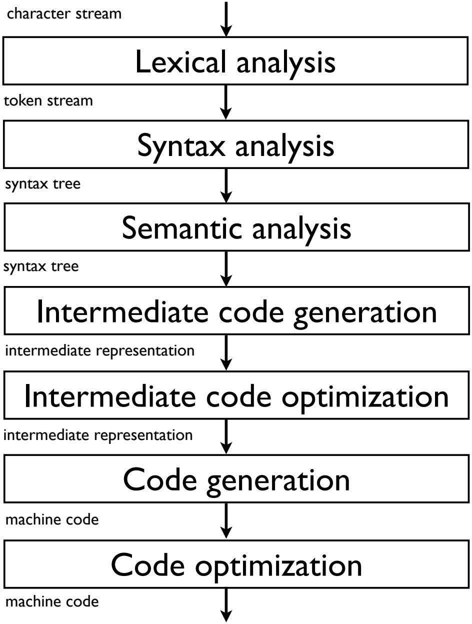
Part 5

Intermediate code generation

Structure of a compiler



Outline

1. Intermediate representations

2. Illustration

3. Optimization

Intermediate code generation

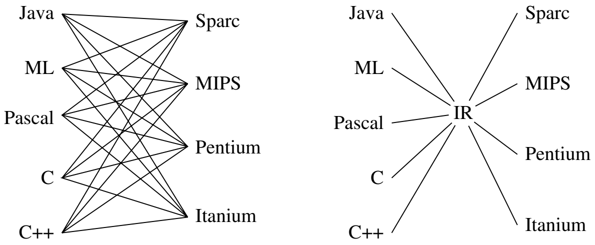
- The final phase of the compiler front-end
- Goal: translate the program into a format expected by the compiler back-end
- In typical compilers: followed by intermediate code optimization and machine code generation

- Techniques for intermediate code generation can be used for final code generation

Intermediate representations

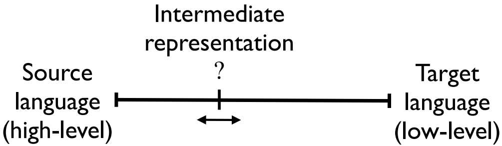
Why use an intermediate representation?

- It's easy to change the source or the target language by adapting only the front-end or back-end (portability)
- It makes optimization easier: one needs to write optimization methods only for the intermediate representation
- The intermediate representation can be directly interpreted



(Appel)

Intermediate representations



- How to choose the intermediate representation?
  - It should be easy to translate the source language to the intermediate representation
  - It should be easy to translate the intermediate representation to the machine code
  - The intermediate representation should be suitable for optimization
- It should be neither too high level nor too low level
- One can have more than one intermediate representation in a single compiler

Some common intermediate representations

General forms of intermediate representations (IR):

- Graphical IR (parse tree, abstract syntax trees, DAG...)
- Linear IR (ie., non graphical)
- Three Address Code (TAC): instructions of the form "result=op1 operator op2"
- Static single assignment (SSA) form: each variable is assigned once
- Continuation-passing style (CPS): general form of IR for functional languages

Some common intermediate representations

Examples:

- Java bytecode (executed on the Java Virtual Machine)
- LLVM (Low Level Virtual Machine): SSA and TAC based
- C is used in several compilers as an intermediate representation (Lisp, Haskell, Cython. . .)
- Microsoft's Common Intermediate Language (CIL)
- GNU Compiler Collection (GCC) uses several intermediate representations:
  - Abstract syntax trees
  - GENERIC (tree-based)
  - GIMPLE (SSA form)
  - Register Transfer Language (RTL, inspired by lisp lists)

(Google them)

Static Single-Assignment Form (SSA)

- A naming discipline used to explicitely encode information about both the flow of control and the flow of data values
- A program is in SSA form if:
  1. each definition has a distinct name
  2. each use refers to a single definition
- Example:

Original code	SSA form
$y = 1$	$y_1 = 1$
$y = 2$	$y_2 = 2$
$x = y$	$x_1 = y_2$

- Main interest: allows to implement several code optimizations.
  - In the example above, it is clear from the SSA form that the first assignment is not necessary.

