PC–Dee: Syntax and Semantics

Peter Grogono

February 1994

Department of Computer Science
Concordia University
1455 de Maisonneuve Blvd West
Montréal, Québec
Canada H3G 1M8

1 Syntax

A Dee source program is a sequence of *tokens* that obeys the rules of a context-free grammar. Section 1.1 describes the tokens that may appear in the program and Section 1.2 describes the rules of the grammar.

1.1 Lexical Structure

The lexical structure of a language determines the form of its tokens. Each token in Dee is a string of characters chosen from a subset of the ASCII character set.

1.1.1 Character Set

Dee uses a character set that is a subset of the ASCII graphic character set and that consists of the following characters:

```
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
0 1 2 3 4 5 6 7 8 9
+ - * / \ < > . ; : ! " @ ( ) [ ] { }
```

All ASCII characters, and characters in the PC-DOS "extended" character set (codes 128–255) can be used in string literals.

Following convention, we refer to the characters `a`, `b`, ..., `z`, `A`, `B`, ..., `Z` as *letters* and the characters `0`, `1`, ..., `9` as *digits*.

1.1.2 Tokens

Tokens are terminal symbols of the grammar. In a source program, tokens must be separated by "white space". White space consists of zero or more space, tab, new line, or form feed characters. There must be at least one white space character between two tokens whose juxtaposition could create a longer token. Comments in Dee are tokens, not "white space".

A token may be a keyword, an identifier, a literal, an operator, a comment, or a punctuation symbol. The following list describes each kind of token.

Keywords Keywords consist entirely of lower case letters. Figure 1 shows the keywords of Dee. Keywords in parentheses, such as (**abstract**), may appear in interfaces but not in source programs. Keywords in brackets, such as [**ensure**], indicate future extensions and are not recognized by the current parser.

Identifiers Identifiers consists of one or more letters, digits, and underscore characters (_). Internal identifiers (that is, identifiers generated and used by the Dee system) may also contain the character $. The first character of an identifier must not be a digit. The keywords of Dee may not be used as identifiers. Dee is a case-sensitive language; the identifiers *DEE*, *Dee*, and *dee* are distinct.

(abstract)	class	else	fi	[invariant]	od	signal	var
(ancestors)	cons	elsif	from	link	or	then	while
and	const	end	(func)	method	private	true	
attempt	continue	[ensure]	handle	new	(proc)	undefined	
begin	debug	extends	if	nil	public	until	
break	do	false	inherits	not	[require]	(uses)	

Figure 1: The Keywords of Dee

Literals Instances of the basic classes of Dee are constructed by literals. The basic classes are *Int*, *Float*, *Bool*, and *String*. The literals false and true denote the instances of the standard class *Bool*. Other literal constants are either *numeric literals* or *string literals.*

There is also a literal nil denotes the unique, undefined object.

Numeric Literals A numeric literal is a token of the form

$$\text{digit}^{+}\ [\ .\ \text{digit}^{+}\ ]\ [\ (\texttt{E} \mid \texttt{e})\ [+ \mid -]\ \text{digit}^{+}\ ]$$

in which digit^{+} denotes one or more digits and the characters "E", "e", "+", "−", and "." denote themselves. If the numeric literal consists only of digits, it is interpreted as a constant object of class *Int*, otherwise it is interpreted as a constant object of class *Float*. A leading minus sign (−) is treated as a unary operator.

String Literals A string literal begins with the character " and continues until the next " on the same line. The quote characters are not part of the string, but all of the characters between them are part of the string. A string literal is interpreted as a constant object of class String.

Within the string, the character `\` acts as an escape character. The sequence `\n` denotes a new line and `\t` denotes a tab character. The sequence `\c`, where `c` is any character other than `n` or `t`, denotes the character `c` itself. In particular, `\\` translates to `\` and `\"` translates to `"`.

The sequence `\nnn`, where `nnn` is a sequence of up to three decimal digits, denotes the ASCII character with the corresponding code. The scanner reads digits until three digits have been read or until a non-digit character has been read. Figure 2 compares the source string in the Dee program and the actual string generated by the compiler. The expression `<n>` in the generated string denotes the character whose ASCII code is `n`. Note that the codes are decimal, not octal as in C. The character "␣" denotes a blank.

Source string	Generated string
`"\1\2 \3 2"`	`<1><2>␣<3>␣2`
`"\132 \0132"`	`<132>␣<13>2`

Figure 2: String Encoding

Unary Operators The unary operators of Dee are:

`-`   `not`   `undefined`

The operator `-` is the unary minus, which is translated by the parser to *$negate* (see Section 1.2). The operator **not** is Boolean negation. The operator **undefined** is a predicate which is true if its argument is the special value **nil** (the "undefined value").

Binary Operators Figure 3 shows the operators of Dee and explains their meanings. The first column of the table is the operator symbol. The second column explains the meaning of the operator. The table does not include the symbol ".", which separates the receiver and the method name in a message.

Symbol	Meaning
<	less than
<=	less than or equal
=	equal to
~=	not equal to
>=	greater than or equal to
>	greater than

Symbol	Meaning
+	plus
-	binary minus
-	unary minus
*	multiply
/	divide
\	modulus

Figure 3: The Binary Operators of Dee

Assignment Operators The assignment operators of Dee are: `:=`, `+=`, `-=`, `*=`, `/=`, and `\=`. As in C, the statement X `+=` Y is an abbreviation for $X := X+Y$. The other assignment operators are defined in a similar way, as shown in Figure 4.

Abbreviated Form	Expanded Form
X += *E*	*X* := *X* + *E*
X -= *E*	*X* := *X* - *E*
X *= *E*	*X* := *X* * *E*
X /= *E*	*X* := *X* / *E*
X \= *E*	*X* := *X* \ *E*

Figure 4: Abbreviated Assignment Statements

Punctuation The following symbols are used as punctuation symbols and delimiters in Dee programs:

`:`   `;`   `@`   `(`   `)`   `[`   `]`

Comments A comment begins with a left brace (`{`) and ends with a right brace (`}`), as in Pascal. Comments are terminal symbols of the grammar.

