Compilation — a primer

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau
http://www.softlang.org/
Compilation versus interpretation

- **Interpretation**: An interpreter executes elements of a software language, for example, to produce the I/O behavior of a function, the query result of a database query, or the object graph corresponding to method invocations in an OO program.

- **Compilation**: A compiler translates (transforms) elements of an executable software language into elements of another executable software language. This translation may or may not lower the level of abstraction.

Let's discuss examples!
Simplified data flow in a compiler. The rectangles with rounded edges represent logical phases of compilation. The remaining nodes (rectangles and triangles) correspond to input and output, as expressed by the direction of the arrows.

In addition to the basic dichotomy of compilation versus interpretation, there is also the specific classification of implementation strategies for domain-specific languages—internal versus external versus embedded DSL, as discussed in Section 1.2.4.

1.3.2.2 Architecture of a compiler

Compilers and interpreters consist of several components—a compiler's decomposition into standard components with the associated data flow is summarized in Figure 2; see textbooks on compiler construction [Aho+06; Lou97; AP02] for an in-depth discussion. Thus, the source code ('text') is mapped to a parse tree (or an abstract syntax tree (AST)) which is then further enriched with attributes and links. Eventually, code of a virtual or actual machine is generated. These conceptual phases may be properly separated ('multi-pass compilation') or integrated into one actual phase ('single-pass compilation'). The components are explained more in detail as follows:

**Parser**

A parser verifies the conformance of given input (i.e., text) to the syntax rules of a language and represents the input in terms of the structure defined by the rules. A parser performs parsing. Compilers and interpreters begin by parsing. Many other language processors, as discussed below, also involve parsing.

**Semantic analysis**

A parse tree only represents the structure of the source code. For any sort of non-trivial treatment such as code generation, the parse tree needs to be enriched with attributes and links related to typing and name binding. Names with their bindings and other attributes may be aggregated in a data structure which one refers to as symbol table or environment.

**Code generator**

The enriched parse tree is translated, more or less directly, into machine code, i.e., code of some actual or virtual machine. In particular, code generation involves resource and storage decisions such as register location, i.e., assigning program variables to processor registers of the target machine.
1.3.2.2 Architecture of a compiler

Compilers and interpreters consist of several components—a compiler's decomposition into standard components with the associated data flow is summarized in Figure 2; see textbooks on compiler construction [Aho+06; Lou97; AP02] for an in-depth discussion. Thus, the source code ('text') is mapped to a parse tree (or an abstract syntax tree (AST)) which is then further enriched with attributes and links. Eventually, code of a virtual or actual machine is generated. These conceptual phases may be properly separated ('multi-pass compilation') or integrated into one actual phase ('single-pass compilation'). The components are explained more in detail as follows:

- **Parser**
  A parser verifies the conformance of given input (i.e., text) to the syntax rules of a language and represents the input in terms of the structure defined by the rules. A parser performs parsing. Compilers and interpreters begin by parsing. Many other language processors, as discussed below, also involve parsing.

- **Semantic analysis**
  A parse tree only represents the structure of the source code. For any sort of non-trivial treatment such as code generation, the parse tree needs to be enriched with attributes and links related to typing and name binding. Names with their bindings and other attributes may be aggregated in a data structure which one refers to as symbol table or environment.

- **Code generator**
  The enriched parse tree is translated, more or less directly, into machine code, i.e., code of some actual or virtual machine. In particular, code generation involves resource and storage decisions such as register location, i.e., assigning program variables to processor registers of the target machine. In

- **Optimization**
  Optimization may be expressed at the level of enriched parse trees, IR, and assembly code.

- **An extra IR (intermediate representation) may be leveraged.**

- **An extra assembly code representation may be leveraged, subject to an assembler.**

- **Optimization may be expressed at the level of enriched parse trees, IR, and assembly code.**
Euclidian division in imperative language

```
{ 
  // Sample operands for Euclidian division
  x = 14;
  y = 4;

  // Compute quotient q=3 and remainder r=2
  q = 0;
  r = x;
  while (r >= y) {
    r = r - y;
    q = q + 1;
  }
}
```
Euclidian division as a parse tree

```haskell
-- // Compute quotient \( q \) and remainder \( r \) for dividing \( x \) by \( y \)
-- \( q = 0; r = x; \) while \( (r \geq y) \) \{ \( r = r - y; q = q + 1; \) \}

`euclideanDiv :: Stmt`

```
`euclideanDiv =`

```
    Seq (Assign "q" (IntConst 0)) (Seq (Assign "r" (Var "x"))
        (While
            (Binary Geq (Var "r") (Var "y"))
            (Seq (Assign "r" (Binary Sub (Var "r") (Var "y")))
                (Assign "q" (Binary Add (Var "q") (IntConst 1)))))))
```

We use Haskell as the implementation language here.

