Telelogic Logiscope

Writing C Rules Using RuleChecker Tcl Verifier

Version 6.5

Before using this information, be sure to read the general information under “Notices” section, on page **27**.

Table of Contents

About this manual

This manual is a complement to the *Telelogic Logiscope RuleChecker & QualityChecker – C Reference Manual* where the **Tcl Verifier** data model and main support procedures are described.

Reading first the above document is mandatory.

What is important to remember is that the data model mainly describes an abstract syntax tree of the code, with some semantic information already resolved and attached to the syntax tree.

Audience

This manual is intended for **Telelogic® Logiscope™** *RuleChecker C* users who want to verify new programming rules using the **Tcl Verifier** and develop the associated scripts.

Overview

This document describes some fine points and how C constructs translate to the data model used by the Logiscope **Tcl Verifier**.

Section 1 explains some key support procedures of the **Tcl Verifier**.

Section 2 gives examples of how C code is translated into the data model.

Section 3 provides usual shortcuts when using the **Tcl Verifier**.

Section 4 addresses some special cases.

.

Conventions

The following typographical conventions are used in this manual:

italics	names of textual elements (filename), notes, documentation titles.
`typewriter`	screen and file examples.

Contacting IBM Rational Software Support

Support and information for Telelogic products is currently being transitioned from the Telelogic Support site to the IBM Rational Software Support site. During this transition phase, your product support location depends on your customer history.

Product support

- If you are a heritage customer, meaning you were a Telelogic customer prior to November 1, 2008, please visit the Logiscope Support Web site.

  Telelogic customers will be redirected automatically to the IBM Rational Software Support site after the product information has been migrated.

- If you are a new Rational customer, meaning you did not have Telelogic-licensed products prior to November 1, 2008, please visit the IBM Rational Software Support site.

Before you contact Support, gather the background information that you will need to describe your problem. When describing a problem to an IBM software support specialist, be as specific as possible and include all relevant background information so that the specialist can help you solve the problem efficiently. To save time, know the answers to these questions:

- What software versions were you running when the problem occurred?
- Do you have logs, traces, or messages that are related to the problem?
- Can you reproduce the problem? If so, what steps do you take to reproduce it?
- Is there a workaround for the problem? If so, be prepared to describe the workaround.

Other information

For Rational software product news, events, and other information, visit the IBM Rational Software Web site.

Bibliography

[TCL94] JOHN K. OUSTERHOUT

Tcl and the Tk Toolkit - Addison-Wesley Professional Computing Series

1994 ISBN 0-201-63337-X

[TCL03] BRENT WELCH, KEN JONES, JEFFREY HOBBS

Practical Programming in Tcl and Tk (4th Edition) – Prentice Hall

2003 ISBN 0-130-38560-3

[C90] ISO/IEC 9899:1990

International standard Programming languages - C

1. Support Procedures

1.1. MapRole

The main support procedure is `MapRole`. The main purpose of this procedure is to allow navigation in the data model, as described in the *Telelogic Logiscope RuleChecker & QualityChecker C Reference Manual* .

But it can also be used in other ways. The purpose of this procedure is to allow actions on the target objects of a link in the data model, but it returns a count of objects on which the action has been performed.

For example, if you want to know whether a type is qualified with const, you may use the fact that there is a `QualifierConst` object in the `qualifier` role of the type:

```
if {[MapRole $type qualifier -filterclass QualifierConst {expr 0;#}]}
```

Here, the action is to return 0, which stops the `MapRole` as soon as a `QualifierConst` is encountered during the navigation on the role. `;#` introduces a TCL comment, so that the exact action performed, `expr 0;# <handle on qualifier>`, does not see the handles on the qualifier objects. This `MapRole` will return `0` if no action is performed (i.e. there is no `QualifierConst` object in the role), or `1` if there is at least one `QualifierConst`. The `MapRole` will stop as soon as a `QualifierConst` object is processed.

Another example: you may compute the number of operands of an `ExpressionComplex` (i.e an expression with operator or function) with:

```
set argumentCount [MapRole $expression operand]
```

(a missing action is like having an action that always returns `1`).

A filter can be inserted between the name of the link that is to be followed and the action. The filter restricts the objects that are subject of the action. Note that these objects are thus not counted in the result of MapRole.

The filter can be:

- `-filter <script fragment>`. The `<script fragment>` is evaluated, like the action, with the object handle appended. If the filter returns `0`, the action is not evaluated; if the filter returns `1`, the action is evaluated.
- `-filterclass <class list>` . The action is evaluated only if the object is an instance of one of the classes of `<class list>`.

For example, to perform an action on all expressions using the ternary operator (`?:`):

```
proc isTernary {expression} {
     expr {[isClassObject ExpressionComplex $expression] && \
     [isClassObject FunctionTernary [GetRole $expression function]]}
}


MapRole application expression -filter isTernary action
```

To count the number of `typedef` in a translation unit `$scopeTU`:

```
set typedefCount [MapRole $scopeTU symbolDef \
                         -filterclass SymbolType]
```

1.2. Violation

The other important support procedure is `Violation`:

```
Violation $object $::thisRule "message"
```

The global variable `thisRule` is always set to the value of the `.KEY` keyword of the rule file before evaluation of the rule code. But what is interesting here is the `$object` part: this must be an object handle, and the object must belong to a class which inherits of the class `Origin`; the object handle is used to know where (file, line, function, if applicable) the violation will be shown. So you can play tricks with it. For example, if you want to flag a non conforming identifier for a variable, it may be best to issue a violation notice on all declarations and definitions of the variable.

1.3. IsClassObject

The `isClassObject` procedure performs the same function as the `-filterclass` filter of `MapRole`: it allows to efficiently test the class of a data model object.

```
set clist {InstructionDefinition InstructionTentativeDefinition}
if {[isClassObject $clist $object]} {
     ...
}
```

is equivalent to

```
if {[lsearch -exact $clist [Class $object]] >= 0} {
     ...
}
```

except that the `isClassObject` procedure is more memory and time efficient, and that it checks the validity of the class names in `$clist`.

2. From C Code to Data Model

2.1. Scopes and Symbols

A `Symbol` is an identifier in a scope. `Scope` objects are name spaces for identifiers,

The identifiers visible in the whole application are in the role `symbolDef` of the `ScopeGlobal` (there may be here only instances of `SymbolVariable` and `SymbolFunction`). There is only one instance of `ScopeGlobal`.

Every C file introduces a `ScopeTranslationUnit` which is the name space representing the C file with all included files expanded. Here are `SymbolVariable` and `SymbolFunction` declared with the keyword `static`, `SymbolType` (`typedef` identifiers), `SymbolTag` (tags of structures, unions and enumerations) and `SymbolEnum` (enumeration constant) that are declared at file level, and all `SymbolMacro` encountered in the C file and the included files.

Every structure and union introduces a new name space, represented by a `ScopeStructure`, that contains the `SymbolField` (field identifiers).

Every defined function introduces a new name space (`ScopeFunction`), that contains the parameter identifiers (of class `SymbolVariable`) and the `goto` labels (`SymbolLabel`).

Every macro function introduces a new name space (`ScopeFunction`), that contains the parameter identifiers (`SymbolVariable`). Note that the `Variable` objects linked to these parameters have no `type` role.

Every block of instructions introduces a new name space (`ScopeBlock`), that contains all the identifiers declared and defined in the block.

The `Scope*` objects, besides holding `Symbol*` objects in the `symbolDef` role, also hold the `Variable` (`variableDef` role) and `Function` (`functionDef` role) objects which are valid within the scope: functions being either `extern` or `static`, their containing scope may only be the `ScopeGlobal` or a `ScopeTranslationUnit`; `Variable` objects may be automatic, `extern` or `static`, thus their containing scope may be a `ScopeBlock`, the `ScopeGlobal` or a `ScopeTranslationUnit`, respectively. Note that `static` block variables are represented by a `Variable` object in a `ScopeBlock`, with the attribute `permanent` set to true.

`Scope*` objects also hold the declarations and definitions of the variables and functions in the `instructionDef` role. The `InstructionDeclaration` (for variables and functions), `InstructionDefinition` and `InstructionTentativeDefinition` (for variables) are described below.

```
/* The C file (and all its includes) introduces a
     ScopeTranslationUnit */
struct { /* Introduces a new ScopeStructure,
           subScope of the ScopeTranslationUnit */
   int a; /* a SymbolField (name = "a") within the ScopeStructure */
}
/* A SymbolVariable within the ScopeTranslationUnit */
static int a;
```

```
/* A SymbolVariable within the ScopeGlobal */
extern int b = 3;
/* A SymbolFunction within the ScopeGlobal */
/* The parameter list introduces a new ScopeFunction,
     subScope of the ScopeGlobal, since the function is external */
void f(int c) {
     /* The body of the function introduces a new BlockScope */
     /* Same Variable object as above but different Symbol,
          this one being in the BlockScope */
     extern int a;
     {
          /* another BlockScope, the superScope of which is
               the previous BlockScope */
          /* same Variable object as above but different Symbol */
          extern int b;
     }
}
```

2.2. Types

Types come in different flavors:

- Built in types, represented by the classes `TypeVoid`, `TypeInt`, etc.
- Constructed types, such as pointers, arrays and function types.
- Enumeration types.
- Structure and union types.
- Symbolic types, defined with typedef.