1.1.3 Operator Precedence

Figure 5 shows the precedence of operators in Dee. The lowest precedence is 1 and the highest is 5. Binary minus (`-`) has precedence 2 and unary minus (the same symbol: `-`) has precedence 5. As in Pascal, arithmetic expressions must be parenthesized in an expression that contains both Boolean and arithmetic operators.

Level	Operators
1	`<` `<=` `=` `!=` `>=` `>`
2	`+` `-` `or`
3	`*` `/` `\` `and`
4	`.`
5	`-` `not` `undefined`

Figure 5: Operator Precedence

1.2 Grammar

We use the following typographical conventions to describe the grammar of Dee.

- Non-terminal symbols are written in *slanted type* and have an initial upper case letter. Examples: *Factor*, *Feature*.
- Keywords are written in **sans serif type** and consist entirely of lower case letters. Examples: **begin**, **undefined**.
- Terminal symbols that have structure not shown by the grammar, such as numbers, are written in roman type. Examples: num, string.
- Other terminal symbols are written in `typewriter style` and quoted. Examples: "`.`", "`(`".
- The following symbols are metasymbols of the grammar:

  $\rightarrow$ separates the defined symbol from the defining expression;

  | indicates alternatives;

  (. . .) indicate grouping;

  [. . .] enclose an optional item (zero or one occurrences);

  {. . .} enclose a repeated item (zero or more occurrences).

The productions in the sections below define all of the non-terminal symbols of Dee. The start symbol of the grammar is *Class* defined in Section 1.2.7. The grammar includes syntax which is legal in a machine-readable interface but not in a Dee source program. For example, the interface of a class may contain a `uses` list and a method whose body is `abstract`, but neither of these should appear in source files. (The parser does not report errors if they do, however.)

1.2.1 Expressions

Literal	→	num | string | true | false | nil
Factor	→	*Message*
	|	"(" *Expr* ")"
	|	not *Factor*
	|	new *Type*
	|	undefined *Factor*
	|	*Literal*
Term	→	*Factor* { *Multop Factor* }
Simp	→	[*Sign*] *Term* { *Addop Term* }
Expr	→	*Simp* [*Compop Simp*]
Exprs	→	{ *Expr* [";"] }

Literal expressions other than nil, correspond to instances of the basic classes *Int*, *Float*, *String*, and *Bool*. Section 1.1.2 describes both numeric literals (num) and string literals (string).

There are only three levels of precedence for binary operators: *Multop* has the highest precedence and *Compop* has the lowest. See Section 1.2.2 below.

The semicolon is provided as an optional separator for expression lists. It is occasionally required to ensure correct parsing.

Example 1.1 These are literals: 256, `"John Dee"`. A literal number must be positive: -99 is a simple expression (*Simp*), not a literal. □

Example 1.2 The message a.*get*(2 −5) has one argument: $2-5=-3$. The programmer probably intended two arguments, and should have written a.*get*(2; −5). □

1.2.2 Operators

Multop	→	and | "*" | "/" | "\\"
Sign	→	"+" | "-"
Addop	→	or | *Sign*
Compop	→	"=" | "~=" | "<" | "<=" | ">" | ">=" |
Op	→	*Multop* | *Addop* | *Compop*

All operators are infix binary with precedences shown in Figure 5. The operator "-" may also be used as a prefix unary operator.

Example 1.3 Expressions such as $x * (1 - y)$ are allowed in Dee and have their conventional meanings. □

1.2.3 Messages

$$\textit{Message} \quad \to \quad (\ \mathrm{id}\ [\ @\ \mathrm{id}\ ]\ |\ \text{“(”}\ \textit{Expr}\ \text{“)”}\ )\ \{\ \text{“.”}\ \mathrm{id}\ [\ \text{“(”}\ \textit{Exprs}\ \text{“)”}\ ]\ \}$$

A message is the object oriented form of a procedure or function call. A message is either an expression or a statement, depending on its context.

The part of the message that precedes the dot is called the *receiver* of the message. The form of the receiver determines the kind of message.

- The receiver may be a simple variable.
- The receiver may itself be a message. In this case, the message will contain two or more dots. The dots associate to the left: $r.s.m$ stands for $(r.s).m$.
- By default, Dee uses dynamic binding: the method is selected at run-time according to the class of the receiver. If the receiver has the form $r@C$, the compiler treats the receiver as an instance of class C and selects the method accordingly.
- The receiver may be an expression that is neither a variable or a message. In this case, the syntax requires that it be enclosed in parentheses.

The identifier that follows a dot is a *method name*. The method name is optionally followed by a list of arguments, each of which is an expression.

Example 1.4 In the first line of the following program, the receiver is a simple variable, *phonebook*. In the second line, the receiver is the expression *f* . *format* (53). The parentheses around the receiver are not necessary: the third line is equivalent to the second. In the last line, however, the receiver is an infix expression, $n + 5$, and the parentheses are required.

```
phonebook . get ("Fred")
(f . format (53)) . substr (45)
f . format (53) . substr (45)
(n + 5) . show
```

□

Example 1.5 This example demonstrates the use of static binding: the receiver *stdnt* is assumed to be an instance of class *Person* and the method *setid* is obtained from this class.

```
stdnt@Person . setid (temp)
```

□

1.2.4 Statements

```
Assop  →  ":=" | "+=" | "-=" | "*=" | "/=" | "\="

Stmt   →  comment
       |  Message
       |  id Assop Expr
       |  if Expr then Stmts
          { elsif Expr then Stmts }
          [ else Stmts ] fi
       |  do Stmts od
       |  from Stmts
          ( until | while ) Expr [ comment ]
          do Stmts od
       |  break
       |  continue
       |  signal Expr
       |  attempt Stmts
          { handle Signature Stmts }
          end
       |  instr
       |  debug Expr

Stmts  →  { Stmt [ ";" ] }
```

The syntax for loops that begin with **from** is ambiguous. To avoid problems, do not write loop initialization statements that contain loops.