Converting to SSA

- Converting a program into a SSA form is not a trivial task

Original code	SSA form
$x = 5$	$x_1 = 5$
$x = x - 3$	$x_2 = x_1 - 3$
if $x < 3$	**if** $x_2 < 3$
$\quad y = x * 2$	$\quad y_1 = x_2 * 2$
$\quad w = y$	$\quad w_1 = y_1$
else	**else**
$\quad y = x - 3$	$\quad y_2 = x_2 - 3$
$w = x - y$	$w_2 = x_2 - y_?$
$z = x + y$	$z_1 = x_2 + y_?$

- Need to introduce a special statement: Φ-functions

Converting to SSA

Original code

$x = 5$
$x = x - 3$
if $x < 3$
    $y = x * 2$
    $w = y$
else
    $y = x - 3$
$w = x - y$
$z = x + y$

SSA form

$x_1 = 5$
$x_2 = x_1 - 3$
if $x_2 < 3$
    $y_1 = x_2 * 2$
    $w_1 = y_1$
else
    $y_2 = x_2 - 3$
$y_3 = \Phi(y_1, y_2)$
$w_2 = x_2 - y_3$
$z_1 = x_2 + y_3$

- $\Phi(y_1, y_2)$ is defined as y_1 if we arrive at this instruction through the THEN branch, y_2 if through the ELSE branch.
- One needs to introduce Φ functions at every point of the program where several "paths" merge.

SSA form

- Given an arbitrary program, finding where to place the Φ functions is a difficult task.
- However, an efficient solution is available, based on the control flow graph of the program (see later).

- In practice, the Φ functions are not implemented. They indicate to the compiler that the variables given as arguments need to be stored in the same place.
- In the previous example, we can infer that y_1 and y_2 should be stored in the same place

Continuation-passing style (CPS)

- A programming style in functional languages where control is passed explicitly as argument to the functions, in the form of a *continuation*.
- A continuation is an abstract representation of the control state of a program, most of the time in the form of a first-class function.
- Like SSA, CPS is often used in intermediate representation in compilers of functional languages.

CPS: examples

Direct style

```
(define (pyth x y)
 (sqrt (+ (* x x)(* y y))))
```

CPS style (k is the continuation)

```
(define (pyth& x y k)
 (*& x x (lambda (x2)
           (*& y y (lambda (y2)
                     (+& x2 y2 (lambda (x2py2)
                                 (sqrt& x2py2 k))))))))
(define (*& x y k)
 (k (* x y)))
(define (+& x y k)
 (k (+ x y)))
(define (sqrt& x k)
 (k (sqrt x)))
```

CPS

- The main interest of CPS is to make explicit several things that are typically implicit in functional languages: returns, intermediate values (= continuation arguments), order of argument evaluation...
- Like for SSA, the main interest is to ease optimizations.
- Theoretically, SSA and CPS are equivalent: a program in SSA form can be transformed into a CPS program and vice versa.
- Previous program can be rewritten as:

$$
\begin{aligned}
&x2 = x * x \\
&y2 = y * y \\
&x2py2 = x2 + y2 \\
&res = sqrt(x2py2)
\end{aligned}
$$

Outline

The intermediate language

We will illustrate the translation of typical high-level language constructions using the following low-level intermediate language:

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & [\ \mathit{Instructions}\ ] \\
\\
\mathit{Instructions} & \rightarrow & \mathit{Instruction} \\
\mathit{Instructions} & \rightarrow & \mathit{Instruction}\ ,\ \mathit{Instructions} \\
\\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathbf{unop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathbf{id}\ \mathbf{binop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := M[\mathit{Atom}] \\
\mathit{Instruction} & \rightarrow & M[\mathit{Atom}] := \mathbf{id}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{Instruction} & \rightarrow & \texttt{LABEL}\ \mathbf{labelid} \\
\mathit{Instruction} & \rightarrow & \texttt{GOTO}\ \mathbf{labelid} \\
\\
\mathit{Instruction} & \rightarrow & \texttt{IF}\ \mathbf{id}\ \mathbf{relop}\ \mathit{Atom}\ \texttt{THEN}\ \mathbf{labelid}\ \texttt{ELSE}\ \mathbf{labelid} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \texttt{CALL}\ \mathbf{functionid}(\mathit{Args}) \\
\\
\mathit{Atom} & \rightarrow & \mathbf{id} \\
\mathit{Atom} & \rightarrow & \mathbf{num} \\
\mathit{Args} & \rightarrow & \mathbf{id} \\
\mathit{Args} & \rightarrow & \mathbf{id}\ ,\ \mathit{Args}
\end{array}
$$