`TypeMeta` may not be encountered in an instantiation of the data model.

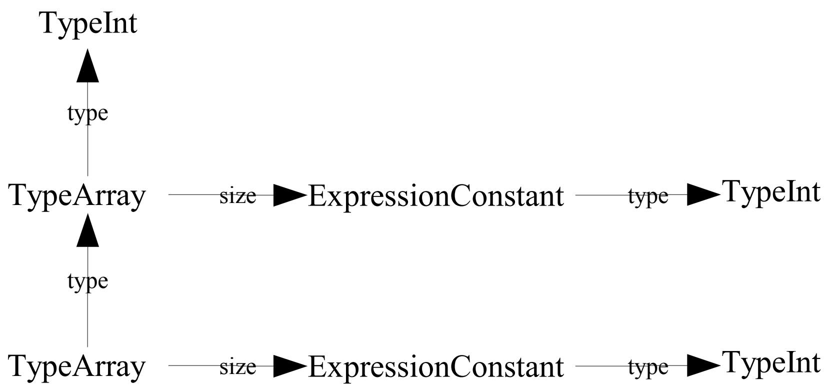
The instantiation of the data model for the different types is illustrated below.

`int a[5];`

(only the type of `a` is represented here, not the whole declaration)

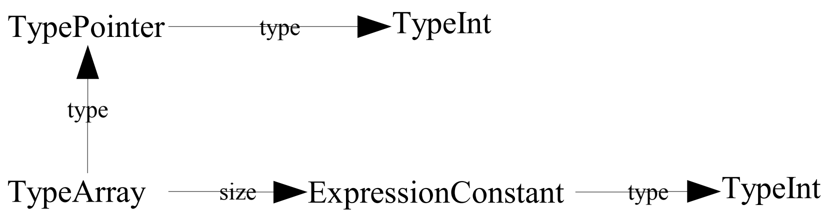
```
int a[3][2];
```

(only the type of `a` is represented here, not the whole declaration)



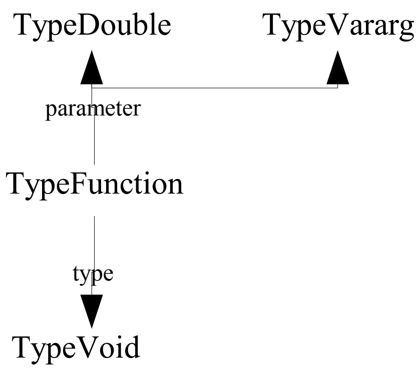

```
int *a[5];
```

(only the type of `a` is represented here, not the whole declaration)

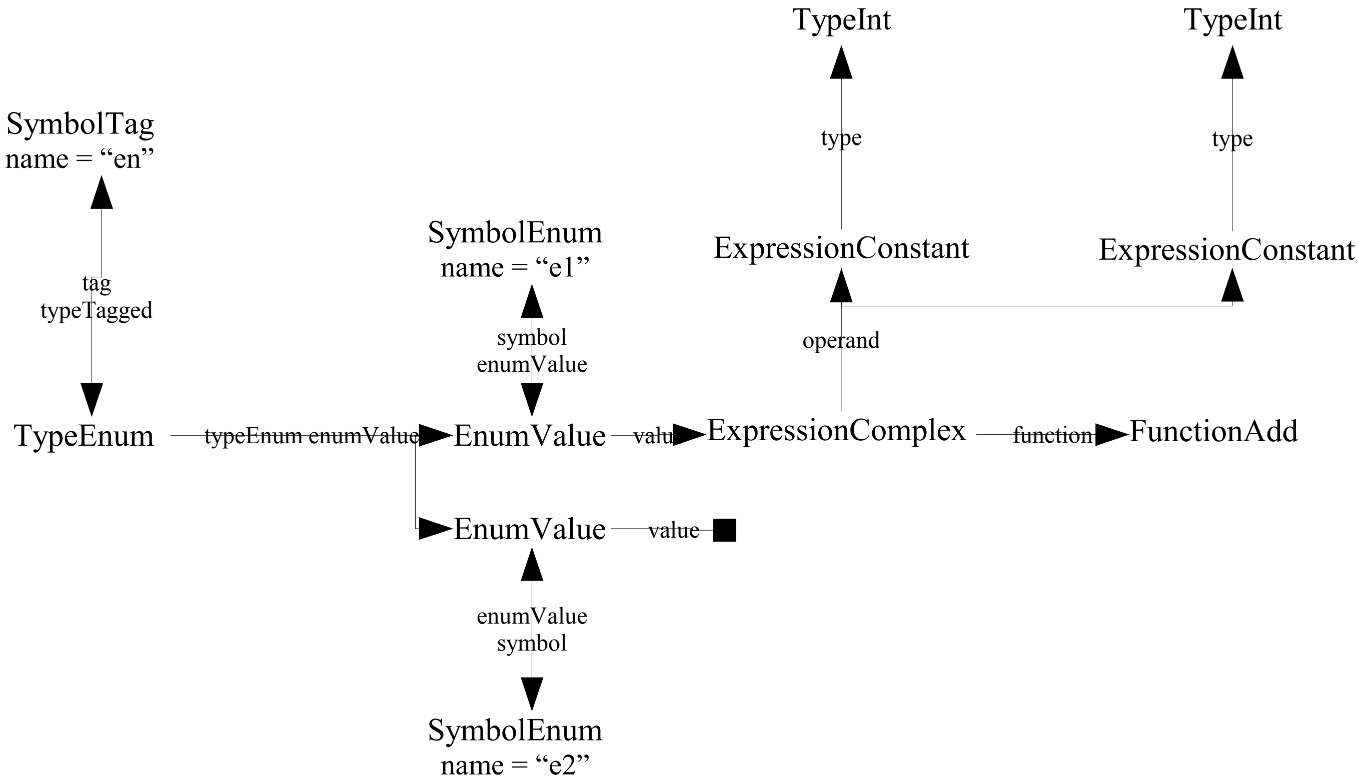



```
extern void f(double d, ...);
```

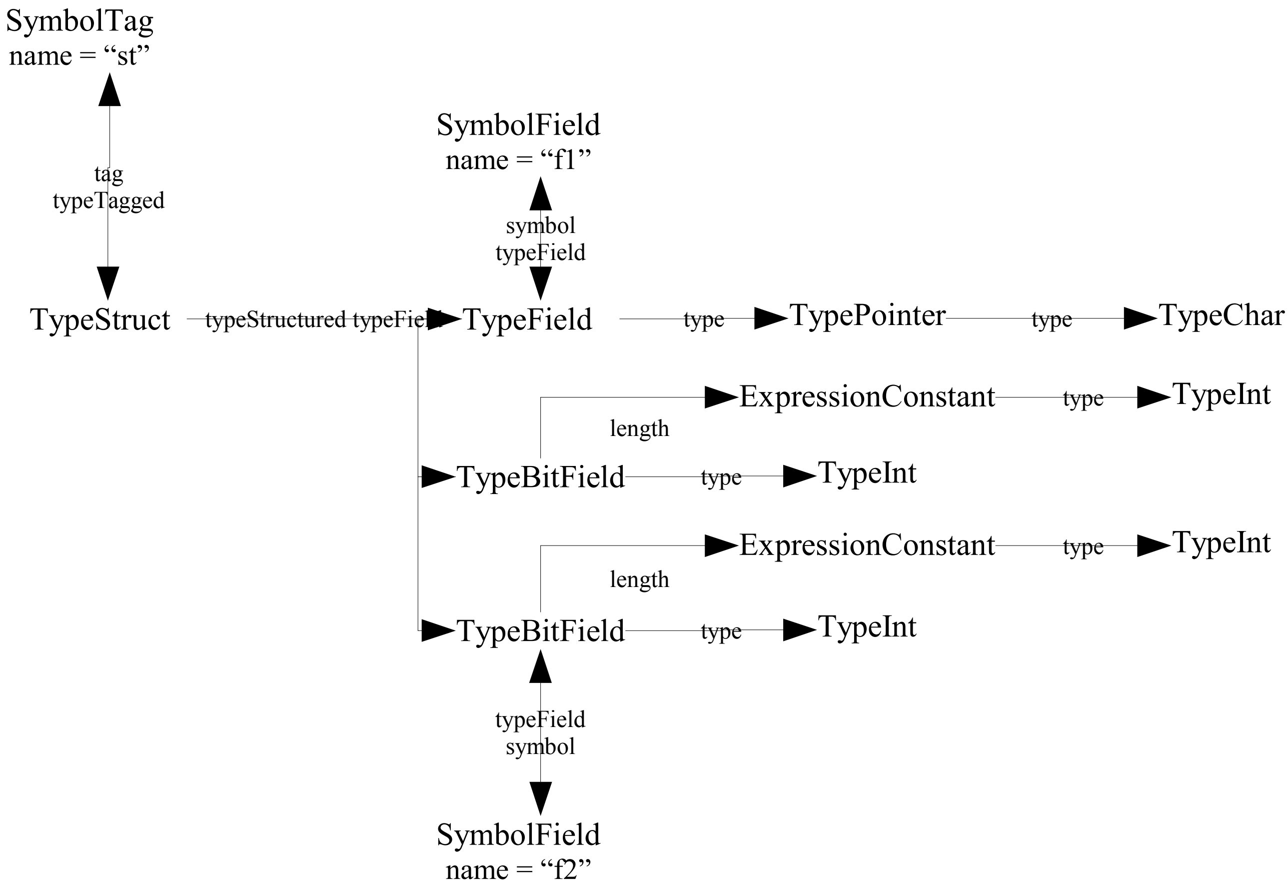
(only the type of `f` is represented here, not the whole `InstructionDeclaration`)



```
enum en {
     e1 = 3 + 4,
     e2
};
```



```
struct st {
     char *f1;
     int :2;
     int f2:6;
};
```