A comment is allowed in any context that permits a statement. The statement list separator, semicolon, is required on rare occasions.

Example 1.6 The parser will try to interpret

$$out\,.\,close \quad (x+1)\,.\,show$$

as if $(x+1)$ was the argument of *close*. To correct this, write $out\,.\,close;\ (x+1)\,.\,show$. □

1.2.5 Signatures

```
Type       →  id [ "[" { Type } "]" ]

Signature  →  id ":" Type [ comment ]
```

Dee types may have arguments. For example, the type *Table*[*String Widget*] might denote the type of a table containing widgets and indexed by strings.

1.2.6 Features

Constdec → const id "=" *Literal* [comment]

Vardec → (var | link) *Signature*

Methdec → (method | func | proc | cons)
(id | *Op*)
["(" { *Signature* } ")"]
[":" *Type*]
[comment]
[abstract | from id | [var { *Signature* }] begin *Stmts* end]

Feature → [public | private] (*Constdec* | *Vardec* | *Methdec*)

A *feature* of a class is either an *attribute* or a method. An attribute is either a *constant* or an *instance variable*. A method may be a *procedure*, *function*, or *constructor*. The keywords method and cons are used in source programs; the compiler writes func and proc in interfaces to indicate the results of side-effect analysis.

1.2.7 Classes

Class → class id ["[" { *Signature* } "]"]
[comment]
{ comment | ((extends | inherits | uses | ancestors) { *Type* }) }
{ comment | *Feature* }
eof

The terminal symbol "eof" denotes the end of the input file.

2 Semantics

A Dee program consists of a collection of classes that contains a unique *root class*. A class is determined by its *features*.

Classes play three roles in Dee.

- A class is the *unit of compilation*: each time the compiler is invoked, it compiles exactly one class.
- A class defines a *scope*. The features of a class have unlimited lifetimes, but are visible only within the class, or through its instances.
- A class defines a *collection of objects*. These objects, the *instances* of the class, all share the characteristics defined by the class.

There are four *basic classes* which have special status. They are:

- *Int*, the class of integers;
- *Float*, the class of floating point numbers;
- *String*, the class of finite strings over the ASCII character set; and
- *Bool*, the class of Boolean values.

These classes differ from other classes in two ways: their instances are created by literals (Section 2.1.1) rather than constructors (Section 2.1.4), and it is not possible to inherit from them (Section 2.6).

All run-time entities in a Dee program are *objects*. You can think of the run-time data structure as a directed graph in which each vertex corresponds to an object and each edge corresponds to an instance variable. An object is created by:

- evaluating a literal (Section 2.1.1);
- invoking a constructor (Section 2.1.4), or
- evaluating a **new** expression (Section 2.1.4).

Once created, an object remains a part of the object graph until it becomes inaccessible (no edges lead to it). After becoming inaccessible, it remains in memory until the garbage collector reclaims the memory that it occupies.

In the remainder of this section, we describe the semantics of Dee programs informally. The sequence of exposition is "bottom up", starting with the simplest program units and moving towards larger entities.

2.1 Expressions

The evaluation of a Dee expression yields a reference to an object. Sometimes evaluation requires the creation of one or more new objects, but most expressions do not create new objects. If the expression invokes a constructor, a new object will be created. If the expression contains a literal, a new object may be created. The run-time system does not create duplicate copies of common objects, such as 0, 1, **true**, and **false**.

2.1.1 Literals

Literals are values that denote themselves, such as 23, `"Strings can be literal, too."`, and **false**. The classes *Int*, *Float*, *Bool*, and *String* provide literals.

- Literals of the class *Int* are strings of decimal digits. The value of a literal is the conventional decimal value. The largest integer literal permitted has the value $2^{31} - 1$.

  Dee allows expressions such as -3, but interprets such expressions as a unary prefix operator ("$-$") followed by an integer literal.
- Literals of the class *Float* are strings representing floating point numbers. Section 1.1.2 describes the precise syntax.
- Literals of the class *String* are quoted strings. Section 1.1.2 describes the precise syntax.
- The class *Bool* provides exactly two literals: **true** and **false**.
- **nil** is a literal: see Section 2.1.6.

Example 2.1 These are literals:

```
39   3.14159263   "My name is John Dee""   true   nil
```

□

2.1.2 Variables

A variable is either an *instance variable* or a *local variable*. An instance variable names a component of an object and is valid for as long as the object exists. A local variables is a temporary name used by a method; it is valid only for the duration of an invocation of the method.

Variable names in Dee denote references to values, not the values themselves. Several variables may refer to the same object. A variable that does not refer to any object is said to be *undefined*.

Section 2.4.2 describes instance variables and Section 2.3.3 describes local variables. Section 2.5 describes the scope rules of Dee.

2.1.3 Messages

A *message* is the object oriented equivalent of a procedure or function invocation. Messages can be arbitrarily complex (see Section 1.2.3), but the basic form of all messages is $r \,.\, m(a_1, \ldots, a_n)$. The first component, r is the *receiver* of the message and may be a simple name, a message, or a general expression. The second component, m, is the name of a method. The third component, $a_1, \ldots, a_n$ is a list of expressions that are evaluated and passed as arguments to the method.

The evaluation of the message $r \,.\, m(a_1, \ldots, a_n)$ proceeds as follows.

- Evaluate the receiver, r.
- Evaluate the arguments, $a_1, \ldots, a_n$.
- The receiver is an object that belongs to a particular class C. Invoke the method m in the class C with the evaluated arguments.