Simplified three-address code, very close to machine code

See chapter 5 and 7 of (Mogensen, 2010) for full details

The intermediate language

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & [\ \mathit{Instructions}\ ] \\
\\
\mathit{Instructions} & \rightarrow & \mathit{Instruction} \\
\mathit{Instructions} & \rightarrow & \mathit{Instruction}\ ,\ \mathit{Instructions} \\
\\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathbf{unop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := \mathbf{id}\ \mathbf{binop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \mathbf{id} := M[\mathit{Atom}] \\
\mathit{Instruction} & \rightarrow & M[\mathit{Atom}] := \mathbf{id} \\
\\
\mathit{Atom} & \rightarrow & \mathbf{id} \\
\mathit{Atom} & \rightarrow & \mathbf{num}
\end{array}
$$

- All values are assumed to be integer
- Unary and binary operators include normal arithmetic and logical operations
- An atomic expression is either a variable or a constant
- $M[Atom] :=$ **id** is a transfer from a variable to memory
- **id** $:= M[Atom]$ is a transfer from memory to a variable

The intermediate language

$$
\begin{array}{lcl}
\mathit{Instruction} & \rightarrow & \texttt{LABEL}\ \textbf{labelid} \\
\mathit{Instruction} & \rightarrow & \texttt{GOTO}\ \textbf{labelid} \\
\mathit{Instruction} & \rightarrow & \texttt{IF}\ \textbf{id}\ \textbf{relop}\ \mathit{Atom}\ \texttt{THEN}\ \textbf{labelid}\ \texttt{ELSE}\ \textbf{labelid} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \texttt{CALL}\ \textbf{functionid}(\mathit{Args}) \\
\mathit{Atom} & \rightarrow & \textbf{id} \\
\mathit{Atom} & \rightarrow & \textbf{num} \\
\mathit{Args} & \rightarrow & \textbf{id} \\
\mathit{Args} & \rightarrow & \textbf{id}\ ,\ \mathit{Args}
\end{array}
$$

- LABEL only marks a position in the program
- **relop** includes relational operators $\{=, \neq, <, >, \leq$ or $\geq\}$
- Arguments of a function call are variables and the result is assigned to a variable

Principle of translation

- Syntax-directed translation using several attributes:
  - Code returned as a synthesized attribute
  - Symbol tables passed as inherited attributes
  - Places to store intermediate values as synthesized or inherited attributes
- Implemented as recursive functions defined on syntax tree nodes (as for type checking)
- Since translation follows the syntax, it is done mostly independently of the context, which leads to suboptimal code
- Code is supposed to be optimized globally afterwards

Expressions

$$
\begin{array}{lcl}
Exp & \rightarrow & \textbf{num} \\
Exp & \rightarrow & \textbf{id} \\
Exp & \rightarrow & \textbf{unop}\ Exp \\
Exp & \rightarrow & Exp\ \textbf{binop}\ Exp \\
Exp & \rightarrow & \textbf{id}(Exps) \\
\\
Exps & \rightarrow & Exp \\
Exps & \rightarrow & Exp\ ,\ Exps
\end{array}
$$

(source language grammar !)

Principle of translation:

- Every operation is stored in a new variable in the intermediate language, generated by a function *newvar*
- The new variables for sub-expressions are created by parent expression and passed to sub-expression as inherited attributes (synthesized attributes are also possible)

Expressions

$Trans_{Exp}(Exp, vtable, ftable, place)$ = case Exp of	
num	$v = getvalue(\mathbf{num})$ $[place := v]$
id	$x = lookup(vtable, getname(\mathbf{id}))$ $[place := x]$
unop Exp_1	$place_1 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $op = transop(getopname(\mathbf{unop}))$ $code_1 +\!\!+ [place := op\ place_1]$

where to place the translation of Exp_1 (inherited attribute)

String concatenation

- *getopname* retrieves the operator associated to the token **unop**. *transop* translates this operator into the equivalent operator in the intermediate language
- $[place := v]$ is a string where *place* and v have been replaced by their values (in the compiler)
  - Example: if $place = t14$ and $v = 42$, $[place := v]$ is the instruction [t14:=42].

