# Bramspr

report on the Compiler Construction final project

H.S.J. Driessens | B.C. Leenders

# Introduction

One of the final courses for the Computer Science bachelor's program at the University of Twente is *Compiler Construction*. After some introductory assignments, the bulk of the course is the final project. In this project, a language needs to be designed and an accompanying language processor, in the form of a compiler, needs to be built.

We have designed a procedural language with an ALL(*)-grammar and a Java-like syntax. Using frameworks ANTLR 4[1] for parser-generation and ASM 4[2] for Java bytecode generation, we have built a corresponding compiler that generates JVM-executables. Naming it after ourselves, and with a nod to the parser generator that we have been making grateful use of during this project, we dubbed this language Bramspr.

This report contains all there is to the language, including a full language specification, a discussion on design choices and problems that arose during the process, code templates for compiling to Java bytecode, an overview of the compiler's software architecture and the testing scheme we have subjected the compiler to.

Note: the specification, grammar, parser, lexer and checker all have support for arrays. However, we unfortunately didn't finish the code generation for them. The rest of this report is written as if arrays were fully implemented.

[1] http://www.antlr.org/
[2] http://asm.ow2.org/

# Bramspr: a brief description

Bramspr is a procedural language with a Java-like syntax. The feature set of Bramspr includes that of the *Basic Expression Language*, augmented with control flow statements (*if* and *while*), functions, procedures, arrays, enumerated types, records and strings, amongst others. Here follows a conspectus of Bramspr's most notable features and concepts.

## Entities

Bramspr knows four kinds of entities: values, variables, functions and types. A variable may be declared as constant. Functions can be overloaded, and may or may not have a return expression. Types can be composite types, array-types, enumerated types or built-in types. Variables, functions and enumerated and composite types must be declared before they can be used.

## Swap

As a handy feature, Bramspr has a swap-operator:

```
x: integer := 4;
y: integer := 10;
x <> y;                  // now x = 10 and y = 4
```

## Constancy

Variables can be declared as constant, which makes their values immutable. Most languages support such a feature. However, in Bramspr, the concept of constancy is broader than that: values, functions and other entities have, in addition to a type, a constancy-property. *An entity is constant if its value can be determined at compile-time.* Using this definition, the context checker follows the 'chain of constancy' and determines what is constant and what is not. This creates an opportunity for optimization of the code by substituting constant entities with their resulting values, speeding up the compiled program. The details of this mechanism are specified further on in this report, as part of the contextual constraints. (Note: we have not actually exploited aforementioned opportunity.)

## Declare anywhere, hide anything

In Bramspr, the four types of declarations (variables, enumerated types, composite types, functions) can be made anywhere statements can be made. It is, for example, possible to declare functions in functions.

Apart from in the same scope, the four types of declarations can be made multiple times for the same identifier, each time hiding the older declarations. This way, even built-in types and functions can be hidden.  For example:

```
type integer {foo: boolean};
x: integer;              // x's type is not the 'normal', built-in integer type
```

## Variadic comparative operators

In most programming languages, such as Java, comparing multiple integers has to be done in parts. Consider the following example of Java code:

```
int x = 8;
boolean xIsBetween5And12 = 5 < x && x < 12;      // xIsBetween5And12 = true
```

To make common comparisons like these less cluttered, Bramspr supports variadic comparative operators:

```
integer ageOfLisa := 19;
integer ageOfPeter := 22;
```

```
boolean perfectConditions = 18 < lisasAge < petersAge < 25;    // true
boolean thisIsFalse := 1 = 4 = 9 = 10 = 3;                     // false
```

## Context-sensitive enumeration shorthand

Normally, denoting a value of an enumeration goes like this:

```
enumeration dateOptions {CINEMA, ROMANTIC_DINNER, PICNIC};
lisasFavorite: dateOptions;
lisasFavorite := enumeration.dateOptions.ROMANTIC_DINNER;
```

As a shorthand, the following is also allowed:

```
lisasFavorite := dateOptions.ROMANTIC_DINNER;
```

However, when there is a name conflict with a composite type or variable, the latter has precedence over the enumeration and the full notation is required to denote the enumeration. This allows for name collisions to be circumvented, so programmers do not have to worry about having variables or types with the same names as enumerations.

# Problems and solutions

## LL(1) versus ALL(*)

The first practical problem we encountered, is that ANTLR 4 does not support the construction of LL(*k*)-grammars. Quoting Sam Harwell, one of the ANTLR 4 developers: "*the entire point of ANTLR 4 is to remove traditional limitations like LL(1) from consideration during development and implementation of a new language.*" [3]

Instead, ANTLR 4 works exclusively with a form of grammar the developers have dubbed Adaptive LL(*), or ALL(*). Because we were interested in exploring ANTLR 4, we have adopted ALL(*). For details on ALL(*), we refer to a paper by Terence Parr, Sam Harwell and Kathleen Fisher: *Adaptive LL(*) Parsing: The Power of Dynamic Analysis*. [4]

## 'Suit' and the chain of constancy

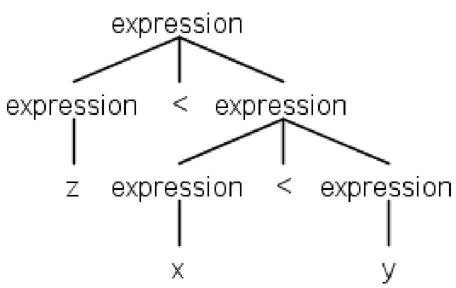
In Bramspr, an entity is constant if its value can be determined at compile-time. With this definition, constancy is a context-sensitive aspect, so it is evaluated in the context checker. It also brings to life the 'chain of constancy': constancy propagates through the context-checking hierarchy. For instance, the expression `x + someArray[someIndex]` yields a constant value if `x`, `someArray` and `someIndex` are all constant.

To keep track of this, it was not enough for `BramsprChecker`'s visit methods to only return a type. We introduced suit: a property of a Bramspr-entity which is a pair of a type and a truth value, the latter indicating the constancy of the entity. For example, the following declaration causes the return suit of `doILikePie` to be *constant boolean*:

```
function doILikePie{
   constant result : boolean := true;
   return result;              // function return types are implicit in Bramspr
}
```

## Parsing variadic operators

Before introducing Arithmetic and Atomic, our grammar did not have a parsing hierarchy for expressions: everything was a direct production rule of Expression. This approach gave problems parsing expressions like `z < x < y` as a single node, as they should be parsed. Using production rules such as Expression ⇒ Expression (< Expression)+, the aforementioned expression would be parsed like this:




[3] http://stackoverflow.com/questions/24001958/antlr4-force-ll1
[4] http://www.antlr.org/papers/allstar-techreport.pdf

We tried augmenting the rules with the ANTLR-property for right-associativity or greediness, which didn't help. Then we realized that variadic operators such as *smaller than* always compare expressions yielding integer values and thus could never have a variadic comparator as operand again (since these yield boolean values). That meant we could separate the arithmetic expressions in the grammar, solving the problem.

The only comparators for which this reasoning didn't hold were *equals to* and *not equals to*, since they can have non-integer-yielding operand expressions. But we introduced the variadic comparators to handily compare multiple arithmetic expressions in the first place, so we argued it would be a reasonable constraint to only have variadic comparators for these expressions and have regular binary comparators for all others. Therefore, we have a separate binary universal-Equals-to-Expression that compares any expression and a variadic Equals-to-Expression for comparing arithmetic expressions.

## Context-sensitiveness of enumerations

Enumerations, using the shorthand notation, cannot be parsed context-freely because the denotation of an enumeration-value is syntactically indistinguishable from the field-access of a composite variable:

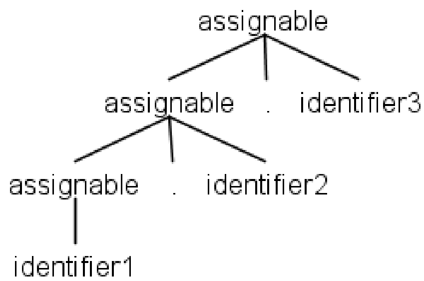
```
x := ambigous.something;      // Value 'something' of enumeration 'ambiguous'?
    // Or variable 'ambiguous' of a composite type containing field 'something'?
```

The relevant parts of the grammar used to be like this:

```
Expression ⇒          Arithmetic ⇒         Atomic ⇒           Assignable ⇒
     ...                   Atomic               ...                  | Assignable . Identifier
     | Arithmetic          | ...                | Assignable         | ...
     | ...                 | ...                | ...                | Identifier
```

Parsing the right-hand part of the following assignment would then yield the following parse tree:

```
x := identifier1.identifier2.identifier3;
```




Context-checking the right-hand part would be done recursively. The context checker would call the visit method of `identifier3`, which would verify its legality and look up its type by requesting `identifier2`'s type, which, in turn, would request `identifier3`'s type. In other words, the context checker can only 'see' two levels at once. As a consequence, the context checker would need to make the decision 'is this an enumeration-value or a field access of a composite?' *at every level*. This should not be necessary: it is evident that, in this example, these are not enumeration-values. After all, the 'chain length' is more than two, while the only ambigous case is Identifier . Identifier.

Instead, we wanted the context checker to only have to make the decision in truly ambigous cases. We identified four potential ways to do this.

### Syntactic predicates

ANTLR provides the option to execute Java code while parsing and make parsing decisions based on that with syntactic predicates. Using these, we could keep track of declared enumerations and variables of composite type, giving us the required information at parse-time. But in that case, some sort of primitive symbol table would need to be maintained during the parsing phase. This would form a burden on parsing and cause the context checker to do redundant work. We also did not like the idea of cluttering our grammar specification with Java code.

### Separate 'special case' production rule

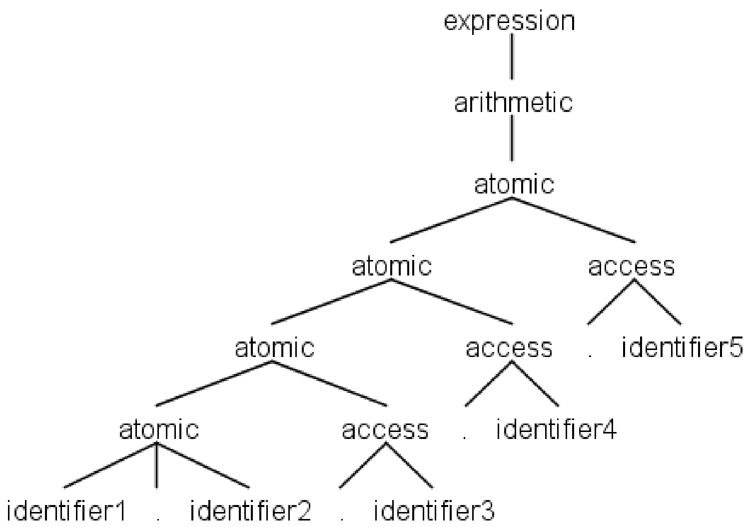
Another option was to add to the grammar a specific production rule for this case:

Atomic ⇒
    ...
    | Identifier **.** Identifier
    | Assignable
    | ...

But this would make parse trees less unifom, as the first two elements of an 'access-chain' would always be parsed as a bundle – even though the only ambiguous case is with a chain of two. For example, consider this phrase:

```
x := identifier1.identifier2.identifier3.identifier4.identifier5;
```

With the special case production rule, this would be the resulting parse tree of the right-hand part:




### Flattening

Yet a different approach is to not parse Identifier**.**Identifier**.** ... **.**Identifier-chains recursively, but flatten them, so they get parsed as one node with all identifiers as children. This makes for uniform parsing as well as easy decision-making in the context checker (before consulting the symbol table, easily check the chain length, for only with a length of 2 there is ambiguity).

