# Appendix B

# Leda Grammar

In this appendix we describe the grammar of the language Leda as it is used in this text. Note that the language has evolved over time, and this grammar differs slightly from the language used in some of the earlier papers on Leda, especially [Budd 91a] and [Budd 92].

The grammar will be defined using a combination of BNF and commentary. Keywords are given in **bold**. Non-keyword lexical items are written in *italic*. Nonterminals symbols are written in Roman font. The symbol $\epsilon$ is used to represent "nothing", that is, an empty sequence of characters.

## B.1 Overall Program Structure

| | | |
|---|---|---|
| program | ::= | declarations body **;** |
| declarations | ::= | $\epsilon$ |
| | \| | declarations declaration |
| body | ::= | **begin** statements **end** |

The overall structure of a program is a sequence consisting of zero or more declarations, followed by a single compound statement. The compound statement given in the body is the statement initially executed when the program is invoked.

Definitions: declaration: Section B.2, statements: Section B.6.

## B.2 Declarations

| | | |
|---|---|---|
| declaration | ::= | **const** constantDefinitions |
| | \| | **var** variableDefinitions |
| | \| | **type** typeDefinitions |
| | \| | functionDeclaration |
| | \| | classDeclaration |
| constantDefinitions | ::= | constantDefinition |
| | \| | constantDefinitions constantDefinition |
| constantDefinition | ::= | *identifier* **:=** expression **;** |
| variableDefinitions | ::= | variableDefinition |
| | \| | variableDefinitions variableDefinition |
| variableDefinition | ::= | identifierList **:** type **;** |
| identifierList | ::= | *identifier* |
| | \| | identifierList **,** *identifier* |
| typeDefinitions | ::= | typeDefinition |
| | \| | typeDefinitions typeDefinition |
| typeDefinition | ::= | *identifier* **:** type **;** |

Unlike Pascal, in Leda the different forms of declaration can be listed in any order, and the same type of declaration section may appear more than once in a given scope (although individual identifiers cannot be declared more than once, see below).

The expression assigned to a constant identifier need not be resolvable at compile time. Such values will be calculated in the order they are given prior to the execution of any statement in the context in which they are defined.

All declared identifiers (constant, type, variable, function and class names) must be unique within the scope in which they are declared. However, names can match declarations in surrounding scopes, in which case within the local scope the new name temporarily hides the surrounding name binding.

Although type declarations are permitted in Leda they tend to be used far less frequently than in Pascal, since the vast majority of type declarations are replaced by classes.

Identifiers declared within a function can only be accessed by statements defined in the function scope.

Identifiers created within a class definition can be accessed outside the class scope only if qualified by an expression of the appropriate class type (see Section B.8).

Definitions: type: Section B.3, functionDeclaration: Section B.4, classDeclaration: Section B.5, expression: Section B.7.

## B.3 Types

type ::= *identifier*
  | *identifier* [ typeList ]
  | **function** ( optionalArguments ) optionalReturnType

typeList ::= type
  | typeList , type

optionalArguments ::= $\epsilon$
  | formalList

formalList ::= parameterMode type
  | formalList , parameterMode type

parameterMode ::= $\epsilon$
  | **byRef**
  | **byName**

optionalReturnType ::= $\epsilon$
  | $->$ type

Types can be divided into class types (which are types generated automatically by a class definition), resolved types (produced by binding qualified type parameters on a generic function or class), or function types.

A type-list used to qualify a generic function or class type must match in number and compatibility the type arguments given in the declaration of the underlying type.

The data-type `relation`, used in the back-tracking mechanism, is the only type predefined by the Leda system (as opposed to being defined in Leda itself in the standard library). Although a relation can be thought of as a form of boolean, in fact a relation is more accurately described as equivalent to the following definition:

```
type
    relation : function (relation)->boolean;
```

That is, a relation is a function which takes as argument another relation and returns a boolean value. Functions that manipulate relations are written in Leda and are defined as part of the standard library. A number of other types are also defined in the standard library.

There are three parameter passing mechanisms used by Leda. In no parameter mode is specified then parameters are passed by value. The `byRef` keyword indicates that parameters are to be passed by reference, in which case assignments to the argument value made within the function are reflected in changes to the actual parameter value. The `byName` keyword indicates that parameters are to be passed by name. In this case the evaluation of the parameter is delayed until it is used within the function.

