Dr. Annette Bieniusa
M.Sc. Peter Zeller

TU Kaiserslautern
Fachbereich Informatik
AG Softwaretechnik

Lab 4: Compiler and Language Processing Tools (WS 2018)

Deadline: 07.01.2019, 14:00

Editor support (optional)

In this optional exercise you will extend your MiniJava compiler and add support for interactive editors. Since there are a lot of different editors and different languages, it makes sense to use a standardized interface to provide language specific features to an editor. The state of the art for editor support is the *Language Server Protocol* (`https://microsoft.github.io/language-server-protocol/`.

If you download the template files for this exercise sheet you will get some basic language server support for showing syntax and type errors in Java files. To try out this feature, follow these steps:

1. Build your project with `./gradlew install`
   This will generate a jar-file `projectname.jar` in `./build/install/projectname/lib`.
2. Install Visual Studio Code.
   `https://code.visualstudio.com/`
3. Install our MiniJava extension from the marketplace.
   `https://marketplace.visualstudio.com/items?itemName=peterzeller.minijava-vscode`
4. Go to the Visual Studio Code settings and change the entry for `minijava.minijavaJar` to the location of your `projectname.jar` file from step 1.
5. Open a Java file with type errors – you should now see the error messages from your type checker.

Now you can extend your language server with some additional features by working on the implementation in the `MinijavaTextDocumentService` class:

1. Add support for hover texts by implementing the method `hover`. To do this you can the AST and try to find the element in the AST that is closest to the position of the cursor. Then you can return a message based on the found element.
2. Add support for going to the definition of variables and methods. For this you have to implement the method `definition`. Similarly to `hover` you can find the AST element at the cursor and if it is a `MJVarRef` you can return a link to the definition by using the results from your type analysis.
3. Add support for autocomplete of methods. For this you have to add the completion provider capability in the `initialize` method of class `MinijavaLanguageserver` by removing the comment for the respective line. Then you have to add an implementation for the `completion` method in `MinijavaTextDocumentService`. You can start with an approach similar to the other two features. If the cursor is at `MJMethodCall`, then you can determine the type of the receiver object using code from your type analysis and then use your class table to find all possible methods that can be suggested.

   One difficulty here is that autocomplete is usually invoked on incomplete programs. You will typically have the start of a method call like `obj.` but still miss the method name and the argument list. Therefore you will have to make your parser more robust so that it can produce an AST for these incomplete inputs.

From MiniJava to MiniLLVM (Part 1)

For the next step in our project, we will use MiniLLVM as an intermediate language. The structure and semantics of MiniLLVM are defined at the end of this exercise sheet.

The input to the translation phase is the annotated AST from the analysis phase. Please contact us, if you did not complete the last exercise and we will provide you the required material.

Using the AST and the results of the name- and type-analysis, you have to generate a MiniLLVM program with equivalent behavior to the MiniJava program given as input. Equivalence means that running the Java program and the corresponding MiniLLVM program produce the same output, when executed with the same inputs. When the Java program terminates with an exception, the corresponding MiniLLVM program should terminate using the special `HaltWithError` instruction. When the Java program terminates normally, the MiniLLVM program should terminate with exit code zero. The programs may differ, when the stack or heap memory is exhausted.

Translation of statements and expressions (10 points)

As the first part of the translation, you should handle the translation of statements and expressions. You only have to handle the translation of the main method and can ignore other classes and methods. Your translation must handle all statements and expressions, except for the ones related with object oriented programming and arrays, which you will have to handle later.

More precisely, your compiler should be able to translate the following statements and expressions: `Block, StmtIf, StmtWhile, StmtReturn, StmtPrint, StmtExpr, StmtAssign, ExprBinary, ExprUnary, BoolConst, VarUse, Number`.

Translation of arrays (5 points)

Extend your compiler with the translation of arrays, which includes the translation of the following expressions: `ArrayLookup, ArrayLength, ExprNull, NewIntArray`.

Hints

- Translation of the object oriented constructs (including the expressions `FieldAccess, MethodCall, ExprThis, NewObject`) will be handled on the next exercise sheet. Design your compiler, so that you can easily add these features later.
- You may assume that you are translating only name- and type-correct MiniJava programs.
- The AST elements (see description in Section 1.3) have two different kind of attributes: Attributes marked with `ref` and normal attributes.