Note that the data model cannot distinguish between:

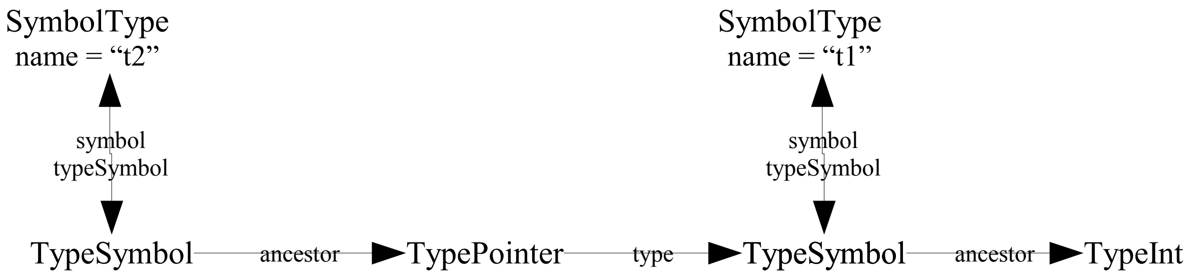
```
struct st {
     char *f1;
     int :2;
     int f2:6;
};
```

and:

```
struct st {
      char *f1;
      int :2, f2:6;
};
```

Beware that the names `TypeField` and `TypeBitField` may be confusing: their instances are not types, but fields of structures and unions.

```
typedef int t1; typedef t1 *t2;
```



(note that the role `expansion` does not work).

2.3. Function Declaration and Definition

Function objects have `InstructionDeclaration`, but no `InstructionDefinition`, nor `InstructionTentativeDefinition`.

As may be expected, an `InstructionDeclaration` is created when

```
extern int function(int i);
```

is encountered in the code. The `InstructionDeclaration` object is linked to the `SymbolFuction`, which has `function` as attribute `name`. By following the link from the `SymbolFunction` object to the `Function` object, all other `SymbolFunction` objects for the same function may be retrieved, and thus all the declarations for the function.

Retrieving the function definition and code is a bit trickier, and is covered below.

As a special case,

```
extern int f(void), g(int i);
```

creates two `InstructionDeclaration` objects, one for `f` and one for `g`.

2.4. Variable Declaration and Definition

Variable objects have `InstructionDeclaration`, `InstructionDefinition`, and `InstructionTentativeDefinition`.

`InstructionDefinition` objects are created for every declaration of a variable that reserves memory for the variable:

- Every variable declaration with an initializer.
- Every variable declaration in a block that is not introduced with the keyword `extern`.

`InstructionDeclaration` objects are created for every declaration of a variable that cannot reserve memory for the variable:

- Every variable declaration without initialization that is introduced with the keyword extern.

All other variable declarations are represented by a `InstructionTentativeDefinition` object. It is an obscure feature of the C language that such declarations do not reserve memory for the variable by themselves: if no definition is found for the variable at the end of the translation unit, the C compiler will reserve memory for the variable. A common extension found in C compilers on UNIX systems allows the linker to merge the memory allocated by the compiler for the different tentative definitions of the same variable name with external linkage.

Most of the time, `InstructionTentativeDefinition` objects are to be used like `InstructionDefinition` objects in rules.

When several variables are declared or defined in the same statement, one `InstructionDeclaration`, `InstructionDefinition` or `InstructionTentativeDefinition` object is created for each variable.

Examples:

```
static int a;      /*  InstructionTentativeDefinition */
int b;             /*  InstructionTentativeDefinition */
extern int c;      /* InstructionDeclaration  */
extern int b = 3;  /* same b, InstructionDefinition */
int a = 4;         /* same a, InstructionDefinition */
void f(void) {
     extern int a; /* same a InstructionDeclaration  */
     int d, e;     /* two InstructionDefinition */
}
```

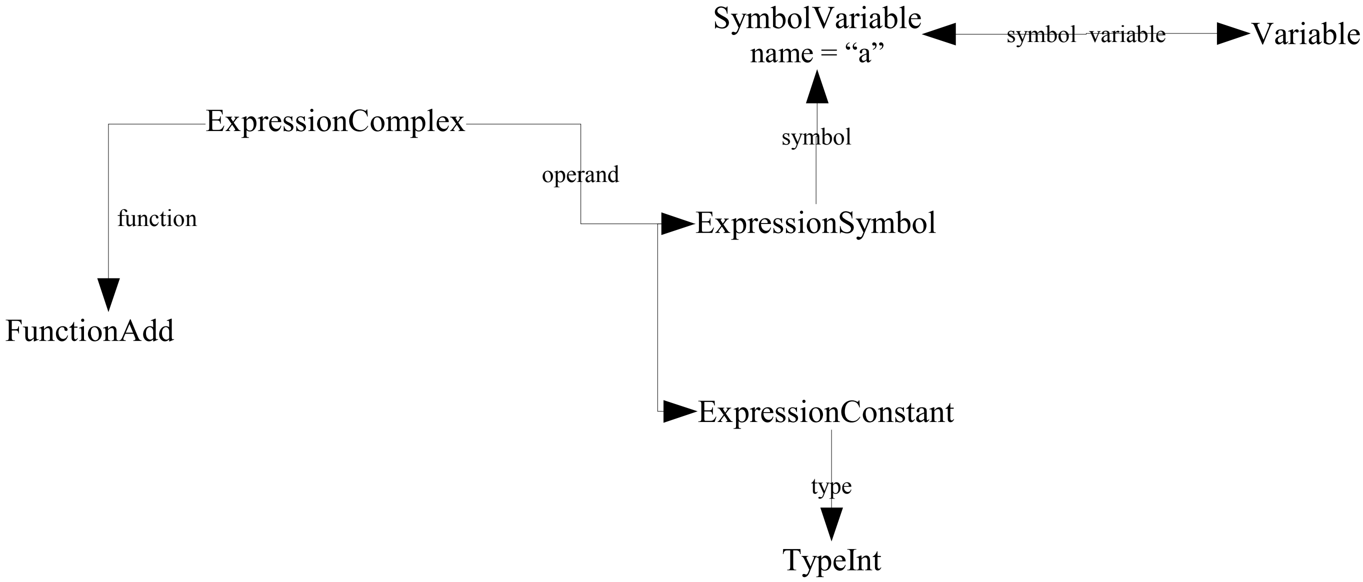
2.5. Expressions

Expressions come in different flavors:

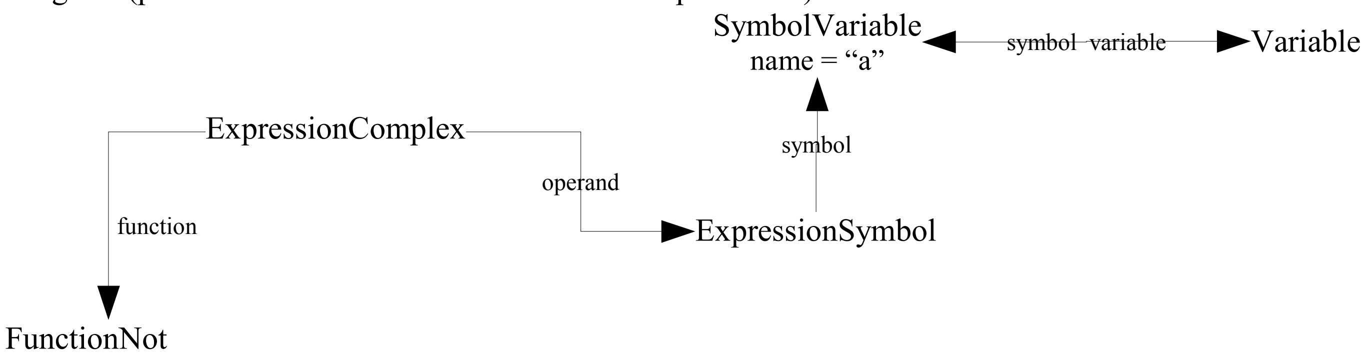
- `ExpressionConstant` represents the literal constant (numeric or string).
- `ExpressionSymbol` represents the symbolic expressions, such as a variable name, a function name, an enumeration value name, a structure or union field name.
- `ExpressionType` represents a type, when used in an expression, as part of a `sizeof` argument or in a cast.
- `ExpressionComplex` represents an expression with an operator and its operands, or a function call.

