



Compiler Construction

Lecture 15: Code Generation I (Intermediate Code)

Summer Semester 2017

Thomas Noll

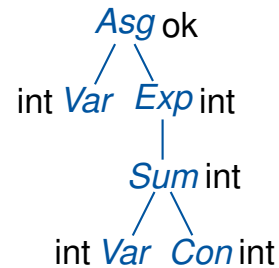
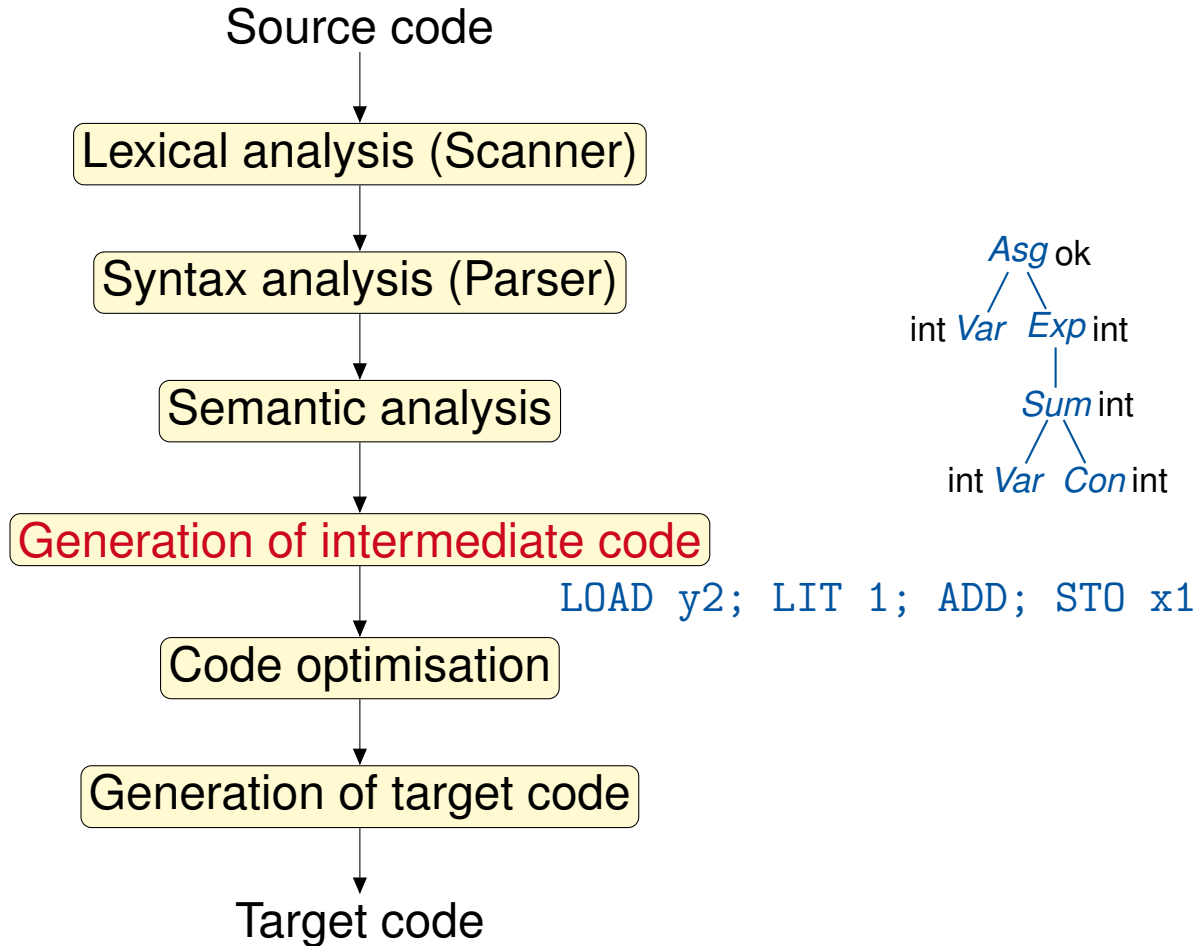
Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ss-17/cc/>

