Compiler Construction

Lecture 15: Code Generation I (Intermediate Code)

Thomas Noll

Lehrstuhl für Informatik 2
(Software Modeling and Verification)

RWTH AACHEN UNIVERSITY

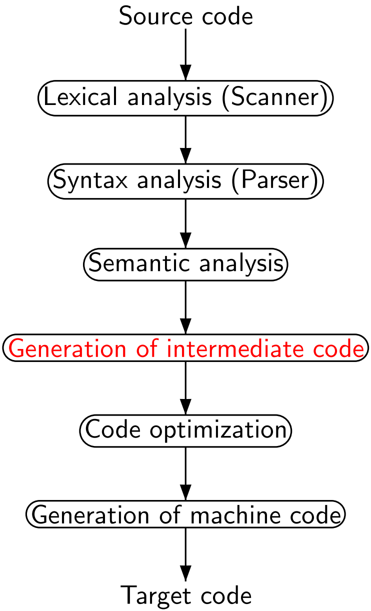
noll@cs.rwth-aachen.de

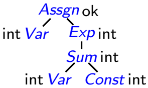
http://moves.rwth-aachen.de/teaching/ss-14/cc14/

Summer Semester 2014

Outline

Conceptual Structure of a Compiler





tree translations

```
LOAD y2; LIT 1; ADD; STO x1
```

Modularization of Code Generation I

Splitting of code generation for programming language PL:

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

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

Backend: code generates actual machine code (MC)

Modularization of Code Generation I

Splitting of code generation for programming language PL:

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

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

Backend: code generates actual machine code (MC)

Advantages: IC machine independent $\Longrightarrow$

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

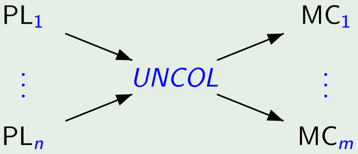
Code size: IC programs usually smaller than corresponding MC programs

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

Modularization of Code Generation II

Example 15.1

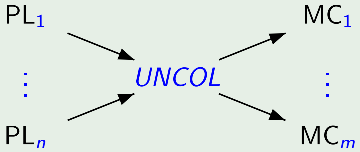
1. UNiversal Computer-Oriented Language (UNCOL; ≈ 1960; http://en.wikipedia.org/wiki/UNCOL): universal intermediate language for compilers (never fully specified or implemented; too ambitious)



only $n + m$ translations (in place of $n \cdot m$)

Example 15.1

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

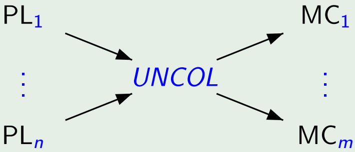


only $n + m$ translations (in place of $n \cdot m$)

2. Pascal's pseudocode (P-code; ≈ 1975; http://en.wikipedia.org/wiki/P-Code_machine)

Example 15.1

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



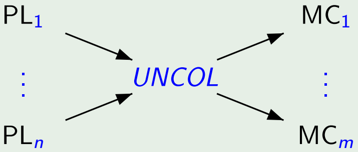
only $n + m$ translations (in place of $n \cdot m$)

2. Pascal's pseudocode (P-code; ≈ 1975; http://en.wikipedia.org/wiki/P-Code_machine)
3. The Amsterdam Compiler Kit (TACK; ≈ 1980; http://tack.sourceforge.net/)

Modularization of Code Generation II

Example 15.1

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



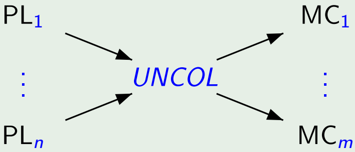
only $n + m$ translations
(in place of $n \cdot m$)

2. Pascal's pseudocode (P-code; ≈ 1975;
http://en.wikipedia.org/wiki/P-Code_machine)
3. The Amsterdam Compiler Kit (TACK; ≈ 1980;
http://tack.sourceforge.net/)
4. Java Virtual Machine (JVM; Sun; ≈ 1996;
http://en.wikipedia.org/wiki/Java_Virtual_Machine)

Modularization of Code Generation II

Example 15.1

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



only $n + m$ translations (in place of $n \cdot m$)

2. Pascal's pseudocode (P-code; ≈ 1975; http://en.wikipedia.org/wiki/P-Code_machine)
3. The Amsterdam Compiler Kit (TACK; ≈ 1980; http://tack.sourceforge.net/)
4. Java Virtual Machine (JVM; Sun; ≈ 1996; http://en.wikipedia.org/wiki/Java_Virtual_Machine)
5. Common Intermediate Language (CIL; Microsoft .NET; ≈ 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 (sequences, branching statements, loops, ...)
- Procedures and functions
- Modularity: blocks, modules, and classes

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 (sequences, branching statements, loops, ...)
- Procedures and functions
- Modularity: blocks, modules, and classes

