"
```

The example above shows a typical usage of a parameter of type **type**. The routine **alloc** accepts a type as a parameter and returns an object whose type is a reference to the argument type.

To perform an operation upon an object of this new type, the operation would have to be passed as a parameter as well. Figure 3.4 shows a memory allocator which zeros the allocated memory before returning it, and a typical call to this allocator.

```
\ A memory allocator which zeros memory
zalloc: [ t: type, zero: [ ref t ] routine void ]
        routine ref t == {
    x: ref t == alloc[ t ]
    zero[ x ]
    return[ x ]
}
\ A typical call to this memory allocator
ptr: ref ref int
zero: [ i: ref int ] routine void == { i := 0 }
ptr := zalloc[ int, zero ]
```

Figure 3.4: Parameter of type type

Query Parameters

Often it may be desirable to implement a routine which can accept an object of any type as a parameter and perform some operation upon it. By using parameters of type **type** this can be achieved, but by using query parameters one can avoid specifying the type of the object as an additional parameter.

```
toss: [ x: ?t ] routine void == ...
```

A query declaration is used to declare the type of the parameter in the above example. A typical call to this routine might be

```
a: ref int
!toss[ a ]
```

The routine **toss** has two parameters, but only one argument must be supplied when **toss** is called. The value of the argument is bound to **x**, and the type of the argument is bound to **t**. This declaration of **toss** achieves the same effect as the declaration

```
toss2: [ t: type, x: t ] routine void == ...
```

but a call to **toss2** would require an additional argument.

```
a: ref int
!toss2[ ref int, a ]
```

It is also possible to partially constrain the type of a polymorphic argument. Query parameters are used for only those parts of the type which are to be polymorphic. For example,

```
stuff[ x: ref ?t ] routine void == ...
```

stuff accepts only arguments of type **ref** to anything. When the type of an argument is to be constrained to be some parameterized type, extra ?s must be added to export the type parameter declaration to the appropriate level of [] nesting. Figure 3.5 shows the use of two ?'s to declare a routine which accepts any argument which is a routine accepting one parameter and returning void.

```
\ A memory allocator requiring a zero function
zalloc: [ zero: [ ref ??t ] routine void ]
        routine ref t == {
    x: ref t == alloc[ t ]
    zero[ x ]
    return[ x ]
}
\ A typical call to this allocator
ptr: ref ref int
zero: [ i: ref int ] routine void == { i := 0 }
ptr := zalloc[ zero ]
```

Figure 3.5: Query parameter

The query identifier is declared K–1 levels of brackets outward from its enclosing parameterization, where K is the number of ?s preceding the identifier.

The previous examples have all been examples of first order polymorphism. In general, a *Nth* order polymorphic routine is one which accepts as an argument a *N*–1 order polymorphic routine. The following example shows the use of a query declaration to declare a second order polymorphic routine. This routine accepts one argument which is a routine accepting one parameter of any type and returning that same type.

```
poly2: [ x: [ ?t ] routine t ] routine int
```

Automatic Parameters

Automatic parameters allow a routine to require certain operations be available from the environment in which it is used. The effect of an automatic parameter is just as if the user passed it explicitly. An automatic parameter consists of both a type and a name. The compiler searches the symbol table at the call site for a definition of the name with the appropriate type, and passes the object as any other parameter. In Figure 3.6, the routine `zero` is an automatic parameter and so must be available at the call site with the required type.

```
\ A zeroing allocator with a clean interface
zalloc: [ t: type || zero: [ ref t ] routine void ]
        routine ref t == {
    x: ref t == alloc[ t ]
    zero[ x ]
    return[ x ]
}
\ A typical call to this allocator
ptr: ref ref int
zero: [ i: ref int ] routine void == { i := 0 }
ptr := zalloc[ int ]
```

Figure 3.6: Automatic parameter

Automatic parameters are intended to be used with query declarations in implementing polymorphic abstract data types. A polymorphic routine requires specific operations to implement its function. For example, the following definition of `square` can square an argument of any type, provided the operation * is available for that type.

```
square: [ x: ?t || '*': [ t, t ] t ] t == {
    return[ x * x ]
}
```

Type Equivalence

In **ForceOne** a type is simply a representation for a set of values that is used by the compiler to ensure that values from different sets are not inappropriately mixed. A new type, distinct from all others in the program, may be created by an equivalence declaration from some other implementation type. The new type may be converted to its implementation type and vice versa by typemod expressions.

Structural equivalence is used up to a point to determine if two types are the same. For instance,

```
i: ref int
j: ref int
r: ref real
```

the variables `i` and `j` have the same type. The variables `i` and `r` have different types. Structure equivalence is not used to compare records or enumeration types.

```
a: record[ i: ref int ]
b: record[ i: ref int ]
```

The variables `a` and `b` in the above example do not have the same type. Another way of saying this is that the record type generator and the enumeration type generator yield new types, whereas the other type generators such as the reference type generator simply modify their argument type.

As mentioned earlier, two typemod operations, `detype` and `retype`, are available for use with new types created by an equivalence declaration. If the equivalence declaration is an outermost declaration, then these operations are available only to the source file containing the declaration and to its descendants. This supports the construction of abstract data types, because placing the type declaration in the node at the root of the ADT allows only routines implementing the ADT to have access to the internal representation of the type. Furthermore, since the detype and retype operations are explicitly provided, both the external and the internal representation of the type are available within the ADT.

Figure 3.7 is an implementation of a modulo 7 arithmetic package. It supports three operations, +, *, and `put`. The mainline routine uses it to add 6 and 5. The operation * is implemented in a particularly naive manner, however it shows the use of the external representation of the abstract data type.

