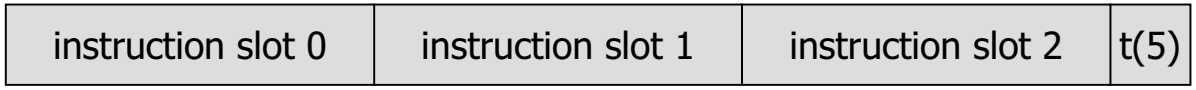
IA-64 Architecture

- Review of bundling and dispersal.
- Review of control and data speculation.
- Intel Itanium.

IA-64 Architecture

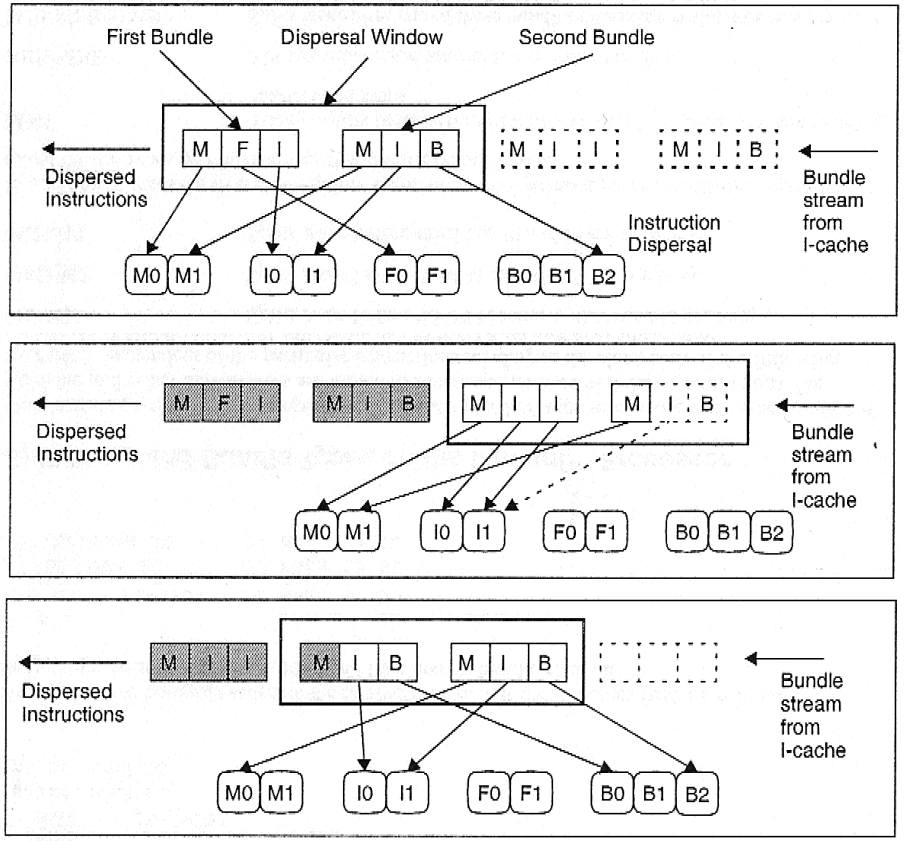
- The IA-64 defines 128-bit long instruction words (called bundles) consisting of three 41-bit microoperations and a 5-bit template field. Multiple bundles can be issued per clock cycle (number is defined by implementation).



- Template field:
  - helps decode and route instructions
  - indicates the location of stops that mark the end of groups of microoperations that can execute in parallel

- Dispersal: process of sending instructions to functional units.

IA-64 Architecture: Bundling and Dispersal on Itanium



IA-64 Architecture

- Between instruction groups: instructions behave as though read accesses occur after the update of all instructions from the previous group.

- Within instruction groups: instructions behave as though
  - memory read accesses occurred after the update of the memory of all prior instructions in the same group.
  - register read accesses occurred before the update of the register state by any instruction (prior or later) in the same instruction group – with the exception of some dependence requirements (see next slide).

