Concepts of Programming Languages - boLang

Burak Karakoc

August 29, 2022

1 Introduction

This document describes the project of Concepts of Programming Languages SoSe-2022. In this project, a Java-like, basic programming language has developed.

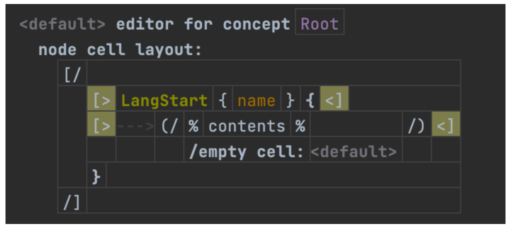
2 Language

The programming language boLang only accepts int, string, and boolean types of variables currently, however, since the structure of the development is conducted in an object-oriented fashion, new types can easily be added.

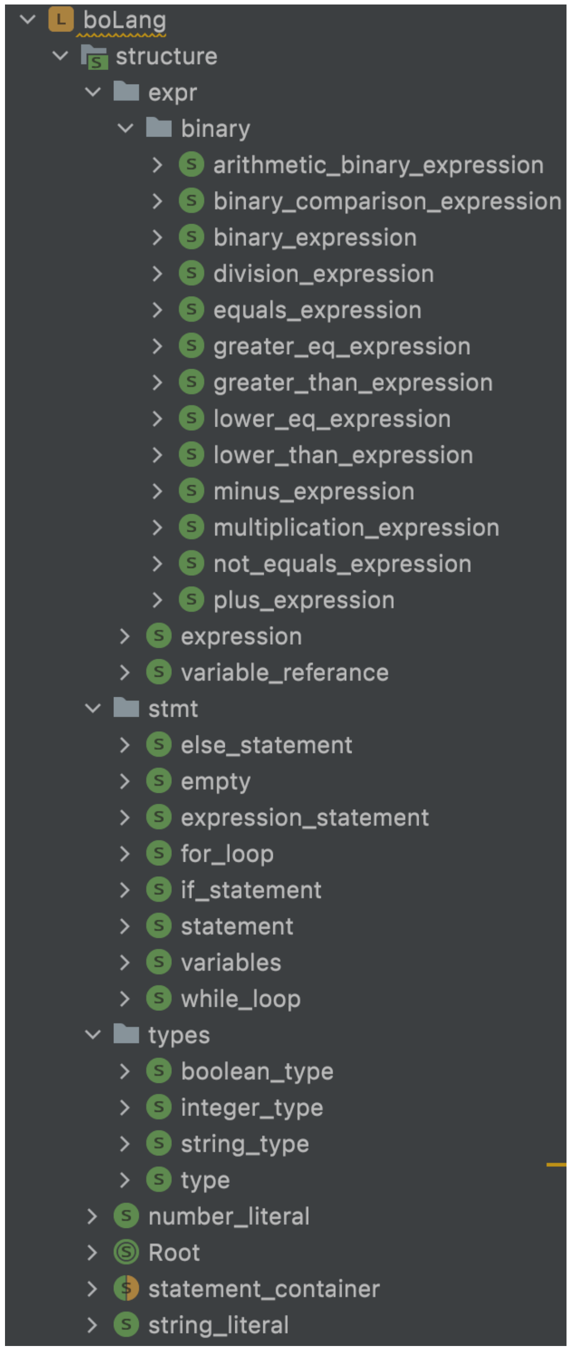
Worksheets can either contain:
- Variable declarations (with duplicate checks),
- Variables have also scopes - i.e. the one defined outside the scope, is not reachable outside,
- Expressions including arithmetic, comparison, equality,
- All expressions have appropriate type checks and priorities,
- Control flow (i.e. if-else, for-loop, while-loop),
- Tests (able to combined and used in single node - worksheet)
- Also, spaces and end-lines supported.

3 Structure

The worksheet(root) contains 0..n statement nodes. In other words, every node that extends abstract statement concept, can be contained inside the root, such as, variable definitions, assignments, and control statements like if-else, for, while. For structure details, and editor scheme of language see figures below.



