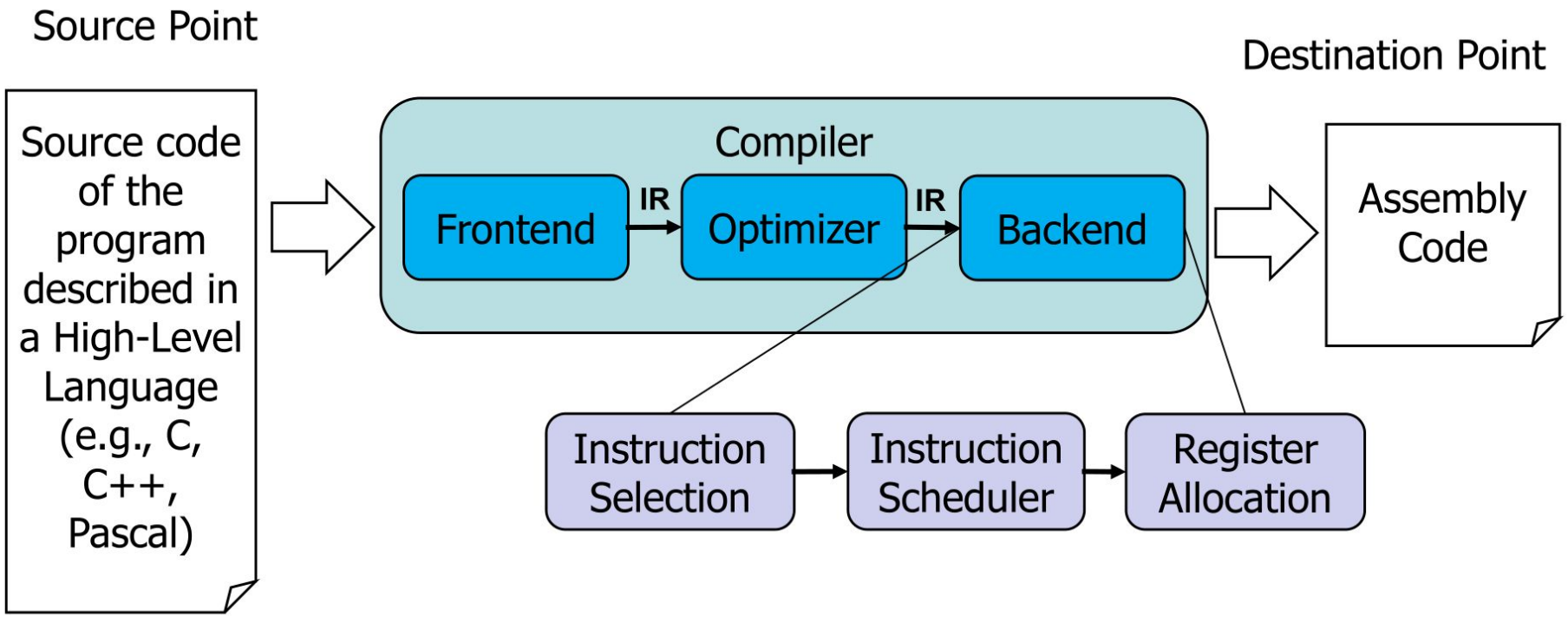
Index

Compiler Stages



Alignment and Packing

- Alignment requirements:
  - Integer types **int** (4 bytes) start in addresses with 2 LSBs == "00"
  - Integer types **short** (2 bytes) start in addresses with LSB == '0'
- Alignment requires:
  - Filling between fields to ensure alignment
  - Packing of fields to ensure memory savings

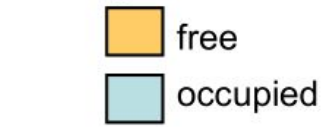
Alignment

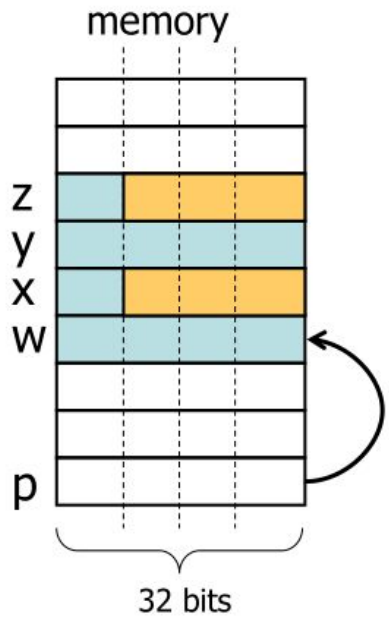
Naive organization

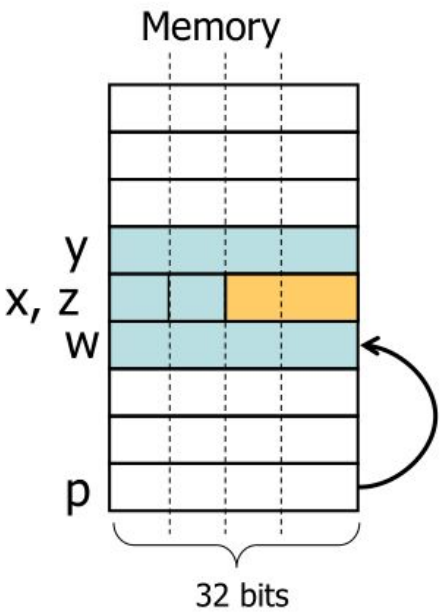
Packing (optimized)
(saves 4 bytes)

```
typedef struct {
  int w;
  char x;
  int y;
  char z;
} foo;

foo *p;
```