Use of procedures and blocks:

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

Language Structures II

Structures in machine code: (von Neumann/SISD)

Memory hierarchy: accumulators, registers, cache, 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)

Language Structures II

Structures in machine code: (von Neumann/SISD)

Memory hierarchy: accumulators, registers, cache, 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

Outline

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 ($\Longrightarrow$ dynamic memory management using runtime stack with static links)
- (not considered: procedure parameters and [dynamic] data structures)

Definition 15.2 (Syntax of EPL)

The syntax of EPL is defined as follows:

$$
\begin{array}{lll}
\mathbb{Z}: & z & \text{(* } z \text{ is an integer *)} \\
\mathit{Ide}: & I & \text{(* } I \text{ is an identifier *)} \\
\mathit{AExp}: & A ::= z \mid I \mid A_1 + A_2 \mid \ldots & \\
\mathit{BExp}: & B ::= A_1 < A_2 \mid \texttt{not}\ B \mid B_1\ \texttt{and}\ B_2 \mid B_1\ \texttt{or}\ B_2 & \\
\mathit{Cmd}: & C ::= I\ \texttt{:=}\ A \mid C_1 \texttt{;} C_2 \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid & \\
 & \qquad \texttt{while}\ B\ \texttt{do}\ C \mid I\texttt{()} & \\
\mathit{Dcl}: & D ::= D_C\ D_V\ D_P & \\
 & D_C ::= \varepsilon \mid \texttt{const}\ I_1\ \texttt{:=}\ z_1\texttt{,} \ldots \texttt{,} I_n\ \texttt{:=}\ z_n\texttt{;} & \\
 & D_V ::= \varepsilon \mid \texttt{var}\ I_1\texttt{,} \ldots \texttt{,} I_n\texttt{;} & \\
 & D_P ::= \varepsilon \mid \texttt{proc}\ I_1\texttt{;} K_1\texttt{;} \ldots \texttt{;proc}\ I_n\texttt{;} K_n\texttt{;} & \\
\mathit{Blk}: & K ::= D\ C & \\
\mathit{Pgm}: & P ::= \texttt{in/out}\ I_1\texttt{,} \ldots \texttt{,} I_n\texttt{;} K\ \texttt{.} &
\end{array}
$$

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.
```

Outline

Static Semantics of EPL I

- All identifiers in a declaration D have to be different.

Static Semantics of EPL I

- 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 Semantics of EPL I

- 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.
- Multiple declarations of an identifier in different blocks are possible. Each usage in a command C refers to the "innermost" declaration.

Static Semantics of EPL I

- 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.
- Multiple declarations of an identifier in different blocks are possible. Each usage in a command C refers to the "innermost" declaration.
- 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).

Static Semantics of EPL II

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() ...] .
```

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

Static Semantics of EPL II

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

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
- Static scoping: body of P can refer to x, y, z

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
- Static scoping: body of `P` can refer to `x, y, z`
- Later declaration: call of `R` in `P` followed by declaration (in Pascal: `forward` declarations for one-pass compilation)

Dynamic Semantics of EPL

(omitting the details)

- To "run" a program, execute the main block in the state which is given by the input values

Dynamic Semantics of EPL

(omitting the details)

- To "run" a program, execute the main block in the state which is given by the input values
- Effect of statement = modification of state
  - assignment I `:=` A: update of I by current value of A
  - composition C_1 ; C_2: sequential execution
  - branching `if` B `then` C_1 `else` C_2: test of B, followed by jump to respective branch
  - iteration `while` B `do` C: execution of C as long as B is true
  - call I`()`: transfer control to body of I and return to subsequent statement afterwards

Dynamic Semantics of EPL

(omitting the details)

- To "run" a program, execute the main block in the state which is given by the input values
- Effect of statement = modification of state
  - assignment I `:=` A: update of I by current value of A
  - composition C_1 ; C_2: sequential execution
  - branching `if` B `then` C_1 `else` C_2: test of B, followed by jump to respective branch
  - iteration `while` B `do` C: execution of C as long as B is true
  - call I`()`: transfer control to body of I and return to subsequent statement afterwards
- Consequently, an EPL program $P =$ `in/out` $I_1, \ldots, I_n; K. \in Pgm$ has as semantics a function

$$[\![P]\!] : \mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$$

Dynamic Semantics of EPL

(omitting the details)

- To "run" a program, execute the main block in the state which is given by the input values
- Effect of statement = modification of state
  - assignment I `:=` A: update of I by current value of A
  - composition C_1 `;` C_2: sequential execution
  - branching `if` B `then` C_1 `else` C_2: test of B, followed by jump to respective branch
  - iteration `while` B `do` C: execution of C as long as B is true
  - call I`()`: transfer control to body of I and return to subsequent statement afterwards
- Consequently, an EPL program $P =$ `in/out` $I_1, \ldots, I_n; K. \in Pgm$ has as semantics a function

$$[\![P]\!] : \mathbb{Z}^n \dashrightarrow \mathbb{Z}^n$$

Example 15.5 (Factorial function; cf. Example 15.3)

here $n = 1$ and $[\![P]\!](x) = x!$ (where $x! := 1$ for $x \leq 1$)

Outline

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).

