

# Asterix reference manual

John Romein <john@cs.vu.nl>

January 8, 2010

## 1 Introduction

This manual gives a definition of the language Asterix. Asterix is a simple imperative programming language that can be extended to support modern programming paradigms like object orientation and garbage collection.

## 2 Lexical conventions

This section describes the classes of lexical tokens.

### 2.1 White space

Blanks, tabs, new-lines, and comments form white space. White space is ignored except when it separates adjacent tokens. All input between **(\*** and **\*)** is considered a comment. Comments do nest. A program may not end with a non-terminated comment.

### 2.2 Identifiers

An identifier consists of a letter or underscore, followed by any number of letters, underscores, and digits. Identifiers are case sensitive. Identifiers are at least 30 characters significant.

### 2.3 Keywords

The following tokens are keywords and are reserved. They cannot be used as regular identifiers.

<b>and</b>	<b>array</b>	<b>begin</b>	<b>bool</b>	<b>by</b>	<b>char</b>	<b>delete</b>	<b>do</b>	<b>else</b>
<b>end</b>	<b>false</b>	<b>for</b>	<b>function</b>	<b>if</b>	<b>include</b>	<b>int</b>	<b>is</b>	<b>new</b>
<b>not</b>	<b>null</b>	<b>of</b>	<b>or</b>	<b>procedure</b>	<b>real</b>	<b>repeat</b>	<b>return</b>	<b>size</b>
<b>string</b>	<b>then</b>	<b>to</b>	<b>true</b>	<b>until</b>	<b>var</b>	<b>while</b>		

**include** is only a keyword at the beginning of a line.

### 2.4 Integer constants

An integer constant is defined exactly as in ANSI-C (including octal and hexadecimal constants), except that there are no L, l, U, and u suffixes. An integer constant is of type **int** and cannot be

used as lvalue.

## 2.5 Character constants

A character constant is defined exactly as in ANSI-C, except that there is no `L` prefix for wide-character constants. A character constant is of type `char` and cannot be used as lvalue.

## 2.6 Real constants

A real constant is defined exactly as in ANSI-C, except that there are no `F`, `f`, `L`, and `l` suffixes. A real constant is of type `real` and cannot be used as lvalue.

## 2.7 Bool constants

There are two constants of type `bool`: `true` and `false`. They cannot be used as lvalue.

## 2.8 String constants

A string constant is defined exactly as in ANSI-C, except that there is no `L` prefix for wide-character string constants, and that strings are not implicitly terminated by `'\0'`. A string constant is of type `string` and cannot be used as lvalue. Deleting a string constant causes undefined behavior.

## 2.9 The null constant

The value `null` is of the implicit `null` type and denotes a null reference. `null` cannot be used as lvalue.

## 2.10 File inclusion

A file can be included by specifying `include file.inc`; the keyword `include` must begin a new line and may not be preceded by white space. It is recommended but not necessary to end a file with `.inc`. The search path can be modified by using the `-I` compiler switch.

# 3 Types

Asterix is a strongly typed language. There are basic types (`bool`, `char`, `int`, and `real`), and compound types, as described below.

## 3.1 int

A value of type `int` denotes a signed integer. The minimum and maximum values of an `int` depends on the implementation, but the maximum is at least 32767, and the minimum at most -32768.

## 3.2 char

A value of type `char` is in the range `'\000'` to `'\377'`.

### 3.3 real

The range of values of type **real** is machine and implementation dependent.

### 3.4 bool

A value of type **bool** is either **true** or **false**.

### 3.5 array

type → **array of** type<sub>1</sub>

type → **string**

An array is a sequence of values of a base type, type<sub>1</sub>. A variable of an **array** type is a reference to an array, which must be dynamically allocated by a call to the **new** operator, and destroyed with a **delete** statement. The size of an array is not part of its type, but must be specified when using the **new** operator. If the size of the array is  $n$ , the array can be indexed by a value  $0 \leq i < n$ . There are no multi-dimensional arrays, but the base type of an array may be an array. This way even non-rectangular data structures can be composed. **string** is a shorthand for **array of char**. Unlike C strings, Asterix strings are not terminated by '\0'.

### 3.6 The null type

There is an implicit type, the null type. No variables of the null type can be declared, but the **null** constant is of the null type.