Within the body of the method, the receiver (named *self*), the result (named *result*, if it exists), and the arguments $a_1, \ldots, a_n$ are visible. See Section 2.3 and Section 2.5 for further details of methods and scope rules respectively.

The value of the message depends on the class of the receiver, which is determined at run-time (*dynamic binding*). You can fix the class of the method invoked at compile-time (*static binding*) using the notation $r@C\,.\,m(a_1, \ldots, a_n)$, in which C is a class name. This message is evaluated in the same way as $r\,.\,m(a_1, \ldots, a_n)$, except that r is considered to be an instance of class C and so the method m of class C is invoked.

Evaluation of a message will fail if the receiver is undefined (see Section 2.1.6). Other conditions for successful evaluation, such as the existence of the method in the appropriate class and the conformance of arguments to parameters, are checked at compile-time.

A message can be used in a statement context as well as an expression context: see Section 2.2.2.

2.1.4 Constructors and `new` Expressions

New instances of basic classes are created by literals, as described in Section 2.1.1. New instances of all other classes are created by invoking *constructors*. The syntax of the invocation is the same as for a method, but the receiver of a constructor is usually an uninitialized variable. After the message has been evaluated, the variable refers to a newly-created object. A constructor may return another reference to the same object as its value: such a constructor can be used as an expression. Arguments of the constructor initialize the instance variables of the new object.

Example 2.2 If *anna* has been declared as a variable of class *Person*, and *Person* has a constructor *make_friend* which takes the person's name as an argument, the expression

anna . *make_friend* (`"Anna"`)

would create the new friend. If the constructor *make_friend* returns the new object, we could write

my_friend := *anna* . *make_friend* (`"Anna"`)

□

A concrete class in a Dee program must contain at least one constructor, but may contain several.

The name of a constructor is usually *make* (or has *make* as a prefix), but this is merely a convention. For objects which require explicit destruction, such as files, a different convention is used: the object is constructed with *open* and destroyed with *close*. The method *close* does not de-allocate the memory used by the object: this is done, as usual, by the garbage collector.

During the evaluation of a message $r\,.\,m(a_1, \ldots, a_n)$, in which m is the name of a constructor, the run-time system:

- allocates memory for the new object;
- binds the variable r to the new object; and
- invokes the constructor m with the argument(s) $a_1, \ldots, a_n$.

The purpose of combining these tasks is to ensure that variables correspond to initialized objects wherever possible. Occasionally, however, it is convenient to perform each task separately: new expressions are provided for this purpose.

The expression

```
new T
```

yields a new, uninitialized instance of the type T. We say "type", rather than "class", because Dee allows expressions such as *Set*[*Int*] (see Section 2.7.4).

The class corresponding to T must be a concrete class (see Section 2.6.4).

Example 2.3 The method *makeit* creates an instance of class *Special* and returns it as an instance of class *General*. Dee allows this, provided that *Special* is a descendant of *General*, and constructions of this kind are quite common.

```
method makeit: General
  var inst: Special
  begin
    inst . make (0)
    result := inst
  end
```

We can avoid the need for the local variable *inst* in the method above by using new, as the following, equivalent, definition demonstrates.

```
method makeit: General
  begin
    result := (new Special) . make (0)
  end
```

□

Dee allows a new expression in any context that is legal for an expression. In practice, new expressions are most often used as receivers, as in the example above. Although new expressions may also be used, for instance, to pass uninitialized objects to methods, extensive use of uninitialized objects is considered poor style in Dee.

2.1.5 Operators

Dee provides a number of unary and binary operators, as described in Section 1.1.2 and Section 1.1.3. Each operator expression is translated during parsing into a message. For example, $x + y$ translates to x . *\$plus* (y). There is a method name corresponding to each operator in the way that *\$plus* corresponds to $+$. The operator $-$ corresponds to two method names: *\$minus* for binary $-$ and *\$negate* for unary $-$.

The precedence of operators is fixed. The meaning of an operator is determined by the declaration of the corresponding method in a class. Thus we cannot say that $x + y$ means "add y to x", although style suggests that the operation should at least resemble addition semantically.

Example 2.4 The expression $x*y-z$ translates to x . *\$mul* (y) . *\$minus* (z) and the expression $x + y/z$ translates to x . *\$plus* (y . *\$div* (z)) □

2.1.6 nil and undefined

A variable usually refers to an object. If it does not, it is said to be *undefined*. For instance, a local variable (not a parameter) of a method is initially undefined. There are several ways in which a variable may be defined: by construction (Section 2.1.4), by assignment (see below and Section 2.2.3), by argument passing (Section 2.3.2), or by an exception handler (Section 2.2.6).

After the assignment statement $x :=$ nil has been executed, the variable x is undefined.

The expression undefined E yields true if the expression E evaluates to nil and false otherwise.

It might seem that we could avoid using the keyword undefined simply by writing $x =$ nil. The compiler would translate this expression into the internal form x . *\$equal* (nil). Evaluation of this message would fail if x was nil, because messages cannot be sent to nil. Consequently, $x =$ nil is not a useful expression and we have to write undefined x instead.

2.1.7 Expression Evaluation

Example 2.5 describes the evaluation of several expressions.

- `"The tide has turned."` is a string.
- *self* . *show* is a simple message, consisting of a receiver, *self*, and a method name, *show*. There are no arguments. It is not necessary to write an empty argument list ("()") if there are no arguments.
- In the message (3) . *float* the receiver, 3, is a literal. The parentheses are required: "3 . *show*" is syntactically incorrect.
- The message *rcvr* . *substr* (3 4) + `"."` is transformed by the compiler into the form

  rcvr . *substr* (3 4) . *\$plus* (`"."`)

  At run-time, the variable *rcvr* is evaluated, yielding a string; the method *substr* extracts four characters (the third through sixth) of this string; and the string `"."` is appended to the result, yielding the final value of the expression. The original string, *rcvr*, is unchanged.
- The run-time system evaluates the message *rcvr* . *make* (`"big"`), in which *make* is a constructor, by first allocating memory for *rcvr* and then evaluating the message as usual. Any previous binding of the variable *rcvr* is lost.
- The run-time system evaluates the message (new *Widget*) . *doit* (0.0 1.0) by first allocating memory for a new instance of *Widget* and then invoking the method *doit* on the new object. Dee does not require *doit* to be a constructor but, since the new object has not been initialized, a constructor is normally used after new.