```
\ Source file "main"
main: routine void == {
    put[ "6 plus 5 mod 7 is " ];
    put[ const7[ 6 ] + const7[ 5 ] ];
    put[ "\n" ]
}

\ Source file "main/mod7"
mod7: type == 0..6
const7: [ 0..6 ] mod7 == {{ retype[ 0 ], retype[ 1 ],
                             retype[ 2 ], retype[ 3 ],
                             retype[ 4 ], retype[ 5 ],
                             retype[ 6 ] }}

overload '+': [ i: mod7, j: mod7 ] routine mod7 == {
    retype[ (i + j) % 7 ]
}

overload '*': [ i: mod7, j: mod7 ] routine mod7 == {
    count: ref int
    sum: ref mod7

    sum := j
    loop[ count := i - 1, count > 0, --count ]
        sum := sum + j
    return[ sum ]
}

overload put: [ i: mod7 ] routine void == {
    put[ detype[ i ] ]
}
```

Figure 3.7: A modulo 7 arithmetic package

Overload and Coercion Selection

The selection of the proper meaning of an overloaded identifier and the insertion of coercions where necessary are closely related problems. The following example shows why.

```
widen int_to_real: [ int ] routine real == ...
overload v: ref int
overload v: ref real
r: ref real
r := @v
```

Either definition of `v` can be selected for the assignment statement. If the `ref int` definition of `v` is selected then the coercion `int_to_real` must be applied. **ForceOne** uses a minimum cost algorithm to disambiguate such expressions.

1. Coercion Selection Rules

User defined coercions are classified into *widening* and *narrowing* coercions. Defining a coercion as a narrowing is intended to mean that the coercion discards information. For example, converting from `real` to `int` would be a narrowing coercion as the fractional part of the number must be discarded. A widening coercion is just the opposite; it does not lose information. Following are the rules used to select coercions and overloaded definitions for an expression. The rules are listed in decreasing order of importance.

1. The number of narrowings is minimized.
2. Widening each parameter of an instance selection is cheaper than widening the result.
3. The number of widenings is minimized.
4. Narrowings are done as late as possible.
5. Widenings are done as early as possible.
6. Only one user defined coercion will be inserted at any point in the tree.

Rules 1 and 3 are fairly clear. Because the rules are listed in order, the number of narrowings is first minimized, then amongst all solutions with that minimum number of narrowings the number of widenings is minimized. Saving one narrowing is more important than any number of widenings. Rule 2 says that if a widening can be applied either before or after an explicit operation, the widening before the operation will be chosen. Rule 4 ensures that when the result type of an operation is narrower than the operand types, the operation will be carried out in the wider operand type. Rule 5 ensures that if an operation may be carried out in two or more types, the widest type will be chosen to minimize information loss. Rule 6 is necessary primarily due to implementation considerations.

Consider the following situation.

```
widen  int_to_real: [ int ] routine real == ...
narrow real_to_int: [ real ] routine int == ...
overload '+': [ int, int ] routine int == ...
overload '+': [ real, real ] routine real == ...
i: ref int

i := 1 + 2.3
```

The compiler has two choices: it can coerce the `1` to `real`, use the `real +`, and narrow the result so that it can be assigned to `i`; or, it can coerce the `2.3` to `int` and use the `int +`. Since the number of narrowings is the same for both choices, rule 3 governs and the compiler takes the first choice.

```
i := !real_to_int[ !int_to_real[ 1 ] + 2.3 ]
```

Since automatic dereferencing and calling are not considered user defined coercions, any number of dereferences and calls followed by one user defined coercion may be applied to an expression. When a type declaration is made, a user defined coercion which detypes an object of the external type to one of the internal type is created. If the programmer defines his own coercion from the external type to the internal type this coercion overrides the automatically defined one. Of course, the effect of the automatically defined one is still available through the explicit detype expression.

There are two errors which may occur during overload resolution. The compiler may be unable to select any set of coercions and overloaded symbols to resolve some expression, or it may find several solutions yielding the same minimum cost. These errors may be caused by several situations. Most often a simple typographical or logical error in the expression is the cause; however, it is also possible for a poorly defined environment to render seemingly innocuous expressions ambiguous.

```
widen    int_to_real: [ int ] routine real == ...
overload '+': [ int, real ] routine real == ...
overload '+': [ real, int ] routine real == ...
r: ref real

r := 1 + 2
```

In the example above, the compiler will be unable to choose between the two minimum cost solutions shown below.

```
r := !int_to_real[ 1 ] + 2
r := 1 + !int_to_real[ 2 ]
```

There is a command line option on the compiler which requests that the selected coercions and overloaded definitions be displayed.

2. Weak Contexts

Many language constructs can be resolved without any contextual information. Non-overloaded identifiers and constants are simple examples of such constructs; their type can be determined independent of context. These constructs are known as *context independent* constructs. Other constructs, such as overloaded identifiers, require contextual information to be resolved. These constructs are known as *context dependent* constructs. A context which provides a finite set of possible types is known as a *strong* context. A context which provides no information is known as a *weak* context. The selector expression of a `select` expression is an example of a weak context. Context dependent constructs may not appear in

weak contexts. Appendix D lists the context dependent constructs and weak contexts.

One specific instance of the weak context problem is the balancing problem, introduced first by Algol 68 [Wij 76]. Certain language constructs return a type which is the same as their arguments, although no one argument governs the choice of this type. The inline if expression is the best example of this. Both the if expression and the else expression must be resolved to the same target type, which is also the type returned by the whole expression. Such an expression cannot be used in a weak context, such as the following case.