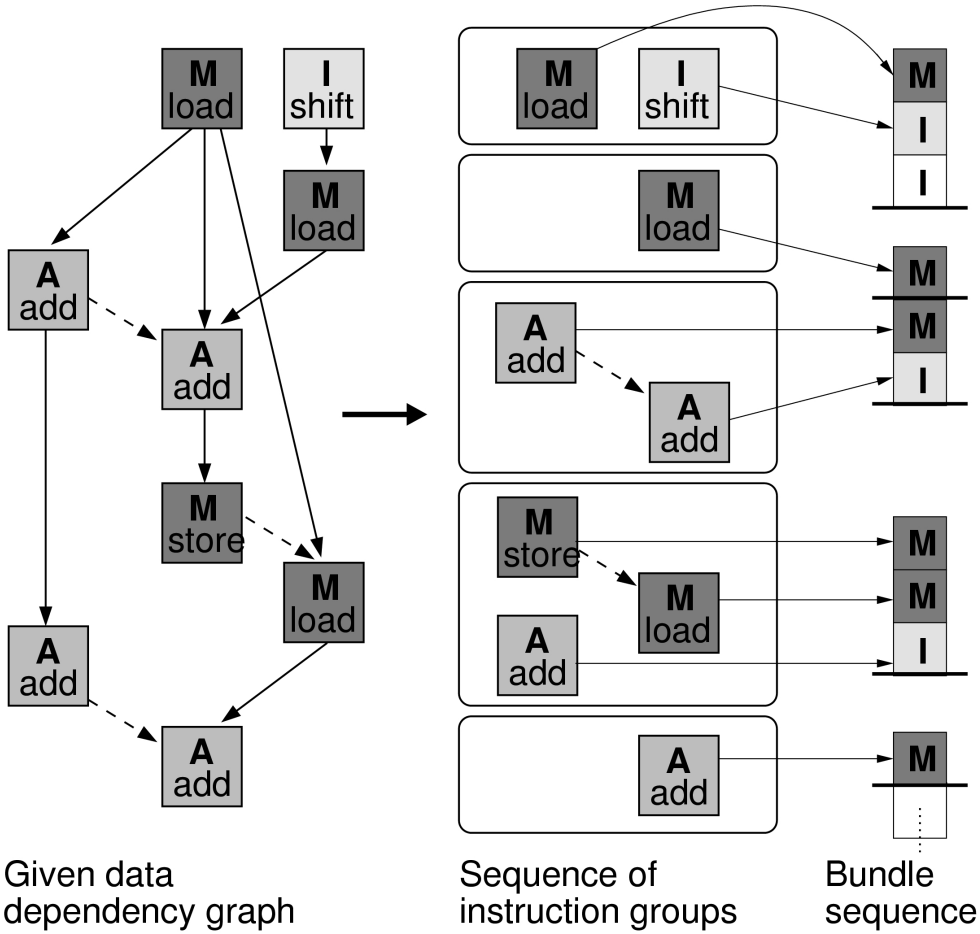
IA-64 Architecture

- Register dependencies: Within an instruction group, read-after-write (RAW) and write-after-read (WAR) register dependencies are not allowed (with respect to explicit and implicit register accesses), except for some special cases. Write-after-read register dependencies (WAR) are allowed, except for some special cases.

- If the program violates the dependency rules within an instruction group, processor behavior is undefined.

- Memory dependencies: Within an instruction group, RAW, WAW, and WAR memory dependencies are allowed. A load will observe the results of the most recent store to the same memory address.

IA-64 Architecture



IA-64 Architecture

Control speculation:

```
(p1)    br.cond label
        ld8 r4=[r3];;
        add r4=8,r4;;
```



```
        ld8.s r4=[r3];;
// ...
(p1)    br.cond label;;
        chk.s r4, recovery;;
back:   add r4=8,r4;;
// ...
recovery:
        ld8 r4=[r3];;
        br back;;
```

IA-64 Architecture

- Data Speculation:

```
//...
st8 [r4]=r12
ld8 r6=[r8];;
add r5=r6,r7;;
st8 [r18]=r5
```

```
        ld8.a r6=[r8];;
        // ...
        add r5=r6,r7;;
        // ...
        st8 [r4]=r12
        chk.a.clr r6, recovery
back:   st8 [r18]=r5
        // ...
recovery:
        ld8 r6=[r8];;
        add r5=r6,r7
        br back;;
```

IA-64 Architecture

- Multimedia instructions supporting packed arithmetic: a general purpose register is considered as concatenated of 8 8-bit, 4 16-bit, or 2 32-bit elements. Operations:
  - parallel modulo addition (padd), parallel modulo subtraction (psub), parallel compare (pcmp), etc.
  - Data arrangement: interleave odd elements from both sources (mix.l), convert from larger to smaller elements with signed saturation (pack.uss), etc.
- Integer (fixed-point) multiply is executed in the floating-point unit (multiply/accumulate). Example: xma f1=f3, f4, f2 computes f1:=f2+f3*f4.

Intel Itanium

- First IA-64 implementation.
- Functional units:
  - 2 M-type
  - 2 I-type
  - 2 F-type
  - 3 B-type
- All functional units are fully pipelined.
- Functional units of the same type are not identical; seldomly used operations usually can only be executed on the first unit of each type.

Intel Itanium

- Dispersal window contains two bundles allowing up to 6 microoperations to be issued in one clock cycle, among which at most 2 I-type, 2 M-type, 2 F-type, 3 B-type operations.
- The functional unit a microoperation is executed by is determined by its type and its position in the bundle.
- Execution time:
  - 1-2 cycles for most integer operations
  - 2-7 cycles for F-type operations
- 10-stage pipeline, divided into two parts:
  - the 3-stage *front end* reads the instruction stream
  - the 7-stage main pipeline executes the instructions
  - *front end* and main pipeline are decoupled; they are connected by a FIFO buffer that can contain up to 8 bundles.

Orientation

- ✓ Embedded systems: Application areas and characteristics.
- ✓ Digital signal processing: important concepts and algorithms.
- ✓ Real-life hardware architectures for embedded applications:
  - Analog Devices ADSP-2106x SHARC
  - Infineon TriCore
  - Philips TriMedia TM1000
  - Intel IA-64 Architecture & Intel Itanium (currently used for GPP)
- Today:
  - Summary of features of embedded processors
  - Basics of code generation.

Characteristics of DSPs

- Multiply-accumulate units: multiplication and accumulation in a single clock cycle (vector products, digital filters, correlation, fourier transforms, etc)
- Multiple-access memory architectures for high bandwidth between processor and memory
  - Goal: throughput of one operation per clock cycle.
  - Required: several memory accesses per clock cycle.
  - Separate data and program memory space: harvard architecture.
  - Multiple memory banks
  - arithmetic operations in parallel to memory accesses. But often irregular restrictions.

Characteristics of DSPs

- Specialized addressing modes:
  - linear addressing mode: addition/subtraction of offset to/from base address, often with auto-modify (pre-/post increment/decrement)
  - circular addressing: address results from incrementing an address register and taking the remainder of the division of this value by a constant.
  - bit-reverse addressing: carry bits are propagated in bit-reverse order (-> FFT)

Characteristics of DSPs

- Branches depending on control bits: early branch detection allowing for small interlock penalties in combination with typically short pipelines
- Residual control: execution behavior depends on specific control bits set by a preceding instruction.
- Predicated/guarded execution: instruction execution depends on the value of explicitly specified bit values or registers.
- Increasingly: SIMD instructions for packed arithmetic (multimedia applications).