Expressions: binary operators and function call

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`	
Exp_1 **binop** Exp_2	$place_1 = newvar()$ $place_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, place_2)$ $op = transop(getopname(\textbf{binop}))$ $code_1 +\!\!+ code_2 +\!\!+ [place := place_1\ op\ place_2]$
$\textbf{id}(Exps)$	$(code_1, [a_1, \ldots, a_n])$ $= Trans_{Exps}(Exps, vtable, ftable)$ $fname = lookup(ftable, getname(\textbf{id}))$ $code_1 +\!\!+ [place :=$ `CALL` $fname(a_1, \ldots, a_n)]$

Expressions: function arguments

$Trans_{Exps}(Exps, vtable, ftable) = \texttt{case}\ Exps\ \texttt{of}$	
Exp	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_1, [place])$
$Exp\ ,\ Exps$	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_2, args) = Trans_{Exps}(Exps, vtable, ftable)$ $code_3 = code_1 +\!\!+ code_2$ $args_1 = place :: args$ $(code_3, args_1)$

Expressions: example of translation

Translation of 3+f(x-y,z):

```
    t1 := 3
            t4 := v0
            t5 := v1
        t3 := t4 - t5
        t6 := v2
    t2 := CALL _f(t3,t6)
t0 := t1+t2
```

Assuming that:

- x, y, and z are bound to variables v0, v1, and v2
- Expression is stored in t0
- New variables are generated as t1, t2, t3...
- Indentation indicates depth of call to $Trans_{Exp}$

Statements

$$
\begin{array}{lcl}
Stat & \rightarrow & Stat \ ; \ Stat \\
Stat & \rightarrow & \textbf{id} := Exp \\
Stat & \rightarrow & \texttt{if}\ Cond\ \texttt{then}\ Stat \\
Stat & \rightarrow & \texttt{if}\ Cond\ \texttt{then}\ Stat\ \texttt{else}\ Stat \\
Stat & \rightarrow & \texttt{while}\ Cond\ \texttt{do}\ Stat \\
Stat & \rightarrow & \texttt{repeat}\ Stat\ \texttt{until}\ Cond \\
 & & \\
Cond & \rightarrow & Exp\ \textbf{relop}\ Exp
\end{array}
$$

Principle of translation:

- New unused labels are generated by the function *newlabel* (similar to *newvar*)
- These labels are created by parents and passed as inherited attributes

Statements: sequence of statements and assignment

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
$Stat_1$; $Stat_2$	$code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 +\!\!+ code_2$
$\mathbf{id} := Exp$	$place = lookup(vtable, getname(\mathbf{id}))$ $Trans_{Exp}(Exp, vtable, ftable, place)$

Statements: conditions

$Trans_{Stat}(Stat, vtable, ftable) = \texttt{case}\ Stat\ \texttt{of}$	
$\texttt{if}\ Cond$ $\texttt{then}\ Stat_1$ $\texttt{else}\ Stat_2$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_3 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 +\!\!+ [\texttt{LABEL}\ label_1] +\!\!+ code_2$ $\quad +\!\!+ [\texttt{GOTO}\ label_3,\ \texttt{LABEL}\ label_2]$ $\quad +\!\!+ code_3 +\!\!+ [\texttt{LABEL}\ label_3]$

$Trans_{Cond}(Cond, label_t, label_f, vtable, ftable) = \texttt{case}\ Cond\ \texttt{of}$	
$Exp_1\ \textbf{relop}\ Exp_2$	$t_1 = newvar()$ $t_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, t_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, t_2)$ $op = transop(getopname(\textbf{relop}))$ $code_1 +\!\!+ code_2 +\!\!+ [\texttt{IF}\ t_1\ op\ t_2\ \texttt{THEN}\ label_t\ \texttt{ELSE}\ label_f]$

Statements: while loop

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
`while` $Cond$ `do` $Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_2, label_3, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $[\texttt{LABEL}\ label_1] +\!\!+ code_1$ $\quad +\!\!+ [\texttt{LABEL}\ label_2] +\!\!+ code_2$ $\quad +\!\!+ [\texttt{GOTO}\ label_1,\ \texttt{LABEL}\ label_3]$