## B.4 Function Declarations

| | | |
|---|---|---|
| functionDeclaration | ::= | **function** *identifier* typeArguments valueArguments optionalReturnType ; declarations body ; |
| typeArguments | ::= | $\epsilon$ |
| | \| | [ argumentList ] |
| valueArguments | ::= | ( ) |
| | \| | ( argumentList ) |
| argumentList | ::= | parameterMode identifierList : type |
| | \| | argumentList , parameterMode identifierList : type |

A function declaration provides both the function signature (argument types and return type) and the function body.

The keyword **function** is used to define both functions that return values and functions that do not return any value (the latter are sometimes referred to as procedures).

Functions can be made generic by defining type parameters. If type parameters are omitted then the square brackets are omitted as well. However, the parenthesis that surround value parameters must be written, even if no such parameters are used.

Identifiers declared within a function definition can be accessed only by statements that originate within the scope of the function.

The meaning of operator symbols can be defined by providing a function which uses the textual-name for the operator symbol. See Section B.11 for a list of the textual-names for operators.

Function names cannot be overloaded, with the exception of functions that are defining the meaning of operator symbols, which can be overloaded only at the global scope.

Definitions: declarations, body: Section B.1, identifierList: Section B.2, parameterMode, type, optionalReturnType: Section B.3.

## B.5 Class Declarations

| | | |
|---|---|---|
| classDeclaration | ::= | classHeading declarations **end** ; |
| classHeading | ::= | className ; |
| | \| | className **of** *identifier* ; |
| | \| | className **of** *identifier* [ typeList ] ; |
| className | ::= | **class** *identifier* typeArguments |

Class definitions define a new identifier scope, in a similar manner to function definitions. In addition, class declarations define the class name as a new type.

The optional identifier following the keyword `of` must denote a class name, which is the parent class from which the new class will inherit. If the parent class is generic (defined using type-parameters) then type-parameters must be provided in the bracket-surrounded type argument list in order to fully resolve the generic class parameters. Furthermore, such parameters must match in type and in number of type arguments given in the underlying class declaration. Such a type-list cannot be used on a non-generic class. If no parent class is specified the class `object` is assumed.

Definitions: typeList: Section B.3, typeArguments: Section B.4.

## B.6 Statements

| | | |
|---|---|---|
| statements | ::= | $\epsilon$ |
| | \| | statements statement **;** |
| statement | ::= | reference **:=** expression |
| | \| | **return** |
| | \| | **return** expression |
| | \| | **begin** statements **end** |
| | \| | **if** expression **then** statement |
| | \| | **if** expression **then** statement **else** statement |
| | \| | **while** expression **do** statement |
| | \| | **for** expression **do** statement |
| | \| | **for** expression **to** expression **do** statement |
| | \| | **for** reference **:=** expression **to** expression **do** statement |
| | \| | procedureCall |
| | \| | $\epsilon$ |
| procedureCall | ::= | functionCall **(** optionalExpressionList **)** |
| | \| | **cfunction** *identifier* **(** optionalExpressionList **)** |
| optionalExpressionList | ::= | $\epsilon$ |
| | \| | expressionList |
| expressionList | ::= | expression |
| | \| | expressionList **,** expression |

The semicolon is used as a statement terminator, not a statement separator.

In an assignment statement, the expression to the right of the assignment symbol must have a type that is legally assignment-compatible with the reference described to the left of the assignment symbol. Assignment is performed using *pointer semantics.* Following the assignment statement the value of the reference on the left of the assignment arrow is exactly the same as the value on the right of the assignment arrow. If the right side is a simple reference, then changes made to one variable will be reflected in changes made to the other, and vice-versa. This situation will remain until one or the other reference is subsequently reassigned, or until one or the other goes out of scope.

A value is assignment-compatible with a variable if

1. The type of the value is the same as the type of the variable,
2. The type of the value is an instance of a subclass for the class associated with the variable,
3. The value is the polymorphic constant `NIL`,
4. Both the variable and value are function types, and
   (a) The number of arguments and their storage form match
   (b) The type associated with each by-value or by-name parameter in the function value is type-assignment compatible with the equivalent types in the target variable,
   (c) The type associated with each by-reference parameter is the same in both cases (a test that can be implemented by determining that the values are type-assignable in both directions), and
   (d) The type associated with the returned value is type-assignable to the type of the target.