Characteristics of DSPs

- Hardware loops / zero overhead loops: no explicit loop counter increment/decrement, no loop condition check, no branch back to top of loop
- Restricted interconnectivity between registers and functional units -> more severe phase coupling problems.
- Complex instructions with specific functionality, typical for the application area.
- Strongly encoded instruction formats: a throughput of one instruction per clock cycle requires one instruction to be fetched per cycle. Thus each instruction has to fit in one memory word -> reduction of bit width of the instruction. Techniques:

Characteristics of DSPs

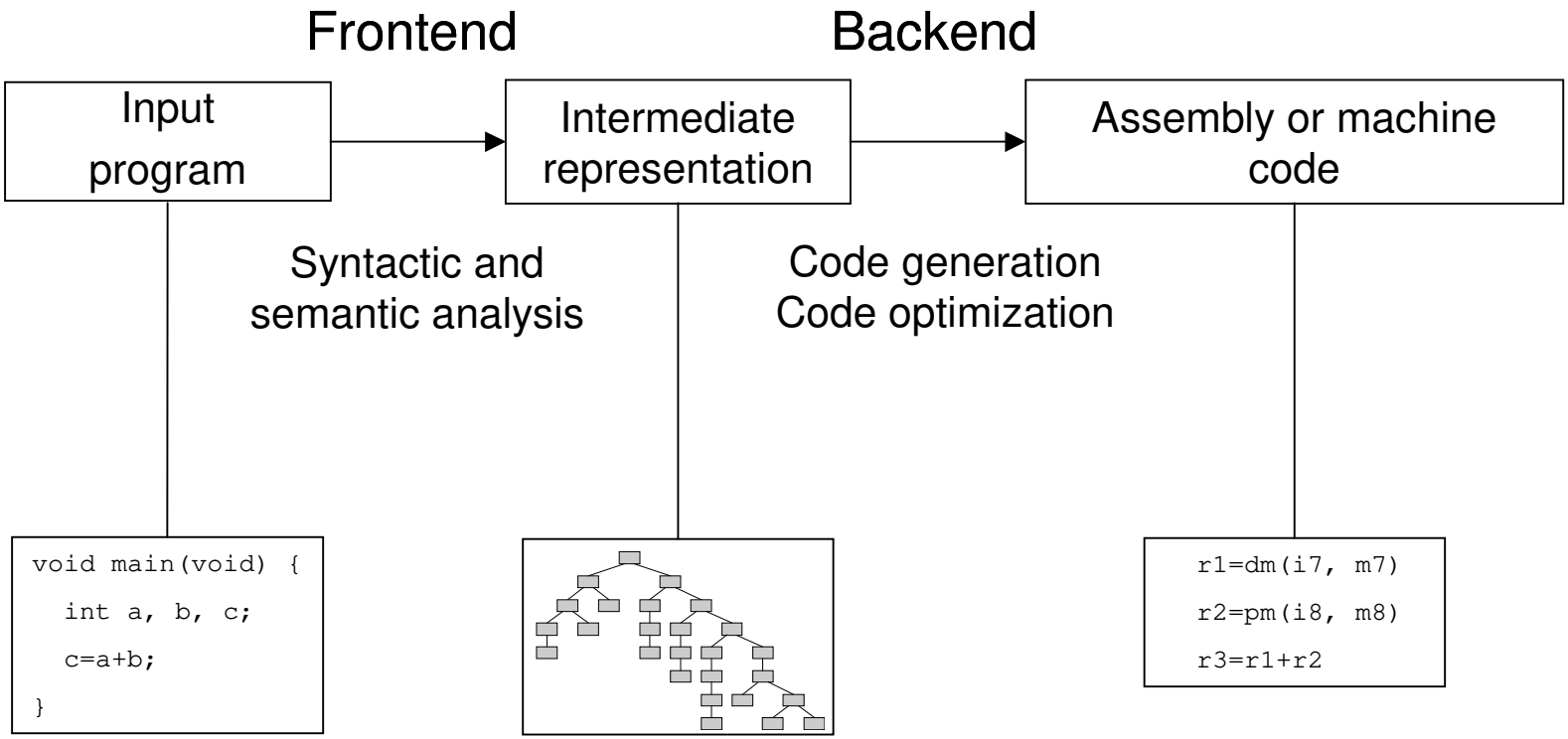
Techniques for reducing the instruction width:

- Reducing the number of addressing modes. Example: immediate memory accesses restricted to a small set of instructions.
- Restricting the set of source and destination operands: short addressing modes for elements of certain register groups, implied operands.
- Mode bits, for example a single shift instruction that performs arithmetic or logical shifts depending on a control bit in a mode register.
- Consequence: Increased irregularity of the instruction set – but reduction of processor and system cost and increased individual instruction performance.