□

2.2 Statements

Most statements have side-effects but do not yield objects. Dee allows a statement to return an object, but the new object is not accessible to the program.

2.2.1 Comments

The grammar allows a comment in any context that permits a statement. The comment has no effect on the evaluation of the program.

2.2.2 Messages

A message can be used as either an expression or a statement. If a message that is used in a statement context yields an object, the object is discarded.

In particular, constructors are often used as statements. Their effect is to bind a reference to a new object to the receiver. The constructor may also return a reference to the new object, and it is this reference that is discarded.

See also Section 2.1.3.

2.2.3 Assignments

The assignment statement $x := E$ evaluates the expression E, yielding an object, and then makes the variable x refer to that object. The left side must be a variable identifier.

Evaluation of E may yield a previously existing object or construct one or more new objects. The assignment $x := E$ does not create new objects other than the objects, if any, created by E.

The semantics of assignment permit aliasing.

The statement $x :=$ nil is legal whatever class x belongs to; it leaves x undefined.

The assignment operators (Section 1.1.2) are purely syntactic. The meanings of statements that use operators such as $+=$ are defined by Figure 4 on page 4.

Example 2.6

- If y is a variable referring to an object then, after the assignment $x := y$, the variables x and y refer to the same object.
- The statement *text* $+=$ " " is equivalent to *text* $:=$ *text* $+$ " ". Assuming that *text* is a string, the method $+$ from class *String* is used to concatenate the strings.
- In the program fragment

  x . *display*
  $x := y$
  y . *change*
  x . *display*

  assume that the method *change* alters the value of y and the method *display* displays the value of x. It is quite possible that the two calls to *display* in the fragment will show different values of x. Since x and y are aliases, y . *change* may change the value of x.

□

2.2.4 Conditional Statements

The conditional statement has conventional form and meaning. In the conditional statement

```
if $C_1$ then $S_1$
elsif $C_2$ then $S_2$
.....
elsif $C_{n-1}$ then $S_{n-1}$
else $S_n$
```

each C_i must be an expression yielding an instance of class *Bool* and each S_i must be a statement. The effect of the statement is the effect of the statement S_i, where i is the smallest value such that evaluation of C_i yields true. If C_i yields false for $i = 1, 2, \ldots, n-1$, then the effect is that of S_n.

Example 2.7 The statement

```
if undefined $x$
  then $x := y$
fi
```

makes x refer to the same object as y if x was previously undefined; otherwise, it has no effect. □

2.2.5 Loop Statements

The effect of the do statement

```
do $S_1 \cdots S_n$ od
```

is to evaluate the statements $S1, \ldots, S_n$ repeatedly. Between do and od:

- the statement break transfers control to the end of the loop, thereby terminating it; and
- the statement continue transfers control to the beginning of the loop, thereby starting it again.

The effect of the from statement

```
from $S$ while $C$ do
  $S_1 \cdots S_n$
od
```

is to first evaluate the statement S and then to evaluate the statements $S_1, \ldots, S_n$ repeatedly. At the beginning of every iteration, the expression C is evaluated and must yield either true or false. If it yields false the loop terminates.

The effect of replacing while by until, as in the statement

```
from $S$ until $C$ do
  $S_1 \cdots S_n$
od
```

is similar, but the loop terminates immediately after the expression C has yielded true. Note that the condition C is evaluated *before* the body of the loop is executed.

Example 2.8

- The statement
  ```
  from $n := 1$ while $n * n \leq s$
    do $n += 1$ od
  ```
  increases n until it is the smallest integer whose square exceeds s.
- The statement
  ```
  from $i := 0$ until $i > 10$
    do $out.put(i.show)$ od
  ```
  loops indefinitely. Although the form of the control statements (unfortunately) suggests a Pascal-style for-loop, in which incrementing is performed implicitly, Dee does *not* change the "controlled" variable inside the loop. It is easy to forget to update the controlled variable, but the Dee convention provides additional flexibility.

□

2.2.6 Exceptions

The signal statement

```
signal $E$
```

evaluates the expression E and then raises an exception. The *value of the exception* is the object yielded by E, which may be of any class.

The exception propagates through dynamic enclosing scopes until it is caught by a *handler*. The run-time system defines a handler which catches all exceptions and whose scope is the entire program. This ensures that all exceptions will be caught.

The attempt statement handles exceptions. It has the form

```
attempt $S$
  handle $H_1$
  handle $H_2$
  .....
  handle $H_n$
end
```

This statement evaluated in the following way. First, the statement sequence S is evaluated. If S terminates normally, without raising an exception, the attempt statement is complete. If an exception is raised during the evaluation of S, either at the current level or in an inner dynamic scope, control passes to the handlers $H_1, \ldots, H_n$. Each handler either accepts the exception or rejects it, as described below. The exception is accepted at most once: if it is accepted by handler H_i, then handlers $H_{i+1}, \ldots, H_n$ are not invoked. If none of the handlers match the exception object, the exception is propagated into the enclosing dynamic scope.

Each handler H_i consists of a signature, $x_i : T_i$, and a statement list S_i. Suppose that the exception object is o and that its class is C_o. If C_o conforms to T_i (see Section 2.7.1), the exception is accepted by the handler, otherwise it is rejected. If the exception is accepted, the object o is bound to the name x_i, and the statements in S_i are evaluated. The scope of the declaration $x_i : T_i$ is just S_i (see Section 2.5.1).