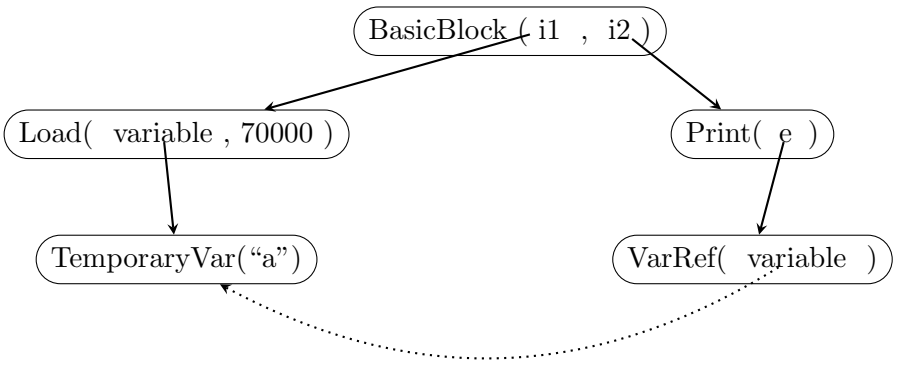
  For normal attributes, the stored element may occur only once in the tree, because they store a reference to their parent element.

  Ref-attributes can refer to elements stored at other places in the tree. Ref-attributes do not affect the parent-attribute of the referenced element and so there can be many references to the same element.

As an example, consider the following declaration and use of a `TemporaryVar a`:

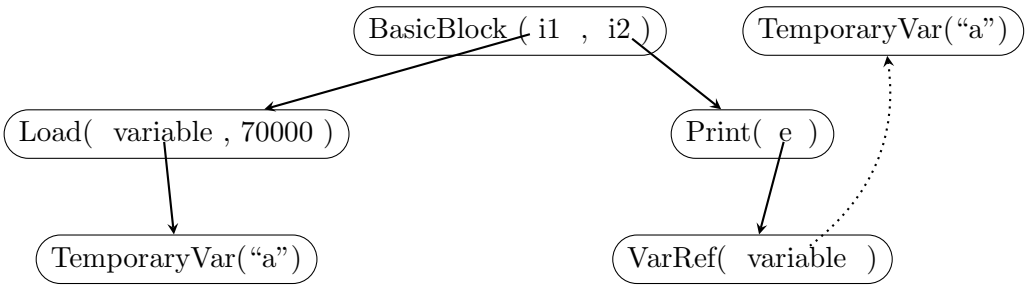
```
TemporaryVar a = TemporaryVar("a");
BasicBlock entry = BasicBlock(
       Load(a, ConstInt(70000)),
       Print(VarRef(a))
);
```

The variable `a` is owned by the `ReadByte` instruction, so the parent of `a` is the `ReadByte` instruction. The same variable `a` is referenced by the `VarRef` operand. This situation can be visualized as follows, using dotted lines for references and solid lines for a containment-relation:



In contrast to the example above, the following example would produce two different `TemporaryVar` elements, of which one is not rooted in the tree:

```
BasicBlock entry = BasicBlock(
       Load(TemporaryVar("a"), ConstInt(70000)),
       Print(VarRef(TemporaryVar("a")))
);
```



- As each element can have only one parent, it is not possible to reuse Operands. Therefore the following code will not work and throw a runtime exception:

  ```
  Operand c = ConstInt(1);
  TemporaryVar t = TemporaryVar("t");
  Instruction i = BinaryOperation(t, c, Add(), c);
  ```

  You can use the `copy` method to solve this problem in some cases.

- As mentioned above, the LLVM AST uses references to represent uses of declarations. This has some benefits: The declarations can be renamed without changing the uses, it is not necessary to do a name analysis after generation, and the API makes better use of the Java type system.

  However, it means that elements have to be created before they can be used. For example you have to create an (empty) `BasicBlock`, before you can create the `Jump` instruction, which jumps to this block.

- When the names inside a LLVM program are not unique, the Printer will automatically make them unique before printing. You should be aware of this fact while debugging.
- Make use of the `CommentInstr` to make the generated code traceable to its source. For example, you can add a comment with the original line number before translating each statement.
- If you want to pass additional information from your analysis phase to the translation phase, adapt the code in `main/MiniJavaCompiler.java`.