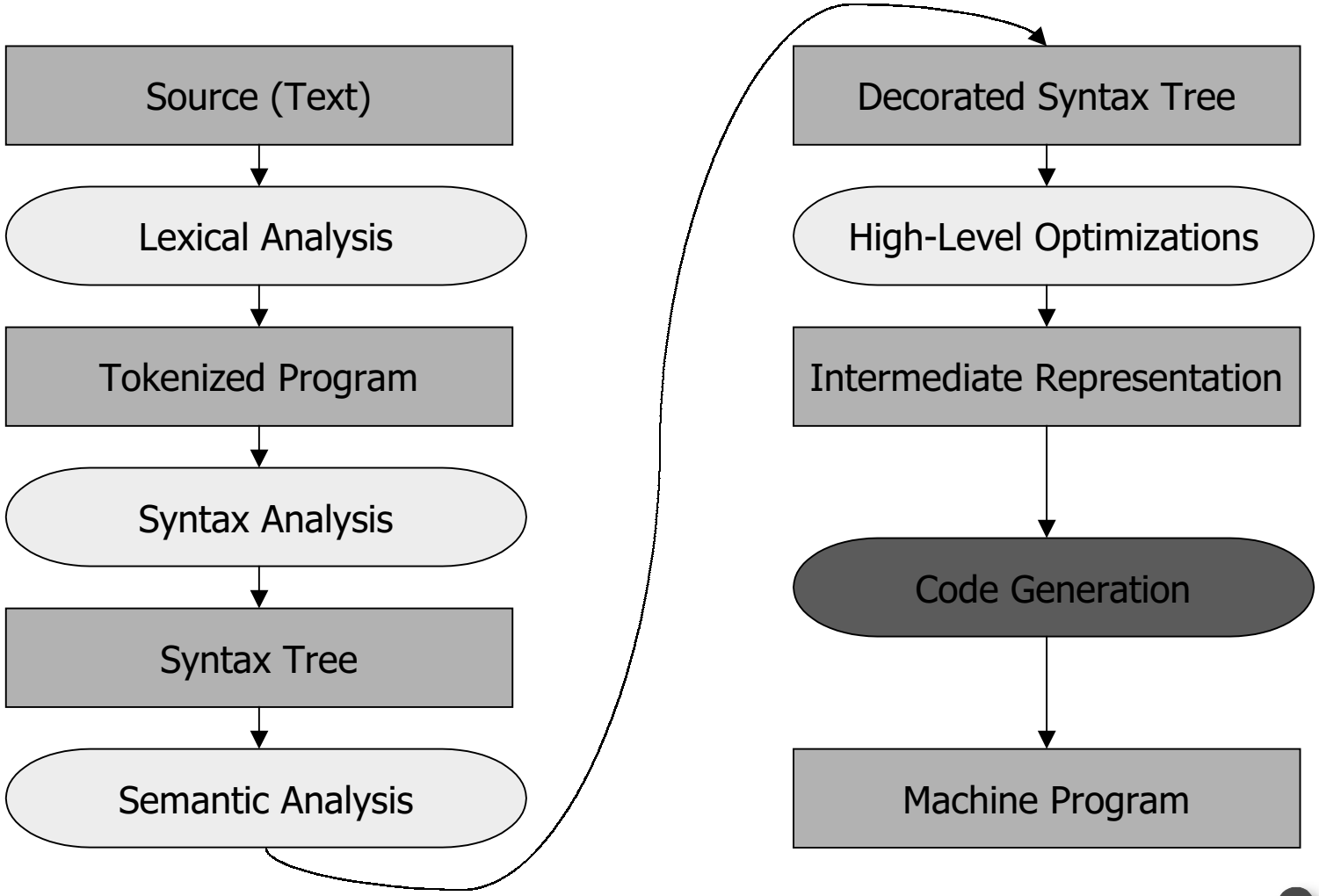
Characteristics of DSPs

- Overall consequence:
  - Irregularity
  - Severe phase coupling problems
  - Need for specialized algorithms