```
push: [ x: ?t ] routine void == ...
push[ if[ a ] 1 else 2.3 ]
```

Chapter 4

Compilation of ForceOne

ForceOne is a separately compilable language, designed to be compiled into standard reusable object units. One object unit is created for each source file compiled. These object units may be linked by the host system's linker to generate a standard code image. Object units generated by other languages may also be linked into **ForceOne** programs.

The Automatic Compilation Mechanism

When changes are made to a program under construction, and that program is recompiled, the effect of the compilation must be as if all previous work done by the compiler were discarded and the entire program recompiled from its source. However, often only a small part of the program is changed, hence recompiling the entire program would result in a lot of duplicated effort. A separate compilation system allows the program to be divided up into many pieces, so that each piece can be compiled separately. Only those pieces which have been changed and those pieces affected by the changes must be compiled in order to recompile the program.

Many contemporary languages such as C provide separate compilation, but since the order of compilation of source files is the programmer's responsibility, bugs can be introduced in the program when source files are not compiled in the correct order. The **ForceOne** compiler determines the order of compilation *automatically* and provides no means for the programmer to specify how a program should be compiled, thereby guaranteeing that no program can be incorrectly compiled. To compile a **ForceOne** program, one simply issues the compile command giving the root directory of the program to be compiled as an argument to the command. The compiler will inform the user of any errors, and if there are no errors an object unit will be created suitable for input to the host's linker.

The *automatic compilation system* integrated into the prototype **ForceOne** compiler does separate compilation on a node, or source file, basis. It determines which nodes are new, which have been changed, and which have been deleted, and classifies the new and changed nodes into one of three states. This state represents the compilation phases which must be applied to the node to produce a properly compiled program. The compiler determines the state of each node by comparing the new symbol table for the program with the old symbol table from the previous compilation, and finding changes in the types of declared objects.

The three states are: *unchanged*, *requires_parsing*, or *requires_resolution*. A production compiler would add a fourth state: *requires_code_generation*. *Parsing* refers to the process of reading the source file, performing lexical analysis, parsing the file according to the language grammar, and generating a parse tree for the node. *Resolution* refers to the process of traversing a parse tree and determining the resolution of overloaded symbols and where coercions should be placed in the tree. *Code Generation* refers to the process of using the resolved parse tree to generate machine code representing the file. A

node which *requires_parsing* necessarily *requires_resolution*, and a node which *requires_resolution* necessarily *requires_code_generation*.

The prototype **ForceOne** compiler uses five phases to compile a program. All phases are always entered, although the amount of work done during a particular phase may vary depending on the extent of changes to the source since the last recompilation. In each phase an algorithm is performed upon each node of the program tree whose state indicates that work needs to be done in this phase. The program tree is traversed in prefix, infix, or postfix order, depending upon the phase, although the order does not matter in some phases.

1. Change Detection

The first phase entered by the compiler is the *Change Detection* phase. During this phase the compiler first reads a symbol table database saved by the last compilation of the program. This database contains the symbol table for the entire program and the modification dates of the source files as of the last compilation. The compiler then traverses the program tree as it is stored in the file system by making the appropriate operating system calls.† By comparing the file system tree and the modification dates of source files with the database, the compiler can determine what nodes have been changed, what nodes are new, and what nodes have been deleted. The file system is traversed in infix order, as this ordering is most conducive to the directory traversal interfaces provided by most operating systems.

New and changed nodes are marked as *requires_parsing*, and will be parsed during the next phase. Deleted nodes may only affect the resolution of symbols in their parent node due to the visibility rules, and so are handled by marking their parent as *requires_resolution*.

2. Lexical Analysis, Parsing, and Type Synthesis

The second phase the compiler enters is the *Lexical Analysis, Parsing, and Type Synthesis* phase. During this phase, the compiler traverses the program tree in postfix order, parsing those files marked as *requires_parsing*, and synthesizing the types of outermost declarations.

Within a routine text, forward references to declarations within that routine text are not allowed, so the types of all symbols declared can be determined easily. However, for outermost declarations, the situation is more complex. If the equivalence declaration

```
a: type == b
```

meant that `a` and `b` were exactly the same type, a closure algorithm over the whole program tree would be necessary to determine the types of all objects. However, the type equivalence rules were chosen so that except within the declaring node and its descendants, the new type is completely different from its representation type. When the additional rule is added that type symbols may not be overloaded, this allows the compiler to determine the type of every non-overloaded symbol in the program in one postfix (or synthesis) pass. Figure 4.1 shows the type synthesis algorithm.

Although the prototype compiler does not implement it, the detection of cycles in types can also be done during synthesis of types.

† This is the only part of the compiler that is operating system specific.

```
Iterate bottom-up over nodes N

  If N is marked requires_parsing
    simultaneously perform lexical analysis and parsing of N

  check that no declarations of children of N conflict

  Iterate top-down over children M of N

    For each child P of M
      resolve type references of M with definitions of P
```

Figure 4.1: The type synthesis algorithm

3. Type Change Propagation

After the types of all objects in the program have been determined, the compiler enters the *Type Change Propagation* phase. The compiler traverses the program tree in a prefix (or inheritance) pass to determine which unmodified nodes are nevertheless affected by changes in the types of objects they reference. Since a change in the type of a symbol can affect how references in other nodes are resolved, a closure algorithm is necessary to propagate such changes to dependent nodes. However, the visibility and type equivalence rules allow a single inheritance pass with closure only over sets of sibling nodes to completely propagate these changes. Figure 4.2 shows the change propagation algorithm.

```
Iterate top-down over sibling sets SS

  For all nodes N in SS marked requires_parsing
    compare declarations of N against old declarations of N,
    marking those which are different changed

  Iterate until convergence
    If an unchanged reference of node N of SS
          depends on a changed declaration of any node
      mark reference changed
      mark N requires_resolution
    If an unchanged declaration of node N of SS
          depends on a changed reference of any node
      mark declaration changed
      mark N requires_resolution

