# The Language BasilIR

BNF-converter

June 6, 2025

This document was automatically generated by the *BNF-Converter*. It was generated together with the lexer, the parser, and the abstract syntax module, which guarantees that the document matches with the implementation of the language (provided no hand-hacking has taken place).

## The lexical structure of BasilIR

### Literals

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

BVTYPE literals are recognized by the regular expression {“bv”}⟨*digit*⟩+

BIdent literals are recognized by the regular expression ‘.’(‘_’ | ⟨*letter*⟩)([“#$._”] | ⟨*digit*⟩ | ⟨*letter*⟩)∗

LocalIdent literals are recognized by the regular expression ([“%_”] | ⟨*letter*⟩)([“#$._”] | ⟨*digit*⟩ | ⟨*letter*⟩)∗

GlobalIdent literals are recognized by the regular expression ‘$’([“#$._”] | ⟨*digit*⟩ | ⟨*letter*⟩)+

BlockIdent literals are recognized by the regular expression ‘#’([“#$._”] | ⟨*digit*⟩ | ⟨*letter*⟩)+

ProcIdent literals are recognized by the regular expression ‘@’([“#$._”] | ⟨*digit*⟩ | ⟨*letter*⟩)+

BeginList literals are recognized by the regular expression ‘[’

EndList literals are recognized by the regular expression ‘]’

BeginRec literals are recognized by the regular expression ‘{’

EndRec literals are recognized by the regular expression ‘}’

Str literals are recognized by the regular expression ‘"’(⟨*anychar*⟩ − [“"\”] | ‘\’[“"\fnrt”]) ∗ ‘"’

IntegerHex literals are recognized by the regular expression {“0x”}([“abcdef”] | ⟨*digit*⟩)+

BitvectorHex literals are recognized by the regular expression {“0x”}([“abcdef”] | ⟨*digit*⟩)+

## Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in BasilIR are the following:

| | | |
|---|---|---|
| `assert` | `assume` | `axiom` |
| `be` | `block` | `bool` |
| `booland` | `boolimplies` | `boolnot` |
| `boolor` | `booltobv1` | `bvadd` |
| `bvand` | `bvashr` | `bvcomp` |
| `bvconcat` | `bvlshr` | `bvmul` |
| `bvnand` | `bvneg` | `bvnor` |
| `bvnot` | `bvor` | `bvsdiv` |
| `bvsge` | `bvsgt` | `bvshl` |
| `bvsle` | `bvslt` | `bvsmod` |
| `bvsrem` | `bvsub` | `bvudiv` |
| `bvuge` | `bvugt` | `bvule` |
| `bvult` | `bvurem` | `bvxnor` |
| `bvxor` | `call` | `ensure` |
| `ensures` | `eq` | `extract` |
| `false` | `goto` | `guarantee` |
| `guard` | `indirect` | `int` |
| `intadd` | `intdiv` | `intge` |
| `intgt` | `intle` | `intlt` |
| `intmod` | `intmul` | `intneg` |
| `intsub` | `invariant` | `le` |
| `load` | `memory` | `neq` |
| `proc` | `prog` | `rely` |
| `require` | `requires` | `return` |
| `sign_extend` | `store` | `true` |
| `unreachable` | `var` | `zero_extend` |

The symbols used in BasilIR are the following:

| | | |
|---|---|---|
| ; | , | : |
| ( | -> | ) |
| := | = | |

### Comments

Single-line comments begin with `//`.
Multiple-line comments are enclosed with `/*` and `*/`.

## The syntactic structure of BasilIR

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and $\epsilon$ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Program*⟩ ::= ⟨*ListDeclaration*⟩

⟨*ListDeclaration*⟩ ::= $\epsilon$
  | ⟨*Declaration*⟩ `;` ⟨*ListDeclaration*⟩

⟨*ListBlockIdent*⟩ ::= $\epsilon$
  | ⟨*BlockIdent*⟩
  | ⟨*BlockIdent*⟩ `,` ⟨*ListBlockIdent*⟩