Compiler Structure



Detailed Compiler Structure

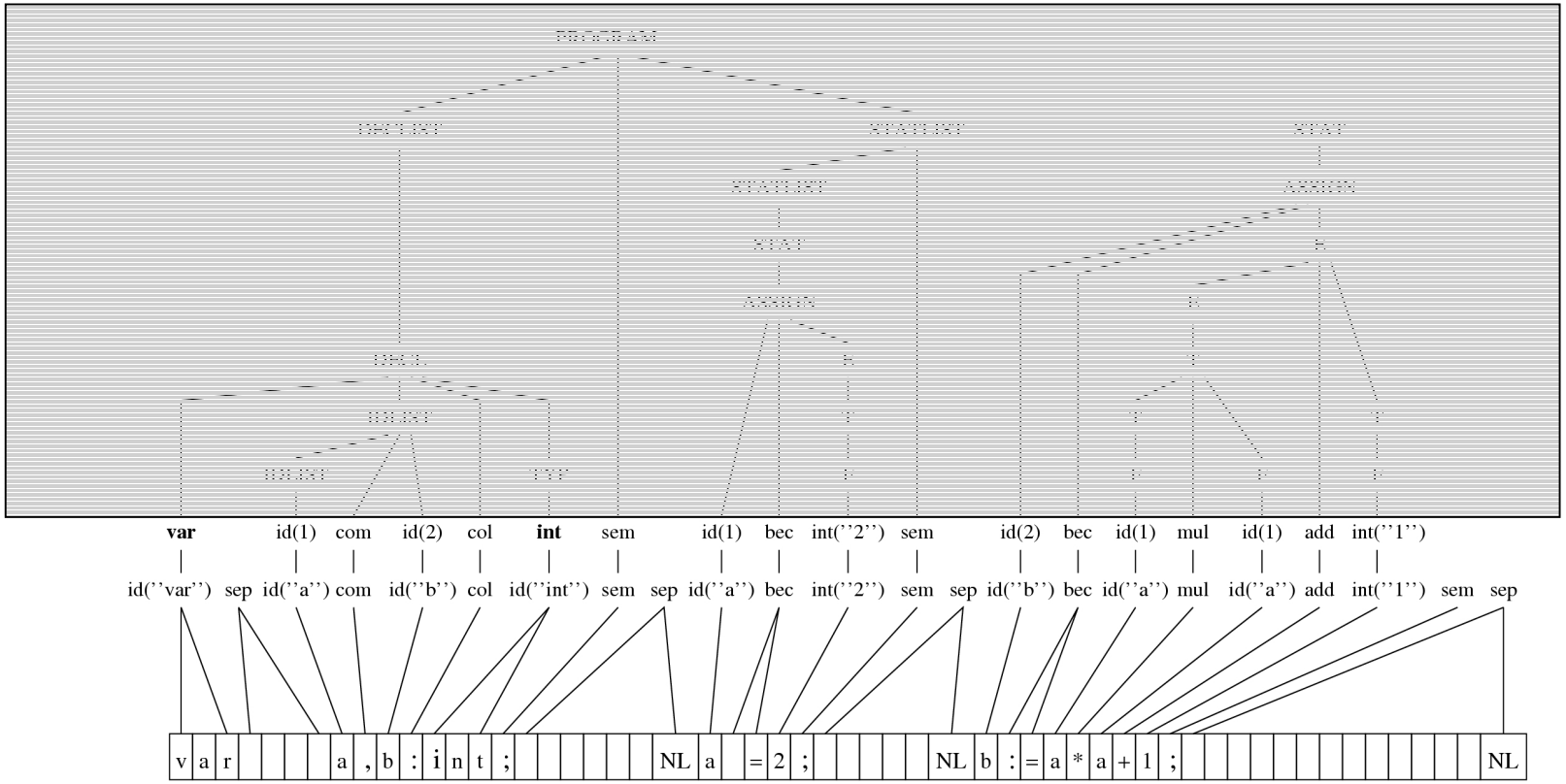


Lexical Analysis (Scanning & Screeing)

- Input: Program text as sequence of characters.
  Output: Program text as sequence of symbols (tokens).

1. Read Input file.
2. Report errors about symbols illegal in the programming language.
3. Screening subtask:
   - Identify language keywords and standard identifiers.
   - Elimininate "white-spaces", e.g. consecutive blanks and newlines.
   - Count line numbers.

Lexical Analysis (Scanning)

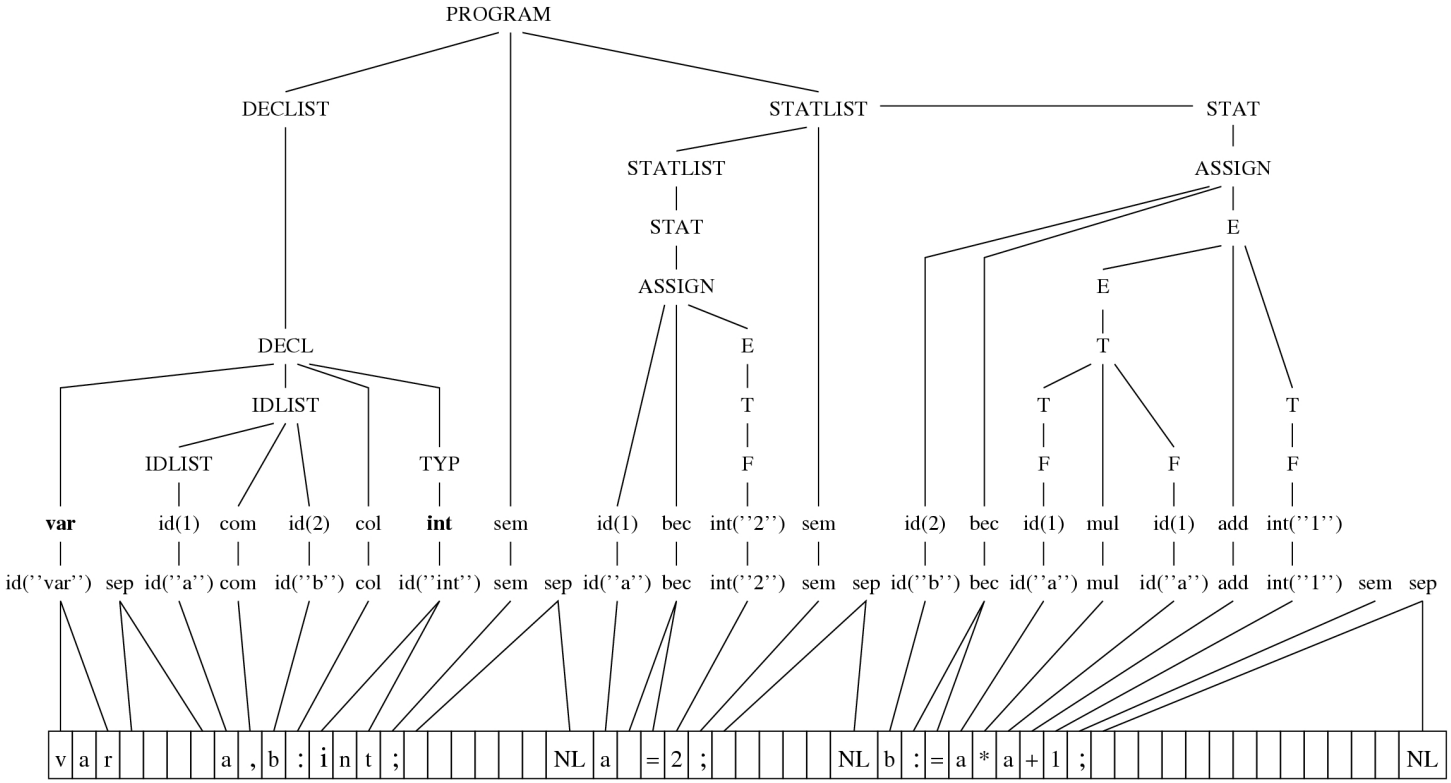


Syntax Analysis (Parsing)

- Input: Sequence of symbols (tokens).
  Output: Structure of the program:
  - concrete syntax tree,
  - abstract syntax tree, or
  - parse (derivation).

- Syntax errors:
  - report (as many as possible) syntax errors,
  - diagnose syntax errors,
  - correct syntax errors.