Submission

- Deadline: 07.01.2019, 14:00
- Late submissions will not be accepted.
- Submit your solution via your group's Git repository (in a folder named "ex4").
- You have to put the materials provided with this assignment into the corresponding folders in your solution for the previous assignment and integrate these materials into your solution.
- You are strongly encouraged to test your solution with your own test cases.
- Write a readme file (in text, markdown, or pdf format) and submit it to the repository. Explain the overall structure of your solution and the pre- and post-conditions for the translation of translating expressions and statements. If the methods for translating expressions and statements have side-effects, describe what they are.

1 The miniLLVM intermediate language

MiniLLVM is an intermediate language which uses static single assignment (SSA) form and is modeled as a subset of LLVM with some macros to simplify the translation.

For this exercise you should install version 3.8, 3.9 or 4.0 of LLVM. The executables `opt` and `lli` should be either in your `PATH`, or in a folder given by the `LLVM_COMPILER_PATH` environment variable.

For Mac, the easiest way to install LLVM is using Homebrew (https://brew.sh/). After installing Homebrew itself using the installation instructions on the Homebrew homepage, you can install LLVM with the command `brew install llvm`. Since the LLVM binaries are not installed into the PATH, you need to set the `LLVM_COMPILER_PATH` environment variable to the installation path (usually `/usr/local/opt/llvm/bin`).

Under Linux, you will most likely find LLVM packages in your package manager. Make sure that you install a package that installs the required links as well (e.g. `llvm` under Ubuntu, not `llvm-3.8`). The LLVM executables need to be callable without version postfix (e.g. `lli`, not `lli-3.8`).

If you are using Windows, you can have to install the ClangOnWin distribution[1] for the two required executables `opt.exe` and `lli.exe` to be installed.

1.1 Further Resources for LLVM

- Reference manual
  `http://llvm.org/docs/LangRef.html`
- LLVM tutorial (the Kaleidoscope language)
  `http://llvm.org/docs/tutorial/index.html`
- Hints on translating high-level constructs to LLVM
  `https://github.com/idupree/llvm-doc/blob/master/MappingHighLevelConstructsToLLVMIR.rst`

1.2 Commands for using LLVM on the commandline

Usually LLVM is integrated into compilers as a library, but it is also possible to use it from the commandline (which we will do in this project). The following commands show the most common use cases:

```
# Compile C file test.c to llvm:
clang test.c -S -emit-llvm

# run file test.ll in the llvm interpreter
lli test.ll

# Optimize a file:
opt -O3 -S test.ll > test2.ll

# Remove alloca instructions (convert to SSA form)
opt -mem2reg -S test.ll > test2.ll

# Compile an llvm file to an executable:
  # 1. llvm -> bitcode
  llvm-as blub.ll
  # 2. bitcode -> object file
  llc -filetype=obj blub.bc
```

[1] available under `https://sourceforge.net/projects/clangonwin/files/MsvcBuild/4.0/`

```
  # 3. link object file:
  clang -o blub blub.o
# Show control flow graph / dominator tree (requires graphviz)
opt -analyze -view-cfg test.ll
opt -analyze -view-cfg-only test.ll
opt -analyze -view-dom-only test.ll
opt -analyze -domfrontier test.ll
```