  For all references of the parent of SS
    If the reference is not overloaded
      If the reference refers to a declaration which
            is changed and is not a type declaration
        mark parent requires_resolution
    Else
      If any visible declaration matching the reference is changed
        mark parent requires_resolution
```

Figure 4.2: The type change propagation algorithm

For each set of sibling nodes, the outermost declarations of those nodes marked *requires_parsing* are sorted by name and compared with the old declarations from the previous compilation. Those declarations whose type is different are marked *changed*. The closure over the set of sibling nodes of those declarations which depend upon *changed* declarations is then computed. Each reference to a *changed* declaration is marked *changed*, and each declaration which refers to a *changed* reference is marked *changed*. When the closure algorithm is complete, any nodes containing a *changed* declaration are marked *requires_resolution*. Finally, if the parent node references any *changed* declarations, the parent node is marked *requires_resolution*.

When an overloaded declaration is marked *changed* by the above algorithm, any node containing a reference which might resolve to that declaration must be reresolved. This is handled by checking all visible declarations which could match the reference and marking the node *requires_resolution* if any of the declarations is *changed*.

When a coercion is marked *changed*, the number of nodes which might be affected by such a change is very large. Cormack [Cor 81] proposes some algorithms for determining more precisely which nodes may be affected, however the prototype **ForceOne** compiler simply assumes that each node in which the coercion is visible is affected. Every descendant of the parent node of a node containing a *changed* coercion is marked *requires_resolution*.

4. Overload Resolution

The *Overload Resolution* phase resolves overloaded symbols and inserts coercions in the parse tree where required by context. A two pass recursive algorithm is executed for each node which is marked *requires_resolution*. This algorithm uses a cost function to select among the many possible interpretations of the parse tree which may result when user defined coercions and overloading are used. Effectively, it assigns a cost to every possible interpretation, and selects the interpretation with minimum cost. If two interpretations exist with the same minimum cost, the parse tree is ambiguous, and an error message is issued. Similarly an error is issued if no interpretation can be found for the parse tree.

The function `selbest` is the recursive routine which implements the heart of the overload resolution algorithm. This routine is called upon a parse tree with a maximum cost, and returns the number of least cost solutions and their cost where this cost is less than the specified maximum. Optionally the routine may mark the least cost solution it finds in the tree. This routine is first called upon a specific parse tree with an infinite maximum cost to determine the cost of the least cost solution for that parse tree. The solution is not marked; only its cost is determined. If there is a unique least cost solution, the routine is called again upon the parse tree with the maximum cost being the least cost found earlier. During this pass the algorithm determines the least cost solution again and marks it in the parse tree. This pass is necessary as in order to mark each node with the correct coercion and overloading interpretation, each node must be last visited with the appropriate target type required for the least cost solution. Figure 4.3 shows the routine `selexpr` which is the driver of `selbest`.

```
selexpr( target_type, tree )

select( n_solns, cost := selbest( target_type, tree, infinity, false ) )
   case 0:
      put( "Expression has no solution" )

   case 1:
      selbest( target_type, tree, cost, true )

   otherwise:
      put( "Expression is ambiguous" )
```

Figure 4.3: The overload algorithm: selexpr

The routine `selbest`, as shown in Figure 4.4 has only three cases: leaves, identifiers, and instance selection. In reality there is a different case for each different kind of parse tree node, however the case for each of the other kinds of nodes is very similar to one of these three cases.

Leaves are the simplest case. If the type of the leaf coerces to the target type, there exists one unique solution for this subtree, whose cost is the cost of that coercion. If this cost is less than or equal to the required maximum, this cost is returned.

```
selbest( target_type, tree, max_cost, mark )
      returns n_solns, best_cost

n_solns := 0

select( kind of tree )
   case Leaf:
      if( (cost := coercible( type of leaf, target_type )) < infinity )
         if( cost <= max_cost )
            if( cost = max_cost ) n_solns += 1
            else max_cost := cost, n_solns := 1
            if( mark )
               save coercion in the tree
      return( n_solns, max_cost )

   case Identifier:
      for( each identifier of this name visible here )
         if( (cost := coercible( type of identifier, target_type )) < infinity )
            if( cost <= max_cost )
               if( cost = max_cost ) n_solns += 1
               else max_cost := cost, n_solns := 1
               if( mark )
                  save coercion in the tree
                  mark identifier in the tree
      return( n_solns, max_cost )

   case Instance_selection:
      while( (n_isolns, cost, type := selnext( tree.expr, max_cost, false )) > 0 )
         if( type is || and tree.n_actuals = type.n_actuals )
            for( each type.formal and tree.actual )
               (n_isolns *=, cost +=)
                     selbest( type.formal, tree.actual, max_cost-cost, false )
            if( (cost += coercible( type.result, target_type )) < infinity )
               if( cost <= max_cost )
                  if( cost = max_cost ) n_solns += 1
                  else max_cost := cost, n_solns := 1
                  if( mark )
                     mark coercion of result in tree
                     selnext( tree.proc_expr, max_cost, true )
                     for( each type.formal and tree.actual )
                        selbest( type.formal, tree.actual, max_cost-cost, true )
      return( n_solns, max_cost )
```

Figure 4.4: The overload algorithm: selbest

Identifiers are not treated the same as leaves because they may overload. The identifier whose type coerces with least cost to the target type is selected as the minimum cost solution.

Instance selection is the hardest case. The cost of selecting any given interpretation of a lambda expression at this node is the sum of the cost for the lambda expression's subtree plus the cost of matching each actual parameter to its corresponding formal parameter. The minimum cost solution may not necessarily involve the minimum cost solution for the lambda expression. The algorithm must match all possible interpretations of the lambda expression with the actual parameters in order to determine the minimum cost solution. The routine `selnext`, shown in figure Figure 4.5 is called upon the lambda expression to obtain a possible interpretation of the subtree. `selnext` parallels `selbest` in structure, but operates like a coroutine, returning one possible interpretation of the subtree each time it is called. †

† The Instance_selection case of selbest as shown here does not include propagation of ref across parameterization.