### 3.7 Type compatibility

Two expressions are assignment compatible if and only if one of the following rules holds:

- both expressions are of the same basic types,
- both expressions are of the array type with the same (recursive) base type,
- the lvalue is of any array type and the rvalue is of the null type.

## 4 Declarations

program → declarations

declarations → declaration [ declarations<sub>1</sub> ]

declaration → subprogram\_decl | **var** var\_declaration\_list

var\_declaration\_list → var\_declaration ; [ var\_declaration\_list<sub>1</sub> ]

A program consists of at least one declaration. There are two types of declarations: subprogram declarations and variable declarations. The order of declarations is not important: a subprogram or variable may be used before or after it has been textually declared.

## 4.1 Variable declarations

var\_declaration → identifier\_list : type  
 identifier\_list → identifier [ , identifier\_list<sub>1</sub> ]

A `var_declaration` is used to declare a variable in the current scope. Multiple variables can be declared in one declaration.

There are three storage classes: *bss*, *stack*, and *heap* data. There are also three types of variables: *global*, *formal*, and *local*. Whether it is a global, formal, or local variable declaration depends on the context.

**global** A global variable can only be declared in global scope. Its storage class is *bss*. Before `main` is executed, all global variables are initialized according to Table 1. The lifetime of the variable spans the execution of `main`.

**formal** A formal parameter can only be declared in a parameter list. If `identifier_list` is prefixed by the keyword `var`, call by reference semantics is used when the subprogram is invoked, otherwise call by value semantics is used. Its storage class is *stack*. The lifetime spans the execution of the subprogram.

**local** A local variable can only be declared in a compound statement. Its storage class is *stack*. The lifetime spans the execution of the compound statement.

A variable may not be declared multiple times in the same scope; neither may it be redeclared as subprogram or vice versa (variables and subprograms share the same, scoped name space). If it is redeclared in a nested scope, it hides the variable (or subprogram) in the surrounding scope.

A variable has either a value of a basic type, or a reference to an array.

Variables are always implicitly initialized upon declaration. The default value depends on the type, as is shown in Table 1.

type	value
<b>array of</b> any type	<b>null</b>
<b>bool</b>	<b>false</b>
<b>char</b>	<b>'\0'</b>
<b>int</b>	<b>0</b>
<b>real</b>	<b>0.0</b>

Table 1: Default initialization values.

## 4.2 Subprogram declarations

subprogram\_decl → subprogram\_header ; | subprogram\_header **is** statement  
 subprogram\_header → procedure\_header | function\_header  
 procedure\_header → **procedure** name\_and\_formals  
 function\_header → **function** name\_and\_formals : type  
 name\_and\_formals → identifier ( [ formal\_parameters ] )  
 formal\_parameters → formal\_parameter [ ; formal\_parameters<sub>1</sub> ]  
 formal\_parameter → [ **var** ] var\_declaration

There are two types of subprogram declarations: procedures and functions. Functions return a value, while procedures do not. If no **return** statement is executed within a function body, then a compiler or run-time error will occur.

A subprogram declaration may come with or without body. A subprogram may only be declared in the global scope. It may be redeclared, provided that:

- at most one declaration contains a subprogram body;
- all declarations either declare a **procedure**, or a **function**;
- in case of a function declaration, the return types must be the same;
- the number of arguments are the same;
- the formal types are the same, including the calling convention (call by value or call by reference) for that parameter.

If no declaration of a subprogram contains a body, then the function must be a standard library function, or a linker error will occur.

The subprogram **main** is special: the runtime system calls this function to start execution of the program. It must be a function returning an **int** and accepting one call by value argument. The type of the argument must be **array of string** (or, equivalently, **array of array of char**). The actual value is a list of the command line arguments. The array has a size of at least one string; the first string is the name of the executable with which the program was invoked. Deleting the array or one of its elements causes undefined behavior. The return value of **main** is used as exit status of the program. Successful program execution should result in a zero return value; unsuccessful program execution should result in a non-zero return value.

## 5 Expressions

If an operator has multiple operands, the order of evaluation is unspecified except when stated otherwise. The precedence and associativity of the operators are listed in Table 2.