Note In the current version of PC-Dee, the run-time system does not check conformance. The handler H_i is evaluated only if C_o and T_i denote the same class.

Example 2.9 The assignment statement in the code below reads a string from the source *in* and uses *stoi* to convert the string to an integer value. If the conversion fails, the run-time system raises an integer-valued exception which is matched by e. The handler displays a warning message and sets n to zero. Strictly, the handler should check the value of e as well as the type. The code given is reasonable if *get* refers to the keyboard, because *stoi* is the only possible source of an exception.

```
attempt
  n := in . get . stoi
handle e: Int
  out . put ("Invalid number!")
  n := 0
end
```

□

2.2.7 debug

As its name suggests, the primary purpose of the **debug** statement is to facilitate debugging. If the switch **debugging** is on, the effect of the statement **debug** E is to evaluate the expression E and then to display a string representation of the object yielded by E. If the switch is off, the **debug** statement has no effect.

A general object is displayed by displaying the values of its instance variables between parentheses: $(F_1\ F_2\ \ldots\ F_n)$. The instance variables refer to objects that are displayed using the same format recursively. Instances of basic classes are displayed as literals. Null objects are displayed using the character with ASCII code 22 (a small block). Components below the sixth level of nesting are displayed using the character with ASCII code 29 (a small, double-ended arrow).

2.2.8 Dee-machine Instructions

Some methods, including most methods of the basic classes, are implemented by *Dee-machine instructions* (DMIs) rather than Dee code. Each DMI begins with "!". DMIs are designed to fulfill a particular purpose and should not be used outside the context for which they were designed.

Example 2.10 The following method from class *Int* illustrates the use of DMIs.

```
public method +(other: Int): Int
  begin !addi end
```

□

2.3 Methods

Methods play the role of procedures in an imperative language and functions in a functional language. Dee recognizes four kinds of method, introduced by one of the keywords **proc**, **func**, **method**, and **cons**. Programmers usually write **method** and **cons** only; the keywords **proc** and **func** appear only in interfaces.

- The keyword proc introduces a method that may have side-effects. A method has a *side-effect* if it may change the value of an instance variable of its receiver or another object.
- The keyword func introduces a method that does not have side-effects.
- The keyword method introduces either a proc or a func.
- The keyword cons introduces a constructor.

A constructor differs semantically from other kinds of method in that its evaluation creates a new object. Memory for the new object is allocated before the constructor is invoked. Within the body of the constructor, the identifier *self* (see Section 2.3.4) initially refers to the new object with instance variables uninitialized.

2.3.1 Names

The name of a method may be either an identifier or a Dee operator. If the name is an operator, the signature of the method must be appropriate for the operator. For example, a method with the name + should have one argument and return a result.

2.3.2 Parameters

Each parameter of a method has a name and a type. On entry to the method, each parameter is a reference to the corresponding argument of the invoking message. Thus Dee implements "call by sharing", in which the argument and the corresponding parameter both refer to the same object.

2.3.3 Local Variables

In addition to parameters, a method may have local variables. On entry to the method, each local variable is *undefined*.

2.3.4 *self*

The implicitly declared local variable *self* refers to the receiver of the message.

Dee does not permit assignment to the variable *self*. Such an assignment would not cause any harm, but could not have any observable effect. It is therefore outlawed on the assumption that it is probably a mistake by the programmer.

2.3.5 *result*

Any method may return a result. If it does:

- there must be a type declaration following the parameter list; and
- the local variable *result* is implicitly declared and is used to store the value of the result. For type checking purposes, *result* has the type declared in the method header.

If a method does not return a result, the use of *result* within it is an error.

Example 2.11 The name of the method below is *valid*. It has a parameter p and a local variable *diff*. If the method is invoked by a . *valid* (b) then, within the body of the method, *self* refers to a and p refers to b. Since the method returns a result, the local variable *result* is implicitly declared with type *Bool*.

```
method valid (p: Person): Bool
  var diff: Float
  begin
    diff := self . age − p . age
    result := diff > 20.0
  end
```

□

2.3.6 Bodies

There are three possible forms of a method body.

- The usual form of the body is a statement sequence enclosed between the keywords **begin** and **end**. When the method is invoked, control is passed to the first statement of this sequence. The method returns when all the statements in its body have been evaluated.
- The body may be empty. In this case the method is called an *abstract method*. Section 2.6.3 describes abstract methods.
- The body may have the form **from** P, where P is a class name. This form is used when the method could be inherited from more than one parent; it indicates that the method is to be inherited from the class P. See Section 2.6.

Example 2.12 The method *sq*, below, is a normal, or "concrete", or "implemented", method. The method *unit* is abstract. The method *dummy* is a concrete method that does not do anything. The method *show* has versions in two or more ancestors of the class in which it is defined, and **from** is required to resolve the ambiguity.

```
public method sq: Int
  begin result := self ∗ self end

public method unit: T

public method dummy
  begin
  end

public method show: String
  from Array
```

□

2.3.7 Constructors

Constructors are introduced by the keyword **cons** but are otherwise syntactically identical to other methods. A constructor may return a result; if it does, the type of the result must be the type of the current class, and the constructor must contain the statement *result* := *self*.

Constructors are evaluated in the same was as other methods except that the run-time system allocates space for the receiver before executing the body of the constructor. On entry to the constructor, the instance variables of the new object are uninitialized. Statements in the body usually include assignments to these instance variables.

2.4 Classes

A class has a number of *features*. A feature is a *constant*, and *instance variable*, or a *method.* Each feature is either declared in the class or inherited from another class declaration.

Each feature has a *name* that must be unique within the class.

The declaration of a feature may begin with either of the keywords **public** or **private**. If neither is present, **private** is assumed. The scope of all features includes the entire class declaration (see Section 2.5).