```
selnext( tree, max_cost, mark )
      returns n_solns, cost, type

select( kind of tree )
   case Leaf:
      return( 1, 0, type of leaf )

   case Identifier:
      if( tree.current = nil )
         tree.current := first possible definition
      else
         tree.current := next possible definition past tree.current
      return( tree.current != nil, 0, type of tree.current )

   case Instance_selection:
      while( (n_solns, cost, type := selnext( tree.expr, max_cost, mark )) > 0 )
         if( type is [] and tree.n_actuals = type.n_actuals )
            for( each type.formal and tree.actual )
               (n_solns *=, cost +=)
                     selbest( type.formal, tree.actual, max_cost-cost, mark )
            if( n_solns > 0 )
               return( n_solns, cost, type )
      return( 0, infinity, nil )
```

Figure 4.5: The overload algorithm: selnext

The routine `coercible`, shown in figure Figure 4.6 is called by `selbest` to determine if a type is coercible to another type.

```
coercible( type, target_type )
      returns cost

if( type matches target_type )
   return( 0 )

if( type widens to target_type )
   return( number of nodes in tree currently being examined by selbest )

if( type narrows to target_type )
   return( current depth of nested selbest calls )

return( infinity )
```

Figure 4.6: The overload algorithm: coercible

If the two types are the same type, then they are clearly coercible with zero cost. If the second type is a query declaration, then they are coercible with zero cost, and the actual type matched to the query declaration is stored in the symbol table. Otherwise, as many `refs` and `routines` are stripped from the first type as possible, and a search is made of the symbol table to find a coercion from this type to the desired type. If a widening is found, then the two types coerce with cost equal to the number of subnodes of the current parse tree node. If a narrowing is found, then the two types coerce with cost equal to the depth of the current parse tree node.

A more abstract description of the overload resolution algorithm may be found in [Cor 86a].

5. Code Generation

The output from the overload resolution phase is one parse tree for each source file, decorated with the type yielded by each node of the tree. The *Code Generation* phase converts this set of parse trees into a set of object units suitable for linking. A code generator for **ForceOne** needs little more sophistication than a code generator for Pascal; the only additional capability required is the ability to generate code for objects of varying size. Algorithms for generating such code are simple and already well known [Aho 86]. The code generator in the prototype compiler is incomplete and unexceptional; it will be discussed in little detail.

The prototype compiler simply traverses every tree of the program and emits one Vax 11 assembly source file which is assembled to produce one object unit representing the program. A production compiler would attempt to generate code for only those source files necessary, and would also generate machine code directly. For each source file marked *requires_code_generation*, the code generator would determine the sizes of all types used in the parse tree, and would determine by comparison with the previous sizes which types have changed in size from the previous compilation. In a similar manner to the way in which the type change propagation phase propagates changes in types, those parse trees which depend upon these sizes would be marked *requires_code_generation*. Then each tree whose source file is marked *requires_code_generation* would be traversed to generate one object unit for each source file.

Memory for variables is allocated from a stack, with static links used to maintain the display for each procedure. All objects and temporaries have fixed offsets in the stack frame; variable size objects are represented by a fixed size pointer pointing to a variable size bucket at the end of each stack frame. Symbols at the outermost level are treated as globals. Figure 4.7 shows the stack frame layout.

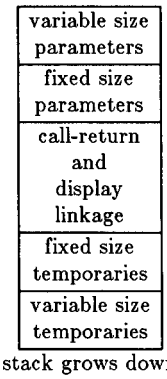

Figure 4.7: Stack frame layout

The Prototype Compiler

All of **ForceOne** has been implemented to some degree in a prototype compiler. All constructs are parsed, and almost all constructs are handled by the overload resolution pass. Code generation, however, has only been implemented for some of the more common constructs due to a lack of time. Completing the prototype compiler would not be difficult; those parts not yet implemented have been sketched out. This implementation has been very useful in highlighting and clarifying several issues and problems in the language design which might never have been discovered if **ForceOne** had simply been designed on paper.

Figure 4.8 outlines the source organization of the prototype compiler. The source comprises about 11,200 lines of C code in total.

codegen code generation 2300 lines	**debug** debugging 500 lines	**error** error reporting 300 lines	**filesys** file system walking 300 lines
findchg change detection 100 lines	**flatten** symbol database IO 1500 lines	**h** include files 700 lines	**lex** lexical analysis 700 lines
misc miscellaneous 500 lines	**node** source file traversal 100 lines	**overload** overload resolution 1700 lines	**parse** parsing 1300 lines
propchg change propagation 500 lines	**symt** symbol tables 500 lines	**typeres** type synthesis 200 lines	

Figure 4.8: Source code organization

Chapter 5

Future Directions

Problems with ForceOne

ForceOne has been an excellent tool for finding problems in the specification and implementation of languages of its class. **ForceOne** exhibits enough problems that of itself it would not make a very good programming language; however, it seems fairly clear that when some of these problems have been resolved a derivative of **ForceOne** could be built which would be much more expressive than many contemporary programming languages.

Failure of the overload algorithm due to insufficient context is clearly undesirable. There will always be weak contexts so long as we have polymorphic procedures. Therefore, the only way we can eliminate this failure is by eliminating context dependent constructs. This may not be feasible either; however, it may be possible to arrange that very seldom in ordinary coding would context dependent constructs appear in weak contexts. The introduction of a `var` type generator may help. This type generator is very similar to the `ref` type generator, but subsumes the role `ref` currently plays in allocating variables. Automatic dereferencing is replaced with automatic *devaring*. The cost of devaring once is $-\epsilon$, where ϵ is very small. There exists a coercion from `ref X` to `var X` of zero cost which may be applied automatically by the compiler. `ref` propagation across parameterization is no longer necessary. In declarations, for identifiers of type `ref X` there may be a value supplied with ==. For identifiers of type `var X` there must be no value supplied. We can now write

```
i: var int
select[ i ]
    ...
```

rather than

```
i: ref int
select[ @i ]
    ...
```

In retrospect, the rule that routine texts yielding `void` are automatically called should be deleted. Instead, parentheses should accept an expression list:

expr ≡ (*expr_list*)

where the last expression is the value returned and all others must yield `void`. Each expression in a routine text, including the last one, yields `void`; falling off the end of a routine text is illegal. The return expression causes a return from the innermost routine text. We now use `()` where `{}` yielding void was used before.