1.3 Abstract Syntax

```
Prog(TypeStructList structTypes,
    GlobalList globals,
    ProcList procedures)

Global(ref Type type, String name, boolean isConstant, Const
    initialValue)

Variable =
      Parameter(ref Type type, String name)
    | TemporaryVar(String name)

Proc(String name,
    ref Type returnType,
    ParameterList parameters,
    BasicBlockList basicBlocks)

// a BasicBlock is a list of instructions
BasicBlock * Instruction


// instructions:
Instruction =
      Assign
    | TerminatingInstruction
    | Print(Operand e)
    | Store(Operand address, Operand value)
    | CommentInstr(String text)

Assign =
      Alloc(TemporaryVar var, Operand sizeInBytes)
    | Alloca(TemporaryVar var, ref Type type)
    | BinaryOperation(TemporaryVar var, Operand left,
          Operator operator, Operand right)
    | Bitcast(TemporaryVar var, ref Type type, Operand expr)
    | Call(TemporaryVar var, Operand function, OperandList arguments)
    | GetElementPtr(TemporaryVar var, Operand baseAddress,
          OperandList indices)
    | Load(TemporaryVar var, Operand address)
    | PhiNode(TemporaryVar var, ref Type type,
                      PhiNodeChoiceList choices)

Operator = Add() | Sub() | Mul() | Sdiv() | Srem()
        | And() | Or() | Xor()
```

```
        | Eq() | Sgt() | Sge() | Slt() | Sle()

PhiNodeChoice(ref BasicBlock label, Operand value)




// terminating instructions:
TerminatingInstruction =
      Branch(Operand condition, ref BasicBlock ifTrueLabel,
           ref BasicBlock ifFalseLabel)
    | Jump(ref BasicBlock label)
    | ReturnExpr(Operand returnValue)
    | ReturnVoid()
    | HaltWithError(String msg)

// operands:
Operand =
      Const
    | VarRef(ref Variable variable)

Const =
      ConstBool(boolean boolVal)
    | ConstInt(int intVal)
    | GlobalRef(ref Global global)
    | ProcedureRef(ref Proc procedure)
    | Nullpointer()
    | Sizeof(ref TypeStruct structType)
    | ConstStruct(ref TypeStruct structType, ConstList values)

// types:
Type =
      TypeArray(ref Type of, int size)
    | TypeBool()
    | TypeByte()
    | TypeInt()
    | TypePointer(ref Type to)
    | TypeNullpointer()
    | TypeProc(TypeRefList argTypes, ref Type resultType)
    | TypeStruct(String name, StructFieldList fields)
    | TypeVoid()

StructField(ref Type type, String name)
```

1.4 Semantics and Types

1.4.1 Programs

A program consists of a list of struct types, a list of global variables, and a list of procedures.

There must be one procedure named `main` with no arguments and return type integer, which is the entry point of the program.

1.4.2 Globals

Globals have a name and a type and are initialized with constant expressions. For constant values the boolean `isConstant` can be set to **`true`**.

At runtime a global has a fixed memory address, which can be retrieved via the `GlobalRef` operand.

1.4.3 Struct Types

Structs are like in C and contain a list of fields. Each field has a name and a type, however the name is only used for debugging and otherwise ignored.

1.4.4 Procedures

A procedure has a name, a list of parameters, a return type, and an implementation given by a list of basic blocks.

1.4.5 Basic blocks

A procedure always starts with executing the first basic block in the list (usually this block is named "entry").

A basic block must always end with a `TerminatingInstruction`.

1.4.6 Instructions

Note that LLVM has a very small instruction sets. Most instructions that could be represented by other instructions are not included in LLVM. For example there are no unary operations, since `xor` and `sub` can be used instead. Similarly, there are no simple move-assignments (assigning the value of one variable to another).

Print prints the given integer. This is not a built-in LLVM instruction, but a macro which is implemented by calling the printf function in the C standard library.

Java representation	printed LLVM
`Print(ConstInt(12))`	`call void @print(i32 12)`

Store stores the given `value` at the given `address`.

`http://llvm.org/docs/LangRef.html#store-instruction`

Java representation	printed LLVM

`TemporaryVar x = TemporaryVar("x");`
`...`
`Alloca(x, TypeInt()),`
`Store(VarRef(x), ConstInt(42))`
	`%x = alloca i32`
	`store i32 42, i32* %x`

CommentInstr can be used to add comments to the generated code.

Java representation	printed LLVM
`CommentInstr("This is a comment")`	`;This is a comment`

1.4.7 Instructions - Assignments

Most instructions are assignments and assign the result of their computation to a new temporary variable. Note that the variable of an assignment is not used with `ref`, which means that the variable is owned by the assignment and **the same variable can only be used by one assignment**. This ensures that the program always is in SSA form. You can use the `Load`, `Store`, and `Alloca` instructions for translating mutable variables.

Alloc allocates some new space **on the heap**. The parameter `size` is an integer value which denotes the number of bytes to allocate. The result of an allocation is a pointer to byte, which can be cast to other pointer types (see `Bitcast` instruction). The result may be **`null`**, if the system cannot allocate enough memory.