The drawback of this approach, however, is that we lose the recursion. The recursion forms a useful level of indirection. For instance, mixed-up field-access occurrences and array-access occurrences, like `pets.dogs[9].age`, are currently parsed homogeneously. With the flattening-approach, the array-access node would 'break up' the chain of the field-access occurrences.

### Molecule

In this approach, an additional level of hierarchy is added to the way expressions are parsed. Currently, the hierarchy is Expression ⇒ Arithmetic ⇒ Atomic. By adding another level between Arithmetic and Atomic, it becomes possible to parse Identifier . Identifier-couples separately while still parsing longer chains the regular way, maintaining recursion. This separates the ambiguous case of two identifiers from the unambiguous cases with only one, or three or more identifiers. We dubbed this new level Molecule, referring to it being 'super-atomic'. This is the approach chosen for Bramspr.

## Mutual left recursion

ANTLR 4 can handle direct left recursion, but not indirect left recursion. Later on, we introduced Assignable as a left-hand side for assignments, but before we did that that, the following production rules were mutually indirect left recursive:

Expression ⇒ Assignment          Assignment ⇒ (Expression :=)+ Expression

We solved this by adding parenthesis to the first rule: Expression ⇒ ( Assignment )

This may look like a hack or a cheat, but it is not. It should have been like this in the first place, for cases such as the following have to be distinguished from each other:

```
x := y := 5 + 6;        // Both x and y become 11.
x := (y := 5) + 6;      // First, y becomes 5, then x becomes 11
```

## Context error handling: `ErrorType` and `Suit.`*`ERROR`*

We wanted to make the context checking errors really sensible and useful. Specifically, we wanted to avoid the following situations:

- just one context error is fatal, causing the whole context checking process to be aborted;
- a context error propagates to other parts of the program, causing the context checking of these parts to raise errors too, obfuscating the actual problem from the programmer.

At first, we wanted the visit methods to return the intended, correct type in case of an error, so the rest of the checking would continue unhindered. However, it cannot always be determined what the intended return type of an erroneous expression ought to be. But we quickly stumbled upon cases in which it wasn't possible to decide the intended type. The compiler should not have to guess, for guessing wrong might cause some very confusing errors, so we created the class `ErrorType` and the static constant `Suit.`*`ERROR`*. These serve as a wildcard type and suit. Comparisons to them (for instance, calling `ErrorType.equals(BramsprChecker.`*`INTEGER`*`)` return true. This solved the problem: the context checking will return one error and then continue checking the rest of the parse tree.

# Bramspr: specification

This section gives a full specification of the Bramspr language. The syntax is formally specified in EBNF notation, while the contextual constrains and semantics are informally described. For readability, EBNF notational elements are colored blue while terminals are **`bolded and in monospace`**.

To prevent the syntax specification from becoming overly verbose, in some parts a shorthand notation is used. In this shorthand notation, the following two grammar specifications have the same meaning:

Some-Non-terminal ⇒
    Some **`ebnf`**? (Production | Rule)    Some-Other-Non-terminal

Some-Non-terminal ⇒
    Some-Other-Non-terminal

Some-Other-Non-terminal ⇒
    Some **`ebnf`**? (Production | Rule)

Although the languages they produce are equivalent, the syntax specification in this report does not completely correspond to the ANTLR syntax specification. This is because in some situations, one way of specifying is conceptually clearer while the other way is programmatically more convenient (for instance by enabling code reuse).

Likewise, the contextual constraints specified for each non-terminal may not always completely match to what is checked in the corresponding visit-methods of `BramsprChecker`, for instance because it is programmatically more convenient if a certain aspect is handled by a different method. The resulting languages, however, are the same.

## Programs

A Program is a manifestation of Bramspr software. It communicates with the user by performing input-output.

### Syntax

Program ⇒ Statement*

### Contextual constraints

A Program is not confined to any contextual constraints.

### Semantics

A Program '$S_1$ ... $S_n$' is run by putting the standard environment into place, opening a new scope, executing each $S_i$ and closing aforementioned scope.

## Statements

Statements are the building blocks of Bramspr and form the smallest executable components.

### Syntax

```
Statement ⇒
        Structure
      | Declaration ;
      | Command ;
      | Function-Call ;
```

### Contextual constraints

A Statement is not confined to any contextual constraints.

### Semantics

A Statement 'S', 'D **;**', 'C **;**' or 'FC **;**' is executed respectively by following S, elaborating D, executing C or performing FC.

## Structures

Structures group statements, control the program flow and dictate the scope hierarchy.

### Syntax

```
Structure ⇒
        { Statement* }                                            Block-Structure
      | if ( Expression ) Block-Structure (else Block-Structure)?  If-Structure
      | while ( Expression ) Block-Structure                      While-Structure

Strucute ➔
        Block-Structure

Block-Structure →
        { Statement* }
```

### Contextual constraints

A Block-Structure '**{** $S_1$ ... $S_n$ **}**' is not confined to any contextual constraints.

In an If-Structure '**if (** E **)** BS' or '**if (** E **)** $BS_1$ **else** $BS_2$' or a While-Structure '**while (** E **)** BS', E must yield a value of type *boolean*.

### Semantics

A Block-Structure '**{** $S_1$ ... $S_n$ **}**' is followed by opening a new scope, executing each $S_i$ in ascending order of i, and closing aforementioned scope.

An If-Structure '**if ( E )** BS' or While-Structure '**while ( E )** BS' is followed as follows. E is evaluated. If its value is *true*, then BS is followed; if its value is *false*, nothing happens.

An If-Structure '**if (** E **)** $BS_1$ **else** $BS_2$' is followed as follows. E is evaluated. If its value is *true*, then $BS_1$ is followed; if its value is *false*, then $BS_2$ is followed.

## Declarations

Declarations define entities in Bramspr-programs and bind symbols, being identifiers or couples of identifiers and parameter signatures, to them.

### Syntax

```
Declaration ⇒
        Function-Declaration
        | Composite-Declaration
        | Enumeration-Declaration
        | Variable-Declaration

Function-Declaration ⇒
        function Identifier (  (Identifier : Type-denoter (, Identifier : Type-denoter)*)? )
                { Statement* (return Expression ;)? }

Composite-Declaration ⇒
        type Identifier { Identifier : Type-denoter (, Identifier : Type-denoter)* }

Enumeration-Declaration ⇒
        enumeration Identifier { (Identifier (, Identifier)*)? }

Variable-Declaration ⇒
        pure-Variable-Declaration
        | instantiating-Variable-Declaration

pure-Variable-Declaration ⇒
        Identifier (, Identifier)* : Type-denoter

instantiating-Variable-Declaration ⇒
        constant? Identifier (, Identifier)* : Type-denoter := Expression
```

### Contextual constraints

In a Function-Declaration '**function** I **(** $I_1$ **:** $T_1$ **,** ... **,** $I_n$ **:** $T_n$ **) {** $S_1$ ... $S_m$ **}**' or '**function** I **(** $I_1$ **:** $T_1$ **,** ... **,** $I_n$ **:** $T_n$ **) {** $S_1$ ... $S_m$ **return** E **; }**':

- I ($T_1$ ... $T_n$) where $T_i$ is the type $T_i$ denotes, must not have already been bound to a function in this scope;
- each $I_i$ must be unique amongst each other.

In a Composite-Declaration '**type** I **{** $I_1$ **:** $T_1$ **,** … **,** $I_n$ **:** $T_n$ **}**':

- I must not have already been bound to a composite type in this scope;
- each $I_i$ must be unique amongst each other.

In an Enumeration-Declaration '**enumeration** I **{** $I_1$ **,** … **,** $I_n$ **}**':

- I must not have already been bound to an enumerated type in this scope;
- each $I_i$ must be unique amongst each other.

In a pure-Variable-Declaration '$I_1$ **,** … **,** $I_n$ **:** T':

- no $I_i$ must have already been bound to a variable in this scope;
- each $I_i$ must be unique amongst each other.

In an instantiating-Variable-Declaration '$I_1$ **,** … **,** $I_n$ **:** T **:=** E' or '**constant** $I_1$ **,** … **,** $I_n$ **:** T **:=** E':

- no $I_i$ must have already been bound to a variable in this scope;
- each $I_i$ must be unique amongst each other;
- E must yield a value of the type denoted by T;
- if there is a **constant**, E must yield a constant value.

## Semantics

A Function-Declaration '**function** I **(** $I_1$ **:** $T_1$ **,** … **,** $I_n$ **:** $T_n$ **)** **{** $S_1$ … $S_m$ **}**' or '**function** I **(** $I_1$ **:** $T_1$ **,** … **,** $I_n$ **:** $T_n$ **)** **{** $S_1$ … $S_m$ **return** E **;** **}**' is elaborated by binding I ($T_1$ … $T_n$) where $T_i$ is the type $T_i$ denotes, to a newly created function with the following properties:

- its parameter signature is $I_1$ $T_1$ … $I_n$ $T_n$;
- its body is $S_1$ … $S_m$;

If there is a **return** E **;**:

- its result expression is E;
- its return suit is the suit of the value yielded by E (E is evaluated to learn this suit).

If instead there is no **return** E **;**:

- the function will have no return expression and the function's return suit will be *constant void*.

This binding stays in effect, and hides previously made bindings of I ($T_1$ … $T_n$) to a function (if there are any), for as long as the current scope is open.

A Composite-Declaration '**type** I **{** $I_1$ **:** $T_1$ **,** … **,** $I_n$ **:** $T_n$ **}**' is elaborated by binding I to a newly created composite type containing, for each $I_i$, a field to which $I_i$ has been bound of the type denoted by $T_i$. This binding stays in effect, and hides previously made bindings of I to a composite type (if there are any), for as long as the current scope is open.

An Enumeration-Declaration '**enumeration** I **{** $I_1$ **,** … **,** $I_n$ **}**' is elaborated by binding I to a newly created enumerated type with, for each $I_i$, a value to which $I_i$ has been bound. This binding stays in effect, and hides previously made bindings of I to an enumerated type (if there are any), for as long as the current scope is open.

A pure-Variable-Declaration '$I_1$ **,** … **,** $I_n$ **:** T', '$I_1$ **,** … **,** $I_n$ **:** T **:=** E' is elaborated by binding each $I_i$ to a newly created uninitialized variable of type *non-constant T*, where *T* is the type denoted by T.

These bindings stay in effect, and hide previously made bindings of each $I_i$ to a variable (if there are any), for as long as the current scope is open.

A an instantiating-Variable-Declaration '**constant** $I_1$ , … , $I_n$ : T := E' is elaborated by binding each $I_i$ to a newly created variable of the type denoted by T, initialized with the value yielded by E (E is evaluated to learn this value) If there is a **constant**-keyword, the constancy of these variables is *constant*. Otherwise, it is *non-constant*. These bindings stay in effect, and hide previously made bindings of each $I_i$ to a variable (if there are any), for as long as the current scope is open.

## Commands

Commands update variables with new values.

### Syntax

Command ⇒

| | |
|---|---|
| (Assignable **:=**)+ Expression | Assignment |
| \| Assignable **<>** Assignable | Swap |

### Contextual constraints

For an Assignment '$A_1$ **:=** … $A_n$ **:=** E':

- each $A_i$ must be non-constant;
- the type of each $A_i$ must be the type of the value yielded by E;
- the yielded suit is that of E.