⟨*GobbleScolon*⟩ ::= $\epsilon$
  | ⟨*GobbleScolon*⟩ `;`

⟨*Declaration*⟩ ::= `axiom` ⟨*AttrDefList*⟩ ⟨*Expr*⟩
  | `memory` ⟨*GlobalIdent*⟩ `:` ⟨*Type*⟩
  | `var` ⟨*GlobalIdent*⟩ `:` ⟨*Type*⟩
  | `prog` ⟨*AttrDefList*⟩ ⟨*BeginList*⟩ ⟨*ListProgSpecDecl*⟩ ⟨*EndList*⟩
  | `prog` ⟨*AttrDefList*⟩
  | ⟨*ProcDef*⟩

⟨*IntType*⟩ ::= `int`

⟨*BoolType*⟩ ::= `bool`

⟨*MapType*⟩ ::= `(` ⟨*Type*⟩ `->` ⟨*Type*⟩ `)`

⟨*BVType*⟩ ::= ⟨*BVTYPE*⟩

⟨*Type*⟩ ::= ⟨*IntType*⟩
  | ⟨*BoolType*⟩
  | ⟨*MapType*⟩
  | ⟨*BVType*⟩

```
⟨ListExpr⟩  ::=  ϵ
             |   ⟨Expr⟩
             |   ⟨Expr⟩ , ⟨ListExpr⟩

⟨IntVal⟩  ::=  ⟨IntegerHex⟩
           |   ⟨Integer⟩

⟨BVVal⟩  ::=  ⟨IntVal⟩ : ⟨BVType⟩

⟨Endian⟩  ::=  le
           |   be

⟨ListStatement⟩  ::=  ϵ
                  |   ⟨Statement⟩ ; ⟨ListStatement⟩

⟨Assignment⟩  ::=  ⟨LVar⟩ := ⟨Expr⟩

⟨Statement⟩  ::=  ⟨Assignment⟩
              |   ( ⟨ListAssignment⟩ )
              |   ⟨LVar⟩ := load ⟨Endian⟩ ⟨GlobalIdent⟩ ⟨Expr⟩ ⟨IntVal⟩
              |   store ⟨Endian⟩ ⟨GlobalIdent⟩ ⟨Expr⟩ ⟨Expr⟩ ⟨IntVal⟩
              |   ⟨CallLVars⟩ call ⟨ProcIdent⟩ ( ⟨ListExpr⟩ )
              |   indirect call ⟨Expr⟩
              |   assume ⟨Expr⟩ ⟨AttrDefList⟩
              |   guard ⟨Expr⟩ ⟨AttrDefList⟩
              |   assert ⟨Expr⟩ ⟨AttrDefList⟩

⟨ListAssignment⟩  ::=  ⟨Assignment⟩
                   |   ⟨Assignment⟩ , ⟨ListAssignment⟩

⟨LocalVar⟩  ::=  ⟨LocalIdent⟩ : ⟨Type⟩

⟨GlobalVar⟩  ::=  ⟨GlobalIdent⟩ : ⟨Type⟩

⟨ListLocalVar⟩  ::=  ⟨LocalVar⟩
                 |   ⟨LocalVar⟩ , ⟨ListLocalVar⟩

⟨CallLVars⟩  ::=  ϵ
              |   var ( ⟨ListLocalVar⟩ ) :=
              |   ( ⟨ListLVar⟩ ) :=

⟨Jump⟩  ::=  goto ( ⟨ListBlockIdent⟩ )
         |   unreachable
         |   return ( ⟨ListExpr⟩ )

⟨LVar⟩  ::=  var ⟨LocalVar⟩
         |   ⟨GlobalVar⟩
```

```
⟨ListLVar⟩ ::= ⟨LVar⟩
             | ⟨LVar⟩ , ⟨ListLVar⟩

⟨ListBlock⟩ ::= ϵ
              | ⟨Block⟩
              | ⟨Block⟩ ; ⟨ListBlock⟩

⟨Block⟩ ::= block ⟨BlockIdent⟩ ⟨AttrDefList⟩ ⟨BeginList⟩ ⟨ListStatement⟩ ⟨Jump⟩ ; ⟨EndList

⟨AttributeItem⟩ ::= ⟨BIdent⟩ = ⟨IntVal⟩
                  | ⟨BIdent⟩ = ⟨BVVal⟩
                  | ⟨BIdent⟩ = ⟨Expr⟩
                  | ⟨BIdent⟩ = ⟨Str⟩

⟨ListAttributeItem⟩ ::= ϵ
                      | ⟨AttributeItem⟩
                      | ⟨AttributeItem⟩ ; ⟨ListAttributeItem⟩

⟨AttrDefList⟩ ::= ⟨BeginRec⟩ ⟨ListAttributeItem⟩ ⟨GobbleScolon⟩ ⟨EndRec⟩
                | ϵ

⟨Params⟩ ::= ⟨LocalIdent⟩ : ⟨Type⟩

⟨ListParams⟩ ::= ϵ