This is not a built-in LLVM instruction[2], but a marco which is printed as a call to the malloc function in the C standard library.

Java representation	printed LLVM
`Alloc(TemporaryVar("t"), ConstInt(100))`	`%t = call i8* @malloc(i32 100)`

Alloca allocates some new space on the **stack**. This space is freed again after the current procedure invocation ends. The operation will allocate space for the given type and save the pointer to this space in the given variable.

`http://llvm.org/docs/LangRef.html#alloca-instruction`

Java representation	printed LLVM
`Alloca(TemporaryVar("x"), TypeInt())`	`%x = alloca i32`

BinaryOperation computes a normal binary operation. All operators are executed in a strict way, in particular `And` and `Or` are not lazy (or "short-circuiting").

`http://llvm.org/docs/LangRef.html#binary-operations`
`http://llvm.org/docs/LangRef.html#icmp-instruction`
`http://llvm.org/docs/LangRef.html#bitwise-binary-operations`

[2]alloc was a built-in instruction in earlier versions of LLVM

Java representation	printed LLVM

`BinaryOperation(x,`
`  ConstInt(5), Add(), ConstInt(4)),`
`BinaryOperation(y,`
`  VarRef(x), Sdiv(), ConstInt(2)),`
`BinaryOperation(z,`
`  VarRef(x), Slt(), VarRef(y))`
	`%x = add i32 5, 4`
	`%y = sdiv i32 %x, 2`
	`%z = icmp slt i32 %x, %y`

Bitcast interprets an expression as having a different type.

`http://llvm.org/docs/LangRef.html#bitcast-to-instruction`

Java representation	printed LLVM

`Alloc(x, ConstInt(128)),`
`Bitcast(y, TypePointer(myStruct),`
`  VarRef(x))`
	`%x = call i8* @malloc(i32 128)`
	`%y = bitcast i8* %x to %myStruct*`

Call calls the method `function` with the given Arguments `args`. The procedure `proc` must evaluate to a pointer to a procedure (see `ProcedureRef` operand). The argument types must be equal to the types of the formal parameters of the procedure.

`http://llvm.org/docs/LangRef.html#call-instruction`

Java representation	printed LLVM
`Call(x, ProcedureRef(f), OperandList(` `  ConstInt(4), ConstBool(true)))`	`%x = call i32 @f(i32 4, i1 1)`

GetElementPtr is used to get the address of a subelement of an aggregate data structure (struct or array). It performs address calculation only and does not access memory. The `baseAddress` must be a pointer to a struct or array. The `indices` are used to index into the pointer. The first index is used to index in the given `baseAddress`. The subsequent indices are used to index into the array or struct being pointed to. For structs the number represents the index of the field in the struct definition (starting with index 0 for the first field).

`http://llvm.org/docs/LangRef.html#getelementptr-instruction`

Java representation	printed LLVM

`TypeStruct myStruct = TypeStruct(`
`  "myStruct", StructFieldList(`
`  StructField(TypeBool(), "a"),`
`  StructField(TypeBool(), "b"),`
`  StructField(TypeInt(), "c")));`
`// if p is a pointer to myStruct, load`
`// value of field b of that struct:`
`GetElementPtr(x, VarRef(p),`
`  OperandList(ConstInt(0), ConstInt(1))),`
`Load(y, VarRef(x))`
	`%myStruct = type {`
	`  i1 ; a`
	`  ,i1 ; b`
	`  ,i32 ; c`
	`}`
	`...`
	`%x = getelementptr %myStruct,`
	`  %myStruct* %p, i32 0, i32 1`
	`%y = load i1, i1* %x`

Load loads a value from the given `address`.

`http://llvm.org/docs/LangRef.html#load-instruction`

Java representation	printed LLVM

`Alloca(x, TypeInt()),`
`Store(VarRef(x), ConstInt(32)),`
`Load(y, VarRef(x))`
	`%x = alloca i32`
	`store i32 32, i32* %x`
	`%y = load i32, i32* %x`

PhiNode Selects a value based on which BasicBlock was executed directly before the current block. PhiNodes can only be used at the beginning of a basic block.