All expressions, with built in operators or function call follow a unified model. Here are two examples, the first with a binary operator, the second with a unary operator:

`a + 3` gives (provided that `a` is a variable or a function parameter):



`!a` gives (provided that `a` is a variable or a function parameter):



The data model is analogous for all unary and binary operators, only the function differs:

Table 1Arithmetic operators

Operator	***Class***
`+ (unary)`	`FunctionPlus`
`- (unary)`	`FunctionMinus`
`+`	`FunctionAdd`
`-`	`FunctionSub`
`*`	`FunctionMul`
`/`	`FunctionDiv`
`%`	`FunctionMod`

Table 2Bitwise operators

<table>
<tr><th>Operator</th><th>Class</th></tr>
<tr><td>>></td><td>FunctionRsh</td></tr>
<tr><td><<</td><td>FunctionLsh</td></tr>
<tr><td>&</td><td>FunctionBand</td></tr>
<tr><td>|</td><td>FunctionBor</td></tr>
<tr><td>^</td><td>FunctionBxor</td></tr>
<tr><td>~</td><td>FunctionBnot</td></tr>
</table>

Table 3Relational and logical operators

<table>
<tr><th>Operator</th><th>Class</th></tr>
<tr><td><</td><td>FunctionLt</td></tr>
<tr><td><=</td><td>FunctionLe</td></tr>
<tr><td>></td><td>FunctionGt</td></tr>
<tr><td>>=</td><td>FunctionGe</td></tr>
<tr><td>==</td><td>FunctionEq</td></tr>
<tr><td>!=</td><td>FunctionNe</td></tr>
<tr><td>&&</td><td>FunctionAnd</td></tr>
<tr><td>||</td><td>FunctionOr</td></tr>
<tr><td>!</td><td>FunctionNot</td></tr>
</table>

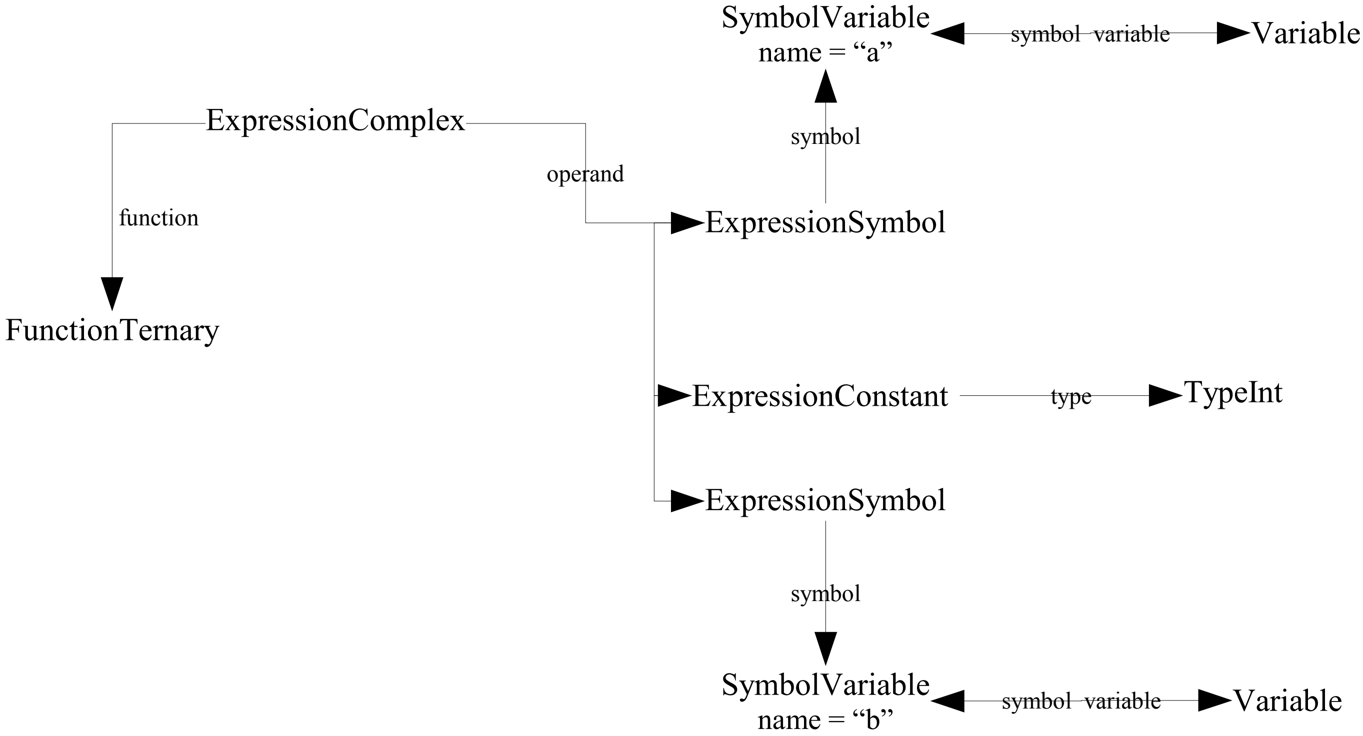
Table 4Assignment operators

<table>
<tr><th>Operator</th><th>Class</th></tr>
<tr><td>=</td><td>FunctionAssign</td></tr>
<tr><td>+=</td><td>FunctionAddAssign</td></tr>
<tr><td>-=</td><td>FunctionSubAssign</td></tr>
<tr><td>*=</td><td>FunctionMulAssign</td></tr>
<tr><td>/=</td><td>FunctionDivAssign</td></tr>
<tr><td>%=</td><td>FunctionModAssign</td></tr>
<tr><td>>>=</td><td>FunctionRshAssign</td></tr>
<tr><td><<=</td><td>FunctionLshAssign</td></tr>
<tr><td>&=</td><td>FunctionBandAssign</td></tr>
<tr><td>|=</td><td>FunctionBorAssign</td></tr>
<tr><td>^=</td><td>FunctionBxorAssign</td></tr>
<tr><td>++ (prefix)</td><td>FunctionPreInc</td></tr>
<tr><td>++ (postfix)</td><td>FunctionPostInc</td></tr>
</table>

Operator	*Class*
`-- (prefix)`	`FunctionPreDec`
`-- (postfix)`	`FunctionPostDec`

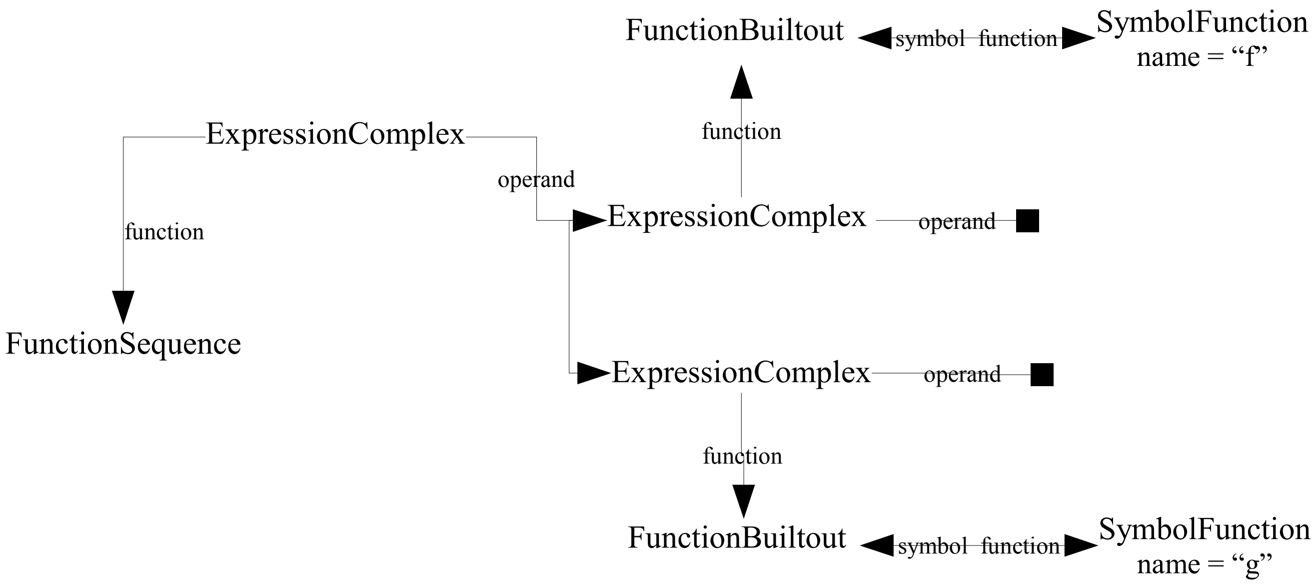
All other operators follow the same model:

`a ? 'a' : b`

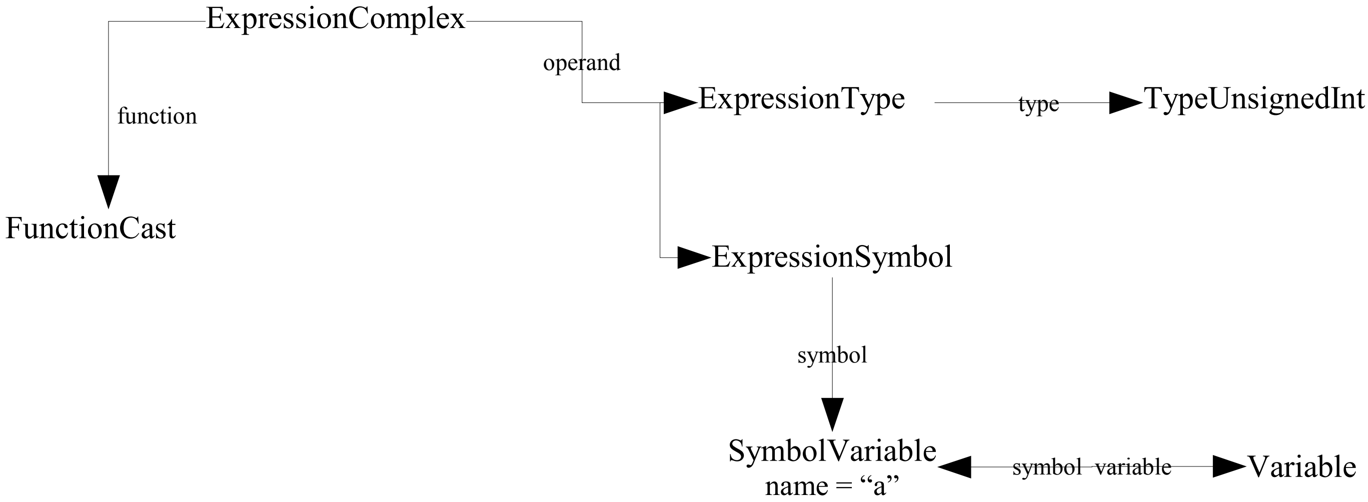


(note that `'a'` has type `int` in C).

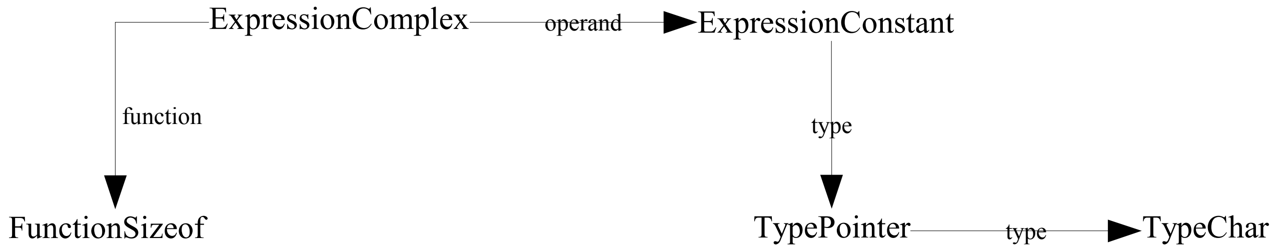
`f(), g()` (this is the comma operator, not the argument separator)



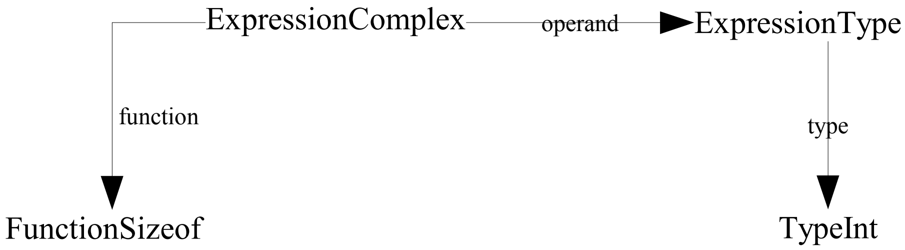
`(unsigned int)a`



`sizeof("a")`

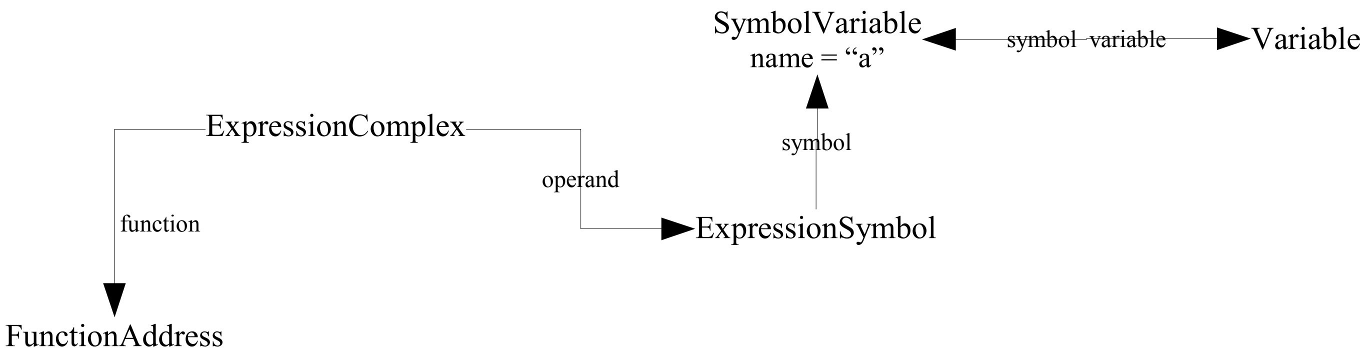


`sizeof(int)`

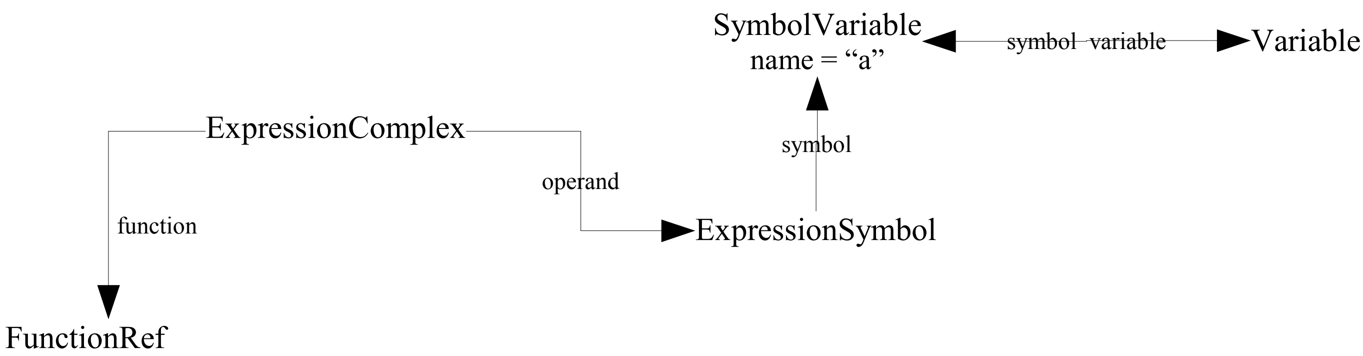


(note that `ExpressionType` is merely an adapter that allows to use a type as an expression.)

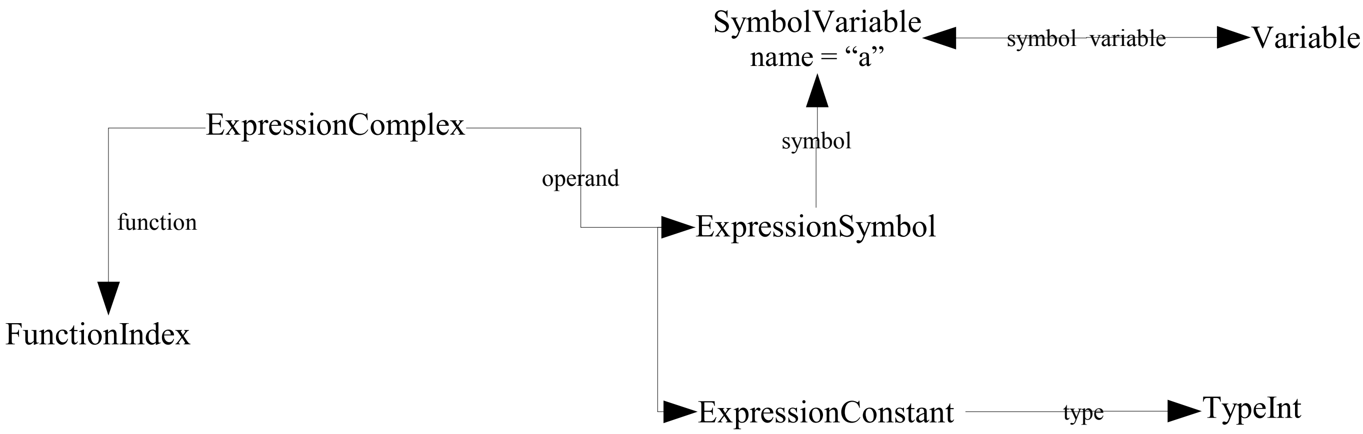
`&a`


SymbolVariable
name = "a"
symbol
variable
Variable
ExpressionComplex
operand
symbol
function
ExpressionSymbol
FunctionAddress