               | ⟨Params⟩
               | ⟨Params⟩ , ⟨ListParams⟩

⟨ProcSig⟩ ::= proc ⟨ProcIdent⟩ ( ⟨ListParams⟩ ) −> ( ⟨ListParams⟩ )

⟨ProcDef⟩ ::= ⟨ProcSig⟩ ⟨AttrDefList⟩ ⟨ListFunSpecDecl⟩
            | ⟨ProcSig⟩ ⟨AttrDefList⟩ ⟨ListFunSpecDecl⟩ ⟨BeginList⟩ ⟨ListBlock⟩ ⟨EndList⟩

⟨Expr⟩ ::= ⟨BVVal⟩
         | ⟨IntVal⟩
         | true
         | false
         | ⟨LocalVar⟩
         | ⟨GlobalVar⟩
         | ⟨GlobalIdent⟩ ( ⟨ListExpr⟩ )
         | ⟨BinOp⟩ ( ⟨Expr⟩ , ⟨Expr⟩ )
         | ⟨UnOp⟩ ( ⟨Expr⟩ )
         | zero_extend ( ⟨IntVal⟩ , ⟨Expr⟩ )
         | sign_extend ( ⟨IntVal⟩ , ⟨Expr⟩ )
         | extract ( ⟨IntVal⟩ , ⟨IntVal⟩ , ⟨Expr⟩ )
         | bvconcat ( ⟨Expr⟩ , ⟨Expr⟩ )
```

```
⟨BinOp⟩  ::=  ⟨BVBinOp⟩
           |  ⟨BVLogicalBinOp⟩
           |  ⟨BoolBinOp⟩
           |  ⟨IntLogicalBinOp⟩
           |  ⟨IntBinOp⟩
           |  ⟨EqOp⟩

⟨UnOp⟩  ::=  ⟨BVUnOp⟩
          |  boolnot
          |  intneg
          |  booltobv1

⟨EqOp⟩  ::=  eq
          |  neq

⟨BVUnOp⟩  ::=  bvnot
            |  bvneg

⟨BVBinOp⟩  ::=  bvand
             |  bvor
             |  bvadd
             |  bvmul
             |  bvudiv
             |  bvurem
             |  bvshl
             |  bvlshr
             |  bvnand
             |  bvnor
             |  bvxor
             |  bvxnor
             |  bvcomp
             |  bvsub
             |  bvsdiv
             |  bvsrem
             |  bvsmod
             |  bvashr

⟨BVLogicalBinOp⟩  ::=  bvule
                    |  bvugt
                    |  bvuge
                    |  bvult
                    |  bvslt
                    |  bvsle
                    |  bvsgt
                    |  bvsge
```

```
⟨IntBinOp⟩  ::=  intadd
             |   intmul
             |   intsub
             |   intdiv
             |   intmod

⟨IntLogicalBinOp⟩  ::=  intlt
                    |   intle
                    |   intgt
                    |   intge

⟨BoolBinOp⟩  ::=  booland
              |   boolor
              |   boolimplies

⟨RequireTok⟩  ::=  require
               |   requires

⟨EnsureTok⟩  ::=  ensure
              |   ensures

⟨FunSpecDecl⟩  ::=  ⟨RequireTok⟩ ⟨Expr⟩
                |   ⟨EnsureTok⟩ ⟨Expr⟩
                |   invariant ⟨BlockIdent⟩ ⟨Expr⟩

⟨ProgSpecDecl⟩  ::=  rely ⟨Expr⟩
                 |   guarantee ⟨Expr⟩

⟨ListFunSpecDecl⟩  ::=  ϵ
                    |   ⟨FunSpecDecl⟩ ; ⟨ListFunSpecDecl⟩

⟨ListProgSpecDecl⟩  ::=  ϵ
                     |   ⟨ProgSpecDecl⟩ ; ⟨ListProgSpecDecl⟩
```