Generation of Intermediate Code

Conceptual Structure of a Compiler



tree translations

Generation of Intermediate Code

Modularisation of Code Generation I

Splitting of code generation for programming language PL:

$$PL \xrightarrow{\text{trans}} IC \xrightarrow{\text{code}} MC$$

Frontend: `trans` generates **machine-independent intermediate code** (IC) for abstract (stack) machine

Backend: `code` generates **actual machine code** (MC)

Advantages: IC machine independent \implies

Portability: much easier to write IC compiler/interpreter for a new machine (as opposed to rewriting the whole compiler)

Fast compiler implementation: generating IC much easier than generating MC

Code size: IC programs usually smaller than corresponding MC programs

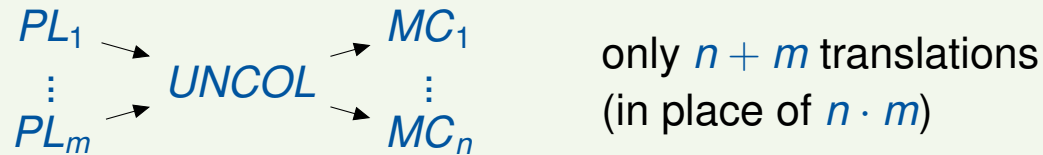
Code optimisation: division into machine-independent and machine-dependent parts

Generation of Intermediate Code

Modularisation of Code Generation II

Example 15.1

1. UNiversal Computer-Oriented Language (UNCOL; \approx 1960;
<http://en.wikipedia.org/wiki/UNCOL>): **universal** intermediate language for compilers
(never fully specified or implemented; too ambitious)



2. Pascal's portable code (P-code; \approx 1970;
http://en.wikipedia.org/wiki/P-Code_machine)
3. Java Virtual Machine (JVM; Sun; \approx 1995;
http://en.wikipedia.org/wiki/Java_Virtual_Machine)
4. Common Intermediate Language (CIL; Microsoft .NET; \approx 2002;
http://en.wikipedia.org/wiki/Common_Intermediate_Language)

Language Structures I

Structures in high-level programming languages

- Basic data types and basic operations
- Static and dynamic data structures
- Expressions and assignments
- Control structures (branching, loops, ...)
- Procedures and functions
- Modularity: blocks, modules, and classes

Use of procedures and blocks

- FORTRAN: non-recursive and non-nested procedures
 - ⇒ **static** memory management (requirements determined at compile time)
- C: recursive and non-nested procedures
 - ⇒ dynamic memory management using **runtime stack** (requirements only known at runtime), no static links
- Algol-like languages (Pascal, Modula): recursive and nested procedures
 - ⇒ dynamic memory management using **runtime stack with static links**
- Object-oriented languages (C++, Java): object creation and removal
 - ⇒ dynamic memory management using **heap**

Generation of Intermediate Code

Language Structures II

Structures in machine code (von Neumann/SISD)

Memory hierarchy: accumulators, registers, caches, main memory, background storage

Instruction types: arithmetic/Boolean/... operation, test/jump instruction, transfer instruction, I/O instruction, ...

Addressing modes: direct/indirect, absolute/relative, ...

Architectures: RISC (few [fast but simple] instructions, many registers), CISC (many [complex but slow] instructions, few registers)

Structures in intermediate code

- **Data types and operations** like PL
- **Data stack** with basic operations
- **Jumping instructions** for control structures
- **Runtime stack** for blocks, procedures, and static data structures
- **Heap** for dynamic data structures

The Example Programming Language EPL

The Example Programming Language EPL

Structures of EPL:

- Only integer and Boolean **values**
- Arithmetic and Boolean **expressions** with strict and non-strict semantics
- **Control structures**: sequence, branching, iteration
- Nested **blocks** and recursive **procedures** with local and global variables
(\implies dynamic memory management using runtime stack with static links)
- (not/later considered: procedure **parameters** and [dynamic] **data structures**)

The Example Programming Language EPL

Syntax of EPL

Definition 15.2 (Syntax of EPL)