The "." operator in a message opens a scope in which only public features of the class of the receiver are visible. Private features are not visible "outside the class".

2.4.1 Constants

A constant declaration binds a literal to an identifier. The value of a constant is either **nil** or an instance of one of the classes *Int*, *Float*, *Bool*, or *String*. Dee infers the type of the constant from its lexical structure.

2.4.2 Instance Variables

The instance variables in a class declaration determine the form of the instances of the class at run-time. Each instance variable has a *name* and a *type*.

The declaration of an instance variable can be written as either **var** $n\colon T$ or **link** $n\colon T$. The idea is that a **var** is a "part of" the object whereas a **link** is an "association" to another object. The current compiler, however, makes no semantic distinction between the two forms of declaration.

When a new object is created by a constructor, its instance variables are initially undefined. Constructors usually contain assignments, or calls to other constructors, to initialize the instance variables of new objects.

2.4.3 Methods

The methods in a class declaration determine the operations that instances of the class can perform at run-time. Each method has a *signature* consisting of its name, parameters, and result type.

2.5 Scope Rules

Dee provides three lexical scope levels: *handler*, *local*, and *global*. In addition to these, inheritance declarations open scopes, as described in Section 2.6.

The scopes are nested: the innermost scope is handler, the outermost is global, and local comes in between. Dee does not allow a name to be defined more than once in any scope. If a name is defined at more than one scope level, the inner definition hides the outer definition(s).

2.5.1 Handler Scope

An **attempt** statement (see Section 2.2.6) has the form

```
attempt $S$
handle $x_1$: $C_1$ ...
handle $x_2$: $C_2$ ...
.....
handle $x_n$: $C_n$ ...
end
```

The scope of the exception variable, x_i is from its declaration to the next occurrence of **handle** or **end**.

2.5.2 Local Scope

The scope of the parameters and local variables of a method is the body of the method—that is, the statement sequence between **begin** and **end**. The implicitly declared local variables *self* and *result* have the same scope as local variables.

2.5.3 Global Scope

The scope of a feature is the entire class declaration. Dee does not require a declaration to appear before the declared feature is used. Declaration before use, however, is recommended because it improves readability.

The public features of a class can be referred to outside the class by subscripting the name of an instance of the class. If r is an instance of class C, and C has a public feature f, then $r.f$ is allowed. It is this usage that the term "global scope" is intended to suggest.

The private features of a class can be referred to only within the class declaration.

Inheritance extends the scope of global variables, as described in the next section.

2.6 Inheritance

A class C may inherit one or more classes $P_1, P_2, \ldots, P_n$. We refer to the current class C as the *child class* and the inherited classes as *parent classes*. The inheritance declaration appears in the program as shown below.

```
class C
  inherits P_1 P_2 ... P_n
```

The effect of an inheritance declaration is, roughly speaking, to bring all of the features of the parent classes into the scope of the current class. If there are no name collisions, this is exactly what happens. If a feature in a parent class has the same name as a feature in the current class, or a feature in another parent class, however, the rules in Section 2.6.2 apply.

The relation *ancestor* is the reflexive and transitive closure of the relation *parent*. Every class belongs to the set of its ancestors, as do its parents, the parents of its parents, and so on.

The relation *descendant* is the reflexive and transitive closure of the relation *child*. Every class belongs to the set of its descendants, as do its children, the children of its children, and so on.

A child class inherits all of the features of its parents (Section 2.6.1). The child class may redefine its inherited features (Section 2.6.2) and define its own features. A feature, as usual, may be a constant, instance variable, or method.

Dee does not allow a child class to hide features provided by its ancestors. Semantic analysis of Dee programs depends on this rule: *if class P provides feature f, and class C inherits P, then C provides f.*

It is not possible to inherit from any of the basic classes *Int*, *Float*, *Bool*, or *String*.

Note The current compiler does not implement all of the inheritance rules exactly as they are described here.

2.6.1 Inheriting Features without Redefinition

If a feature is declared in the parent class but not in the child class, we say that it is *inherited without redefinition*.

If two parents of a class have features with the same name, the names *collide* in the child class. The following rules apply in the event of name collisions. We assume that class C inherits classes P and Q.

- If P and Q both declare a constant named c, the constant must have the same type and value in P and Q.
- If P and Q both declare an instance variable named v, the variable must have the same type in both classes.
- If P and Q both declare a method named m, the method must either be redefined or be declared with a from body in class C. In this case, the body would be either from P or from Q.

  A from clause may not be used to inherit an abstract method (see Section 2.6.3).

2.6.2 Inheriting Features with Redefinition

A feature that is inherited from a parent class may be redefined in the child class. This is called *inheritance with redefinition*.

By default, an inherited feature has the same visibility (public or private) in the child class that it had in the parent class. A feature that is private in the parent class may be redefined as public in the child class.

The rules for inheritance with redefinition follow. Note that, although redefinition of constants and variables is allowed, it is not particularly useful.

- A constant may be redefined provided that its value is the same in the parent class and the child class.
- A variable may be redefined provided that its type is the same in the parent class and the child class.
- An inherited method may be redefined in a child class. The signature of the redefined method must conform to the signature of the method in the parent class. (Section 2.7.1 defines type conformance.)

2.6.3 Abstract Methods

A method with an empty body (that is, no **begin** ... **end** part or **from** clause) is called an *abstract method.* A method that is not abstract is called a *concrete method.* Abstract methods are used to ensure signature conformance between subclasses of an abstract class.

2.6.4 Abstract Classes

A class that contains one or more abstract methods is called an *abstract class.* A class that contains no abstract methods is called a *concrete class.*

A class may contain abstract methods or constructors, but not both. Consequently, an abstract class cannot contain constructors and a concrete class cannot contain abstract methods.

There are two important consequences of the rule that an abstract class has no constructors:

- an abstract class cannot have instances; and
- an abstract method can never be invoked.