For a Swap '$A_1$ **<>** $A_2$':

- $A_1$ and $A_2$ must not be constant;
- $A_1$ and $A_2$ must be of the same type.

### Semantics

An Assignment '$A_1$ **:=** … $A_n$ **:=** E' is executed as follows:

- E is evaluated to yield a value;
- each $A_i$ is resolved;
- for each $A_i$, the entity referred to is updated with this value;
- for each $A_i$, if it was previously uninitialized, it is now initialized;
- the value is yielded.

A Swap '$A_1$ **<>** $A_2$' is executed as follows.

- $A_1$ and $A_2$ are resolved;
- the entities referred to by $A_1$ and $A_2$ switch values.

## Function calls

Function calls invoke parameterized parts of a Bramspr program.

### Syntax

Function-Call ⇒ Identifier **(** (Expression (**,** Expression)*)? **)**

### Contextual constraints

In a Function-Call 'I **(** $E_1$ **,** … **,** $E_n$ **)**':

- I ($T_1$ … $T_n$), where $T_i$ is the type of the value yielded by $E_i$, must have been bound to a function;
- the suit of the Function-Call will be the return suit of the last function I ($T_1$ … $T_n$) has been bound to.

### Semantics

A Function-Call is performed as follows.

- each $E_i$ is evaluated to yield a value;
- the current environment is replaced with the environment of the last function I ($T_1$ … $T_n$) has been bound to (meaning only the bindings that were in effect at the time of the declaration of this function are now in effect);
- a new scope is opened;
- for each $I_i$ $T_i$ in the parameter signature of this function, $I_i$ is bound to a newly created variable of type $T_i$, initialized with the value yielded by $E_i$;
- each $S_i$ of the body of this function is executed in ascending order of i;
- if the function has one, its return expression E is evaluated and the yielded value is yielded;
- the current environment is replaced with the environment as before the Function-Call.

## Assignables

Assignables are references to entities that can be assigned values. They include variables, array-elements and composite-fields.

### Syntax

Assignable ⇒

| | |
|---|---|
| \| Identifier | basic-Assignable |
| \| Assignable **[** Expression **]** | Array-access-on-Assignable |
| \| Assignable **.** Identifier | Field-access-on-Assignable |

### Contextual constraints

In a basic-Assignable 'I', I must have been bound to a variable. The suit of the basic-Assignable is that of the referred variable.

In an Array-access-on-Assignable 'A **[** E **]**', A must be array-typed and E must yield a value of type *integer*. The suit of the Array-access-on-Assignable is of the type of the referred field, and it is constant if both A and E are constant.

In a Field-access-on-Assignable 'A **.** I', A must be of composite type and the composite type must contain a field to which I has been bound. The suit of the Field-access-on-Assignable is of the type of the referred field, and of the constancy of A.

### Semantics
When a basic-Assignable 'I' is resolved, it refers to the last variable that I has been bound to.

An Array-access-on-Assignable 'A **[** E **]**' is resolved as follows. First, A is resolved; then, E is evaluated to yield an *integer* value n; lastly, the Array-access-on-Assignable refers to the nth element of the array-typed entity referred to by A.

A Field-access-on-Assignable 'A **.** I' is resolved as follows. First, A is resolved; then, the Field-access-on-Assignable refers to the field of the entity to which A refers to which I has been bound.

## Type-denoters
Type-denoters denote a certain data type. This includes array-types, composite types and enumerated types.

### Syntax
Type-denoter ⇒

| | |
|---|---|
| Identifier | base-Type-denoter |
| \| **[** Number **]** Type-denoter | Array-Type-denoter |
| \| **enumeration .** Identifier | Enumerated-Type-denoter |

### Contextual constraints
In a base-Type-denoter 'I', at least one of the following must be the case:

1. I must have been bound to a composite type;
2. I must have been bound to an enumerated type.

An Array-Type-denoter '**[** N **]** T' is not confined to any contextual constraints.

In an Enumerated-Type-denoter '**enumeration .** I', I must have been bound to an enumerated type.

### Semantics
A base-Type-denoter 'I' denotes the last composite type I has been bound to. If I has not been bound to a composite type, it denotes the last enumerated type I has been bound to.

An Array-Type-denoter '**[** N **]** T' denotes the type [*n*]*T*, where *T* is the type T refers to and *n* is N interpreted as a decimal integer.

An Enumerated-Type-denoter '**enumeration .** I' denotes the last enumerated type I has been bound to.

# Expressions

Expressions are phrases that yield values.

To achieve the right precedence during parsing, there is a hierarchy to how expressions are structured syntactically. On top, there are comparative and logical expressions. They are composed of arithmetic expressions. One level deeper are molecules, and lastly there are the atomics, which form the elemental building blocks of expressions.

## Syntax

```
Expression ⇒
      ! Expression                                 Not-Expression
    | Arithmetic                                   Arithmetic-Expression
    | Arithmetic (= Arithmetic)+                   Equals-to-Expression
    | Arithmetic (=/= Arithmetic)+                 Not-equals-to-Expression
    | Expression = Expression                      universal-Equals-to-Expression
    | Expression =/= Expression                    universal-Not-equals-to-Expression
    | Arithmetic = Arithmetic +- Arithmetic        Plus-minus-Expression
    | Arithmetic (> Arithmetic)+                   Greater-than-Expression
    | Arithmetic (>= Arithmetic)+                  Greater-than-Equals-to-Expression
    | Arithmetic (< Arithmetic)+                   Smaller-than-Expression
    | Arithmetic (<= Arithmetic)+                  Smaller-than-Equals-to-Expression
    | Expression & Expression                      And-Expression
    | Expression | Expression                      Or-Expression

Arithmetic ⇒
      Molecule                                     Molecule-Expression
    | (+ | - ) Arithmetic                          Sign-Expression
    | Arithmetic ^ Arithmetic                      Power-Expression
    | Arithmetic ( * | / | % ) Arithmetic          Multiplication-Expression
    | Arithmetic ( + | - ) Arithmetic              Addition-Expression

Molecule ⇒
      Identifier . Identifier                      potential-Enumeration-Literal
    | Atomic                                       Atomic-Expression

Atomic ⇒
      ( Assignment )                               Assignment-Expression
    | ( Expression )                               Parenthesis-Expression
    | Assignable                                   Assignable-Expression
    | Function-Call                                Function-Call-Expression
    | Literal                                      Literal-Expression
    | Atomic [ Expression ]                        Array-access-on-Atomic
    | Atomic . Identifier                          Field-access-on-Atomic
```

## Contextual constraints

An Arithmetic-Expression ‘A’ is not confined to any context restraints. Its suit is that of the value yielded by A.

In an Expression ‘$A_1$ **=** … **=** $A_n$’, ‘$A_1$ **=/=** … **=/=** $A_n$’, ‘$A_1$ **=** $A_2$ **+-** $A_3$’, ‘$A_1$ **>** … **>** $A_n$’, ‘$A_1$ **>=** … **>=** $A_n$’, ‘$A_1$ **<** … **<** $A_n$’, or ‘$A_1$ **<=** … **<=** $A_n$’, each $A_i$ must yield a value of type *integer*. The suit of the

Expression will be of type *boolean*, and it constancy will be *constant* if and only if each $A_i$ yields a constant value.

In an Expression '**!** E', '$E_1$ **&** $E_2$' or '$E_1$ **|** $E_2$', each E or $E_i$ must yield a value of type *boolean*. The suit of the Expression will be of type *boolean*, and it constancy will be *constant* if and only if each $A_i$ yields a constant value.

In an Expression '$E_1$ **=** $E_2$' or '$E_1$ **=/=** $E_2$', both $E_1$ and $E_2$ must yield a value of the same type. The suit of the Expression will be of type *boolean*, and it constancy will be *constant* if and only if each $A_i$ yields a constant value.

A Molecule-Expression 'M' is not confined to any context restraints. Its suit is that of the value yielded by M.

In an Arithmetic '**+** A', '**−** A', '$A_1$ **^** $A_2$', '$A_1$ ***** $A_2$', '$A_1$ **/** $A_2$', '$A_1$ **%** $A_2$', '$A_1$ **+** $A_2$' or '$A_1$ **–** $A_2$', each A or $A_i$ must yield a value of type *integer*. The suit of the Arithmetic will be of type *integer*, and it constancy will be *constant* if and only if each $A_i$ yields a constant value.

In a potential-Enumeration-Literal '$I_1$ **.** $I_2$', one of the following must be the case:

1. $I_1$ must have been bound to a variable of composite type and the composite type must contain a field bound to $I_2$. The suit of the potential-Enumeration-Literal will then be that of the referred field;
2. $I_1$ must not have been bound to a variable of composite type; $I_1$ must have been bound to an enumerated type and that type must contain a value bound to $I_2$. The suit of the potential-Enumeration-Literal will then be of that enumerated type and *constant*.

An Atomic-Expression 'A' is not confined to any contextual restraints. Its suit is that of the value yielded by A.

An Atomic '**(** A **)**', '**(** E **)**', 'A', 'FC' or 'L' is not confined to any context restraints. Its suit is that of the value yielded by, respectively, A, E, A, FC or L.

In an Array-access-on-Atomic 'A **[** E **]**', A must be array-typed and E must yield a value of type *integer*. The suit of the Atomic is of the type of the referred array element, and of the constancy of A.

In a Field-access-on-Atomic 'A **.** I', A must be of composite type and the composite type must contain a field bound to I. The suit of the Atomic is of the type of the referred field, and of the constancy of A.

## Semantics

An Arithmetic-Expression 'A' is evaluated by evaluating A and yielding its value.

An Equals-to-Expression '$A_1$ **=** … **=** $A_n$' is evaluated as follows. First, $A_1$ and $A_2$ are evaluated (in that order) to yield a value. Then, *false* is yielded if these are equal. If they are equal, $A_3$ is evaluated and its value is compared with that of $A_2$, and so on until $A_n$ is evaluated and its value is compared with that of $A_{n-1}$. If these are equal as well, *true* is yielded.

A Not-equals-to-Expression '$A_1$ **=/=** … **=/=** $A_n$' is evaluated as follows. First, $A_2$ and $A_1$ are evaluated to yield a value. Then, *false* is yielded if these are equal. If they are unequal, $A_3$ is evaluated and its value is compared with that of $A_2$, and so on until $A_n$ is evaluated and its value is compared with that of $A_{n-1}$. If these are unequal as well, *true* is yielded.

A Plus-minus-Expression '$A_1$ **=** $A_2$ **+−** $A_3$' is evaluated as follows. First, each $A_i$ is evaluated to yield a value. Then, *true* is yielded if the value yielded by $A_1$ lies in the range [value yielded by $A_2$ - value yielded by $A_3$; value yielded by $A_2$ + value yielded by $A_3$], inclusive. Otherwise, *false* is yielded.

A Greater-than-Expression '$A_1$ **>** … **>** $A_n$', is evaluated as follows. First, $A_2$ and $A_1$ are evaluated to yield a value. Then, *false* is yielded if the algebraic equation value of $A_2 >$ value of $A_1$ does not hold. If it does hold, $A_3$ is evaluated and its value is compared with that of $A_2$, and so on until $A_n$ is evaluated and its value is compared with that of $A_{n-1}$. If the equation holds for these values as well, *true* is yielded.

A Greater-than-Equals-to-Expression '$A_1$ **>=** … **>=** $A_n$', is evaluated as follows. First, $A_2$ and $A_1$ are evaluated to yield a value. Then, *false* is yielded if the algebraic equation value of $A_2 \geq$ value of $A_1$ does not hold. If it does hold, $A_3$ is evaluated and its value is compared with that of $A_2$, and so on until $A_n$ is evaluated and its value is compared with that of $A_{n-1}$. If the equation holds for these values as well, *true* is yielded.