Syntax Analysis (Parsing)

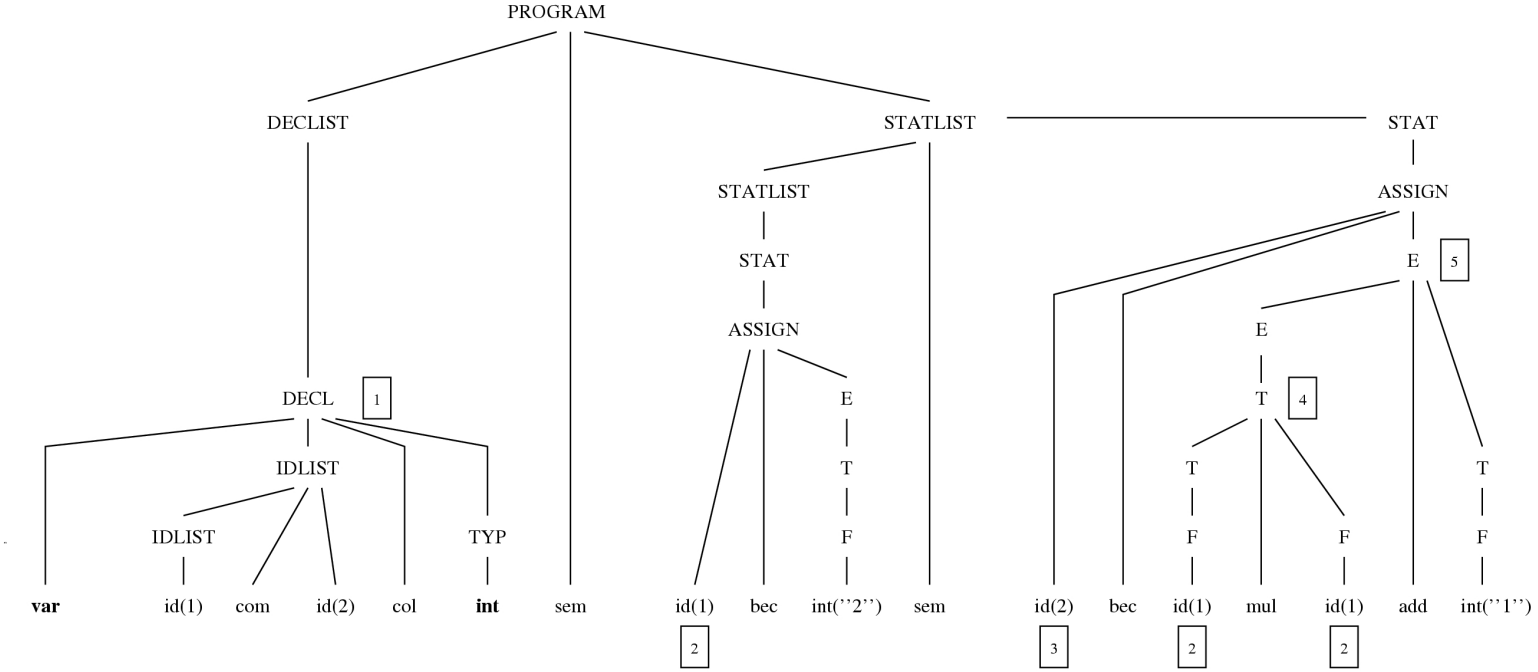


Semantic Analysis

- Input: Abstract syntax tree.
  Output: Abstract syntax tree decorated with attributes, e.g. types of subexpressions.

- Report semantic errors, e.g. undeclared variables, type mismatches.
- Resolve usage of variables: identify applied occurrences of variables with their declarations.
- Compute type of every (sub-)expression.

Semantic Analysis



1 (id(1),(var,int))
(id(2),(var,int))

2 (var,int)

3 (var,int)

4 int

5 int

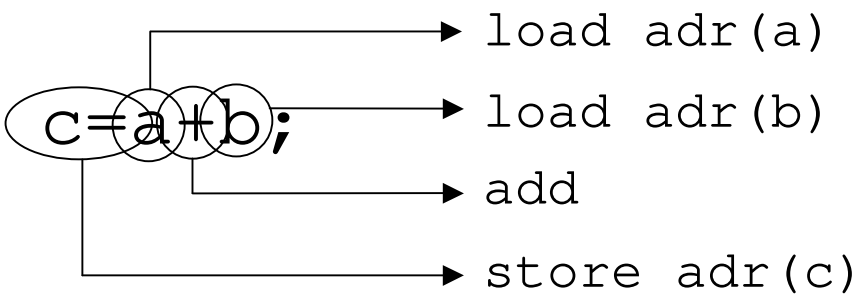
1 (id(1),(var,int,0))
(id(2),(var,int,0))

2 (var,int,0)

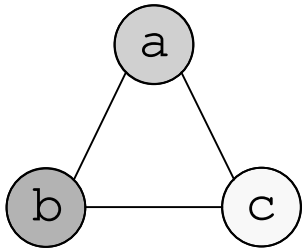
3 (var,int,1)

Code Selection, Register Allocation, and Instruction Scheduling

Code selection



Register allocation



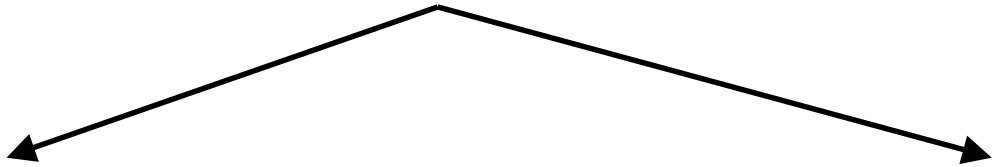
$a \mapsto r1$
$b \mapsto r2$
$c \mapsto r3$

Instruction scheduling

```
r1=load adr(a) || r2=load adr(b)
r3=add r1, r2
store adr(c), r3
```

Phase Coupling Problem

```
d1=s1*s1;
d2=s1+s2;
```



```
r3=r1*r1
store r3
r3=r1+r2
store r3
```

```
r3=r1*r1, r4=r1+r2
store r3
store r4
```

Optimize number of used registers

Optimize code speed and size

Main Tasks of Code Generation (1)

- Code selection: Map the intermediate representation to a semantically equivalent sequence of machine operations that is as efficient as possible.