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Interpretation

—— Execution of statements
execute :: Stmt → Store → Store

—— Results of expression evaluation
type Value = Either Int Bool

—— Stores as maps from variable names to values
type Store = Map String Value

▶ execute euclideanDivision (fromList ["x", Left 13], "y", Left 4])
fromList [("q", Left 3), "r", Left 2), ("x", Left 14), ("y", Left 4)]
When defining the type `Store`, we use Haskell’s library type `Map` to model stores as maps (say, dictionaries) from variable names to values. Here is how we expect to use the interpreter.

Interactive Haskell session:

```
execute euclideanDivision (fromList [("x", Left 13), ("y", Left 4)])
```

Thus, we start from a store with suitable arguments `x` and `y` for division; interpretation returns a store with the variables `x` and `y` unchanged and with `q` and `r` bound to the computed quotient and remainder. We are ready to present the interpreter.

**Illustration 5.5 (A BIPL interpreter)**

```haskell
-- Execution of statements
execute :: Stmt → Store → Store
execute Skip m = m
execute (Assign x e) m = insert x (evaluate e m) m
execute (Seq s1 s2) m = execute s2 (execute s1 m)
execute (If e s1 s2) m = execute (if Right b = evaluate e m then s1 else s2) m
execute (While e s) m = execute (If e (Seq s (While e s)) Skip) m

-- Evaluation of expressions
evaluate :: Expr → Store → Value
evaluate (IntConst i) _ = Left i
evaluate (Var x) m = m!x
evaluate (Unary o e) m = uop o (evaluate e m)
evaluate (Binary o e1 e2) m = bop o (evaluate e1 m) (evaluate e2 m)

-- Interpretation of unary operators
uop :: UOp → Value → Value
uop Negate (Left i) = Left (negate i)
uop Not (Right b) = Right (not b)

-- Interpretation of binary operators
bop :: BOp → Value → Value → Value
bop Add (Left i1) (Left i2) = Left (i1+i2)
```