A Smaller-than-Expression '$A_1$ **<** … **<** $A_n$', is evaluated as follows. First, $A_2$ and $A_1$ are evaluated to yield a value. Then, *false* is yielded if the algebraic equation value of $A_2 <$ value of $A_1$ does not hold. If it does hold, $A_3$ is evaluated and its value is compared with that of $A_2$, and so on until $A_n$ is evaluated and its value is compared with that of $A_{n-1}$. If the equation holds for these values as well, *true* is yielded.

A Smaller-than-Equals-to-Expression '$A_1$ **<=** … **<=** $A_n$', is evaluated as follows. First, $A_2$ and $A_1$ are evaluated to yield a value. Then, *false* is yielded if the algebraic equation value of $A_2 \leq$ value of $A_1$ does not hold. If it does hold, $A_3$ is evaluated and its value is compared with that of $A_2$, and so on until $A_n$ is evaluated and its value is compared with that of $A_{n-1}$. If the equation holds for these values as well, *true* is yielded.

A Not-Expression '**!** E' is evaluated as follows. First, E is evaluated to yield a value of type *boolean*. Then, if that value is *true*, *false* is yielded. Otherwise, *true* is yielded.

An And-Expression '$E_1$ **&** $E_2$' is evaluated as follows. First, $E_1$ and $E_2$ are evaluated to yield values of type *boolean*. Then, if these values are both *true*, *true* is yielded. Otherwise, *false* is yielded.

An Or-Expression '$E_1$ **|** $E_2$' is evaluated as follows. First, $E_1$ and $E_2$ are evaluated to yield values of type *boolean*. Then, if these one of these values is *true*, *true* is yielded. Otherwise, *false* is yielded.

A universal-Equals-to-Expression '$E_1$ **=** $E_2$' is evaluated as follows. First, $E_1$ and $E_2$ are evaluated to yield a value. Then, if these values are the same, *true* is yielded. Otherwise, *false* is yielded. Array-typed values are said to be the same if, for each value i in their index ranges, every ith element of both array-values is the same. Values of composite types are said to be the same if, for each of their fields, the value of that field is the same for both composite values.

A universal-Not-equals-to-Expression '$E_1$ **=/=** $E_2$' is evaluated as follows. First, $E_1$ and $E_2$ are evaluated to yield a value. Then, if these values are the same, *false* is yielded. Otherwise, *true* is yielded.

A Molecule-Expression 'M' is evaluated by evaluating M and yielding its value.

A Sign-Expression '**+** A' is evaluated by evaluating A and yielding its value.

A Sign-Expression '**−** A' is evaluated by first evaluating A to yield the value x, and then yielding the value of the algebraic expression -x.

A Power-Expression '$A_1$ **^** $A_2$' is evaluated by first evaluating $A_1$ and $A_2$ to yield values $x$ and $y$ respectively, and then yielding the value of the algebraic expression $x^y$. However, if that value exceeds $2^{31} - 1$, $2^{31} - 1$ will be yielded instead.

A Multiplication-Expression '$A_1$ ***** $A_2$' is evaluated by first evaluating $A_1$ and $A_2$ to yield values $x$ and $y$ respectively, and then yielding the value of the algebraic expression $xy$.

A Multiplication-Expression '$A_1$ **/** $A_2$' is evaluated by first evaluating $A_1$ and $A_2$ to yield values $x$ and $y$ respectively, and then yielding the value of the algebraic expression $\lfloor x \div y \rfloor$. This will cause a runtime error if $A_2$ yields the value *0*.

A Multiplication-Expression '$A_1$ % $A_2$' is evaluated by first evaluating $A_1$ and $A_2$ to yield values $x$ and $y$ respectively, and then yielding the value of the algebraic expression $x \bmod y$.

An Addition-Expression '$A_1$ **+** $A_2$' is evaluated by first evaluating $A_1$ and $A_2$ to yield values $x$ and $y$ respectively, and then yielding the value of the algebraic expression $x + y$.

An Addition-Expression '$A_1$ **–** $A_2$' is evaluated by first evaluating $A_1$ and $A_2$ to yield values $x$ and $y$ respectively, and then yielding the value of the algebraic expression $x - y$.

An Assignable-Expression 'A' is evaluated by resolving A and yielding the value of the entity it refers to. If that entity is uninitialized, depending on its type, the following things happen:

- if its type is *integer*, value *0* will be yielded;
- if its type is *character*, value *a* will be yielded;
- if its type is *string*, the empty string will be yielded;
- it its type is *boolean*, value *false* will be yielded;
- if its type is composite, enumerated or an array-type, this will cause a run-time error.

A potential-Enumeration-Literal '$I_1$ **.** $I_2$', is evaluated in one of the following ways:

1. if $I_1$ is bound to a variable of composite type, the potential-Enumeration-Literal is evaluated by yielding the value of the field of that variable to which $I_2$ has been bound;
2. else, the potential-Enumeration-Literal is evaluated by yielding the value of the enumerated type to which $I_1$ has been bound, that has $I_2$ bound to it.

An Atomic '**(** A **)**', '**(** E **)**', 'A', 'FC' or 'L' is evaluated by, respectively, executing A and yielding the yielded value, evaluating E and yielding the yielded value, resolving A and yielding the value of the entity that it refers to, performing FC and yielding the result, or yielding the value of L.

An Array-access-on-Atomic 'A **[** E **]**' is evaluated as follows. First, A is evaluated; then, E is evaluated to yield an *integer* value n; lastly, the value of the nth element of the array-typed value yielded by A is yielded.

A Field-access-on-Atomic 'A **.** I' is evaluated as follows. First, A is evaluated; then, the value of the field of the composite typed value yielded by A that has I bound to it is yielded.

## Literals

Literals directly denote a value of a certain type.

### Syntax

Literal ⇒

| | |
|---|---|
| Number | Integer-Literal |
| \| Character | Character-Literal |
| \| String | String-Literal |
| \| Boolean | Boolean-Literal |
| \| **[** (Expression (**,** Expression)*)? **]** | Array-Literal |
| \| Identifier **{** Identifier **:=** Expression (**,** Identifier **:=**Expression)* | Composite-Literal |
| \| **enumeration .** Identifier **.** Identifier | explicit-Enumeration-Literal |

### Contextual constraints

A Literal 'N', 'C', 'S' or 'B' is not confined to any contextual constraints. Their suits are, respectively, *constant integer*, *constant character, constant string* or *constant boolean*.

In an Array-Literal '**[** $E_1$ **,** … **,** $E_n$ **]**':

- each $E_i$ must be of the same type;
- the type of the Literal is [n]*T*, where *T* is the type of each $E_i$;
- the Literal is constant if every $E_i$ is constant.

In a Composite-Literal 'I **{** $I_1$ **:=** $E_1$ **,** … **,** $I_n$ **:=** $E_n$ **}**':

- I must have been bound to a composite type;
- for each $I_i$, the last composite type to which I has been bound must contain a field to which $I_i$ is bound;
- for each field of the last composite type to which I has been bound, there must be a $I_i$ that is bound to that field;
- each $I_i$ must be unique amongst each other;
- for each $I_i$, $E_i$ must be of the type of the field $I_i$ is bound to;
- the type of the Literal is the last type that I has been bound to;
- the literal is constant if every $E_i$ is constant.

In an explicit-Enumeration-Literal '**enumeration .** $I_1$ **.** $I_2$':

- $I_1$ must have been bound to an enumerated type;
- $I_2$ must have been bound to a value of the last enumerated type $I_1$ has been bound to;
- the suit of the Literal is constant, and of the type of the last enumerated type $I_1$ has been bound to.

## Semantics

The value of an Integer-Literal 'N' is the number N interpreted as a decimal integer.

The value of a Character-Literal 'C' is the graphic character C.

The value of a String-Literal 'S' is the string S.

The value of Literal 'B' is the truth value B.

The value of Literal '`[` $E_1$ `,` … `,` $E_n$ `]`' is an array value with the values yielded by $E_1$ … $E_n$ as elements.

The value of Literal 'I `{` $I_1$ `:=` $E_1$ `,` … `,` $I_n$ `:=` $E_n$ `}`' is a composite value where, for each $I_1$, the field to which $I_i$ is bound to is updated with the value yielded by $E_i$.

The value of Literal '**`enumeration`** `.` $I_1$ `.` $I_2$' is the value to which $I_2$ has been bound to of the last enumeration that $I_1$ has been bound to.

## Lexicon

Program ⇒ (Token | Annotation | Whitespace)*

Token ⇒ Number | Character | String | Boolean | Keyword | Identifier | Operator | Interpunction

Number ⇒ Digit+
Character ⇒ ' Graphic '
String ⇒ " Graphic+ "
Boolean⇒ **`true`** | **`false`**
Keyword ⇒ **`if`** | **`then`** | **`else`** | **`while`** | **`function`** | **`type`** | **`enumeration`**
| **`return`** | **`constant`**
Identifier ⇒ Letter (Letter | Digit)*
Operator ⇒ `:=` | `<>` | `+` | `-` | `*` | `/` | `%` | `<` | `<=` | `>` | `>=` | `=` | `=/=` | `+-` | `^` | `!` | `&` | `|`
Interpunction ⇒ `:` | `;` | `(` | `)` | `{` | `}` | `[` | `]` | `,` | `.`

Annotation ⇒ Comment | Block-Comment
Comment ⇒ `//` (Graphic | Blank)* End-of-line
Block-Comment ⇒ `/*` (Graphic | Whitespace)* `*/`

Whitespace ⇒ Blank | End-of-line
Blank ⇒ tab | space
End-of-line ⇒ carriage-return | newline | form-feed

Letter: `a` | `b` | `c` | `d` | `e` | `f` | `g` | `h` | `i` | `j` | `k` | `l` | `m` | `n` | `o` | `p` | `q` | `r` | `s` | `t` | `u` | `v` | `w`
| `x` | `y` | `z` | `A` | `B` | `C` | `D` | `E` | `F` | `G` | `H` | `I` | `J` | `K` | `L` | `M` | `N` | `O` | `P` | `Q` | `R` | `S`
| `T` | `U` | `V` | `W` | `X` | `Y` | `Z`

Digit: `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Graphic: any UTF-16 character. Matches non-greedy.

## Standard environment

Bramspr comes with a few built-in types and functions to which identifiers have been bound by default. The set of these bindings is called the *standard environment*. The programmer can hide these bindings by declaring other types and functions with the same identifiers.

### Types

Bramspr's built-in primitive types are implemented as composite types without fields.

| name | value set | identifier |
|---|---|---|
| *boolean* | The truth values. | **`boolean`** |
| *integer* | All integers ranging from $-2^{31}$ to $2^{31}$-1, inclusive. | **`integer`** |
| *character* | Any UTF-16 character. | **`character`** |
| *string* | Any concatenation of UTF-16 characters. | **`string`** |

In addition, as implicit built-in types, there is the type set of array types [*n*]*T*, where *n* is any number between 0 and $2^{31}$-1, inclusive, and T is any other type in Bramspr, including array types. A value of a type [*n*]*T* has an index range whose lower bound is zero and whose upper bound is one less than *n*. Such an array value has one element of type *T* for each value in its index range.