No automatic conversions are done by Leda,[1] not even the conversion of integer to real. All such changes must be explicitly specified by the user. (Mixed mode arithmetic operations are implemented by defining several forms of the arithmetic operators at the global level).

Parameters that are passed by-name cannot be used as the target of an assignment, nor can identifiers that are declared as referring to constant values. Parameters that are passed by-value are treated the same as local variables that have been initialized with the value of the actual argument. Note that this is a form of assignment, and thus the comments relative to the semantics of assignment (above) are true for by-value parameters as well. This is occasionally a source of confusion, when an operation that changes the state of a formal parameter value will be observed to have also changed the state of the actual parameter value used in a function call.

Return statements are not permitted in the body defined at the global level. A return statement with a value can only be used in a function that returns a value of a type to which

[1] There is one exception to the rule that Leda performs no automatic conversions. Relational values will be automatically converted into a boolean value when they are used in the test expression in an `if` or `while` statement.

the expression can be assigned. Conversely, if a return statement is used within a function that does not produce a value, then no expression can be used with the return statement.

Return statements are not permitted within the statement associated with the relational version of the `for` statement.

The expression used to control execution in an `if` statement or a `while` loop must be either boolean type, or relational type (which will be converted into boolean type).

An `else` clause is matched with the closest surrounding `if` statement.

The first expression used in the first and second form of the `for` statement must be of type relation. In the second form of the `for` statement the second expression must be type boolean. In the form of the `for` statement using assignment (the third form of `for` statement), the reference and all expressions must be type integer.

Parentheses must be used in a procedure call, even if no actual arguments are being passed.

The keyword **cfunction** introduces a run-time system call; an invocation of an underlying operation that is normally not type-checked. It is the responsibility of the programmer to ensure that no type errors can be introduced through the use of this mechanism. Normally calls on **cfunctions** are hidden within other functions. Implementations may impose further restrictions on the use of this mechanism.

In a procedure call the actual arguments being passed must match in number and compatibility the arguments given in the declaration of the target function.

Definitions: expression, functionCall : Section B.7, reference: Section B.8.

## B.7 Expressions

| | | |
|---|---|---|
| expression | ::= | andExpression |
| | \| | expression *orSymbol* andExpression |
| andExpression | ::= | notExpression |
| | \| | andExpression *andSymbol* notExpression |
| notExpression | ::= | relationalExpression |
| | \| | ˜ notExpression |
| | \| | reference **is** type |
| | \| | reference **is** type ( identifierList ) |
| relationalExpression | ::= | plusExpression |
| | \| | plusExpression *relationalOperator* plusExpression |
| | \| | reference $<-$ plusExpression |
| plusExpression | ::= | timesExpression |
| | \| | plusExpression *plusOperator* timesExpression |
| timesExpression | ::= | functionCall |
| | \| | timesExpression *timesOperator* functionCall |
| | \| | *plusOperator* functionCall |
| functionCall | ::= | basicExpression |
| | \| | functionCall ( optionalExpressionList ) |
| | \| | **cfunction** *identifier* ( optionalExpressionList ) $->$ type |

The *or* symbol is `|`. The *and* symbol is `&`. The six relational operators are $<$, $<=$, $=$, $<>$, $>=$ and $>$, The two plus operators are $+$ and $-$. The three times operators are $*$, / and `%`, the latter denoting remainder. The **cfunction** keyword is described in Section B.6.

The **is** keyword introduces a type pattern matching operation. The expression returns a boolean true value if the expression to the left of the **is** is an instance or subclass of the type represented by the identifier to the right of the keyword. The identifiers in the optional list following the test must match the data fields defined for the type. If the identifier list is present and the test is successful, the data fields are copied out of the value into the variables, in effect undoing the actions of the constructor for the class.

The $<-$ operator used in relational expressions is used to tentatively bind an identifier to an expression during the process of conducting a relational search. The result is a value of type `relation`. A subsequent failure of the relational search may force the binding to be broken, while search continues for an alternative value.

In a function call the actual arguments being passed must match in number and compatibility the arguments given in the declaration of the target function. Arguments passed to by-reference parameters need not be references; if non-reference a temporary variable will be generated, the value of the parameter will be assigned to the temporary, and the reference passed to the function will be that of the temporary.