A basic block A is called a predecessor of a block B, if A contains a jump or branch instruction to go to block B. A basic block can have several predecessors. Phi nodes are used to handle SSA form in this case. A phi node has a list of choices, which are (predecessor, operand) pairs. When a phi node is executed, the operand is chosen based on which basic block was executed directly before. Consequently a phi node should have one entry for each possible predecessor.

`http://llvm.org/docs/LangRef.html#i-phi`

Java representation	printed LLVM

`BasicBlock block1 = BasicBlock(`
`  Load(a1, VarRef(x))`
`);`
`block1.setName("b1");`
`BasicBlock block2 = BasicBlock(`
`  Load(a2, VarRef(y))`
`);`
`block2.setName("b2");`
`BasicBlock block3 = BasicBlock(`
`  PhiNode(a, TypeInt(), PhiNodeChoiceList(`
`  PhiNodeChoice(block1, Ast.VarRef(a1)),`
`  PhiNodeChoice(block2, Ast.VarRef(a2))`
`  ))`
`);`
`block3.setName("b3");`
`block1.add(Jump(block3));`
`block2.add(Jump(block3));`
 	
`BasicBlockList blocks = BasicBlockList(`
`  block1, block2, block3`
`);`
	`b1:`
	`  %a1 = load i32, i32* %x`
	`  br label %b3`
	
	`b2:`
	`  %a2 = load i32, i32* %y`
	`  br label %b3`
	
	`b3:`
	`  %a = phi i32 [ %a1, %b1 ], [ %a2, %b2 ]`

1.4.8 Terminating Instructions

Branch is a conditional jump. It jumps to the ifTrueLabel if the condition is true, otherwise it jumps to the ifFalseLabel.

`http://llvm.org/docs/LangRef.html#br-instruction`

Java representation	printed LLVM
`Branch(VarRef(c), ifTrue, ifFalse)`	`br i1 %c, label %ifTrue, label %ifFalse`

Jump is an unconditional jump, which just transfers control to the given label.

`http://llvm.org/docs/LangRef.html#br-instruction`

Java representation	printed LLVM
`Jump(block2)`	`br label %block2`

ReturnExpr returns the given return value from the current procedure.

`http://llvm.org/docs/LangRef.html#ret-instruction`

Java representation	printed LLVM
`ReturnExpr(ConstInt(42))`	`ret i32 42`

ReturnVoid returns from a procedure with no return value.

`http://llvm.org/docs/LangRef.html#ret-instruction`

Java representation	printed LLVM
`ReturnVoid()`	`ret void`

HaltWithError aborts the program with the given error message.

This is not a built-in LLVM instruction, but a macro which is printed as a call to the exit function from the C standard library with exit code 222, followed by the `unreachable` terminating instruction from LLVM.

Java representation	printed LLVM

`HaltWithError("description")`	`; ERROR: description`
	`call i32 (i8*, ...) @printf(i8*`
	`  getelementptr inbounds ([13`
	`  x i8], [13 x i8]*`
	`  @.print_message_1, i32 0,`
	`  i32 0))`
	`call void @exit(i32 222)`
	`unreachable`

1.4.9 Operands

Operands are either a single variable reference or a constant.

VarRef is a reference to a temporary variable or parameter.

ConstBool is a constant boolean value.

ConstInt is a constant integer value.

GlobalRef returns a pointer to the given global variable.

ProcedureRef returns a pointer to the given procedure.

Nullpointer is a special pointer with no memory region assigned.

Sizeof is a constant which computes the size of a struct type.

ConstStruct is a constant to create a value of a struct. This expression can only be used to initialize global variables.

1.4.10 Types

TypeArray The array type. Use a size of zero, if the size is not known at compile-time. (`[size x t]`)

TypeBool Type for booleans (1 bit, `i1`)

TypeByte Type for bytes (8 bits, `i8`)

TypeInt Type for integers (32 bits, `i32`)

TypePointer A pointer type (`t*`)

TypeNullpointer The type of the null-constant. This is not a real LLVM type, but is useful for typechecking.

TypeProc A procedure type (`t(t,...,t)`).

TypeStruct The type of a struct. A struct has a list of fields. LLVM only uses the type of the fields, we also store a name for debugging and printing.

TypeVoid Return type of void functions (**`void`**).