### Functions

| identifier | parameter signature | return suit | effect |
|---|---|---|---|
| **`getBool`** | | *non-constant boolean* | Returns type-matching console input. |
| **`getInt`** | | *non-constant integer* | Returns type-matching console input. |
| **`getChar`** | | *non-constant character* | Returns type-matching console input. |
| **`getString`** | | *non-constant string* | Returns type-matching console input. |
| **`putInt`** | *integer* | *constant void* | Prints the argument to the console. |
| **`putChar`** | *character* | *constant void* | Prints the argument to the console. |
| **`putBool`** | *boolean* | *constant void* | Prints the argument to the console. |
| **`putString`** | *string* | *constant void* | Prints the argument to the console. |

For **`getBool()`**, **`getInt()`**, **`getChar()`**, if the user enters input that is not type-matching, this will cause a run-time error. (If, for a certain application, this is not acceptable, the programmer can easily fix this by hiding these default functions and make a custom implementation using **`getString()`**.)

# Bramspr to Java bytecode: code generation

The following terminology will be used in describing the translation of Bramspr source code to Java bytecode (JBC). The term *side-effects* denotes either updating variables or performing input-output.

| phrase class | code function | effect of generated JBC |
|---|---|---|
| Program | run P | Run P and halt, starting and finishing with an empty stack. |
| Structure | follow S | Follow S. Does not expand or shrink the stack but may have side-effects. |
| Command | execute C | Execute C. Does not expand or shrink the stack but may have side-effects. |
| Expression | evaluate E | Evaluate E, pushing its result on the stack. May have side-effects if E contains a function call or an assignment. |
| Literal | interpret L | Interpret L, pushing its value on the stack. Does not cause side-effects. |
| Assignable | load A | Push the value of A on the stack top. |
| Assignable | store A | Pop the top value from the stack and store it in assignable A. |

For Java 8 and above, stack map frames are required. The Bramspr Compiler is compatible with Java 8. So while they are handled by in the Bramspr Compiler, note that we do not discuss stack map frames in this section; they are an implementation detail but otherwise do not add functionality to the language. For the details on this aspect of the implementation, see the Bramspr Compiler source code.[5]

[5] *http://chrononsystems.com/blog/java-7-design-flaw-leads-to-huge-backward-step-for-the-jvm*
For Java 6 and below, stack map frames are optional or unimplemented. Java 7 can compile with the `-XX:-UseSplitVerifier` option to avoid keeping track of stack map frames.

## Program

Executing a Program does not leave additional values on the stack upon termination.

execute [$S_1$ ... $S_n$] =
    execute $S_1$
    ...
    execute $S_n$

## Statements

Executing a Statement does not leave additional values on the stack upon termination. However, since Assignments leave a value on the stack, these values have to be popped.

execute [S] =
    execute S

execute [D `;`] =
    execute D

execute [C `;`] =
    execute C

execute [A `;`] =
    execute A
    `POP`

## Structures

Following a Block-Structure first executes all the statements inside, and then adds a label that denotes the end of the scope.

follow[ **`{`** $S_1$ ... $S_n$ **`}`** ] =
    for all $S_i$
        execute $S_i$
end:     this label is the end of the scope

Following an If-Structure opens a new scope for the if- and else-blocks, but since these are Block-Structures, this is taken care of implicitly. Note that there are small differences between an If-Structure with and without else-block. This difference is made in order to save a jump.

follow[**`if (`** E **`)`** BS ] =
    evaluate E
    `IFEQ e1`     jumps over the if-block if E yields a value equal to *false*
    follow BS
e1:

follow[**`if (`** E **`)`** $BS_1$ **`else`** $BS_2$] =
    evaluate E
    `IFEQ e1`     jumps over the if-block if E yields a value equal to *false*
    follow $BS_1$
    `GOTO e2`     jumps over the else-block
e1:     follow $BS_2$
e2:

A While-Structure starts with an unconditional jump to the Expression, and jumps back up each time it evaluates it and *true* is yielded.

```
follow[while ( E ) BS ] =
        GOTO e2
e1:     follow BS
e2:     evaluate E
        IFNE e1                 jump back up if E yields a value not equal to false
```

# Declarations

Declaring functions or enumerated types does not produce any bytecode and does not modify the stack. Both functions and enumerated types are inlined, so only an applied occurrence of an enumeration or function call adds bytecode.

## Composite declarations

Declaring a composite type creates a new inner class with fields matching the fields of the composite type. This new inner class is written to a separate file, as Java does not allow the bytecode of an inner class to be in the same file as its outer class. To avoid this problem, the code generator does not return a single piece of bytecode, but multiple pieces, each of which should be written to a separate file.

For example, if a user defines two types, for instance `Person{name:string}` and `Chair{legCount:integer}`, the compiler will generate three classes: a main class (`myProgram.class`) and two classes for the types (`myProgram$CA.class` and `myProgram$CB.class`). The main class will reference to the two other classes if the types are used in the program. The class files can also refer to each other, if a user-defined type type has a field containing another user-defined type, for instance `Surface{length:integer, depth:integer}` and `Room{size:Surface}`.

It should be noted that the names of the fields in the compiled class and the Bramspr source code are equal. The name of the type, however, is changed to avoid naming collisions: the compiler guarantees a unique name for every type declaration, even if the names of two types are equal.

## Scopes of visibility

Declaring a variable requires the scope of the variable to be defined. Upon declaration, a label s is added to indicate the beginning of this variable's visibility. Another label e is added at the end of every scope to indicate the end of its visibility. At the end of the compilation, the scope of every variable is defined using these two labels.

# Commands

Swap loads the values of the Assignables $A_1$ and $A_2$ and stores them in the same chronological order, effectively switching the values of $A_1$ and $A_2$, leaving nothing on the stack.

execute [$A_1$ **<>** $A_2$] =
    load $A_1$
    load $A_2$
    assign $A_1$
    assign $A_2$

Assignment evaluates the Expression E, putting one value on the stack. It then duplicates and stores that value for every Assignable, constantly leaving a single value on the stack. Since Commands cannot expand the stack, they pop the top value from the stack once the value of the Expression is stored in each Assignable.

```
execute[A1 := … An := E] =
        evaluate E
        for all Ai
                DUP
                store Ai
        POP
```

## Function calls

If a function declared as I**(**$A_1$ **,** … **,** $A_n$**)** **{** $S_1$ … $S_m$ **return** $E_r$**}** is called, the Function-Call is translated as follows:

```
evaluate[I ( E1 , … , En) ] =
        for all Ei
                evaluate Ei
                store Ai
        for all Si
                execute Si
        evaluate Er
```

Note that $E_r$ is evaluated, and a value is thus left on the stack, if and only if the function has a return statement.

## Expressions

The variadic comparators Equals-to-Expression, Not-equals-to-Expression, Greater-than-Expression, Greater-than-Equals-to-Expression, Smaller-than-Expression and Smaller-than-Equals-to-Expression evaluate a variable amount of Expressions that yield values of type *integer*. The evaluation is lazy: once a single comparison fails, the Expression immediately yields *false*.

Note that we compare using the negation of a comparator. If the comparison tests equality (=), then the JBC compares using `IF_ICMPNE` (≠). The same holds for the other comparators: inequality (≠) in Bramspr source code is translated to `IF_ICMPEQ` (=), greater-than (>) to `IF_ICMPLE` (≤), greater-than-equals-to (≥) to `IF_ICMPLT` (<), smaller-than (<) to `IF_ICMPGE` (≥) and smaller-than-equals-to (≤) to `IF_ICMPGT` (>). This is because the jump statement should only be executed when *false* is encountered. In the example below, the situation is shown in case of an Equals-to-Expression.

```
evaluate[A1 = A2 = … = An] =
        evaluate A1                     load the value (or reference to value) of A1
        repeat for all An where n > 1
                evaluate An             load the value (or reference to value) of An
                DUP_X1                  copy the value of An and push it two places down
                IF_ICMPNE g             if An ≠ An-1 jump to label g
        POP
        ICONST_1                        load true on the stack
        GOTO h
  g:    POP
        ICONST_0                        load false on the stack
  h:
```

Evaluating a Plus-minus-Expression first evaluates the two Expressions on the right side, and then computes the bounds in which the tested value must be to yield true. All Expressions are evaluated only once. Because this is a slightly more complex piece of code, there are comments on the right that show what the stack looks like after evaluating the command on the left. (In these comments, '$E_i$' means 'value yielded by $E_i$'.

evaluate[$E_1$ **=** $E_2$ **+-** $E_3$] =

```
        evaluate E2           E2
        evaluate E3           E2  E3
        DUP2                  E2  E3  E2  E3
        ISUB                  E2  E3  (E2 - E3)
        DUP_X2                (E2 - E3)  E2  E3  (E2 - E3)
        POP                   (E2 - E3)  E2  E3
        IADD                  (E2 - E3)  (E2 + E3)
        evaluate E1           (E2 - E3)  (E2 + E3)  E1
        DUP_X1                (E2 - E3)  E1  (E2 + E3)  E1
        IF_ICMPLT e1          (E2 - E3)  E1
        IF_ICMPGT e2
        ICONST_1              true
        GOTO e3
e1:     POP
e2:     ICONST_0              false
e3:                           true/false
```

Evaluating an Addition-Expression, Multiplication-Expression (except for the division), Or-Expression or And-Expression generates very similar bytecode:

evaluate[$E_1$ <operator> $E_2$] =

```
        evaluate E1
        evaluate E2
        IADD/ISUB/IMUL/IDIV/IREM/IOR/IAND
```

A dividing Multiplication-Expression is slightly more complex as it includes a check whether the divisor is zero. If it is, a reference to `System.err` is loaded, an error message is printed and the execution halts:

evaluate[$E_1$ **/** $E_2$] =

```
        evaluate E1
        evaluate E2
        DUP
        IFNE d
        GETSTATIC "java/lang/System" "err" "Ljava/io/Printstream"
        string literal: "Divide by zero error: a/b.\nExiting program.\n"
        INVOKEVIRTUAL"java/io/PrintStream"  "println"  "(Ljava/lang/String;)V"
        ICONST_1
        INVOKESTATIC "java/lang/System"  "exit"  "(I)V"
d:      IDIV
```

Since there is no native JVM command for exponentiation, a Java library method is used to translate a Power-Expression:

evaluate[$E_1$ **^** $E_2$] =

```
evaluate E1
I2D
evaluate E2
I2D
INVOKESTATIC "java/lang/Math" "pow" "(DD)D"
D2I
```

A Sign-Expression is translated by first evaluating the Expression and negating the resulting value if the sign is a minus.

evaluate[**-** $E_1$] =

```
evaluate E1
INEG
```

evaluate[**+** $E_1$] =

```
evaluate E1
```

Translating a Not-Expression is done evaluating the Expression and negating the *boolean* result by XOR'ing it with *true*, since the JVM has no native NOT-operator:

evaluate[**!** $E_1$] =

```
evaluate E1
ICONST_1
IXOR
```

## Function calls

As stated before, the code of a function is inlined when a function is called. This means all the code in the function's body is added at the place of a Function-Call.