Precedence	Operators	Associativity
1	( ) (function call), [ ] (indexing)	left
2	<b>not</b> , <b>size of</b> , (unary) -, (unary) +	right
3	*, /	left
4	(binary) +, (binary) -	left
5	<, <=, >, >=	non
6	=, <>	left
7	<b>and</b>	left
8	<b>or</b>	left

Table 2: Operator precedence and associativity.

### 5.1 Grouping

expression            → ( expression<sub>1</sub> )

Yields expression<sub>1</sub>. expression<sub>1</sub> must be of any type. expression is of the same type, and may be used as lvalue if expression<sub>1</sub> may be used as lvalue.

## 5.2 Constants

expression	→ integer_constant
expression	→ character_constant
expression	→ real_constant
expression	→ <b>true</b>   <b>false</b>
expression	→ string_constant
expression	→ <b>null</b>

Any constant is an expression, as defined in Sect. 2.

## 5.3 Identifiers

expression	→ identifier
------------	--------------

An identifier can be used as expression. `identifier` must be visible in the current scope. The type of `expression` depends on the declaration of `identifier`. If it is declared as variable, the type is given by its declaration, and `expression` can be used as lvalue. If it is declared as subprogram, the type is of the implicit subprogram type; the only way to use it is in a subprogram invocation; it cannot be used as lvalue then.

## 5.4 The new operator

expression	→ <b>new array</b> [ <code>expression<sub>1</sub></code> ] <b>of</b> type
------------	---

The **new** operator yields a reference to newly allocated heap space for an array. `expression1` must be of type **int**. `expression1` indicates the size (number of elements) of the array; a runtime error occurs if this value is negative. All elements in the array are initialized according to Table 1. The type of `expression` is **array of** type; it may not be used as lvalue.

## 5.5 The index operator

expression	→ <code>expression<sub>1</sub></code> [ <code>expression<sub>2</sub></code> ]
------------	---

Indexes `expression1` with index value `expression2`. The type of `expression1` must be an array of any type, `expression2` must be of type **int**. A runtime error occurs if `expression1` is a null reference, or if not  $0 \leq \text{expression}_2 < \text{size of } \text{expression}_1$ . The type of `expression` is the base type of `expression1` and may be used as lvalue.

## 5.6 Subprogram invocation

expression	→ <code>expression<sub>1</sub></code> ( [ <code>argument_list</code> ] )
<code>argument_list</code>	→ <code>expression</code> [ , <code>argument_list<sub>1</sub></code> ]

Invokes the subprogram `expression1`. `expression1` must be a procedure or function. If `expression1` is a procedure, `expression` has no value, and may not be used as subexpression. If `expression1` is a function, `expression` has the value of the result of invoking `expression1`; its type becomes the return type of the invoked function; it may not be used as lvalue.

The actual argument list must match the formal parameter list, as defined below. The number of actual arguments must be the same as the number of formal parameters. If the formal parameter corresponding to an actual argument is *not* a **var** parameter, the type of the actual argument

must be assignment compatible with the type of the formal parameter; call by value semantics is used for argument passing. If the formal parameter corresponding to an actual argument *is* a **var** parameter, the type of the actual argument must be exactly the same as the type of the formal parameter; the actual argument must be an lvalue; and call by reference semantics is used for argument passing.

## 5.7 The not operator

expression  $\rightarrow$  **not** expression<sub>1</sub>

Yields the logical negation of expression<sub>1</sub>. expression<sub>1</sub> must be of type **bool**. expression is of type **bool**, and may not be used as lvalue.

## 5.8 The size of operator

expression  $\rightarrow$  **size of** expression<sub>1</sub>

Yields the number of elements in expression<sub>1</sub>. expression<sub>1</sub> must be an array of any type. A runtime error occurs if expression<sub>1</sub> is a null reference. expression is of type **int**, and may not be used as lvalue.

## 5.9 The unary minus operator

expression  $\rightarrow$  **-** expression<sub>1</sub>

Yields the result of subtracting expression<sub>1</sub> from 0 or 0.0. expression<sub>1</sub> must be of type **int** or **real**. expression is of the same type, and may not be used as lvalue.

## 5.10 The unary plus operator

expression  $\rightarrow$  **+** expression<sub>1</sub>

Yields expression<sub>1</sub>. expression<sub>1</sub> must be of type **int** or **real**. expression is of the same type, and may not be used as lvalue.