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 $s = (l, d, p) \in S$ is given by

- a program label $l \in PC$,
- a data stack $d = d.r : \ldots : d.1 \in DS$, and
- a procedure stack $p = p.1 : \ldots : p.t \in PS$.

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`)` ($ca \in PC$)

procedure instructions: `CALL(`ca, dif, loc`)` ($ca \in PC$, $dif, loc \in \mathbb{N}$), `RET`

transfer instructions: `LOAD(`dif, off`)`, `STORE(`dif, off`)` ($dif, off \in \mathbb{N}$),
`LIT(`z`)` ($z \in \mathbb{Z}$)

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

The semantics of an AM instruction O

$$[\![O]\!] : S \dashrightarrow S$$

is defined as follows:

$$
\begin{aligned}
[\![\mathtt{ADD}]\!](l, d : z_1 : z_2, p) &:= (l + 1, d : z_1 + z_2, p) \\
[\![\mathtt{NOT}]\!](l, d : b, p) &:= (l + 1, d : \neg b, p) && \text{if } b \in \{0, 1\} \\
[\![\mathtt{AND}]\!](l, d : b_1 : b_2, p) &:= (l + 1, d : b_1 \wedge b_2, p) && \text{if } b_1, b_2 \in \{0, 1\} \\
[\![\mathtt{OR}]\!](l, d : b_1 : b_2, p) &:= (l + 1, d : b_1 \vee b_2, p) && \text{if } b_1, b_2 \in \{0, 1\} \\
[\![\mathtt{LT}]\!](l, d : z_1 : z_2, p) &:= \begin{cases} (l + 1, d : 1, p) & \text{if } z_1 < z_2 \\ (l + 1, d : 0, p) & \text{if } z_1 \geq z_2 \end{cases} \\
[\![\mathtt{JMP}(ca)]\!](l, d, p) &:= (ca, d, p) \\
[\![\mathtt{JFALSE}(ca)]\!](l, d : b, p) &:= \begin{cases} (ca, d, p) & \text{if } b = 0 \\ (l + 1, d, p) & \text{if } b = 1 \end{cases}
\end{aligned}
$$

Outline

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

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

where

static link sl: points to frame of surrounding declaration environment
$\Longrightarrow$ used to access non-local variables

dynamic link dl: points to previous frame (i.e., of calling procedure)
$\Longrightarrow$ used to remove topmost frame after termination of procedure call

return address ra: program label after termination of procedure call
$\Longrightarrow$ used to continue program execution after termination of procedure call

local variables v_i: values of locally declared variables

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
  ($sl = dl =$ I/O frame)

Structure of Procedure Stack III

Example 15.9 (cf. Example 15.4)

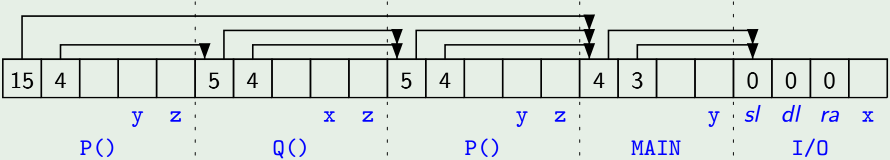
```
in/out x;
  const c = 10;
  var y;
  proc P;
    var y, z;
    proc Q;
      var x, z;
      [... P() ...]
    [... Q() ...]
  proc R;
    [... P() ...]
  [... P() ...].
```

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:



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.

Structure of Procedure Stack IV

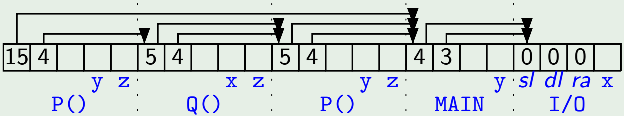
Observation:

- 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:



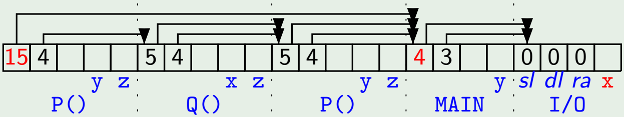
Observation:

- 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 $\Longrightarrow$ *dif* $= 2$

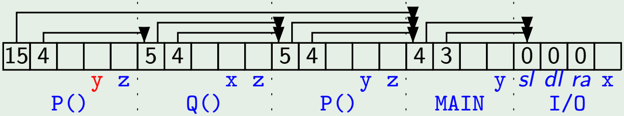
Observation:

- 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 y $\Longrightarrow$ $dif = 0$