```
f: routine int == {
    if[ something ] (
        ...
        ...
        return[ 1 ]
    )
    else
        return[ 0 ]
}
```

The actions of hiding symbols, creating new symbols, and overloading existing symbols, currently controlled by the keyword `overload`, should be separated. In some cases it may be desirable to hide a particular symbol or coercion, without replacing it with anything. It may also be desirable to replace a particular overloaded declaration with a different one, without affecting the other definitions of that symbol.

The order in which the outermost declarations of a program should be elaborated before the mainline procedure for that program is called is not obvious. The prototype compiler elaborates declarations in a top down manner over the program tree. Within a source file, declarations are elaborated in the order found. A better solution would likely be to determine a partial ordering of dependencies and elaborate declarations according to this partial ordering; if such an ordering could not be found then an error would be reported.

The semantics for user defined type generators can be clearly and easily defined without any extensions to the language syntax, however the current implementation of the **ForceOne** compiler does not support these. Parameter substitution for a user defined type generator can be performed at parsing time and so does not affect the rest of the compiler. However, this requires the rule that a file cannot use type generators defined in sibling or descendant files, as all type generators used by a source file must be available before that source file can be parsed. This rule is similar to the rules for type equivalence, and so should not seem too restrictive. An example of a user defined type generator for a linked list follows.

```
linked_list: [t: type] type == record[
    next: linked_list[ t ]
    data: t
]
```

ForceOne allows only a single identifier to be declared with one declaration. This should probably be extended to allow declarations of the form

```
i, j, k: ref int
```

meaning precisely

```
i: ref int
j: ref int
k: ref int
```

If this is allowed for equivalence declarations and the right hand side of the == contains side effects, these must be performed as many times as there are identifiers being declared.

In retrospect, reserving `widen` and `narrow` as keywords is unnecessary. Since the name of the routine defined as a coercion is unnecessary, coercions should be defined by simply overloading the special names `widen` and `narrow`:

```
overload widen:  [ int  ] routine real == ...
overload narrow: [ real ] routine int  == ...
```

Like **widen** and **narrow**, it seems obvious in retrospect that **retype** and **detype** need not be reserved as keywords. A type declaration should automatically create the two procedures

```
overload detype: [ new_type ] base_type
overload retype: [ base_type ] new_type
```

and the coercion

```
overload widen: [ new_type ] base_type == detype
```

Additional Features

ForceOne is not a complete language; there are several features which should be included in the language and many others which seem desirable. Some of these features have already been mentioned; a few more are mentioned here.

Some mechanism to allow extensible syntax should be provided. This mechanism should be sufficiently powerful to allow the definition of an inline `select` expression, so that the present control structures of **ForceOne** can be removed. Also the user should be able to define his own operators, and specify their precedence and associativity. This would allow the deletion of the fixed set of operators currently built in to **ForceOne**. This mechanism might also allow the definition of procedures that can accept an arbitrary number of parameters.

No mechanism for exception handling has been included in **ForceOne**. Language support for exception handling is required in order to allow the dynamic binding of a raised exception with its handler. Some form of `non_local_return` may be sufficient to allow the flow of control to return through several levels when an exception occurs. The exceptions raised by a particular procedure are declared as automatic procedure parameters of that procedure. A handler for a particular exception is declared by declaring a procedure of that name. An exception is raised by invoking the exception name.

```
add: [ a: int, b: int || overflow: routine exception ] == {
    if[ sign[a] = sign[b] && sign[a] != sign[a + b] ]
        !overflow                \ raise overflow exception
    else
        return[ a + b ]
}