Logical operators

- Logical conjunction, disjunction, and negation are often available to define conditions
- Two ways to implement them:
  - Usual arithmetic operators: arguments are evaluated and then the operators is applied. Example in C: bitwise operators: '&' and '|'.
  - Sequential logical operators: the second operand is not evaluated if the first determines the result (lazy or short-circuit evaluation). Example in C: logical operators '&&' and '||'.
- First type is simple to implement:
  - by allowing any expression as condition

$$\mathit{Cond} \rightarrow \mathit{Exp}$$

  - by including '&', '|', and '!' among binary and unary operators
- Second one requires more modifications

Sequential logical operators

$$\begin{array}{lcl} Cond & \to & Exp\ \textbf{relop}\ Exp \\ Cond & \to & \texttt{true} \\ Cond & \to & \texttt{false} \\ Cond & \to & \texttt{!}\ Cond \\ Cond & \to & Cond\ \texttt{\&\&}\ Cond \\ Cond & \to & Cond\ \texttt{||}\ Cond \end{array}$$

$Trans_{Cond}(Cond, label_t, label_f, vtable, ftable) =$ `case` $Cond$ `of`	
`true`	$[\texttt{GOTO}\ label_t]$
`false`	$[\texttt{GOTO}\ label_f]$
`!` $Cond_1$	$Trans_{Cond}(Cond_1, label_f, label_t, vtable, ftable)$
$Cond_1$ `&&` $Cond_2$	$arg_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, arg_2, label_f, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1 +\!\!+ [\texttt{LABEL}\ arg_2] +\!\!+ code_2$
$Cond_1$ `||` $Cond_2$	$arg_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond_1, label_t, arg_2, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond_2, label_t, label_f, vtable, ftable)$ $code_1 +\!\!+ [\texttt{LABEL}\ arg_2] +\!\!+ code_2$

Other statements

More advanced control statements:

- Goto and labels: labels are stored in the symbol table (and associated with intermediate language labels). Generated as soon as a jump or a declaration is met (to avoid one additional pass)
- Break/exit: pass an additional (inherited) attribute to the translation function of loops with the label a break/exit should jump to. A new label is passed when entering a new loop.
- Case/switch-statements: translated with nested if-then-else statements.
- ...

Arrays

Language can be extended with one-dimensional arrays:

$$
\begin{array}{lcl}
Exp & \rightarrow & Index \\
Stat & \rightarrow & Index := Exp \\
Index & \rightarrow & \mathbf{id}[Exp]
\end{array}
$$

Principle of translation:

- Arrays can be allocated statically (at compile-time) or dynamically (at run-time)
- Base address of the array is stored as a constant in the case of static allocation, or in a variable in the case of dynamic allocation
- The symbol table binds the array name to the constant or variable containing its address

Arrays: translation

$Trans_{Exp}(Exp, vtable, ftable, place) =$ `case` Exp `of`	
$Index$	$(code_1, address) = Trans_{Index}(Index, vtable, ftable)$ $code_1 +\!\!+ [place := M[address]]$

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
$Index := Exp$	$(code_1, address) = Trans_{Index}(Index, vtable, ftable)$ $t = newvar()$ $code_2 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_1 +\!\!+ code_2 +\!\!+ [M[address] := t]$

$Trans_{Index}(Index, vtable, ftable) =$ `case` $Index$ `of`	
$\mathbf{id}[Exp]$	$base = lookup(vtable, getname(\mathbf{id}))$ $t = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, t)$ $code_2 = code_1 +\!\!+ [t := t * 4, t := t + base]$ $(code_2, t)$

(Assuming arrays are indexed starting at 0 and integers are 32 bits long)

Multi-dimensional arrays

$$Index \rightarrow \mathbf{id}[Exp]$$
$$Index \rightarrow Index[Exp]$$

Principle of translation:

- Two ways to represent a 2-dimensional array in linear memory:
  - Row-major order: one row at a time. For a 3×2 array: a[0][0], a[0][1], a[1][0], a[1][1], a[2][0], a[2][1]
  - Column-major order: one column at a time. For a 3×2 array: a[0][0], a[1][0], a[2][0], a[0][1], a[1][1], a[2][1]
- Generalization: if $dim_0, dim_1, \ldots, dim_{n-1}$ are the sizes of the dimensions in a n-dimensional array, the element $[i_0][i_1]\ldots[i_{n-1}]$ has the address:
  - Row-major:
    $base + ((\ldots(i_0 \cdot dim_1 + i_1) \cdot dim_2 \ldots + i_{n-2}) \cdot dim_{n-1} + i_{n-1}) \cdot size$
  - Column-major:
    $base + ((\ldots(i_{n-1} \cdot dim_0 + i_{n-2}) \cdot dim_1 \ldots + i_1) \cdot dim_{n-2} + i_0) \cdot size$
- Dimension sizes are stored as constant (static), in variables or in memory next to the array data (dynamic)

Multi-dimensional arrays: translation

$Trans_{Index}(Index, vtable, ftable) =$
$(code_1, t, base, []) = Calc_{Index}(Index, vtable, ftable)$ $code_2 = code_1 ++ [t := t * 4, t := t + base]$ $(code_2, t)$

$Calc_{Index}(Index, vtable, ftable) =$ case $Index$ of	
$\mathbf{id}[Exp]$	$(base, dims) = lookup(vtable, getname(\mathbf{id}))$ $t = newvar()$ $code = Trans_{Exp}(Exp, vtable, ftable, t)$ $(code, t, base, tail(dims))$
$Index[Exp]$	$(code_1, t_1, base, dims) = Calc_{Index}(Index, vtable, ftable)$ $dim_1 = head(dims)$ $t_2 = newvar()$ $code_2 = Trans_{Exp}(Exp, vtable, ftable, t_2)$ $code_3 = code_1 ++ code_2 ++ [t_1 := t_1 * dim_1, t_1 := t_1 + t_2]$ $(code_3, t_1, base, tail(dims))$

(Assume dimension sizes are stored in the symbol table, as constant or variable)

Other structures

- Floating point values: can be treated the same way as integers (assuming the intermediate language has specific variables and operators for floating point numbers)
- Records/structures: allocated in a similar way as arrays
  - Each field is accessed by adding an offset to the base-address of the record
  - Base-addresses and offsets for each field are stored in the symbol table for all record-variables
- Strings: similar to arrays of bytes but with a length that can vary at run-time
- ...

Variable declaration

$$
\begin{array}{lcl}
Stat & \rightarrow & Decl \; ; \; Stat \\
Decl & \rightarrow & \texttt{int} \; \mathbf{id} \\
Decl & \rightarrow & \texttt{int} \; \mathbf{id}[\mathbf{num}]
\end{array}
$$

Principle of translation:

- Information about where to find scalar variables (e.g. integer) and arrays after declaration is stored in the symbol table
- Allocations can be done in many ways and places (static, dynamic, local, global. . .)

Variable declaration

$Trans_{Stat}(Stat, vtable, ftable) =$ `case` $Stat$ `of`	
$Decl\ ;\ Stat_1$	$(code_1, vtable_1) = Trans_{Decl}(Decl, vtable)$ $code_2 = Trans_{Stat}(Stat_1, vtable_1, ftable)$ $code_1 +\!\!+ code_2$

$Trans_{Decl}(Decl, vtable) =$ `case` $Decl$ `of`	
`int` **id**	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\mathbf{id}), t_1)$ $([], vtable_1)$
`int` **id**[**num**]	$t_1 = newvar()$ $vtable_1 = bind(vtable, getname(\mathbf{id}), t_1)$ $([t_1 := HP, HP := HP + (4 * getvalue(\mathbf{num}))], vtable_1)$

(Assumes scalar variables are stored in intermediate language variables and arrays are dynamically allocated on the heap, with their base-addresses stored in a variable. *HP* points to the first free position of the heap.)