For a function declared as "$I_f$ **(** $I_1$ **:** $T_1$ **,** ... **,** $I_n$ **:** $T_n$ **)** **{** $S_1$ ... $S_n$ **return** $E_r$ **}**", a Function-Call later in the program is evaluated as follows:

evaluate[ $I_f$ **(** $E_1$, ..., $E_n$ **)** ] =

```
    for all En
        evaluate Ei
        assign Ei to Ii
    for all Si
        execute Si
    evaluate Er
end                                     label denoting end of the scope
```

If the function is not user-defined, but provided in the standard environment (getInt, putInt, getBool, putBool, getChar, putChar, getString, putString), the following code is added:

execute[**putString ()**] =

```
GETSTATIC   "java/lang/System"   "out"   "Ljava/io/PrintStream;"
INVOKEVIRTUAL   "java/io/PrintStream"  "print"   (Ljava/lang/String;)V"
```

execute [**putChar ()**] =

```
GETSTATIC   "java/lang/System"   "out"   "Ljava/io/PrintStream;"
INVOKEVIRTUAL   "java/io/PrintStream"  "print"   (C)V"
```

execute [**putBool ()**] =

```
GETSTATIC   "java/lang/System"   "out"   "Ljava/io/PrintStream;"
INVOKEVIRTUAL   "java/io/PrintStream"  "print"   (Z)V"
```

execute [**putInt ()**] =

```
GETSTATIC   "java/lang/System"   "out"   "Ljava/io/PrintStream;"
INVOKEVIRTUAL   "java/io/PrintStream"  "print"   (I)V"
```

The **get**...**()** functions all create an instance of `Scanner` (from the Java libraries) and use it to parse `System.in`.

evaluate[**getString ()**] =

```
NEW "java/util/Scanner"
DUP
GETSTATIC "java/lang/System" "in" "Ljava/io/InputStream;"
INVOKESPECIAL "java/util/Scanner" "<init>" "(Ljava/io/InputStream;)V"
INVOKEVIRTUAL "java/util/Scanner" "nextLine" "()Ljava/lang/String;"
```

evaluate[**getBool ()**] =

```
NEW "java/util/Scanner"
DUP
GETSTATIC "java/lang/System" "in" "Ljava/io/InputStream;"
INVOKESPECIAL "java/util/Scanner" "<init>" "(Ljava/io/InputStream;)V"
INVOKEVIRTUAL "java/util/Scanner" "nextBool" "()Z;"
```

evaluate[**getInt ()**] =

```
NEW "java/util/Scanner"
DUP
GETSTATIC "java/lang/System" "in" "Ljava/io/InputStream;"
INVOKESPECIAL "java/util/Scanner" "<init>" "(Ljava/io/InputStream;)V"
INVOKEVIRTUAL "java/util/Scanner" "nextInt" "()I;"
```

evaluate[**getChar ()**] =

```
NEW   "java/util/Scanner"
DUP
GETSTATIC "java/lang/System" "in" "Ljava/io/InputStream;"
INVOKESPECIAL "java/util/Scanner" "<init>" "(Ljava/io/InputStream;)V"
INVOKEVIRTUAL "java/util/Scanner" "next" "()Ljava/lang/String;"
ICONST_0
INVOKEVIRTUAL "java/lang/String" "charAt" "(I)C"
```

Note that enumerations cannot be printed, for that would introduce "magic numbers" (random values with no clear semantics other than uniqueness.

## Assignables

The compiler differentiates between JVM primitives (*integer*, *boolean* and *character*) and non-primitives (*string* and user-defined types). The memory address of a variable (or field) contains a value of a primitive or a reference to a non-primitive. If the reference points to a user-defined type, the object pointed to must have fields, since *string* is the only non-primitive without fields.

Note that array access is not implemented in the code generator, thus has no code templates.

There are two modes for looking up a value/reference; one for reading and one for writing. When accessing in read mode, the value (which may be a reference) is put on the stack. When accessing in write mode, a reference to the value (which may be a reference to a reference) is put on the stack.

If the compiler is in write mode, for example if a variable is being assigned, the following code templates are used:

evaluate[ I ] = `ALOAD <memory address>`

A field access (either on an Assignable or on an atomic such as **`foo()`**) changes the access mode to read (so a field access on a field access will run once in write mode and once in read mode), and uses the following template:

evaluate[ A . $I_2$ ] = evaluate A

Once the compiler is in read mode, for example if a variable is resolved, the following code templates are used:

evaluate[ I ] = `ILOAD <memory address>` if the variable is JVM a primitive
evaluate[ I ] = `ALOAD <memory address>` if the variable is not JVM a primitive

evaluate[ A . I ] =
evaluate A
`GETFIELD <typeDescriptor > I <fieldTypeSignature>`

During the code generation, `<typeDescriptor>` is replaced by a string such as `"Bramspr$CB"`, and `<fieldTypeSignature>` is replaced by a string such as `"I"` (if the field type is *integer*), `"Ljava/lang/String;"` (if the field type is *string*) or `"Bramspr$CA"` (if the field type is user-defined).

## Arrays (general)

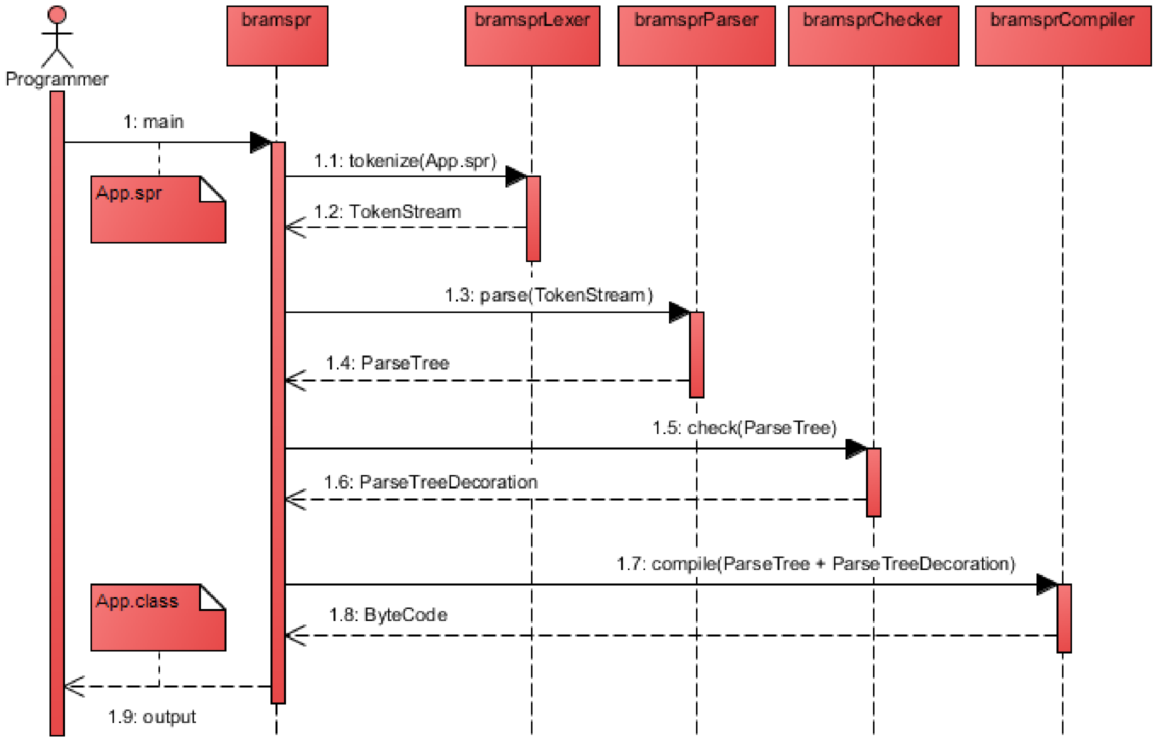
As stated in the introduction, arrays are only implemented in the grammar, parser and checker. The code generator does not support arrays in any form (literals, access, et cetera). This also means that we cannot provide code templates in this document, for they are not specified in the compiler itself. Usage of arrays in Bramspr will, with the current version of the compiler, not produce any bytecode.

# Bramspr Compiler: Java-architecture

Here follows a review of the Bramspr Compiler's software architecture. Only a very high-level overview is given. The Bramspr Compiler documentation is elaborate and the source code is richly commented, so for details please consult these.

## Driver, parser, context checker, code generator

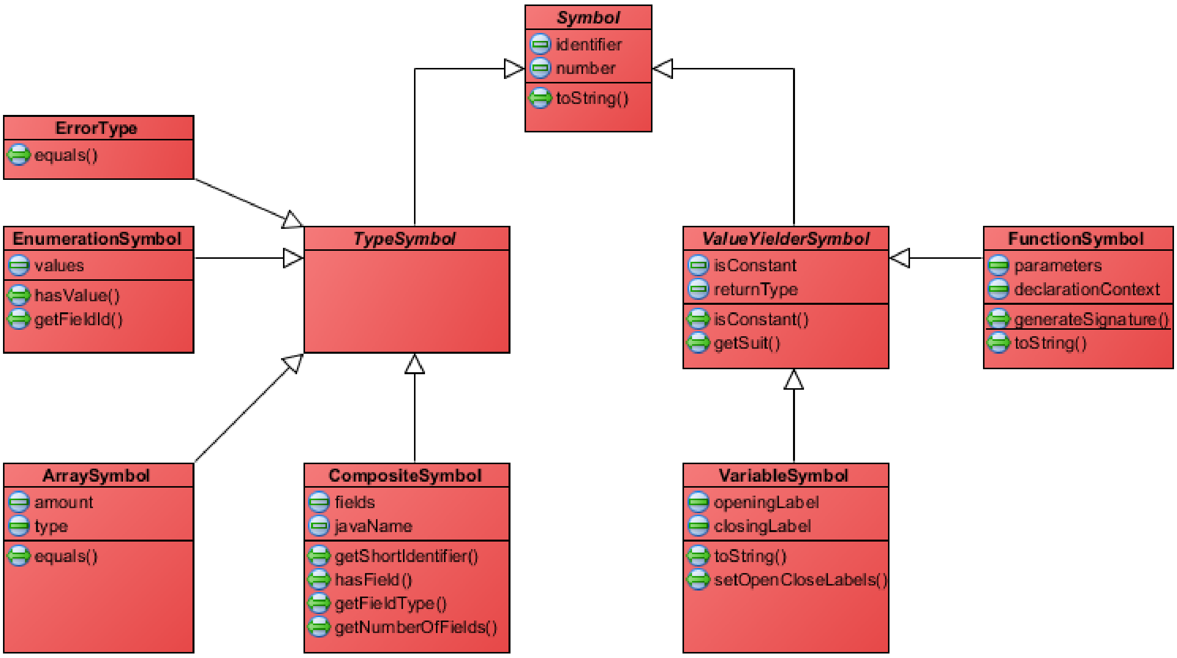
Below is schematically shown how a Bramspr program travels through the compiler and is ultimately translated to a JBC-file. (The denoted 'method calls' and arguments are not actual methods, but slightly simplified conceptualizations.)




The parse tree decoration is of class `ParseTreeProperty<Symbol>` and is a runtime library class of ANTLR 4. Since ANTLR 4 does not work with ASTs anymore, this is considered to be the best practice for storing and communicating decorative information. In essence, it is a collection that maps `ParseTree`-objects to `Symbol`-objects. In the Bramspr Compiler, it is used to decorate the parse tree with contextual information: when `BramsprChecker` encounters a variable, type or function, it links the corresponding `Symbol`-object of that entity (which is created during the visitation of its declaration) to the parse tree node it is currently visiting. The parse tree decoration is passed to `BramsprCompiler`, so the compiler has access to all the required context information while generating object code.

## Symbol tables and the Symbol-hierarchy

`BramsprChecker` keeps track of four `SymbolTable<S extends Symbol>`-objects, parameterized with respectively `EnumerationSymbol`, `CompositeSymbol`, `FunctionSymbol` and `VariableSymbol`. `Symbol`-objects contain information associated with the declaration of an entity in a Bramspr-program, such that the information can later be referenced when visiting an applied occurrence of such an entity. The Symbol-classes are organized in a hierarchy, in which each level adds more specific information. In addition, there is an `ErrorType`-class that is returned in case of a contextual error. For further details, we refer to the documentation of these classes.