overflow: routine exception == {
    put[ "overflow\n" ]          \ handle the exception
    non_local_return[ handling_context ]
}
```

The problem remaining is to determine some means of binding an exception handler to its handling context, which as the example shows is the same problem as determining the target of the `non_local_return`.

Some mechanism should exist to allow access to the program state, in order to implement coroutines and other process control mechanisms. Given a mechanism which allows the saving and restoring of the program state, implementing a coroutine package or cooperating sequential process package in **ForceOne** should be straightforward.

No compiler or language support is provided for libraries or version control. It should be possible to define libraries of routines to be used by many programs, and to make compatible modifications to such libraries without requiring the recomilation of all programs using the libraries. There should be a standard library available to all programs, and users should be able to define their own private libraries.

A production **ForceOne** compiler should generate inline procedure calls wherever possible. Since **ForceOne** programs will use many small polymorphic routines, the generation of inline code for some of the calls to these routines will greatly improve execution efficiency. Here by inline procedure call we mean the generation of inline code and the substitution of arguments for formals in order to avoid

building a new stack frame. Since **ForceOne** is a type secure language, more information is available to the compiler to help it determine when arguments can be substituted for formals without modifying the semantics of the program being compiled.

Contributions of this Thesis

ForceOne is the first attempt at defining a production language encorporating most of the features introduced in the earlier languages L [Cor 81, Cor 83], *L* [Lec 84], and M [Jud 85]. L was mainly a presentation of a set of loosely related language features and algorithms for compiling them. Leclerc's *L* was a more integrated language, but lacked many of the more interesting language features found in **ForceOne**, such as query parameters. M was primarily an investigation into more unconventional language features.

The **ForceOne** compiler is only the second attempted compiler for languages of this class. The first was Leclerc's *L*, however this compiler did not implement many interesting language features as *L* itself did not contain them. **ForceOne** implements overloading, coercions, parameters of type type, query parameters, automatic parameters, and a hierarchical file structure all within the framework of a separate compilation system.

ForceOne contains several new features not present in L, *L*, or M. The file structure of **ForceOne**, which provides a clean module structure while eliminating import and export lists, was inspired in part by Thoth [Car 79]. The realization that a polymorphic procedure has a type and that more than one ? in a query parameter is necessary to describe procedures accepting polymorphic procedures as arguments is new to **ForceOne**. The style of records provided by **ForceOne** with both alternation and concatenation of fields is not used by any popular language.

Chapter three of this thesis is the first complete definition describing both the syntax and semantics of a language of this class.

Conclusion

ForceOne is smaller and more expressive than most contemporary programming languages. Its use should come as a natural extension to programmers already familiar with such languages. Through its greater extensibility **ForceOne** adapts easily to diverse applications and programming methodologies. **ForceOne** facilitates object oriented programming and the creation of reusable libraries of very general purpose software. It is hoped that this presentation of **ForceOne** and the concepts it encapsulates will provide the fabric from which a still simpler and more expressive language will emerge.

References

[Ada 83]
Reference Manual for the Programming Language Ada, U.S. Department of Defense, ANSI/MIL-STD-1815-A (1983)

[Aho 86]
Aho, A.V., Sethi, R. and Ullman J.D., *Compilers*, Addison-Wesley (1986)

[Boe 86]
Boehm, H. and Demers, A., *Implementing Russell*, A.C.M. Sigplan Not. 21:7 (1986), 186-195.

[Car 79]
Cargill, T.A., *A View of Source Text for Diversely Configurable Software*, Univ. Waterloo CS-79-28 (1979)

[Car 85]
Cardelli, L. and Wegner, P., *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys, Vol. 17, No. 4, December 1985

[Cor 81]
Cormack, G.V., *Separate Compilation and New Language Features*, Ph.D. thesis, University of Manitoba (1981)

[Cor 83]
Cormack, G.V., *Extensions to Static Scoping*, A.C.M. Sigplan Not. 18:6 (1983), 187-191.

[Cor 85]
Cormack, G.V., *Zephyr*, unpublished (1985)

[Cor 86a]
Cormack, G.V., and Wright, A.K., *Polymorphism in a Compiled Language*, Univ. Waterloo CS-86-27 (1986)

[Cor 86b]
Cormack, G.V., Judd, M. and Wright, A.K., *Types are not Classes*, Univ. Waterloo CS-86-28 (1986)

[Cor 87]
Cormack, G.V., and Wright, A.K., *Polymorphism in the Compiled Language ForceOne*, Proc. 20th Hawaii Intnl. Conf. Syst. Sci. (Jan 1987)

[Dah 70]
Dahl, O.J., Myhrhaug, B. and Nygaard, K., *The Simula 67 Common Base Language*, Norwegian Computing Centre S-22 (1970)

[Don 85]
Donahue, J. and Demers, A., *Data Types are Values*, A.C.M. Trans. Prog. Lang. Syst. 7:3 (1985), 426-445.

[Gol 83]
Goldberg, A. and Robson, D., *Smalltalk-80, the Language and its Implementation*, Addison-Wesley (1983)

[Jud 85]
Judd, M., *A View of Types and Parameterization in Programming Languages*, M.Sc. thesis, McGill University (1985)

[Ker 78]
Kernighan, B.W. and Ritchie, D.M., *The C Programming Language*, Prentice-Hall (1978)

[Lec 84]
Leclerc, D., *Implementation Considerations for the Programming Language L*, M.Sc. thesis, McGill University (1984)

[Lis 77]
Liskov, B.H., Snyder, A., Atkinson, R. and Schaffert, C., *Abstraction Mechanisms in Clu*, Commun. A.C.M. 20:8 (1977), 564-576.

[Mat 85]
Matthews, D., *Poly and Standard ML*, A.C.M. Sigplan Not. 20:9 (1985), 52-76.

[Mil 78]
Milner, R., *A Theory of Type Polymorphism in Programming*, J. Computer Syst. Sci. 17 (1978), 348-375.

[Mil 85]
Milner, R., *The Standard ML Core Language*, Polymorphism 2:2 (Oct. 1985), 1-28.

[Myl 80]
Mylopoulos, J., Bernstein P.A. and Wong, K.T., *A Language Facility for Designing Database-Intensive Applications*, A.C.M. Trans. Prog. Lang. Syst. 5:2 (1980), 185-207.

[Pas 80]
Specification for the Computer Programming Language Pascal, International Standards Organization, DP-7185 (1980)

[PL1 76]
O.S. PL/1 Checkout and Optimizing Compilers: Language Reference Manual, IBM GC33-0009 (1976)

[Sit 72]
Sites, R.L., *Algol W Reference Manual*, Stanford University STAN-CS-71-230 (1972)

[Str 86]
Stroustrup, B., *The C++ Programming Language*, Addison-Wesley (1986)

[Weg 74]
Wegbreit, B., *The Treatment of Data Types in EL1*, Commun. A.C.M. 17:5 (1974)

[Wij 76]
van Wijngaarden, A., et al., *Revised Report on the Algorithmic Language Algol 68*, Springer Verlag (1976)

[Wir 82]
Wirth, N., *Programming in Modula-2*(second edition), Springer-Verlag (1982)

Appendix A

Reserved Keywords

The following keywords are reserved for use in certain syntactic constructs and may never be used as identifiers.