## 5.11 The multiplication operator

expression  $\rightarrow$  expression<sub>1</sub> \* expression<sub>2</sub>

Yields the product of expression<sub>1</sub> and expression<sub>2</sub>. expression<sub>1</sub> and expression<sub>2</sub> must be both of type **int** or both of type **real**. expression is of the same type, and may not be used as lvalue.

## 5.12 The division operator

expression  $\rightarrow$  expression<sub>1</sub> / expression<sub>2</sub>

Yields the quotient of expression<sub>1</sub> and expression<sub>2</sub>. expression<sub>1</sub> and expression<sub>2</sub> must be both of type **int** or both of type **real**. expression is of the same type, and may not be used as lvalue.

## 5.13 The binary minus operator

expression  $\rightarrow$  expression<sub>1</sub> - expression<sub>2</sub>

Yields the difference between `expression1` and `expression2`. `expression1` and `expression2` must be both of type **int** or both of type **real**. `expression` is of the same type, and may not be used as lvalue.

#### 5.14 The binary plus operator

`expression`             $\rightarrow$  `expression1 + expression2`

Yields the sum of `expression1` and `expression2`. `expression1` and `expression2` must be both of type **int** or both of type **real**. `expression` is of the same type, and may not be used as lvalue.

#### 5.15 The smaller than operator

`expression`             $\rightarrow$  `expression1 < expression2`

Yields **true** if `expression1` is smaller than `expression2`. `<` is non-associative, thus `e1 < e2 < e3` results in an error. `expression1` and `expression2` must be both of type **char**, both of type **int**, or both of type **real**. `expression` is of type **bool**, and may not be used as lvalue.

#### 5.16 The smaller or equal operator

`expression`             $\rightarrow$  `expression1 <= expression2`

Yields **true** if `expression1` is smaller than or equal to `expression2`. `<=` is non-associative, thus `e1 <= e2 <= e3` results in an error. `expression1` and `expression2` must be both of type **char**, both of type **int**, or both of type **real**. `expression` is of type **bool**, and may not be used as lvalue.

#### 5.17 The greater than operator

`expression`             $\rightarrow$  `expression1 > expression2`

Yields **true** if `expression1` is greater than `expression2`. `>` is non-associative, thus `e1 > e2 > e3` results in an error. `expression1` and `expression2` must be both of type **char**, both of type **int**, or both of type **real**. `expression` is of type **bool**, and may not be used as lvalue.

#### 5.18 The greater or equal operator

`expression`             $\rightarrow$  `expression1 >= expression2`

Yields **true** if `expression1` is greater than or equal to `expression2`. `>=` is non-associative, thus `e1 >= e2 >= e3` results in an error. `expression1` and `expression2` must be both of type **char**, both of type **int**, or both of type **real**. `expression` is of type **bool**, and may not be used as lvalue.

#### 5.19 The equality operator

`expression`             $\rightarrow$  `expression1 = expression2`

Yields **true** if `expression1` is equal to `expression2`. `expression1` and `expression2` must be both of type **bool**, both of type **char**, both of type **int**, both of type **real**; or one of the operands is an array of any type and the other is either an array of compatible base type or **null**. Two arrays are equal if the references are equal. `expression` is of type **bool**, and may not be used as lvalue.

## 5.20 The unequal operator

expression → expression<sub>1</sub> <> expression<sub>2</sub>

Is equivalent with **not** ( expression<sub>1</sub> = expression<sub>2</sub> ).

## 5.21 The logical and operator

expression → expression<sub>1</sub> **and** expression<sub>2</sub>

Yields **false** if expression<sub>1</sub> evaluates to **false**, or if expression<sub>2</sub> evaluates to **false**. If expression<sub>1</sub> evaluates to **false**, expression<sub>2</sub> is not evaluated. expression<sub>1</sub> and expression<sub>2</sub> must be both of type **bool**. expression is of type **bool**, and may not be used as lvalue.

## 5.22 The logical or operator

expression → expression<sub>1</sub> **or** expression<sub>2</sub>

Yields **true** if expression<sub>1</sub> evaluates to **true**, or if expression<sub>2</sub> evaluates to **true**. If expression<sub>1</sub> evaluates to **true**, expression<sub>2</sub> is not evaluated. expression<sub>1</sub> and expression<sub>2</sub> must be both of type **bool**. expression is of type **bool**, and may not be used as lvalue.