Language Editor



Structure

4 Grammar Cells

Grammarcells are used to improve the typing exprerience whenever needed. For instance, providing priorities in arithmetic expressions and directly typing the expressions, such as, integers(1,2,3), strings("..."), and booleans([true,false]). Also, grammar cells is using grammar.constant to determine which operation is that. See the figures below for grammarcells usage, priority behaviour definition, and example operation concept(which is a subconcept of abstract binary expression, so that every operation relies to the same rules and behaves exactly the same, except their operation types).

```
<default> editor for concept binary_expression
  node cell layout:
    projection: [> wrap % left % grammar.constant wrap % right % <]
    grammar:    rule: <derive from projection>    %left%  grammar.constant  %right%
```

Grammar Cells

```
concept multiplication_expression extends    arithmetic_binary_expression
                                  implements <none>

  instance can be root: false
  alias: *
```

Example of expression concept (multiplication)

```
public static virtual int priority(concept<> subconConcept) {
  if (subconConcept.conceptAlias == "+" || subconConcept.conceptAlias == "-") { return 0; }
  return 1;
}
```

Priority definition

5 Scope

An interface concept called statement container built to ensure correct scopes for each declaration and reference. Further, includes getStatements() abstract function, which will be used as override in every statement. Basically, it works like getting all the reachable elements and make them reachable at that container. Also, two getScope() function implemented. For further details and an example, see figures below.

```
interface concept behavior statement_container {

  constructor {
    <no statements>
  }

  public virtual abstract sequence<node<statement>> getStatements();

  public virtual Scope getScope(concept<> kind, node<> child)
    overrides ScopeProvider.getScope {
    message error "child" + child.concept, <no project>, <no throwable>;
    if (kind.isSubConceptOf(variables)) {
      ListScope vars = ListScope.forNamedElements(
          getStatements().ofConcept<variables>.where({~it => it.index < child.index; }));
      return new HidingByNameScope(concept/variables/, kind, vars, parent scope);
    }
    return null;
  }

  public virtual Scope getScope(concept<> kind, SContainmentLink link, int index)
    overrides ScopeProvider.getScope {
    if (kind.isSubConceptOf(variables)) {
      ListScope vars = ListScope.forNamedElements(
          getStatements().ofConcept<variables>.where({~it => it.index < index; }));
      return new HidingByNameScope(concept/variables/, kind, vars, parent scope);
    }
    super<ScopeProvider>.getScope(kind, link, index);
  }

}
```

Scope

```
LangStart boLang.example {
  var int i = 1
  if( 1 ) {
    i
    var string trial = "dummy"
  }
  trial
```

Scope example
(i is reachable inside if statement, however, variables declared inside a container can not be reached outside)

6 Types

Type checking is implemented everywhere it is needed. Furthermore, checking implemented for binary operation types too, such as, if two integers compared, this expression should return a boolean([true,false]) type etc. Here are some implementation and real use examples below.

```
overloaded operations rules binary_operation_types

operation concepts: binary_comparison_expression
left operand type: <integer_type()> is exact: false use strong subtyping false
right operand type: <integer_type()> is exact: false use strong subtyping false
is applicable:
<no isApplicable>
operation type:
(operation, leftOperandType, rightOperandType)->node<> {
  return <boolean_type()>;
}
----------------------------------------------
operation concepts: equals_expression
left operand type: <string_type()> is exact: false use strong subtyping false
right operand type: <string_type()> is exact: false use strong subtyping false
is applicable:
<no isApplicable>
operation type:
(operation, leftOperandType, rightOperandType)->node<> {
  return <boolean_type()>;
}
----------------------------------------------
operation concepts: arithmetic_binary_expression
left operand type: <integer_type()> is exact: false use strong subtyping false
right operand type: <integer_type()> is exact: false use strong subtyping false
is applicable:
<no isApplicable>
operation type:
(operation, leftOperandType, rightOperandType)->node<> {
  return <integer_type()>;
}
```

Binary operation check

```
LangStart boLang.example {
  var int a = 1
  var string b = "dummy"
  var bool check = a > b

                 Error: type error[operation is not supported!] is not a subtype of bool
                 Error: opperation not supported!
```

Binary operation check example

```
inference rule typeof_variables {
  applicable for concept = variables as variables
  applicable always
  overrides false

  do {
    if (variables.type != null) {
      typeof(variables) :==: variables.type;
      check(typeof(variables.value) :<=: variables.type);
    } else {
      typeof(variables) :==: typeof(variables.value);
    }
  }
}
```

Variable type check

```
LangStart boLang.example {
  var string b = 1

                   Error: type int is not a subtype of string
```

Variable type check example

```
inference rule typeof_binary_expression {
  applicable for concept = binary_expression as binary_expression
  applicable always
  overrides false

  do {
    when concrete (typeof(binary_expression.left)  as leftType) {
      when concrete (typeof(binary_expression.right)  as rightType) {
        node<> resultType = operation type(binary_expression, leftType, rightType);
        if (resultType != null) {
          typeof(binary_expression) :==: resultType;
        } else {
          typeof(binary_expression) :==: <RuntimeErrorType(errorText: "operation is not supported!")>;
          error "opperation not supported!" -> binary_expression;
        }
      }
    }
  }
}
```