The **syntax of EPL** is defined as follows:

$$\begin{aligned} \mathbb{Z} : & \quad z && \quad (* z \text{ is an integer} *) \\ Ide : & \quad I && \quad (* I \text{ is an identifier} *) \\ AExp : & \quad A ::= z \mid I \mid A_1 + A_2 \mid \dots \\ BExp : & \quad B ::= A_1 < A_2 \mid \text{not } B \mid B_1 \text{ and } B_2 \mid B_1 \text{ or } B_2 \\ Cmd : & \quad C ::= I := A \mid C_1 ; C_2 \mid \text{if } B \text{ then } C_1 \text{ else } C_2 \mid \text{while } B \text{ do } C \mid I() \\ Dcl : & \quad D ::= D_C D_V D_P \\ & \quad D_C ::= \varepsilon \mid \text{const } l_1 := z_1, \dots, l_n := z_n ; \\ & \quad D_V ::= \varepsilon \mid \text{var } l_1, \dots, l_n ; \\ & \quad D_P ::= \varepsilon \mid \text{proc } l_1 ; K_1 ; \dots ; \text{proc } l_n ; K_n ; \\ Blk : & \quad K ::= D C \\ Pgm : & \quad P ::= \text{in/out } l_1, \dots, l_n ; K . \end{aligned}$$

The Example Programming Language EPL

EPL Example: Factorial Function

Example 15.3 (Factorial function)

```
in/out x;
var y;
proc F;
  if x > 1 then
    y := y * x;
    x := x - 1;
  F()
y := 1;
F();
x := y.
```

Static Semantics of EPL

- Usage of identifiers must be **consistent** with declaration:
 - $I := \dots J \dots \implies I$ variable, J constant or variable
 - $I() \implies$ procedure
- All identifiers in a declaration D have to be **different**.
- Every identifier occurring in the command C of a block $D C$ must be **declared**
 - in D or
 - in the declaration list of a surrounding block.
- **Static scoping**: the usage of an identifier in the body of a called procedure refers to its declaration environment (and not to its calling environment).
- **Multiple declarations** of an identifier in different blocks are possible. Each usage in a command refers to the **“innermost” declaration**.

Scoping and Visibility of Identifiers

Scope of an identifier

The **scope** (or: range of validity/visibility) of an identifier's declaration refers to the part of the program where a usage of that identifier can refer to that declaration.

Static vs. dynamic scoping

Static (aka “lexical”) **scoping**: name resolution at **compile time**, depends on location in source code and lexical context (here)

- “part” = area of source code (**block**)
- usually admits at most one declaration of same identifier per block
- but allows additional declarations within **nested** blocks
- **innermost** principle: hide outer declarations (still **valid**, but not **visible**)

Dynamic scoping: name resolution at **runtime**, depends on execution context

- “part” = runtime history, esp. procedure calls

Levels of scope: expression/**block**/procedure/file/module/...

Static Scoping by Example

Example 15.4

```
in/out x;
  const c = 10;
  var y;
  proc P;
    var y, z;
    proc Q;
      var x, z;
      [... z := 1; P() ...]
      [... P() ... R() ...]
    proc R;
      [... P() ...]
      [... x := 0; P() ...] .
```

- “Innermost” principle: use of `x` in main program
- “Innermost” principle: use of `z` in `Q`
- Static scoping: call of `P` in `Q` can refer to `x, y, z`
- Later declaration: call of `R` in `P` followed by declaration (in older languages: *forward* declarations for one-pass compilation)

Semantics of EPL

Dynamic Semantics of EPL

(omitting the details; cf. *Semantics of Programming Languages*)

- To “run” program, execute main block in **state** (memory location \mapsto value) determined by input values
- **Effect of declaration** = modification of **environment**
 - constant \mapsto value, variable \mapsto memory location, procedure \mapsto state transformation
- **Effect of statement** = modification of **state**
 - assignment $l := A$: update of l 's location by current value of A
 - composition $C_1; C_2$: sequential execution
 - branching $\text{if } B \text{ then } C_1 \text{ else } C_2$: test of B , followed by jump to respective branch
 - iteration $\text{while } B \text{ do } C$: execution of C as long as B is true
 - call $l()$: transfer control to body of l and return to subsequent statement afterwards
- EPL program $P = \text{in/out } l_1, \dots, l_n; K. \in \text{Pgm}$ has as **semantics** a function

$$\llbracket P \rrbracket : \mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$$

Example 15.5 (Factorial function; cf. Example 15.3)

here $n = 1$ and $\llbracket P \rrbracket(x) = x!$ (where $x! := 1$ for $x \leq 0$)

Intermediate Code for EPL

The Abstract Machine AM

Definition 15.6 (Abstract machine for EPL)

The **abstract machine for EPL (AM)** is defined by the **state space**

$$S := PC \times DS \times PS$$

with

- the **program counter** $PC := \mathbb{N}$,
- the **data stack** $DS := \mathbb{Z}^*$ (top of stack to the right), and
- the **procedure stack** (or: **runtime stack**) $PS := \mathbb{Z}^*$ (top of stack to the left).