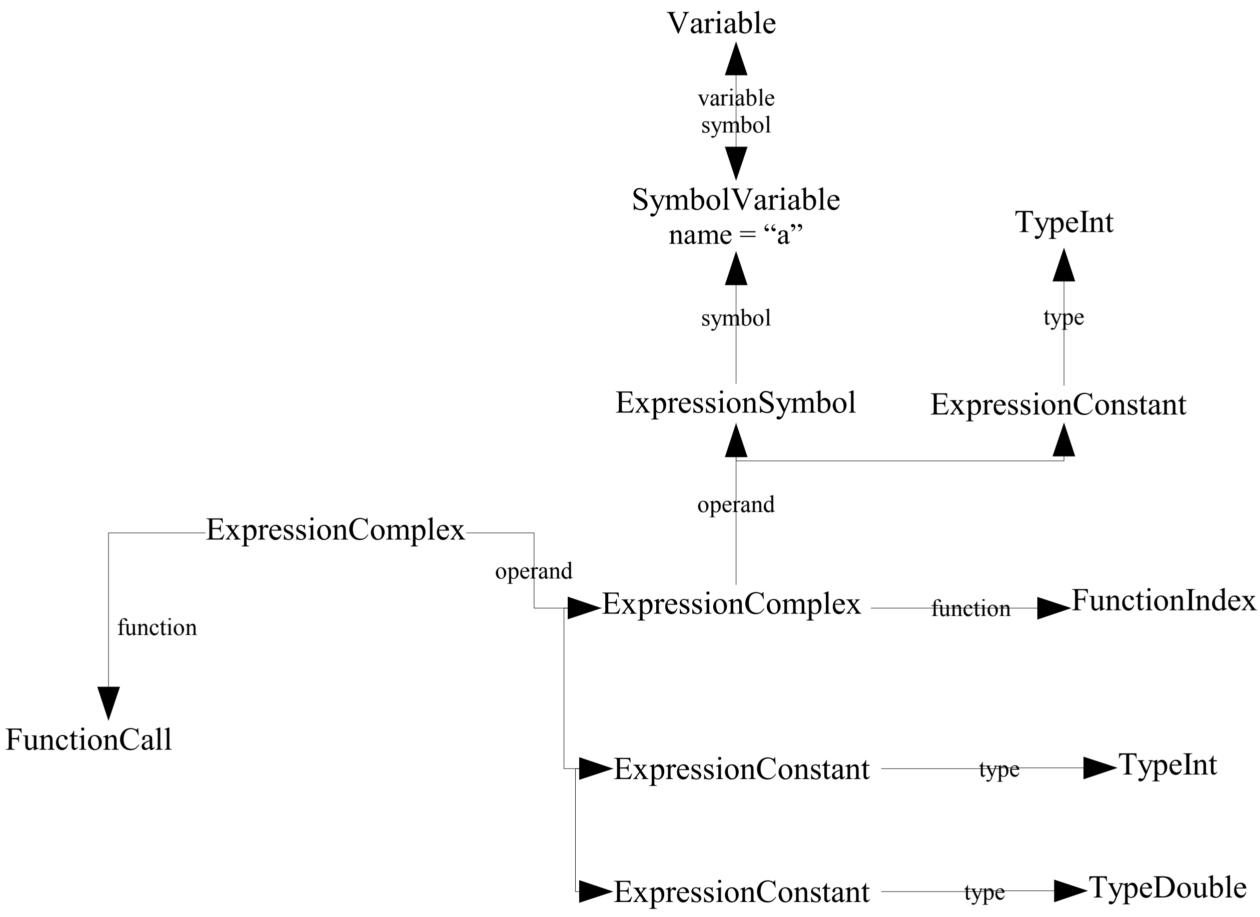
`*a`


SymbolVariable
name = "a"
symbol
variable
Variable
ExpressionComplex
operand
symbol
function
ExpressionSymbol
FunctionRef

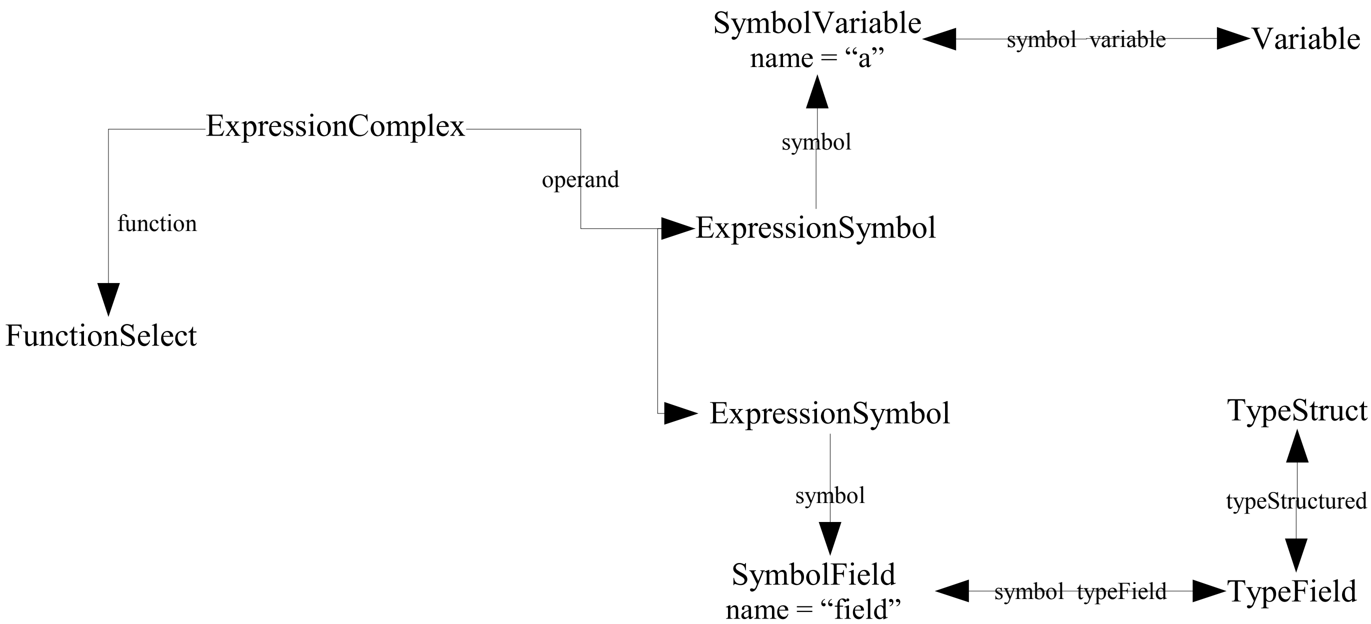
```
a[3]
```


SymbolVariable
name = “a”
symbol
variable
Variable
ExpressionComplex
operand
symbol
function
ExpressionSymbol
FunctionIndex
ExpressionConstant
type
TypeInt

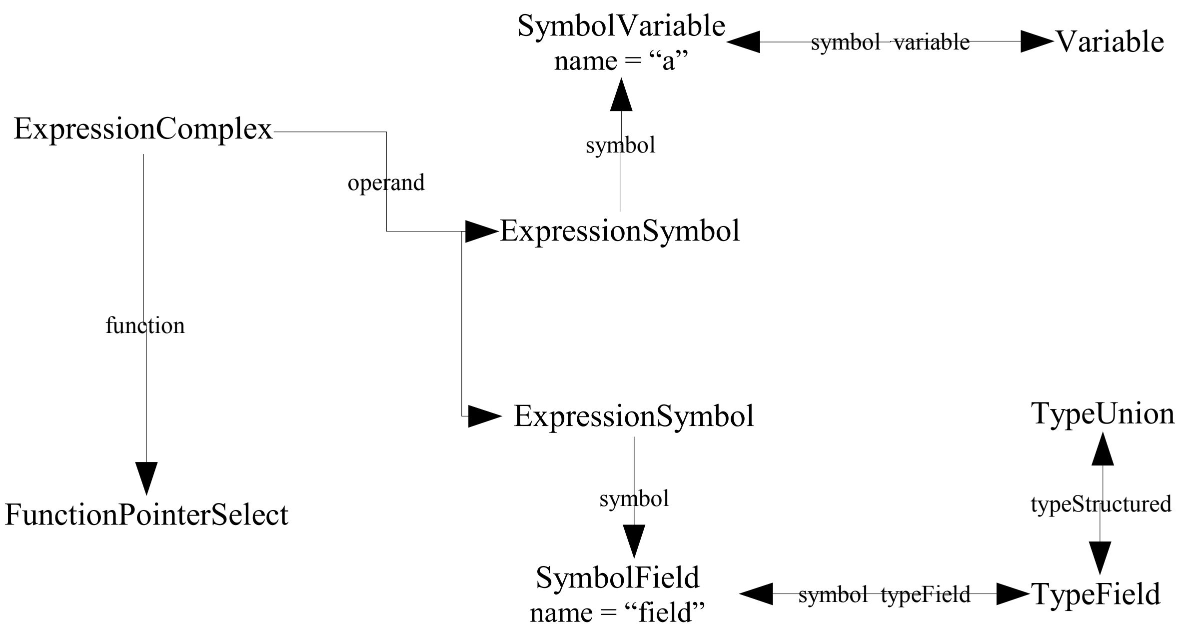
```
(*a[0])(1, 3.0)
```


Variable
variable
symbol
SymbolVariable
name = “a”
TypeInt
symbol
type
ExpressionSymbol
ExpressionConstant
operand
ExpressionComplex
operand
ExpressionComplex
function
FunctionIndex
function
FunctionCall
ExpressionConstant
type
TypeInt
ExpressionConstant
type
TypeDouble