```
case    detype   else     enumerate   external
if      loop     narrow   overload    record
ref     return   retype   routine     select
type    void     widen
```

Appendix B

Escape Sequences

Following is a list of escape sequences recognized within identifiers, character constants, and string constants.

\a	alert or bell
\b	backspace
\f	formfeed
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\'	single quote
\"	double quote
\`	grave accent
\\	backslash
\$nn	numeric constant

In the last form, nn must be a one or two digit hexadecimal number representing the value of the character.

Appendix C
Operators

Following is a table of the builtin operators. The operators are listed in increasing order of precedence. Operators listed on the same line have the same precedence, except for the assignment operators which all have the same precedence. Only the middle group of operators may be defined by the programmer; the others are language operators and are built in to the compiler.

Associativity	Operator						
right	`if`		`loop`		`select`		
right	`else`		`case`				
right	`==`						
none	`..`						
right	`:=`	`+=`	`-=`	`*=`	`/=`	`%=`	
	`&=`	`^=`	`|=`	`<<=`	`>>=`		
left	`|`						
left	`^`						
left	`&`						
left	`=`	`#`					
left	`<`	`<=`	`>`	`>=`			
left	`<<`	`>>`					
left	`+`	`-`					
left	`*`	`/`	`%`				
right	`**`						
right	`+`	`-`	`++`	`--`	`~`	(unary)	
right	`@`	`!`					
right	`ref`		`routine`		`[]`		

Appendix D

Context Dependant Constructs and Weak Contexts

The following constructs are context dependant and may not be used in a weak context.

retype_expr
routine_text
aggregate
string_const
inline *if_expr*

The following constructs generate weak contexts.

`the true and false expressions of an` *if_expr*
`the selector expression of a` *select_expr*
query_decl `formal parameters`

Appendix E
Grammar

Following is the **ForceOne** grammar in a condensed form. The reader should beware that there are semantic restrictions which actually reduce the accepted language. The symbol ␣ means the ascii blank character. The symbol *source_file* is the start symbol for the grammar.

source_file	≡	*expr_sequence*
expr_sequence	≡	$\left[\; expr \; \left[\; ; \; \right] \; \right]^{+}$
expr_list	≡	$expr \; \left[\; , \; expr \; \right]^{*}$
expr	≡	*basic_expr*
	≡	*type_expr*
	≡	*declaration*
	≡	*routine_text*
	≡	*active_expr*
	≡	*control_expr*
	≡	*aggregate*
	≡	*cast*
	≡	*typemod_expr*
	≡	*void_expr*
	≡	*op_expr*
	≡	(*expr*)
basic_expr	≡	*identifier*
	≡	*constant*

type_expr ≡ *ref_tgen*

≡ *routine_tgen*

≡ *param_tgen*

≡ *enum_tgen*

≡ *record_tgen*

≡ *type_tgen*

≡ *subrange_tgen*

≡ *query_decl*

≡ *identifier*

ref_tgen ≡ `ref` *type_expr*

routine_tgen ≡ `routine` [*type_expr* | `void`]

param_tgen ≡ `[` *expr_list* [`||` *expr_list*] `]` *type_expr*

enum_tgen ≡ `enumerate [` *ident_list* `]`

ident_list ≡ *identifier* [`,` *identifier*]*

record_tgen ≡ `record [` *field_list* [`|` *field_list*]* `]`

field_list ≡ [*field*]+

field ≡ `[` *field_list* [`|` *field_list*]* `]`

≡ *storage_decl* [`;`]

type_tgen ≡ `type`

subrange_tgen ≡ *expr* `..` *expr*

declaration ≡ *equiv_decl*

≡ *storage_decl*

equiv_decl ≡ [*decl_type*] *identifier* : *type_expr* == *expr*

≡ [*decl_type*] *identifier* : *type_expr*
== `external` *string_const*

storage_decl ≡ [*decl_type*] *identifier* : *type_expr*

decl_type ≡ `overload` | `widen` | `narrow`

query_decl ≡ [`?`]$^{+}$ *identifier*

routine_text ≡ `{` *expr_sequence* `}`

active_expr ≡ *call*

≡ *deref*

≡ *instance_select*

≡ *field_query*

call ≡ `!` *expr*

deref ≡ `@` *expr*

instance_select ≡ *expr* `[` *expr_list* `]`

field_query ≡ *expr* `?` `[` *expr* `]`

control_expr ≡ *if_expr*

≡ *select_expr*

≡ *loop_expr*

≡ *return_expr*

if_expr ≡ `if [` *expr* `]` *expr* [`else` *expr*]

select_expr ≡ `select [` *expr* `]`

[`case [` *expr_list* `]` *expr*]$^{+}$

[`else` *expr*]

loop_expr ≡ `loop [` [*expr_list*] `]` *expr*

return_expr ≡ `return [` *expr* `]`

aggregate ≡ `{{` *expr_list* `}}`

cast ≡ *type_expr* == *expr*

typemod_expr ≡ *detype_expr*

≡ *retype_expr*

detype_expr ≡ `detype [` *expr* `]`

retype_expr ≡ `retype [` *expr* `]`

void_expr ≡ `void`

op_expr ≡ *expr binary_op expr*

≡ *unary_op expr*

identifier ≡ *alpha* [*alpha* | *digit* | _]*

identifier ≡ ‘ [*src_char* | *escape_seq*]* ‘

constant ≡ *decimal_const*

≡ *based_const*

≡ *real_const*

≡ *char_const*

≡ *string_const*

decimal_const ≡ *digit* [*digit* | _]*

based_const ≡ [*decimal_const*] $ [*alpha* | *digit* | _]+

real_const ≡ *decimal_const* . [*decimal_const*] [*exponent*]

≡ . *decimal_const* [*exponent*]

≡ *decimal_const exponent*

exponent ≡ e | E [+ | -] *decimal_const*

char_const ≡ ' *src_char* | *escape_seq* '

string_const ≡ " [*src_char* | *escape_seq*]* "

binary_op ≡ := | += | -= | *= | /= | %= | &= |
^= | |= | >= | <= | << | >> | ** |
| | ^ | & | = | # | < | > |
+ | - | * | / | % | >>= | <<=

unary_op ≡ + | - | ++ | -- | ^

escape_seq ≡ \a | \b | \f | \n | \r | \t | \v |
\' | \" | \` | \\

≡ \ $ [*alpha* | *digit*] *alpha* | *digit*

alpha ≡

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 ≡

0	1	2	3	4	5	6	7
8	9						

src_char ≡

∅	!	"	#	$	%	&	'
(	)	*	+	,	-	.	/
0	1	2	3	4	5	6	7
8	9	:	;	<	=	>	?
@	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	{	|	}	~	