# 6 Statements

## 6.1 The assignment statement

statement → expression<sub>1</sub> := expression<sub>2</sub> ;

The assignment statement assigns the value of expression<sub>2</sub> to expression<sub>1</sub>. expression<sub>1</sub> must be an lvalue. A compile-time error is produced if the type of expression<sub>2</sub> is not assignment compatible to the type of expression<sub>1</sub>. It is not defined which of the expressions is evaluated first (in case the expressions have side effects).

## 6.2 The expression statement

statement → expression ;

The expression statement evaluates expression only for its side effects. The result is not used.

## 6.3 The delete statement

statement → **delete** expression ;

This statement deletes the object referred to by expression. Expression must be of any array type, or **null**.

## 6.4 The if statement

statement → **if** expression **then** statement<sub>1</sub> [ **else** statement<sub>2</sub> ]

The if statement evaluates expression. expression must be of type **bool**. If expression evaluates to **true**, statement<sub>1</sub> is executed. If expression evaluates to **false** and the optional **else** part is given,

`statement2` is executed. The dangling-else ambiguity is resolved by connecting the **else** with the last encountered **else-less if** at the same block nesting level.

## 6.5 The for statement

`statement` → **for** `expression1` := `expression2` **to** `expression3` [ **by** `expression4` ] **do** `statement1`

The for statement repeats `statement1` a specified number of times. `expression1` denotes the loop value, which must be an lvalue; `expression2` the begin value; `expression3` the end value; `expression4` the step size. If the **by** part is omitted, the step size is 1. All expressions must be of type **int**. Expressions 2, 3, and 4 are evaluated only once, before `statement1` is executed. However, the evaluation order of the expressions is not defined. If the begin value is greater than the end value and the step size is positive, or if the begin value is smaller than the end value and the step size is negative, `statement1` is not executed at all. Otherwise, after each iteration of `statement1`, `expression1` is incremented by the step size. Although considered poor programming practice, it is allowed to use `expression1` as lvalue within the for-loop.

## 6.6 The repeat statement

`statement` → **repeat** `statement1` **until** `expression` ;

The repeat statement repeats `statement1` until `expression` evaluates to true. `statement1` is executed at least once. `expression` must be of type **bool**.

## 6.7 The while statement

`statement` → **while** `expression` **do** `statement1`

`statement1` is executed as long as the evaluation of `expression` yields **true**. `expression` must be of type **bool**.

## 6.8 The return statement

`statement` → **return** [ `expression` ] ;

The return statement exits from the current subprogram. If the subprogram is a procedure, `expression` must be omitted. If the subprogram is a function, `expression` must be provided and assignment compatible to the return type of the function. The return value will be the result after evaluating `expression`.

## 6.9 The compound statement

`statement` → [ **var** `var_declaration_list` ] **begin** [ `statement_list` ] **end**

`statement_list` → `statement` [ `statement_list1` ]

The compound statement is both used for grouping statements, and for introducing local variables. If the optional `var_declaration_list` is provided, a new scope is introduced starting from the keyword **var** up to the keyword **end**.

## 7 The standard library

Asterix comes with a small standard library. Prototypes for the functions can found in the appropriate include files.

### 7.1 `conv.inc`

This module provide functions for standard conversions.

```
function StringToInt(str : string) : int;
function IntToString(i : int) : string;
function StringToReal(str : string) : real;
function RealToString(r : real) : string;
function IntToReal(i : int) : real;
function RealToInt(r : real) : int;
function IntToChar(i : int) : char;
function CharToInt(c : char) : int;
```

### 7.2 `io.inc`

This module provides an interface for I/O operations.

```
function WriteChar(ch : char) : bool;
function WriteInt(i : int) : bool;
function WriteLine() : bool;
function WriteReal(r : real) : bool;
function WriteString(s : string) : bool;
```

### 7.3 `random.inc`

```
function Random() : real;
```

This function yields a random real number  $r$ , such that  $0.0 \leq r < 1.0$ .

### 7.4 `string.inc`

```
function StringCopy(str : string) : string;
```