- Register allocation: Map the values of the intermediate representation to physical registers in order to minimize the number of memory references during program execution.
  - Register allocation proper: Decide which variables and expressions of the IR are mapped to registers and which ones are kept in memory.
  - Register assignment: Determine the physical registers that are used to store the values that have been previously selected to reside in registers.

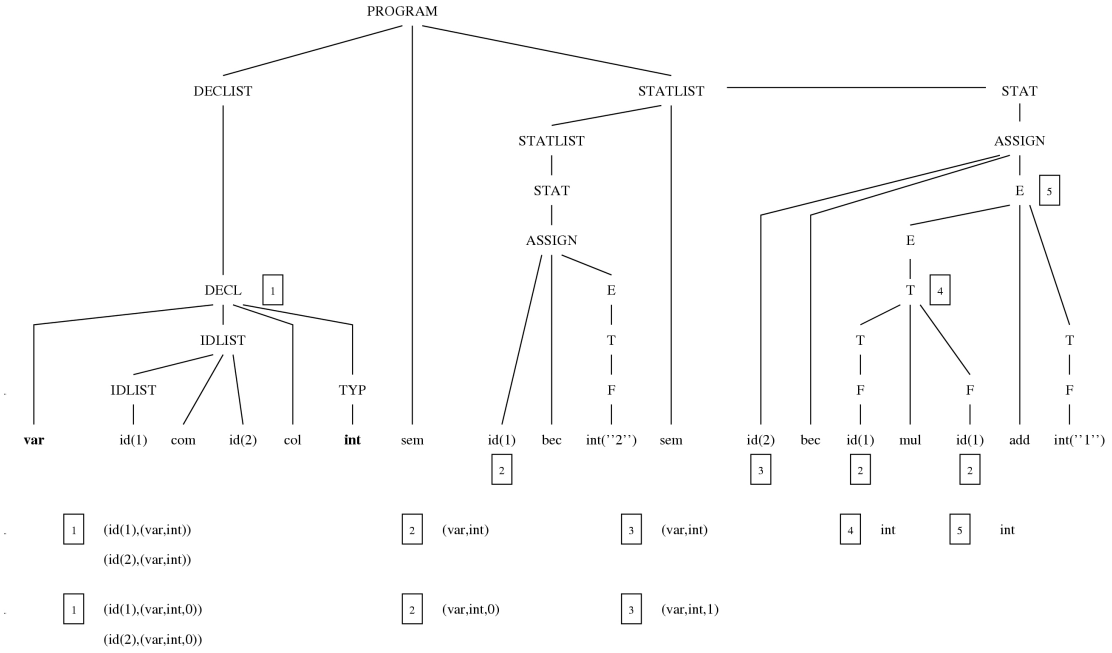
Main Tasks of Code Generation (2)

- Instruction scheduling:
  Reorder the produced operation stream in order to minimize pipeline stalls and exploit the available instruction-level parallelism.

- Resource allocation / functional unit binding:
  Bind operations to machine resources, e.g. functional units or buses.

Program Representations

- Intermediate Representation: typcially the decorated syntax tree, the control flow graph, or the data dependence graph of the input program.



Program Representations: Call Graph

- Call graph:
  - There is a node for the main procedure – being the entry node of the program – and a node for each procedure declared in the program.
  - The nodes are marked with the procedure names.
  - There is an edge between the node for a procedure *p* to the node of procedure *q*, if there is a call to *q* inside of *p*.

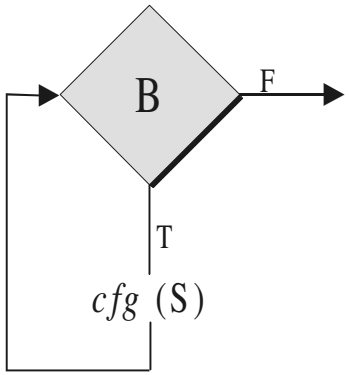
Program Representations: Control Flow Graph

- The control flow graph of a procedure is a directed graph $G_C=(N_C,E_C,n_A,n_\Omega)$ with node and edge labels. For each instruction i of the procedure there is a node n_i that is marked by i. The edges (n,m,λ) denote the control flow of the procedure: $\lambda \in \{T,F,\varepsilon\}$ is the edge label. The nodes for composed statements are shown on the next slide. Edges belonging to unconditional branches lead from the node of the branch to the branch destination. The node n_A is the uniquely determined entry point in the procedure; it belongs to the first instruction to be executed. n_Ω denotes the end node that is reached by any path through the control flow graph.

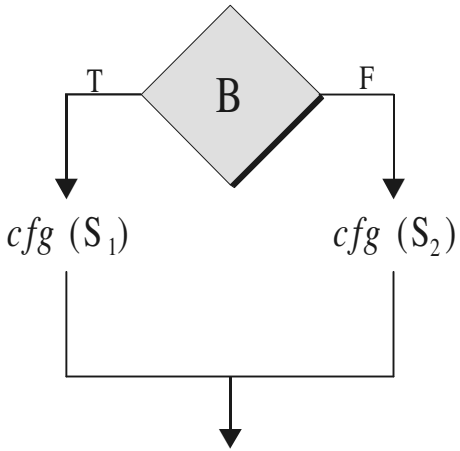
- Nodes with more than one predecessor are called joins and nodes with more than one successor are called forks.