Thus a state $s = (pc, d, p) \in S$ is given by

- a program counter $pc \in PC$,
- a data stack $d = d.r : \dots : d.1 \in DS$, and
- a procedure stack $p = p.1 : \dots : p.t \in PS$.

AM Instructions

Definition 15.7 (AM instructions)

The set of **AM instructions** is divided into

arithmetic instructions: ADD, MULT, ...

Boolean instructions: NOT, AND, OR, LT, ...

jumping instructions: JMP(*ca*), JFALSE(*ca*) (*ca* ∈ PC)

procedure instructions: CALL(*ca*, *dif*, *loc*) (*ca* ∈ PC, *dif*, *loc* ∈ ℕ), RET

transfer instructions: LOAD(*dif*, *off*), STORE(*dif*, *off*) (*dif*, *off* ∈ ℕ), LIT(*z*) (*z* ∈ ℤ)

Semantics of Instructions

Definition 15.8 (Semantics of AM instructions (1st part))

The semantics of an AM instruction O

$$\llbracket O \rrbracket : S \dashrightarrow S$$

is defined as follows:

$$\llbracket \text{ADD} \rrbracket (pc, d : z_1 : z_2, p) := (pc + 1, d : z_1 + z_2, p)$$

$$\llbracket \text{NOT} \rrbracket (pc, d : b, p) := (pc + 1, d : \neg b, p) \quad \text{if } b \in \{0, 1\}$$

$$\llbracket \text{AND} \rrbracket (pc, d : b_1 : b_2, p) := (pc + 1, d : b_1 \wedge b_2, p) \quad \text{if } b_1, b_2 \in \{0, 1\}$$

$$\llbracket \text{OR} \rrbracket (pc, d : b_1 : b_2, p) := (pc + 1, d : b_1 \vee b_2, p) \quad \text{if } b_1, b_2 \in \{0, 1\}$$

$$\llbracket \text{LT} \rrbracket (pc, d : z_1 : z_2, p) := \begin{cases} (pc + 1, d : 1, p) & \text{if } z_1 < z_2 \\ (pc + 1, d : 0, p) & \text{if } z_1 \geq z_2 \end{cases}$$

$$\llbracket \text{JMP}(ca) \rrbracket (pc, d, p) := (ca, d, p)$$

$$\llbracket \text{JFALSE}(ca) \rrbracket (pc, d : b, p) := \begin{cases} (ca, d, p) & \text{if } b = 0 \\ (pc + 1, d, p) & \text{if } b = 1 \end{cases}$$

The Procedure Stack

Structure of Procedure Stack I

The semantics of procedure and transfer instructions requires a particular structure of the procedure stack $p \in PS$: it must be composed of **frames** (or: **activation records**) of the form

$$sl : dl : ra : v_1 : \dots : v_k$$

where

static link sl : points to frame of surrounding declaration environment

⇒ used to access non-local variables

dynamic link dl : points to previous frame (i.e., of calling procedure)

⇒ used to remove topmost frame after termination of procedure call

return address ra : program counter after termination of procedure call

⇒ used to continue program execution after termination of procedure call

local variables v_j : values of locally declared variables

The Procedure Stack

Structure of Procedure Stack II

- Frames are **created** whenever a procedure call is performed
- Two **special frames**:
 - I/O frame: for keeping values of **in/out** variables
($sl = dl = ra = 0$)
 - MAIN frame: for keeping values of top-level block
($sl = dl = \text{I/O frame}$)