Definitions: type: Section B.3, optionalExpressionList: Section B.6, basicExpression: Section B.8.

## B.8 Basic Expression

| | | |
|---|---|---|
| basicExpression | ::= | reference |
| | \| | *constant* |
| | \| | ( expression ) |
| | \| | basicExpression [ typeList ] |
| | \| | [ expressionList ] |
| | \| | **function** valueArguments optionalReturnType **;** declarations body |
| reference | ::= | *identifier* |
| | \| | functionCall **.** *identifier* |

Constants can be either string constants, integer constants, or real (floating-point) constants (see Section B.12).

A reference must denote a declared identifier (which can be either constant, variable, type or function). In the first form the identifier must be accessible in the scope containing the reference, while in the second form the identifier must be accessible in the scope denoted by the function call, which must yield an expression of type class.

A square bracket surrounding a type-list following a basic expression is used to describe the resolution of qualified types in a function expression.

Square brackets surrounding an expression list is used to define an array literal. All the expressions appearing in the list must have the same type.

Definitions: declarations, body: Section B.1, typeList: Section B.3, valueArguments, optionalReturnType: Section B.4, expression, expressionList, functionCall: Section B.7.

## B.9 Comments and Whitespace

Any text between matching curly braces is treated as a comment and will be ignored by the parser. This is the only legal use for curly brace characters.

Outside of literal strings, space, tab and newline characters have no meaning.

## B.10 Identifiers and Keywords

Identifiers must begin with a letter or underscore, and consist of an arbitrary number of letters, digits, or underscore characters.

In addition to the textual names for operator symbols (see Section B.11), the following tokens are reserved as keywords, and cannot be used to define identifiers:

| | | | | |
|---|---|---|---|---|
| begin | byName | byRef | cfunction | class |
| const | defined | do | else | end |
| for | function | if | include | is |
| of | return | then | to | type |
| var | while | | | |

## B.11 Textual names for Operator Symbols

The meaning of unary and binary operators in any given context is provided by the implementation of a function with the *textual name* for the operator symbol. The following table gives the various operators and their associated textual name.

*binary operators*

<table>
<tr><th>symbol</th><th>name</th></tr>
<tr><td>+</td><td>plus</td></tr>
<tr><td>-</td><td>minus</td></tr>
<tr><td>*</td><td>times</td></tr>
<tr><td>/</td><td>divide</td></tr>
<tr><td>%</td><td>remainder</td></tr>
<tr><td>&</td><td>and</td></tr>
<tr><td>|</td><td>or</td></tr>
<tr><td><</td><td>less</td></tr>
<tr><td><=</td><td>lessEqual</td></tr>
<tr><td>></td><td>greater</td></tr>
<tr><td>>=</td><td>greaterEqual</td></tr>
<tr><td>==</td><td>sameAs</td></tr>
<tr><td>~=</td><td>notSameAs</td></tr>
<tr><td>=</td><td>equals</td></tr>
<tr><td><></td><td>notEquals</td></tr>
</table>

*unary operators*

| symbol | name |
|---|---|
| ~ | not |
| - | negation |

Functions that define operator symbols are the only function names that can be overloaded (be associated with more than one function body defined in the same scope). Such overloading can only take place at the global scope.

## B.12 Constants

There are three types of constants in Leda. These are string constants, integer constants, and real (or floating-point) constants.

String constants are surrounded by a matching pair of double-quote marks. String constants cannot span multiple input lines, however the following escape conventions can be used to represent special characters:

| *sequence* | *meaning* |
|---|---|
| `\n` | *newline* |
| `\t` | *tab* |
| `\b` | *backspace* |
| `\\` | *backslash* |
| `\"` | *quote mark* |

There is no separate type for character constants. Strings of length 1 can be used in place of character values.

Integer constants consist of a sequence of digit characters. The underlying architecture may impose restrictions on the size of integer constants that can be recognized.

A real (or floating-point) constant consists of a non-empty sequence of digit characters, followed by a fractional part and/or an exponent part. A fractional part consists of a decimal point (period) followed by a non-empty sequence of digit characters. An exponent part consists of the literal character **E** followed by an optional sign and a non-empty sequence of digit characters.