Control Flow Graph – Composed Statements

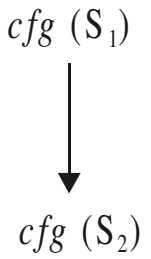
cfg (while B do S od) =



cfg (if B then S_1 else S_2 fi) =



cfg (S_1;S_2) =



Program Representations: Basic Block Graph & Call Graph

- A basic block in a control flow graph is a path of maximal length which has no joins except at the beginning and no forks except possibly at the end.

- The basic block graph $G_B=(N_B, E_B, b_A, b_\Omega)$ of a control flow graph $G_C=(N_C, E_C, n_A, n_\Omega)$ is formed from G_C by combining each basic block into a node. Edges of G_C leading into the first node of a basic block lead to the node of that basic block in G_B. Edges of G_C leaving the last node of a basic block lead out of the node of that basic block in G_B. The node b_A denotes the uniquely determined entry block of the procedure; b_Ω denotes the exit block that is reached at the end of any path through the procedure.

Program Representations: Data Dependence Graph

- Let G_C be a control flow graph. It data dependence graph is a directed graph $G_D=(N_D,E_D)$ with node and edge labels whose nodes are labeled by the operations of the procedure. An edge runs from the node of an operation *i* to the node of an operation *j*, if *i* has to be executed before *j*, i.e. if there is a path from *i* to *j* in the control flow graph and if
  - *i* defines a resource *r*, *j* uses it and the path from *i* to *j* does not contain other definitions of *r* (true dependence, RAW): $(i,j,r,t) \in E_D$
  - *i* uses a resource, *j* defines it and the path from *i* to *j* does not contain any definitions of *r* (anti dependence, WAR): $(i,j,r,a) \in E_D$
  - *i* and *j* define the same resource and the path from *i* to *j* does not contain any uses nor definitions of *r* (output dependence, WAR): $(i,j,r,o) \in E_D$.

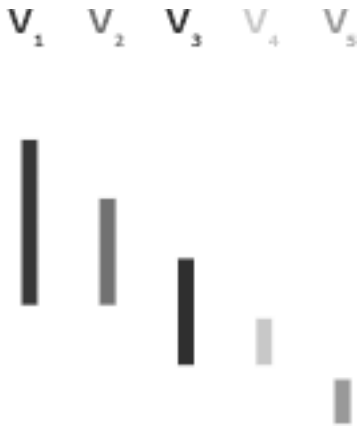
Life Range and Register Interference

- A symbolic register (a variable) r is live at a program point *p*, if there is a program path from the entry node of the procedure to *p* that contains a definition of *r* and there is a path from *p* to a use of *r* on which *r* is not defined. The life range of a symbolic register *r* is the set of program points at which *r* is live.

- Two life ranges of symbolic registers interfere, if one of them is defined during the life range of the other. The register interference graph is an undirected graph whose nodes are life ranges of symbolic registers and whose edges connect the nodes of interfering life ranges.

Life Range and Register Interference

a) $v_1 = Mem[0xAFA0]$
b) $v_2 = Mem[0xAFC0]$
c) $v_3 = v_1 + v_2$
d) $v_4 = v_1 * v_2$
e) $v_5 = v_3 + v_4$
f) return v_5



Standard Algorithms for Code Generation

- Code selection by tree parsing.
- Register allocation by graph coloring.
- Instruction scheduling by
  - list scheduling
  - trace scheduling.
- Optimal algorithms for integrated code selection and register allocation for expression trees and special machine models.

Code Selection

Impact of Code Selection

```
struct {
 unsigned b1:1;
 unsigned b2:1;
 unsigned b3:1;
} s;
s.b3 = s.b1 && s.b2;
```

```
ld.bu  d5, [sp]48
extr.u d5, d5, #0, #1
ne d5, d5, #0
mov16  d4, d5
jeq    d4, #0, _9
ld.bu  d4, [sp]48
extr.u d4, d4, #1, #1
ne     d4, d4, #0
_9:
ld.b   d3, [sp]48
insert d3, d3, d4, #2, #1
st.b   [sp]48, d3
```

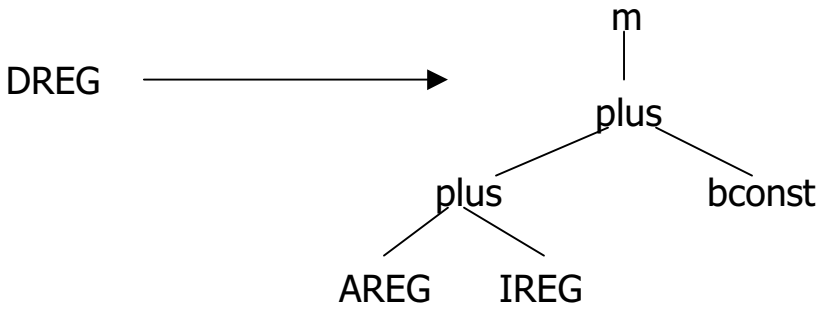
9 cycles, 42 bytes

```
ld.bu  d5, [sp]48
and.t  d3, d5, #0, d5, #1
insert d5, d5, d3, #2, #1
st.b   [sp]48, d5
```

3 cycles, 16 bytes

Generating Code Selectors

- Machine grammar: regular tree grammar;
  - terminals: operators from the program representation
  - non-terminals: represent storage resources
  - often ambiguous
  - each rule has associated costs
  - factorization, e.g. of addressing modes reduces size.



Generating Code Selectors

- A machine grammar enables IR trees for expressions to be derived. The derivation tree for an IR tree represents one possibility of generating code for the IR tree.

- The generated code selector
  - parses intermediate representations of programs
  - computes derivations according to the machine grammar, each corresponding to a sequence of machine instructions
  - has to select a cheapest derivation, corresponding to the (locally) cheapest code sequence
  - may compute costs in states or use dynamic programming