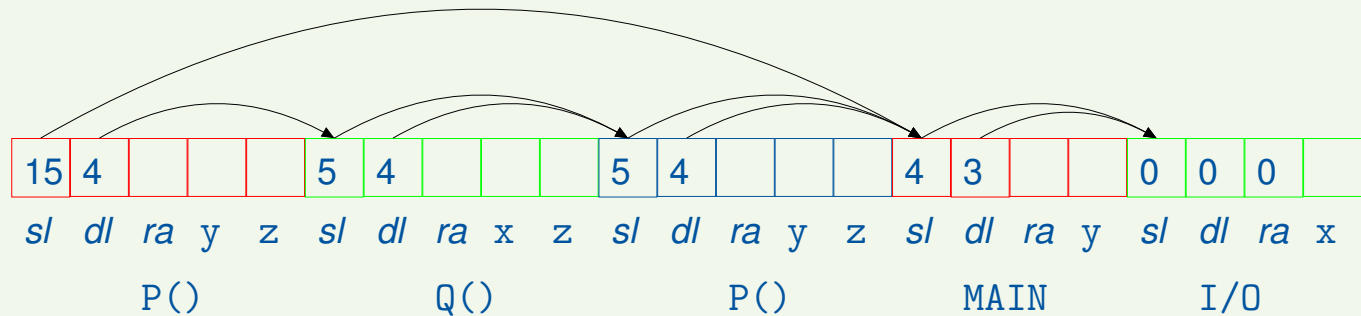
The Procedure Stack

Structure of Procedure Stack III

Example 15.9 (cf. Example 15.4)

```
in/out x;  
const c = 10;  
var y;  
proc P;  
  var y, z;  
  proc Q;  
    var x, z;  
    [... P() ...]  
  [... Q() ...]  
proc R;  
  [... P() ...]  
[... P() ...].
```

Procedure stack after second call of P:



The Procedure Stack

Structure of Procedure Stack IV

Observation:

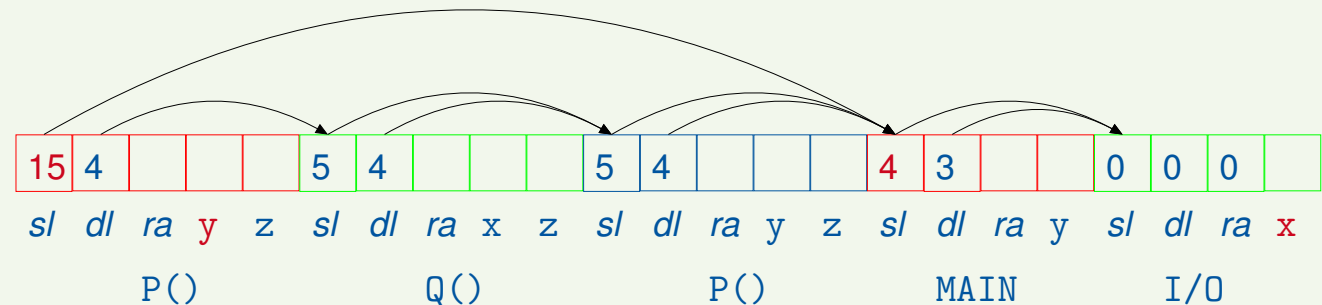
- The usage of a variable in a procedure body refers to its **innermost declaration**.
- If the level difference between the usage and the declaration is *dif*, then a **chain of *dif* static links** has to be followed to access the corresponding frame.

Example 15.10 (cf. Example 15.9)

```

in/out x;
const c = 10;
var y;
proc P;
  var y, z;
  proc Q;
    var x, z;
    [... P() ...]
  [... x ... y ... Q() ...]
proc R;
  [... P() ...]
[... P() ...].
    
```

Procedure stack after second call of P:



P uses x $\implies dif = 2$

P uses y $\implies dif = 0$

The Procedure Stack

Displays

- **Optimisation technique** to replace static links
 - **Implementation:**
 - array of pointers to frames: *disp*
 - Size of the array = maximum block nesting depth of program
 - *disp[i]* points to frame associated with the current scope at nesting depth *i*
 - **Usage:** to access a non-local variable declared at nesting depth *i*,
 1. go to frame pointed to by *disp[i]*
 2. access variable via offset in this frame
- Advantage:** constant-time access (exactly two steps regardless of level difference)
- **Maintenance:**
 - use global display for current procedure activation
 - caller saves display in its frame, and restores it when callee returns
 - display for callee constructed using techniques for static link handling, using static scoping