*Interpreter (Just mentioned in passing)*
Euclidean division in assembly code

euclideanDiv = [
    Const 14, Store "x", -- x = 14;
    Const 4, Store "y", -- y = 4;
    Const 0, Store "q", -- q = 0;
    Load "x", Store "r", -- r = x;
    Label "0", -- Beginning of while loop
    Load "r", Load "y", Geq, -- (r >= y)
    Not, CJump "1", -- Skip while loop
    Load "r", Load "y", Sub, Store "r", -- r = r - y;
    Load "q", Const 1, Add, Store "q", -- q = q + 1;
    Jump "0", -- Next iteration of loop
    Label "1" -- Label to goto when skipping loop
]
Assembly code

```
data Instr
  = Const Int  -- Push a constant onto the stack
  | Store String  -- Store TOS in storage and pop TOS
  | Load String   -- Push a storage cell's content onto stack
  | Label String  -- Place a label as an address for jumps
  | Jump String   -- Jump to a label
  | CJump String  -- Jump to a label, if TOS is nonzero; also pop TOS
  | Not           -- Apply negation to TOS and replace it by result
  | Add           -- Apply addition to the two topmost stack elements; pop them; push result
...```

Importantly, there are instruction forms for accessing the memory in a symbol-based (i.e., name-based) manner. There are instruction forms for jumping both conditionally and unconditionally to a label.

The translation of imperative programs (BIPL) to assembly code (BAL) is essentially a function which maps each statement or expression to a corresponding sequence of assembly instructions. We encode this function as a recursive walk over...
Compiler (core)

compile :: Stmt → [Instr]
compile s = fst (stmt s 0)

stmt :: Stmt → Int → ([Instr], Int)
stmt Skip l = ([], l)
stmt (Assign x e) l = (expr e ++ [Store x], l)
stmt (Seq s1 s2) l0 =
  let
    (zs1, l1) = stmt s1 l0
    (zs2, l2) = stmt s2 l1
  in (zs1 ++ zs2, l2)
stmt (If e s1 s2) l0 =
  let l1 = l0+1
    (zs1, l2) = stmt s1 (l1+1)
    (zs2, l3) = stmt s2 l2
  in (expr e
    ++ (CJump (show l0) : zs2)
    ++ (Jump (show l1) : (Label (show l0) : zs1))
    ++ [Label (show l1)], l3)

expr :: Expr → [Instr]
expr (IntConst i) = [Const i]
expr (Var x) = [Load x]
expr (Unary BIPL.Negate e) = expr (Binary BIPL.Sub (IntConst 0) e)
expr (Unary BIPL.Not e) = expr e ++ [BAL.Not]
expr (Binary o e1 e2) = expr e1 ++ expr e2 ++
  case o of
    BIPL.Add → BAL.Add

We explain the translation rules for the statement forms.

• Skip: The empty instruction sequence '[]' is returned with the unmodified label counter.
• Assign x e: A Store x instruction is added to the instruction sequence for the expression e; the label counter is not modified, as expressions do not involve control flow.
stmt (While e s) l0 =
  let l1 = l0+1
      (zs, l2) = stmt s (l1+1)
  in ([Label (show l0)] ++ expr e
       ++ (Not : (CJump (show l1) : zs))
       ++ [Jump (show l0), Label (show l1)], l2)

expr :: Expr → [Instr]
expr (IntConst i) = [Const i]
expr (Var x) = [Load x]
expr (Unary BIPL.Negate e) = expr (Binary BIPL.Sub (IntConst 0) e)
expr (Unary BIPL.Not e) = expr e ++ [BAL.Not]
expr (Binary o e1 e2) = expr e1 ++ expr e2 ++
  [ case o of
      BIPL.Add → BAL.Add
    ...]
Euclidean division in machine code

euclideanDiv = [Const 14, Store 0, Const 4, Store 1, Const 0, Store 2, Load 0, Store 3, Load 3, Load 1, Geq, Not, CJump 22, Load 3, Load 1, Sub, Store 3, Load 2, Const 1, Add, Store 2, Jump 8]

Assembly code for comparison

euclideanDiv = [
    Const 14, Store "x", —— x = 14;
    Const 4, Store "y", —— y = 4;
    Const 0, Store "q", —— q = 0;
    Load "x", Store "r", —— r = x;
    Label "0", —— Beginning of while loop
    Load "r", Load "y", Geq, —— (r >= y)
    Not, CJump "1", —— Skip while loop
    Load "r", Load "y", Sub, Store "r", —— r = r - y;
    Load "q", Const 1, Add, Store "q", —— q = q + 1;
    Jump "0", —— Next iteration of loop
    Label "1" —— Label to goto when skipping loop
]
Machine code

data Instr
  = Const Int     -- Push a constant onto the stack
  | Store Int     -- Store TOS in storage and pop TOS
  | Load Int      -- Push a storage cell's content onto stack
  | Jump Int      -- Jump to an address
  | CJump Int     -- Jump to an address, if TOS is nonzero; also pop TOS
  | Not           -- Apply negation to TOS and replace it by result
  | Add           -- Apply addition to the two topmost stack elements; pop them; push result
...

Assembly code for comparison

data Instr
  = Const Int     -- Push a constant onto the stack
  | Store String   -- Store TOS in storage and pop TOS
  | Load String    -- Push a storage cell's content onto stack
  | Label String   -- Place a label as an address for jumps
  | Jump String    -- Jump to a label
  | CJump String   -- Jump to a label, if TOS is nonzero; also pop TOS
  | Not            -- Apply negation to TOS and replace it by result
  | Add            -- Apply addition to the two topmost stack elements; pop them; push result
...

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
We need to replace store symbols by store indices. Likewise, we need to replace jump labels by instruction pointers.

assemble :: [BAL.Instr] → [BML.Instr]
assemble zs = concat (map f zs)

where

  f (BAL.Const i) = [BML.Const i]
  f (BAL.Store x) = [BML.Store (cell x)]
  f (BAL.Load x) = [BML.Load (cell x)]
  f (BAL.Label x) = []
  f (BAL.Jump x) = [BML.Jump (instruction x)]
  f (BAL.CJump x) = [BML.CJump (instruction x)]
  f BAL.Not = [BML.Not]
  f BAL.Add = [BML.Add]
  ...

← Map symbol to memory cell

cell :: String → Int

where cell symbol = fromJust (findIndex (==symbol) symbols)

  symbols = nub (concat (map symbol zs))
Haskell module `Language.BML.Syntax`