Comments

- Needs to add error checking in previous illustration (array index out of bounds in arrays, wrong number of dimensions, memory/heap overflow, etc.)
- In practice, results of translation are not returned as strings but either:
  - output directly into an array or a file
  - or stored into a structure (translation tree or linked list)

  The latter allows subsequent code restructuring during optimization
- We have not talked about:
  - memory organization: typically subdivided into static data (for static allocation), heap (for dynamic allocation) and stack (for function calls)
  - translation of function calls: function arguments, local variables, and return address are stored on the stack (similar to what you have seen in INFO-0012, computation structures)

Outline

1. Intermediate representations

2. Illustration

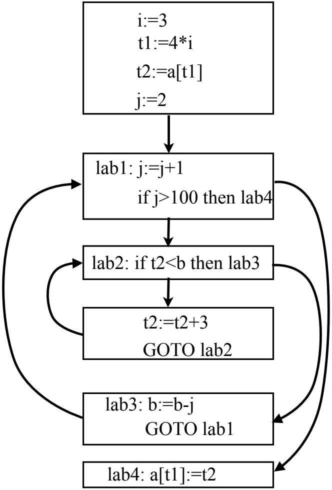
3. Optimization

IR code optimization

- IR code generation is usually followed by code optimization
- Why?
  - IR generation introduces redundancy
  - To compensate for laziness of programmers
- Improvement rather than optimization since optimization is undecidable
- Challenges in optimization:
  - Correctness: should not change the semantic of the program
  - Efficiency: should produce IR code as efficient as possible
  - Computing times: should not take too much time to optimize
- What to optimize?
  - Computing times
  - Memory usage
  - Power consumption
  - ...

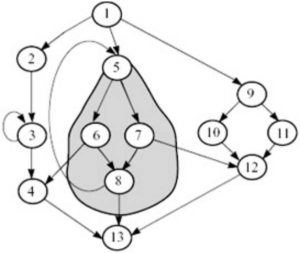
Control-flow graph

- A basic block is a series of IR instructions where:
  - there is one entry point into the basic block, and
  - there is one exit point out of the basic block.
- Control-flow graph: nodes are basic blocks and edges are jumps between blocks



Control-flow graph and SSA

- The control-flow graph (CFG) can be used to determine where to introduce Φ functions when deriving a SSA form:
  - A node A (basic block) of the CFG *strictly dominates* a node B if it is impossible to reach B without going through A. A *dominates* B if A stricly dominates B or $A = B$.
  - B is in the *dominance frontier* of A if A does not strictly dominate B, but dominates some immediate predecessor of B.
  - Whenever node A contains a definition of a variable x, any node B in the dominance frontier of A needs a Φ function for x.
- There exist an efficient algorithm to find the dominance frontier of a node



4,5,12,13 are in the dominance frontier of 5

(Appel)

Local optimizations

Local optimization: optimization within a single basic block

Examples:

- Constant folding: evaluation at compile-time of expressions whose operands are constant
  - 10+2*3 $\rightarrow$ 16
  - [If 1 then Lab1 Else Lab2] $\rightarrow$ [GOTO Lab1]

- Constant propagation: if a variable is assigned a constant, then propagate the constant into each use of the variable
  - [x:=4;t:=y*x;] can be transformed into [t:=y*4;] if x is not used later

Local optimizations

Examples:

- Copy propagation: similar to constant propagation but generalized to non constant values

```
tmp2 = tmp1;
tmp3 = tmp2 * tmp1;
tmp4 = tmp3;
tmp5 = tmp3 * tmp2;
c = tmp5 + tmp4;
```

```
tmp3 = tmp1 * tmp1;
tmp5 = tmp3 * tmp1;
c = tmp5 + tmp3;
```

- Dead code elimination: remove instructions whose result is never used
  - Example: Remove [tmp1=tmp2+tmp3;] if tmp1 is never used

Local optimizations

Examples:

- Common subexpression elimination: if two operations produce the same results, compute the result once and reference it the second time
  - Example: in `a[i]=a[i]+2`, the address of `a[i]` is computed twice. When translating, do it once and store the result in a temporary variable

- Code moving/hoisting: move outside of a loop all computations independent of the variables that are changing inside the loop
  - Example: part of the computation of the address for `a[i][j]` can be removed from this loop

```
while (j<k) {
  sum = sum + a[i][j];
  j++;
}
```

IR code optimization

- Local optimizations can be interleaved in different ways and applied several times each
- Optimal optimization order is very difficult to determine

- Global optimization: optimization across basic blocks
  - Implies performing data-flow analysis, i.e., determine how values propagate through the control-flow graph
  - More complicated than local optimization