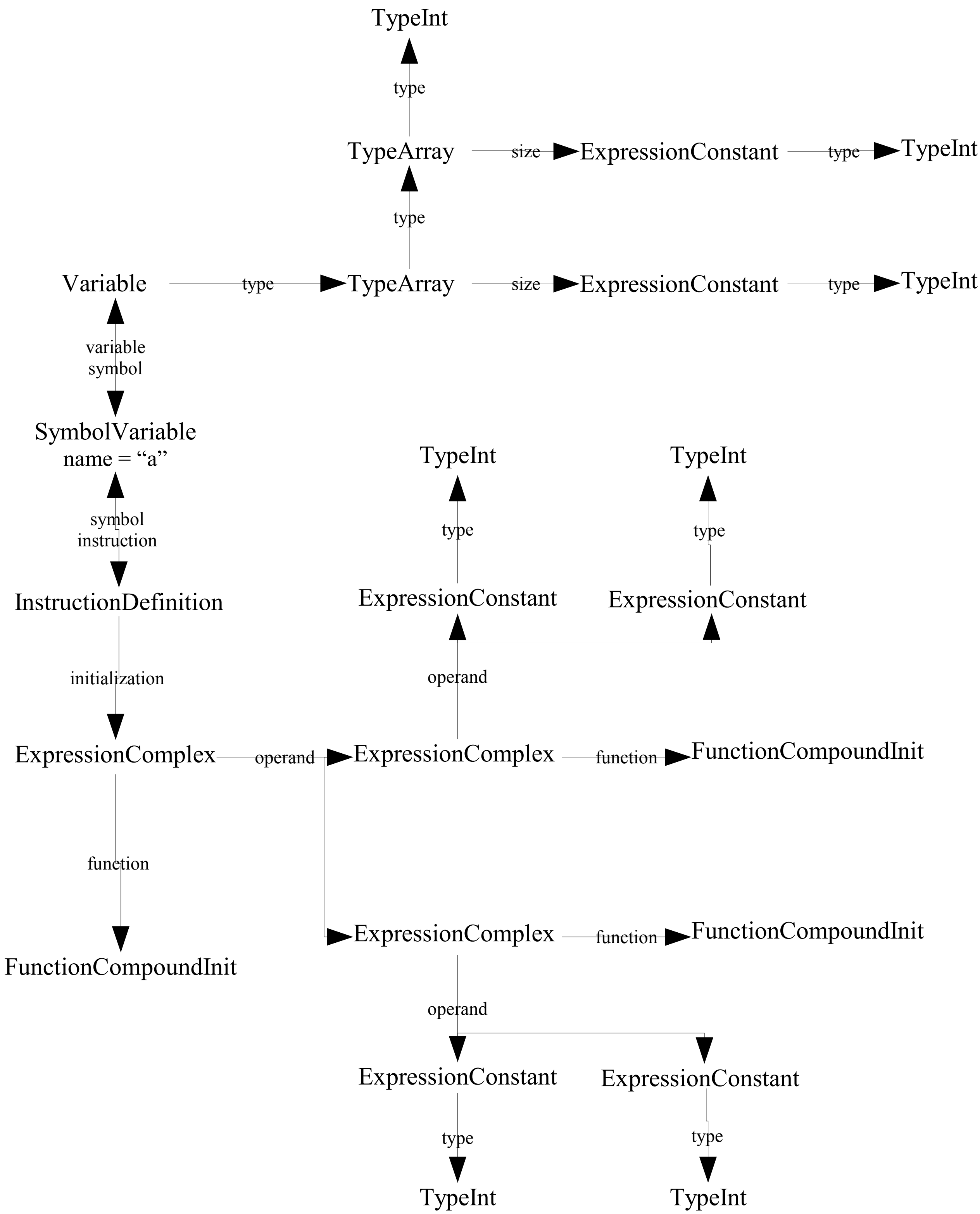
`a.field`



`a->field`



```
int a[2][2] = {{1, 2}, {3, 4}};
```

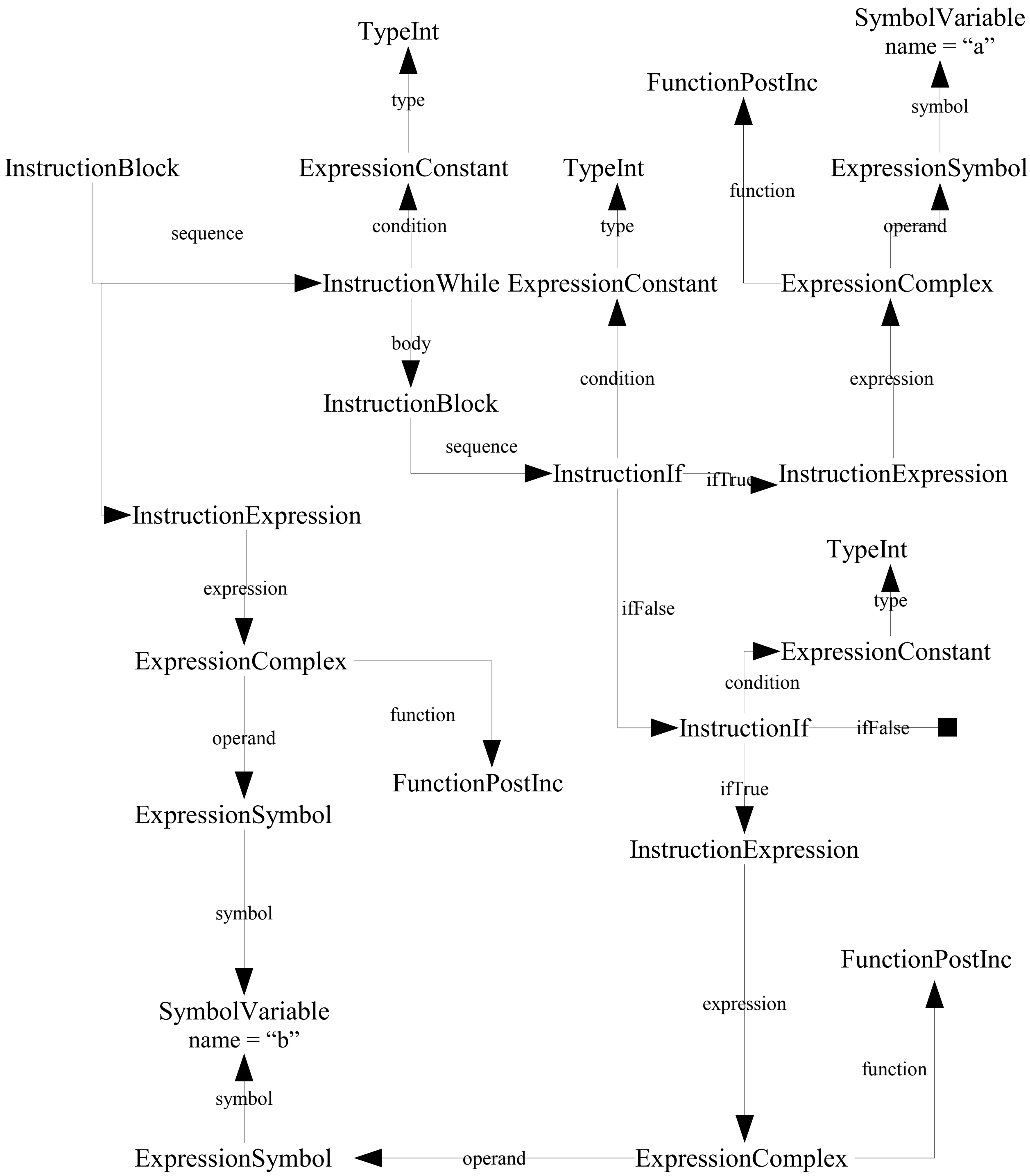


2.6. Instructions and Labels

The data model for instructions is rather straightforward, once `InstructionDeclaration`, `InstructionDefinition`, and `InstructionTentativeDefinition` are explained. The only remaining difficulty is with labels and `switch`.