Binary expression checks

```
LangStart boLang.example {
  var int i = 1
  var string x = "a"
  i - x

  Error: opperation not supported!
```

Binary expression check example

```
LangStart boLang.example {
  var int a = 1
  var string b = "dummy"
  a > b
  Error: opperation not supported!
```

Second example (comparison expression)

```
checking rule check_duplicate {
  applicable for concept = statement_container as sc
  overrides <none>
  do not apply on the fly false

  do {
    set<string> varNames = new hashset<string>;
    foreach variable in sc.getStatements().ofConcept<variables> {
      if (varNames.contains(variable.name)) {
        error "Error: duplicate name found! (" + variable.name + ")" -> variable;
      }
      varNames.add(variable.name);
    }
  }
}
```

Duplicate check

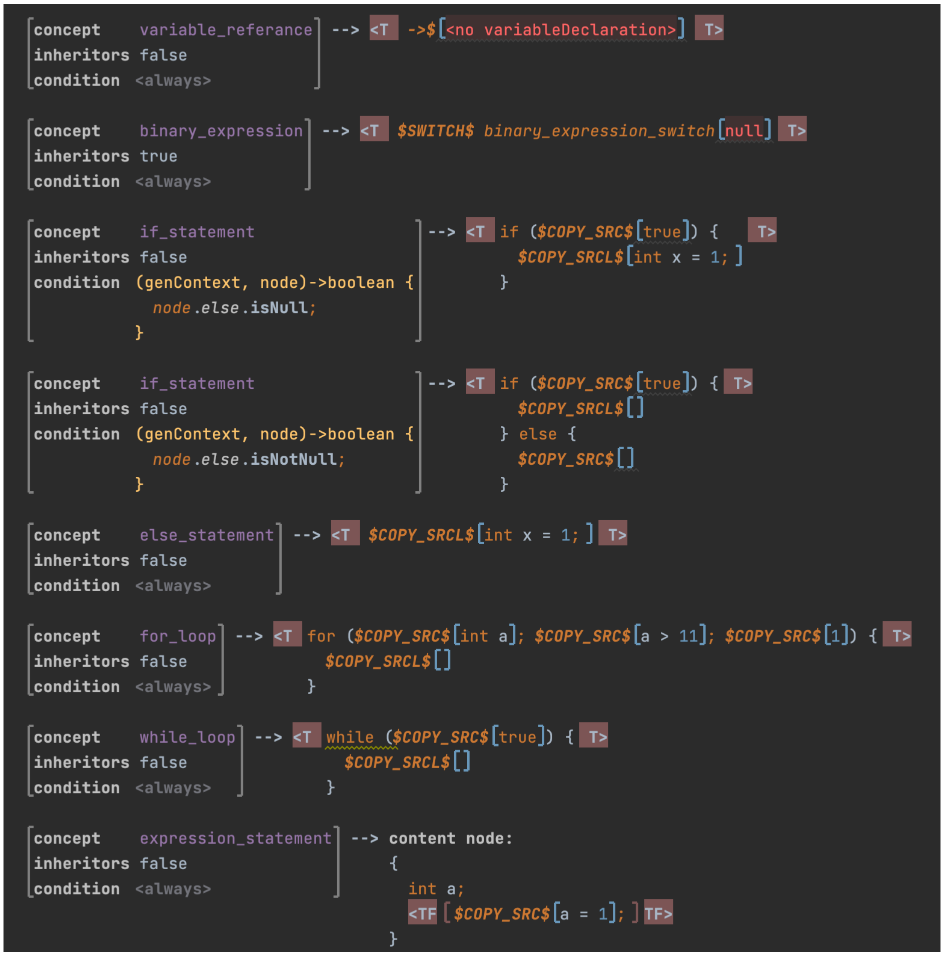
```
LangStart boLang.example {
  var int a = 1
  var int a = 5

            Error: Error: duplicate name found! (a)
```

Duplicate check example

7 Generator

In the semantics part of boLang programing language, generator is decided to used. The generator can map all of the concepts of boLang to Java. For instance, worksheets mapped to public static void main() function. That mapping provides boLang to easily operate over Java with the same concepts. In order to ensure the generic expression mapping to Java, template switch is used. Detailed mapping rules and example of mapping shown in the figures below.