## Type checking (or actually, suit checking)

`BramsprChecker` does suit-checking by comparing `Suit`-objects, which are the return values of the checker's visit methods. A `Suit`-object has a type property, in the form of a `TypeSymbol`-object, and a constancy property, in the form of a `boolean`. `TypeSymbol`-objects can be compared with their `equals`-method.

# Bramspr Compiler: testing scheme

For testing the Bramspr compiler, we have complied with the following scheme:

- one correct master test program, covering all Bramspr's features;
- one 'correct' test program covering parsing and context checking of arrays;
- several incorrect test programs covering all possible run-time errors;
- one incorrect test program covering all possible context errors;
- one incorrect test program covering some common syntactical errors.

To keep things organized, we have bundled tests with the same purpose in one file as much as possible, but of course for testing run-time errors this was not an option.

The separate test program for arrays was included to show that, except for code generation, arrays are supported as well. The program intentionally causes a context error before finishing, to prevent the compiler from attempting code generation.

For the master test program, called `ScoobyTesting.spr`, we have developed a simple Bramspr testing framework to be able to easily do successive unit tests in an organized manner. It contains functions such as `test(encountered: string, expected: string, description: string)`, keeps track of the number of errors and ends with printing a conclusion. The code for this framework can be found in the first 75 lines of `ScoobyTesting.spr`.

All tests produced the right output.

| testing purpose | | file |
|---|---|---|
| Correctness of all features except arrays | | `ScoobyTesting.spr` |
| Correctness of parsing and context checking of arrays | | `Arrays.spr` |
| Run-time errors: | | |
| | Division by zero | `DivZero.spr` |
| | Uninitialized variables | `Uninitialized.spr` |
| | Wrong input **`getInt()`** | `OopsNoInt.spr` |
| | Wrong input **`getBool()`** | `OopsNoBool.spr` |
| | Wrong input **`getChar()`** | `OopsNoChar.spr` |
| Context errors | | `SomeContextPlease.spr` |
| Syntactical errors | | `MidnightProgramming.spr` |

# Conclusions

Designing and implementing a new programming language and formally defining its behavior is no trivial task. Nevertheless, we have managed to make a basic language with some nice additional features such as the Plus-minus-Expression and comparators with plural quantification.

The Bramspr compiler is capable of compiling the not-so-widely used Bramspr language to the very-widely used Java Virtual Machine. Because of its support for the latest JVM features (including the stack map frame), users of the Bramspr language and compiler can be sure of support of their compiled code.

Three (respectively four) years ago, we started at the University of Twente as two n00bs that had trouble installing the JDK on a laptop. Now, nearing the end of our bachelors, we developed our own compiler capable of compiling to that very JDK.

It is truly amazing to look back on both this project and our bachelors to see what an improvement we have made. In just a few weeks, we have revisited most of what we have learnt in our bachelors, made links between the various courses we followed and learning objectives of these courses. We got an opportunity to fully rethink programming concepts we have been using for years, oblivious of all the complexity that was underneath. A roommate of mine, studying technical medicine, walked in last week for some small talk while I was busy programming for Bramspr. She asked what I was doing, so I tried to explain. Her reaction: “Wow. Most of the time when people talk about their courses, I’m able to follow what they’re busy with at least in broad terms, but I don’t even remotely understand what this is about.”

Leaving the university, armed with our own compiler, all that remains to be said is the following: happy coding!

# Appendix A: ANTLR lexer specification

```
lexer grammar BramsprLexer;

/* Operators. */
BECOMES:                ':=';
SWAP:                   '<>';
PLUS:                   '+';
MINUS:                  '-';
MULTIPLICATION:         '*';
DIVISION:               '/';
MODULUS:                '%';
SMALLER_THAN:           '<';
SMALLER_THAN_EQUALS_TO: '<=';
GREATER_THAN:           '>';
GREATER_THAN_EQUALS_TO: '>=';
EQUALS_TO:              '=';
NOT_EQUALS_TO:          '=/=';
PLUSMINUS:              '+-';
POWER:                  '^';
NOT:                    '!';
AND:                    '&' ;
OR:                     '|' ;

/* Keywords. */
IF:                     'if';
THEN:                   'then';
ELSE:                   'else';
WHILE:                  'while';
FUNCTION:               'function';
TYPE:                   'type';
ENUMERATION:            'enumeration';
RETURN:                 'return';
CONSTANT:               'constant';

/* Interpunction. */
COLON:                  ':'     ;
SEMICOLON:              ';'     ;
LEFT_PARENTHESIS:       '('     ;
RIGHT_PARENTHESIS:      ')'     ;
LEFT_BRACE:             '{'     ;
RIGHT_BRACE:            '}'     ;
LEFT_BLOCKBRACE:        '['     ;
RIGHT_BLOCKBRACE:       ']'     ;
COMMA:                  ','     ;
DOT:                    '.'     ;

// Een apostrof, gevolgd door geescapete apostrofes en niet-specialchars.
// De *? (i.t.t. *) maakt hem niet-greedy, dus bij de eerste " stopt hij.
BOOLEAN: 'true' | 'false';
STRING : '"' ( ESCAPED | ~('\n'|'\r') )*? '"';
CHARACTER : '\''  ( '\\\'' | ~('\n'|'\r') )*? '\'';
IDENTIFIER: LETTER (LETTER | DIGIT)*;
NUMBER: DIGIT+;

/* Annotations. */
COMMENT: '//' ~[\r\n\u000C]* -> skip; // Matcht alles wat na // komt
BLOCKCOMMENT: '/*' .*? '*/' -> skip; // Matcht alles (op een non-greedy manier)
tussen /* en */
```

```
WHITESPACE : ( '\t' | ' ' | '\r' | '\n'| '\u000C' )+ -> skip ;

/* Fragments. */
fragment DIGIT: ('0'..'9');
fragment LETTER: ('a'..'z'|'A'..'Z');
fragment ESCAPED: '\\"' | '\\\\' ; // Dit zijn \" en \\. Met andere woorden; " en
\ zoals ze binnen "" staan.
```

# Appendix B: ANTLR parser specification

```
grammar Bramspr;
options { tokenVocab=BramsprLexer; }

program: statement*;

structure: blockStructure
         | ifStructure
         | whileStructure
         ;

blockStructure:     LEFT_BRACE statement* RIGHT_BRACE;
ifStructure:        IF LEFT_PARENTHESIS expression RIGHT_PARENTHESIS blockStructure (ELSE blockStructure)?;
whileStructure:     WHILE LEFT_PARENTHESIS expression RIGHT_PARENTHESIS blockStructure;

statement : structure
          | declaration     SEMICOLON
          | command         SEMICOLON
          | functionCall    SEMICOLON
          ;

command : assignment
        | swap
        ;

declaration: compositeDeclaration
           | functionDeclaration
           | enumerationDeclaration
           | variableDeclaration
           ;

variableDeclaration:
       IDENTIFIER (COMMA IDENTIFIER)* COLON typeDenoter                             # pureVariableDeclaration
      | CONSTANT? IDENTIFIER (COMMA IDENTIFIER)* COLON typeDenoter BECOMES expression      # instantiatingVariableDeclaration
      ;

enumerationDeclaration: ENUMERATION IDENTIFIER LEFT_BRACE (IDENTIFIER (COMMA IDENTIFIER)*)? RIGHT_BRACE;
```

```
functionDeclaration:    FUNCTION IDENTIFIER
                        LEFT_PARENTHESIS
                           (IDENTIFIER COLON typeDenoter (COMMA IDENTIFIER COLON typeDenoter)*)?
                        RIGHT_PARENTHESIS
                        LEFT_BRACE
                           statement*
                           (RETURN expression SEMICOLON)?
                        RIGHT_BRACE
                   ;

compositeDeclaration:   TYPE IDENTIFIER
                        LEFT_BRACE
                            IDENTIFIER COLON typeDenoter (COMMA IDENTIFIER COLON typeDenoter)*
                        RIGHT_BRACE
                    ;

typeDenoter: IDENTIFIER                                                     # baseTypeDenoter
           | LEFT_BLOCKBRACE NUMBER RIGHT_BLOCKBRACE typeDenoter            # arrayTypeDenoter
           | ENUMERATION DOT IDENTIFIER                                     # enumeratedTypeDenoter
           ;

assignment: (assignable BECOMES)+ expression;
swap:       assignable SWAP assignable;
```

```
expression: NOT expression                                                    # notExpression
          | arithmetic                                                        # arithmeticExpression
          | arithmetic (EQUALS_TO arithmetic)+                                # equalsToExpression
          | arithmetic (NOT_EQUALS_TO arithmetic)+                            # notEqualsToExpression
          | expression EQUALS_TO expression                                   # universalEqualsToExpression
          | expression NOT_EQUALS_TO expression                               # universalNotEqualsToExpression
          | arithmetic EQUALS_TO arithmetic PLUSMINUS arithmetic              # plusMinusExpression
          | arithmetic (GREATER_THAN arithmetic)+                             # greaterThanExpression
          | arithmetic (GREATER_THAN_EQUALS_TO arithmetic)+                   # greaterThanEqualsToExpression
          | arithmetic (SMALLER_THAN arithmetic)+                             # smallerThanExpression
          | arithmetic (SMALLER_THAN_EQUALS_TO arithmetic)+                   # smallerThanEqualsToExpression
          | expression AND expression                                         # andExpression
          | expression OR expression                                          # orExpression
          ;

arithmetic: molecule                                                          # moleculeExpression
          | (PLUS | MINUS) arithmetic                                         # signExpression
          | arithmetic POWER <assoc=right> arithmetic                         # powerExpression
          | arithmetic ( MULTIPLICATION | DIVISION | MODULUS ) arithmetic     # multiplicationExpression
          | arithmetic ( PLUS | MINUS ) arithmetic                            # additionExpression
          ;

molecule : IDENTIFIER DOT IDENTIFIER                                          # potentialEnumerationLiteral
         | atomic                                                             # atomicExpression
         ;

atomic : LEFT_PARENTHESIS assignment RIGHT_PARENTHESIS                        # assignmentExpression
       | LEFT_PARENTHESIS expression RIGHT_PARENTHESIS                        # parenthesisExpression
       | assignable                                                           # assignableExpression
       | functionCall                                                         # functionCallExpression
       | literal                                                              # literalExpression
       | atomic access                                                        # accessOnAtomicExpression
       ;
```

```
assignable: assignable access                                              # accessOnAssignable
          | IDENTIFIER                                                     # basicAssignable
          ;

access : DOT IDENTIFIER                                                    # fieldAccess
                 | LEFT_BLOCKBRACE expression RIGHT_BLOCKBRACE             # arrayAccess
                 ;

functionCall: IDENTIFIER LEFT_PARENTHESIS (expression ( COMMA expression)*)? RIGHT_PARENTHESIS
            ;

literal : NUMBER                                                                      # integerLiteral
        | CHARACTER                                                                   # characterLiteral
        | STRING                                                                      # stringLiteral
        | BOOLEAN                                                                     # booleanLiteral
        | LEFT_BLOCKBRACE (expression (COMMA expression)*)? RIGHT_BLOCKBRACE          # arrayLiteral
        | IDENTIFIER LEFT_BRACE IDENTIFIER BECOMES expression
           (COMMA IDENTIFIER BECOMES expression)* RIGHT_BRACE                         # compositeLiteral
        | ENUMERATION DOT IDENTIFIER DOT                                              # explicitEnumerationLiteral
        ;
```