```haskell
data Instr = Const Int
            | Push Int
            | Store Int
            | Load Int
            | Jump Int
            | CJump Int
            | Not
            | Add
```

A translator from BAL to BML may also be referred to as an assembler. The translator essentially translates BAL instruction sequences to BML instruction sequences—one by one.

Haskell module `Language.BAL.Assembler`

```haskell
abless :: [BAL.Instr] -> [BML.Instr]
abless zs = concat (map f zs)
where
  f (BAL.Const i) = [BML.Const i]
  f (BAL.Store x) = [BML.Store (cell x)]
  f (BAL.Load x) = [BML.Load (cell x)]
  f (BAL.Label x) = []
  f (BAL.Jump x) = [BML.Jump (instruction x)]
  f (BAL.CJump x) = [BML.CJump (instruction x)]
  f BAL.Not = [BML.Not]
  f BAL.Add = [BML.Add]
  ...
```

---

Collect all store symbols, order them by first occurrence. Use position as store index.

```haskell
  cell :: String -> Int
  cell x = fromJust (findIndex (==x) symbols)
  where
    symbols = nub (concat (map symbol zs))
    symbol (BAL.Store x) = [x]
    symbol _ = []
```

---

Use instruction position of label in otherwise label-free instruction sequence as instruction pointer.

```haskell
  instruction :: String -> Int
  instruction x = instruction' 0 zs
  where
    instruction' i (BAL.Label x' : zs) = if x==x' then i else instruction' i zs
    instruction' i (_ : zs) = instruction' (i+1) zs
```

© 2017 Ralf Lämmel, Software Languages Team http://www.softlang.org/, University of Koblenz-Landau. All rights reserved.
Machine

We can run machine code, subject to a virtual machine (in fact, an interpreter) for machine code.

run Language.ML.Sample.euclideanDivision
(fromList [(0,14),(1,4),(2,3),(3,2)],[])

type Memory = Map Int Int
type Stack = [Int]

run :: [Instr] → (Memory, Stack)
run zs0 = run' zs0 empty []
  where
  run' :: [Instr] → Memory → Stack → (Memory, Stack)
  run' [] sto sta = (sto, sta)
  run' (z:zs) sto sta = let (zs', sto', sta') = step z in run' zs' sto' sta'
    where
    step :: Instr → ([Instr], Memory, Stack)
    step (Const i) = (zs', sto', sta')
    step (Add x y) = (zs', sto', sta')
    step (Sub x y) = (zs', sto', sta')
    step (Mul x y) = (zs', sto', sta')
    step (Div x y) = (zs', sto', sta')
    step (Jump l) = (zs', sto', sta')
    step (Load l) = (zs', sto', sto!l : sta')
    step (Store l x) = (zs', insert l (head sta') sto', tail sta')

step :: Instr → ([Instr], Memory, Stack)
step (Const i) = (zs, sto, i : sta)
step (Store i) = (zs, insert i (head sta) sto, tail sta)
step (Load i) = (zs, sto, sto!i : sta)
step (Jump i) = (drop i zs0, sto, sta)
step (CJump i) = (if head sta /= 0 then drop i zs0 else zs, sto, tail sta)
step Not = (zs, sto, uop (λ i → if i == 0 then 1 else 0) sta)
step Add = (zs, sto, bop (+) sta)
...
... other operations omitted

— Apply unary operation on ints on stack
uop :: (Int → Int) → Stack → Stack
uop f (i1:sta) = f i1 : sta

— Apply binary operation on ints on stack
bop :: (Int → Int → Int) → Stack → Stack
bop f (i2:i1:sta) = f i1 i2 : sta

We use a variation on continuation-passing style.
Online resources

YAS (Yet Another SLR (Software Language Repository))
http://www.softlang.org/yas
YAS’ GitHub repository contains all code. See here specifically:
See languages:
• BIPL — Basic **Imperative** Programming Language
• BAL — Basic **Assembly** Language
• BML — Basic **Machine** Language

The Software Languages Book
http://www.softlang.org/book
The book discusses compilation in much less detail than textbooks on compilation.