Some of main reduction rules

```
template switch binary_expression_switch extends <none>

parameters
<< ... >>

  null-input message: <none>

  cases:

     [concept     minus_expression] --> <T $COPY_SRC$[1] - $COPY_SRC$[2] T>
     [inheritors false            ]
     [condition  <always>         ]

     [concept     plus_expression] --> <T $COPY_SRC$[1] + $COPY_SRC$[2] T>
     [inheritors false           ]
     [condition  <always>        ]

     [concept     multiplication_expression] --> <T $COPY_SRC$[1] * $COPY_SRC$[2] T>
     [inheritors false                     ]
     [condition  <always>                  ]

     [concept     division_expression] --> <T $COPY_SRC$[1] / $COPY_SRC$[2] T>
     [inheritors false               ]
     [condition  <always>            ]

     [concept     greater_than_expression] --> <T $COPY_SRC$[1] > $COPY_SRC$[2] T>
     [inheritors false                   ]
     [condition  <always>                ]

     [concept     greater_eq_expression] --> <T $COPY_SRC$[1] >= $COPY_SRC$[2] T>
     [inheritors false                 ]
     [condition  <always>              ]

     [concept     lower_than_expression] --> <T $COPY_SRC$[1] < $COPY_SRC$[2] T>
     [inheritors false                 ]
     [condition  <always>              ]
```

Some template switches for expressions

An example of boLang mapping to Java:

```
LangStart boLang.example {
  var int a = 1
  var int b = 2
  a + b

  1 + 4
  1 == 1
  1 - 2 + 1 * 3 / 1
  221

  var int h = a + 1

  var bool check = a > b

  while(check) {
    var int blah = 45
  }

  var string z = "xyz"
  var bool sd = "asd"

  if( 1 ) {
    var int o = 1
  }
  else {
  var string trial = "burak"
  }

  for(a ; a > 1 ; 1 ;) {
    var int a = 1
  }
}
```

```java
/*Generated by MPS */


public class boLang.example {
  public static void main() {
    int a = 1;
    int b = 2;
    a + b;
    1 + 4;
    1 == 1;
    1 - 2 + 1 * 3 / 1;
    221;
    int h = a + 1;
    boolean check = a > b;
    while (check) {
      int blah = 45;
    }
    String z = "xyz";
    boolean sd = "asd";
    if (1) {
      int o = 1;
    } else {
      String trial = "burak";
    }
    for (a; a > 1; 1) {
      int a = 1;
    }
  }
}
```

8 Testing

An additional test language has been developed to execute test cases and assertions. Further, execution of test can be called inside worksheet using boLang too (and can be mapped to Java). There are two main functionalities inside test language, first one is to execute all of the tests inside a spesific test suite, second is executing only one test. Also, of course, they have scopes to ensure that (i.e after selecting a test suite, only test cases included inside that test suite are shown). Some examples demonstrated below.

Example of test declarations:

```
TestSuite: boLang.test

test test_1 {
  var int burak = 11
  var string asd = "asd"
}
test test_2 {
  assert 1 > 3
  var int b = 1
  var int z = 2
  var int g = 1 + b
}
```

```
TestSuite: boLang.test2

test test_1_v.2 {
  if( 1 ) {
    var int i = 1
  }
}
```

Example of test scopes:

```
LangStart boLang.example {
  execute_test -> boLang.test
  execute_test -> boLang.test2

  execute_single -> boLang.test -> t
}
                                    test_1   ^tests (boLang.sandbox.boLang.test)
                                    test_2   ^tests (boLang.sandbox.boLang.test)
                                    Press ⌥⌘B to Show item trace
```

```
LangStart boLang.example {
  execute_test -> boLang.test
  execute_test -> boLang.test2

  execute_single -> boLang.test2 -> t
}
                                    test_1_v.2   ^tests (boLang.sandbox.boLang.test2)
                                    Press ⌥⌘B to Show item trace
```

Additional simple type testing example with test solution

```
Test case types
nodes
  ( [ LangStart root_test_lang {                                   ] )
        var int x = 1

        <check var int x = 2 has error <no errorRef>>

        if( 1 ) {
          var int y = 1
          <check var int y = 2 has error <no errorRef>>
        }

        var bool compstr = <check "a" > "b" has error <no errorRef>>
      }
  ( [ <check LangStart correct_root {                 for error messages> ] )
              <check var int x = 1 + 11 + 41 has type int>
              var int y = x + 10
            }
```