# Appendix C: source code of master test program (`ScoopyTesting.spr`)

```
/*
    First, define some functions that we want to use later on in the program (library-like)
*/

function println()                   { putChar('\n');                 };
function println(toPrint: string)    { putString(toPrint); println(); };
function println(toPrint: integer)   { putInt(toPrint);    println(); };
function println(toPrint: character){ putChar(toPrint);   println(); };
function println(toPrint: boolean)   { putBool(toPrint);   println(); };

/*
    Define the Scooby test functions
*/
errorCount: integer := 0;
testCount: integer := 0;

function test(encountered: integer, expected: integer, description: string) {
    if(expected =/= encountered) {
        errorCount := errorCount + 1;
        println(description);
        putString("    Expected: "); putInt(expected);
        putString(", but encountered: "); putInt(encountered);
        println();
    }
    testCount := testCount + 1;
};

function test(encountered: character, expected: character, description: string) {
    if(expected =/= encountered) {
        errorCount := errorCount + 1;
        println(description);
        putString("    Expected: "); putChar(expected);
        putString(", but encountered: "); putChar(encountered);
        println();
    }
```

```
    testCount := testCount + 1;
};

function test(encountered: boolean, expected: boolean, description: string) {
    if(expected =/= encountered) {
        errorCount := errorCount + 1;
        println(description);
        putString("   Expected: "); putBool(expected);
        putString(", but encountered: "); putBool(encountered);
        println();
    }
    testCount := testCount + 1;
};

function test(encountered: string, expected: string, description: string) {
    if(expected =/= encountered) {
        errorCount := errorCount + 1;
        println(description);
        putString("   Expected: "); putString(expected);
        putString(", but encountered: "); putString(encountered);
        println();
    }
    testCount := testCount + 1;
};

function end() {
    putString("Tests executed: "); println(testCount);
    putString("Fail-O-Meter: "); println(errorCount);

    if (errorCount = 0) {
        println("Nice work Scoob!");
    }
    if (0 < errorCount < 5) {
        println("No Shaggy, we're going to Solve this mystery!");
    }
    if (5 <= errorCount) {
        println("This place makes me so nervous, all I can think of is food!");
    }
```

```
};

/*
    The framework and helper functions are now defined.
    Open a new scope, to let the user hide whatever he/she
    wants without name-collisioning with the functions/variables above.
*/
{

    {   // Mathematics
        test(1+1, 2, "1+1 = 2");
        test(2^0, 1, "2^0 = 1");
        test(2^3, 8, "2^3 = 8");
        test(4^3^2, 262144, "4^3^2 = 4^(3^2)");
        test((4^3)^2, 4096, "(4^3)^2");
        test(10^10^10, 2147483647, "Googleplex is replaced by 2^31-1");
        test(10-5, 5, "10-5");
        test(10-50, -40, "10-50");
    }

    {   // Logic
        test(true | false, true, "true | false = true");

        x: integer := 1000;
        y: integer := 1000000;
        { // < operator
            test(1 < 2, true, "1 < 2");
            test(2 < 1, false, "2 < 1");
            test(1 < 2 < (2^3) < 1000, true, "1 < 2 < (2^3) < 1000");
            test(1 < 2 < (2^3) < x < x+1, true, "1 < 2 < (2^3) < x + 1");
            test(10 < 2 < 2^3 < x, false, "10 < 2 < 2^3 < x");
            test(1 < 2 < 3 < 4 < 5 < 6 < 7, true, "1 < 2 < 3 < 4 < 5 < 6 < 7");
        }
        { // <= operator
            test(1 <= 2, true, "1 <= 2");
            test(2 <= 1, false, "2 <= 1");
            test(1 <= 2 <= (2^3) <= 1000, true, "1 <= 2 <= (2^3) <= 1000");
            test(1 <= 2 <= (2^3) <= x <= x+1, true, "1 <= 2 <= (2^3) <= x + 1");
```

```
        test(10 <= 2 <= 2^3 <= x, false, "10 <= 2 <= 2^3 <= x");
        test(1 <= 2 <= 3 <= 4 <= 5 <= 6 <= 7, true, "1 <= 2 <= 3 <= 4 <= 5 <= 6 <= 7");

        test(1 <= 4 <= 4 <= 10, true, "1 <= 4 <= 4 <= 10");
        test(1 <= 1 <= 4 <= 10, true, "1 <= 1 <= 4 <= 10");
        test(1 <= 4 <= 4 <= 10 <= 2, false, "1 <= 4 <= 4 <= 10 <= 2");
    }
    { // > operator
        test(2 > 1, true, "2>1");
        test(1 > 1, false, "1>1");
        test(2 > 1 > -10, true, "2>1>-10");
    }
    { // >= operator
        test(2 >= 1, true, "2>=1");
        test(1 >= 1, true, "1>=1");
        test(2 >= 1 >= -10, true, "2>=1>=-10");
        test(-2 >= 1 >= -10, false, "-2>=1>=-10");
        test(2 >= 1 >= 10, false, "2>=1>=10");
    }
}

{
    // Declare various types as constants
    constant ONE: integer := 1;
    constant ILikeIcecream: boolean := true;
    constant HELLO: string := "Hello, world!";
    constant INITIAL: character := 'B';

    test(ONE, 1, "ONE");
    test(ILikeIcecream, true, "ILikeIcecream");
    test(HELLO, "Hello, world!", "HELLO");
    test(INITIAL, 'B', "INITIAL");
}

{ // Exponentiation overflow test
    constant MAXINT: integer := 2147483647; // 2^31 - 1
    test(2^30 - 1 + 2^30, MAXINT, "MAXINT");
}
```

```
{ // Simple assignments
    x, y, z: integer;
    z := y := (x:= 5) * 10; // = 15
    test(x, 5, "x, after 'z := y := (x:= 5) * 10;'");
    test(y, 50, "y, after 'z := y := (x:= 5) * 10;'");
    test(z, 50, "z, after 'z := y := (x:= 5) * 10;'");

    constant X: integer := 123;
}

{ // Fancy assignments
    w, x, y, z: integer;

    w := (x := y := (z := 10 * 2) + 1)*2;
    test(w, 42, "w in fancy assignment");
    test(x, 21, "x in fancy assignment");
    test(y, 21, "y in fancy assignment");
    test(z, 20, "z in fancy assignment");
}

{ // While statement
    x: integer := 10;
    i: integer := 0;

    while(i < x) {
        i := i + 1;
    }

    // Now they are equal!
    test(i, x, "i = x after while loop");
}

{   // More complicated testing; lazy evaluation!
    isExecuted: boolean := false;
    function setTrue() { isExecuted := true; return 100; };

    // Dump data to this variable
```

```
        sinkhole: boolean;

        // This should execute lazy; setTrue should not be called.
        sinkhole := 1 < 10 < 2 < setTrue();
        test(isExecuted, false, "'1 < 10 < 2 < setTrue()' was not lazy evaluated");

        // reset
        isExecuted := false;
        sinkhole := 1 < 10 < setTrue() < 10000 < 10;
        test(isExecuted, true, "setTrue() was not evaluated (but should hav been)");
    }

    { // Advanced function testing: hide functions!
        function getTrue() {
            return true;
        };

        test(getTrue(), true, "calling the original getTrue()");

        {
            // This one hides the previous one...
            function getTrue() {
                // But we can still access the previous one, for this one is not yet declared.
                inverse: boolean := !getTrue();
                return false;
            };

            test(getTrue(), false, "calling getTrue(), hiding the original");

            { // We can even change the return type:
                function getTrue() {
                    return 42;
                };
                test(getTrue(), 42, "getTrue() hidden twice!");
            }
        }

        test(getTrue(), true, "calling the original getTrue() again");
```

```
}

{ // Asking user input (not using Scooby framework)
    putString("Please enter a boolean (true/false): ");
    b: boolean := getBool();
    putString("You entered: "); println(b);

    putString("Please enter a character: ");
    c: character := getChar();
    putString("You entered: "); println(c);

    putString("Please enter a number: ");
    i: integer := getInt();
    putString("You entered: "); println(i);

    putString("Please enter a string: ");
    s: string := getString();
    putString("You entered: "); println(s);
}

{ // This is quite a large test... It has multiple types and tests setting/getting fields
    type Stoel {
        aantalPoten: integer
    };

    {   /*
            Simpele read/write test met composite en int:
        */
        s: Stoel := Stoel{ aantalPoten := 4 };
        i:integer := 2;
        test(s.aantalPoten, 4, "s.aantalPoten (before swap)");
        test(i, 2, "i (before swap)");
        s.aantalPoten <> i; // Keert de waardes om!
        test(s.aantalPoten, 2, "s.aantalPoten (after swap)");
        test(i, 4, "i (after swap)");
    }

    type Tafel {
```

```
        aantalZitPlaatsen: integer
    };

    {
        // No errors caused: we can hide the outside type
        type Tafel {
            isOranje: boolean
        };

        t: Tafel := Tafel{ isOranje:= true };

        test(t.isOranje, true, "t.isOranje (we just set it, now accessing its field)");
    }

    type EetKamer {
        eetTafel: Tafel,
        mooisteStoel: Stoel
    };

    type BadKamer {
        heeftBad: boolean
    };

    type Huis {
        eetKamer: EetKamer,
        badKamer: BadKamer
    };

    t1: Tafel := Tafel{ aantalZitPlaatsen := 3 };
    k: EetKamer := EetKamer{ eetTafel := t1, mooisteStoel := Stoel{ aantalPoten := 1 } };
    b: BadKamer;
    h: Huis := Huis {
        badKamer := (b := BadKamer{ heeftBad := true }),
        eetKamer := k
    };

    // De oude stoel heeft maar één poot:
    test(h.eetKamer.mooisteStoel.aantalPoten, 1, "h.eetKamer.mooisteStoel.aantalPoten (oude stoel)");
```

```
        // Nieuwe stoel kopen!
        nieuweStoel: Stoel := Stoel{ aantalPoten := 4 };
        h.eetKamer.mooisteStoel <> nieuweStoel;

        // De nieuwe stoel heeft wel gewoon vier poten:
        test(h.eetKamer.mooisteStoel.aantalPoten, 4, "h.eetKamer.mooisteStoel.aantalPoten (nieuwe stoel)");

        // Nu even testen of we waarden uit literals kunnen opvragen:
        aantalZitPlaatsen: integer :=
            EetKamer{
                eetTafel := Tafel{ aantalZitPlaatsen := 10 },
                mooisteStoel := Stoel{ aantalPoten := 4 }
            }.eetTafel.aantalZitPlaatsen;

        test(aantalZitPlaatsen, 10, "double field access on composite literal (aantalZitPlaatsen)");

        // Speciale aanbieding! 9 poten voor de prijs van vier!
        function buyChair() {
            return Stoel{ aantalPoten:= 9 }; // Composites returnen kan ook gewoon!
        };
        gekochtAantalPoten: integer := (((((buyChair())))))).aantalPoten;
        test(gekochtAantalPoten, 9, "field access on return value of buyChair()");
        // Bij de IKEA kan je nu stoelen met negen poten kopen.
        // (ramp om in elkaar te moeten zetten...
    }
}

// Let's round it up:
end();
```

# Appendix D: object code of `ScoobyTesting.spr`

Truly including this would literally add 98 pages to this report. To save the trees, we have decided not to do this. The object code can be found in file `Appendix_D.txt`.