The purpose of an abstract class is to define a protocol that its descendants may use. Concrete descendant classes redefine the abstract methods with concrete methods.

2.6.5 Extension

The declaration

extends P_1 P_2 ... P_n

is similar to

inherits P_1 P_2 ... P_n

but has a different effect on scopes. Public features in the parent classes become private features of the child class.

Extension provides "protected" inheritance. The child class can use the features of its parents, but clients of the child class cannot use these features.

Example 2.13 The method *show* in class *Dean* below uses *show* in the parent class *Professor* to provide the name of the professor. By writing *self@Professor*, we ensure that the required version of *show* is called and avoid an unpleasant recursion.

```
class Professor
  var name: String
  method show: String
    begin
      result := name
    end

class Dean
  inherits Professor
  method show: String
    begin
      result := "Dean " + self@Professor . show
    end
```

□

2.7 Types and Type Checking

In previous sections, we have used the words "class" and "type" as if they were synonymous. In order to discuss the type rules of Dee, however, we must be more precise.

Class is a run-time, or *dynamic*, concept. During execution, each object is an instance of a particular class.

Type is a compile-time, or *static*, concept. The Dee compiler infers the type of each expression in the source program and uses the type information to ensure that evaluation of the expression at run-time will not fail.

A type is a set of classes. Corresponding to a class C, there is a type $\widehat{C}$ that contains C and all of the subclasses of C.

The declaration $x\colon C$ means that, at run-time, the object referred to be x will be an instance of a class C', where $C' \in \widehat{C}$.

The type *Null* denotes the empty set of classes. The type of nil is *Null*.

A parameterized class does not introduce a type, but each instance of it does. For example, *Array*[*T*] does not correspond to a type, but *Array*[*Int*] does.

There is an important relation on types called *type conformance*. If type T conforms to type T', then an expression of type T can be used in a context that requires type T'. T conforms to T' if and only if $T \subseteq T'$.

Note It is well-known that static type checking of object oriented languages is difficult. The rules used by Dee are unsafe: there are programs accepted by the compiler that fail with a type error during execution. In practice, this occurs rarely, however, and the type system seems to be an acceptable compromise.

Example 2.14

- The expression *Int* denotes both a class whose instances are integers and a type with one member: the class *Int*.
- The expression *Comparable* denotes a class with no instances and a type which includes the class *Comparable* and all its descendants: in this case, all classes whose instances may be compared using =.

□

2.7.1 Type Conformance

Let $C[C_1, \ldots, C_m]$ and $P[P_1, \ldots, P_n]$ be types. Then $C[C_1, \ldots, C_m]$ *conforms to* $P[P_1, \ldots, P_n]$ iff all of the following conditions are true:

- $m = n$, and
- C_i conforms to P_i for $1 \leq i \leq m$, and
- class C is a descendant of class P.

2.7.2 Expressions and Statements

This section summarizes the rules that Dee uses to infer types and to check type correctness. We use E to denote an arbitrary expression in the program.

For the purposes of type checking, all expressions are assumed to be literals, variables, or messages. Constants are replaced with their (literal) values, and infix expressions are replaced by messages, as described in Section 2.1.5.

- Every variable v must have a declaration of the form $v{:}\,C$. When v occurs in an expression or statement, its type is assumed to be $\widehat{C}$.
- If E is a literal, the compiler infers the type from the lexical structure of the literal. An integer literal, such as 99, has type *Int*, and so on. The type of nil is *Null*.
- If E is a message $r\,.\,m(a_1, \ldots, a_n)$, the compiler first determines T_r, the type of r. It then obtains the signature $m(T_1, \ldots, T_k){:}\,T$ of the method m in the class T_r.

  ($T_1, \ldots, T_k$ are the types of the parameters of m. T is the type of the result of m or *Null* if it does not return anything.)

  There must be the same number of arguments and parameters ($k = n$) and the type of each argument a_i must conform to the type of the corresponding parameter T_i. The type of the message is the type of its result, T.
- A message of the form $r@C\,.\,m(a_1, \ldots, a_n)$ is processed in the same way as above, except that the compiler checks that T_r conforms to C and then processes the expression as if the type of the receiver was C.
- An assignment statement $v := E$ is type correct if the type of E conforms to the type of v.
- The expression following one of the keywords if, elsif, while, or until must have type *Bool*.
- Statements have type *Null*. If a message that returns a non-null result is used in a statement context, the result is discarded.

2.7.3 Inheritance

If class C inherits class P, features inherited from P are included in C before type-checking. The following rules must be respected for features that are redefined in class C.

A descendant class may change the visibility of a feature, as described in Section 2.6.2.

- The type of a redefined variable must be the same in both classes.
- The parameters and result of a redefined method must conform to the corresponding types in the parent class. (This is the "covariant" rule. Contravariance is technically safer, but often less useful.)

2.7.4 Parameterized Classes

A parameterized class has a heading of the form

class $C[T_1 \colon D_1; \ldots ; T_n \colon D_n]$

in which $T_1, \ldots, T_n$ are *type parameters* and each $D_1, \ldots, D_n$ is a *constraint*. The class is type checked by (conceptually) replacing each occurrence of T_i with the corresponding constraint D_i.

A parameterized class is instantiated by a declaration of the form $v \colon C[C_1, \ldots, C_n]$, in which $C_1, \ldots, C_n$ are *class arguments*. The compiler checks that the number of arguments is equal to the number of parameters and that each argument conforms to the corresponding constraint.

Example 2.15 The declaration of the class *Set* begins as follows.

class *Set*[T: *Comparable*]

While type-checking this class, the compiler treats every instance of the type T as if it was *Comparable*. This check ensures that members of the set can be compared for equality.

The class *Set* is used in declarations such as

var *Nums*: *Set*[*Int*]

The compiler checks that *Int* conforms to *Comparable*. □