But let's look first at a simple illustration of the data model for instructions:

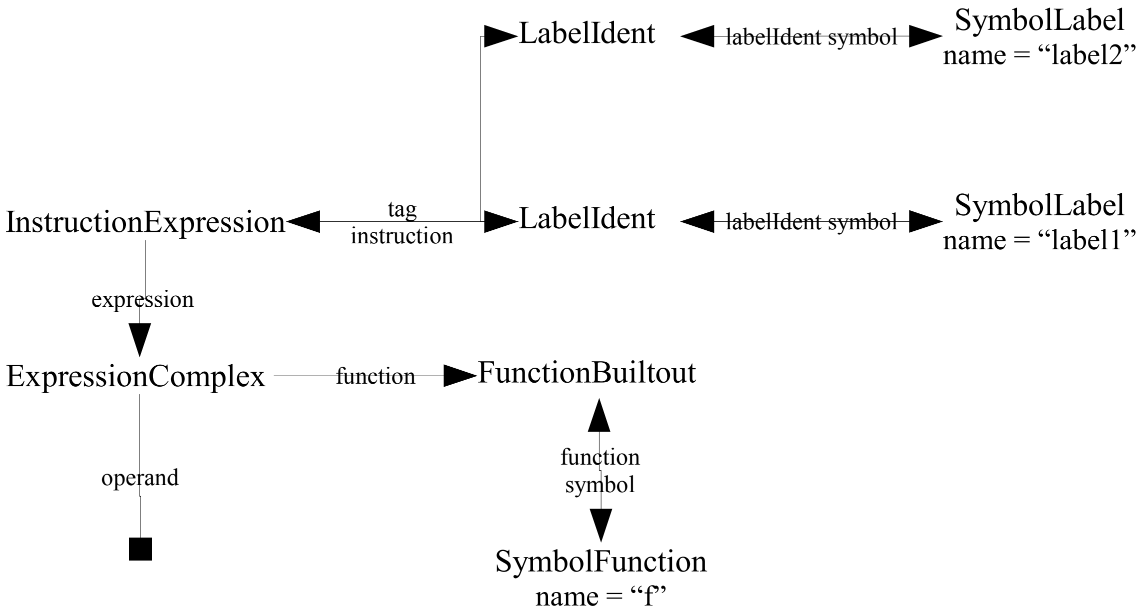
```
{
    while (1) {
        if (0)
            a++;
        else if (1)
            b++;
    }
    b++;
}
```



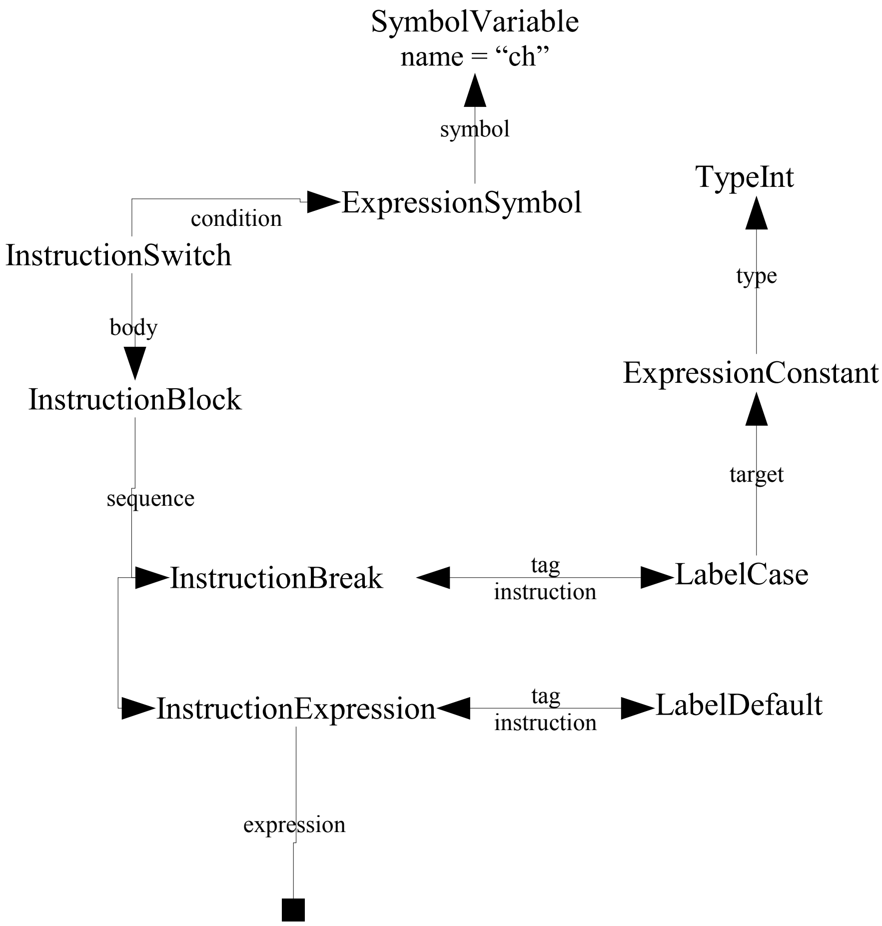
Instructions may be labeled, in order to allow the code to jump to a specific instruction with a `goto` (`InstructionGoto`) or a `switch` (`InstructionSwitch`). Thus the labels come in two flavors:

- Labels that may be used only in the body of a switch instruction: `LabelCase` and `LabelDefault`.
- Labels that may appear anywhere in the code: `LabelIdent`.

```
label1:
label2:
      f();
```



```
switch (ch) {
case 'a':
     break;
default: ;
}
```



3. Shortcuts

The **TCL verifier** defines several shortcuts to ease common tasks:

The `application` object, root of the data model, has roles to most kinds of objects of the data model. Beware, however, that following these links may be costly for large applications, since there may be numerous objects in these roles.

The `Instruction*` objects have a role, `subInstruction`, that allows direct navigation to the `Instruction*` objects that are directly dependent on them.

The `Instruction*` objects have a role, `allInstruction`, that allows direct navigation to all `Instruction*` objects that are dependent on them.

The `Instruction*` objects have a role, `expression`, that allows direct navigation to the `Expression*` objects that are directly dependent on them, for example in the role `condtion`.

The `Expression*` objects have a role, `subExpression`, that allows direct navigation to the `Expression*` objects that are directly dependent on them. For an `ExpressionComplex` object, this is equivalent to the role `operand`.

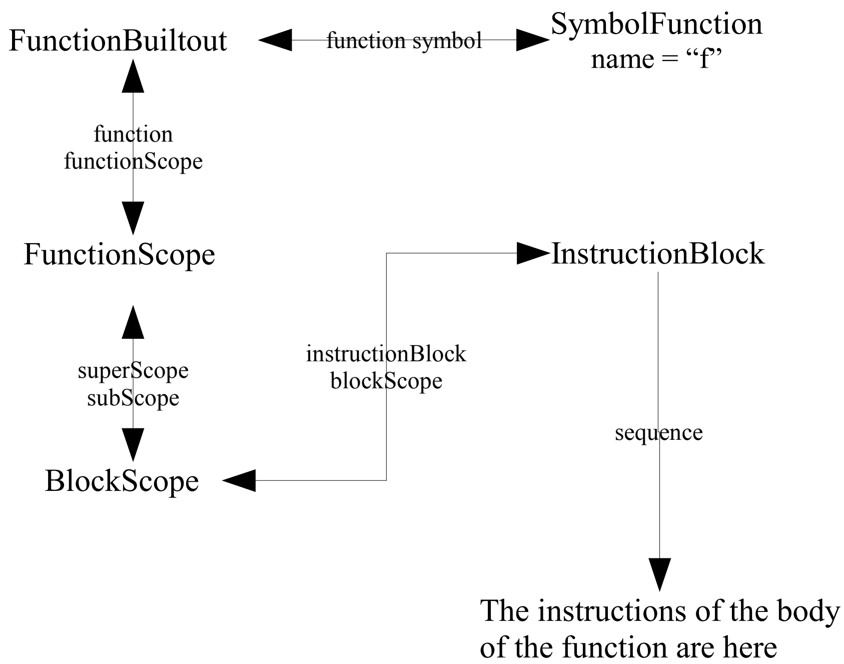
The `Expression*` objects have a role, `allExpression`, that allows direct navigation to all `Expression*` objects that are dependent on them.

The `allInstruction` and `allExpression` roles are very useful when searching for usage of identifiers in the code.

4. Special Cases

4.1. Finding the Function Body

It is often useful to find the body of a function, starting from a `FunctionBuiltout` object or a `SymbolFunction` object. The following schema describes how to retrieve it.



4.2. Implicit Function Declaration

The C language allows to call a function that is not declared. In such a case, the function is considered to be declared as returning int and with an unknown parameter list in the most enclosing scope.

The **TCL verifier** mimics this behavior by creating an `InstructionDeclaration` for a `SymbolFunction` for every undeclared identifier (function name or not). In order to reduce the cluttering of the data model for very old code that relies heavily on implicit function declaration, the `InstructionDeclaration` is created in the `ScopeGlobal`. These are the only `InstructionDeclaration` that may be found in the `instructionDef` role of the `ScopeGlobal`.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for

Copyright license

Trademarks