Conditional Structures/Constructs

- *Branch-delay*
  - The processor always executes the instruction (or a number of instructions) following a conditional branch instruction (being the jump executed or not)
  - When it is not possible to move an instruction to just after the branch instruction a **nop** needs to be inserted

If(a == 1) b = 2;

c = a+1;

- a to $t0; b to $t1

```
...
        addi    $t2, $0, 1
        bne     $t2, $t0, skip_if
        addi    $t3, $t0, 1
        addi    $t1, $0, 2
Skip_if: ...
```

Definition of Formal Languages

- Necessity to define precisely a language
- Definition of the languages structured in layers
  - Start by the set of the symbols of the language (the alphabet, Σ)
  - **Lexical structure** – identifies "words" of the language (each word is a sequence of symbols)
  - **Syntactic structure** – identifies "sentences" in the language (each sentence is a sequence of words)
  - **Semantic** – meaning of the program (specifies the results that should be output for the inputs)

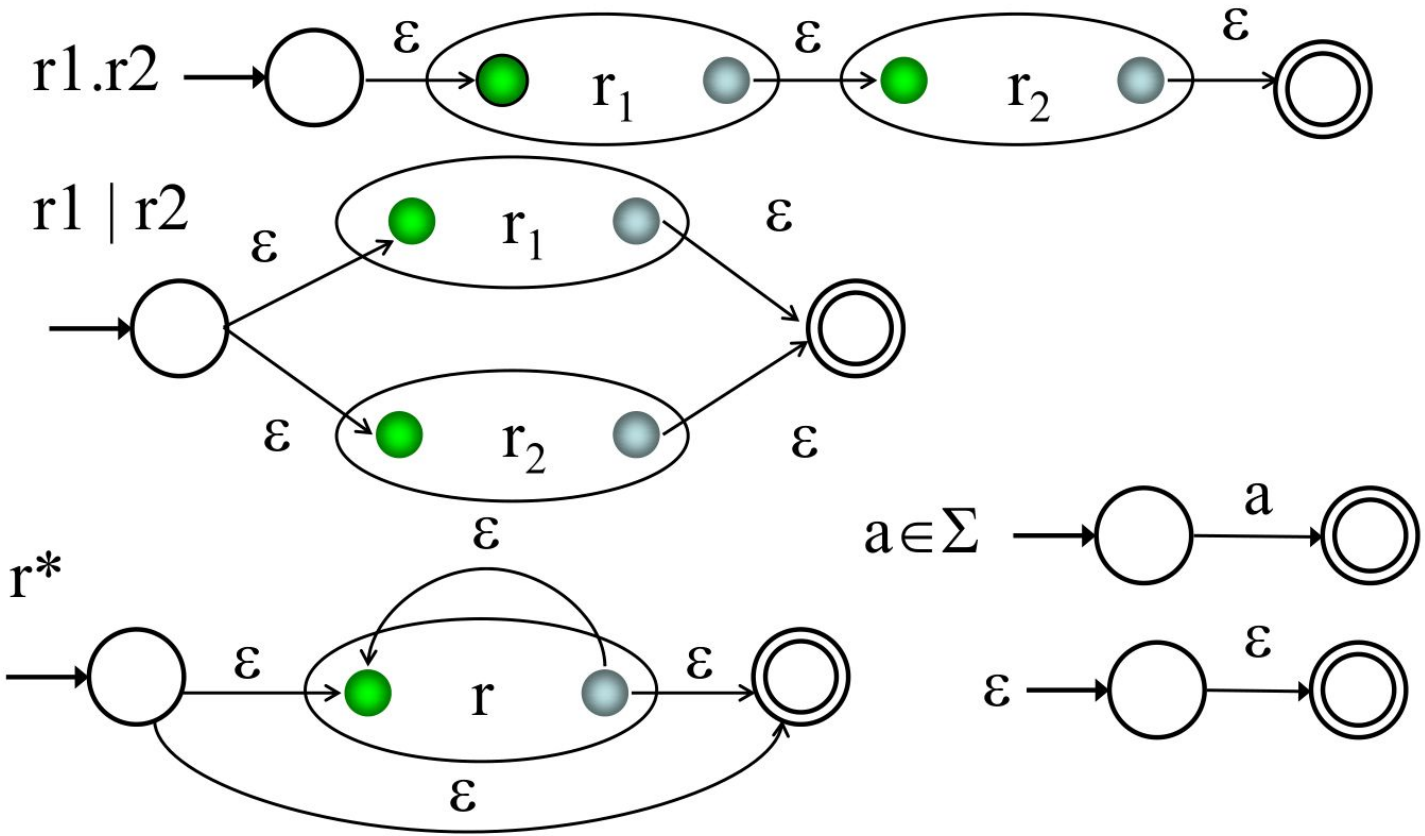
Regular Expressions

- Other constructs:
  - r^+ - one or more occorrunces of r: r | rr | rrr ...
    - Equivalent to: $r.r^*$
  - $r^?$ – zero or one occurrence of r: (r | ε)
  - [ ] – symbol classes:
    - [ac] is the same as: (a | c)
    - [a-c] is the same as: (a | b | c)
    - [a-c0-2] is the same as: (a | b | c | 0 | 1 | 2)

Generative vs Recognize

- Regular expressions are a mechanism to generate the strings of a language
- Finite automata (FAs) are a mechanism to recognize if a string belongs to the language
- Standard approach
  - Use regular expressions when defining the language (regular languages), usually the lexemes of a programming language
  - Translation of the regular expressions to FAs to implement the lexical analysis

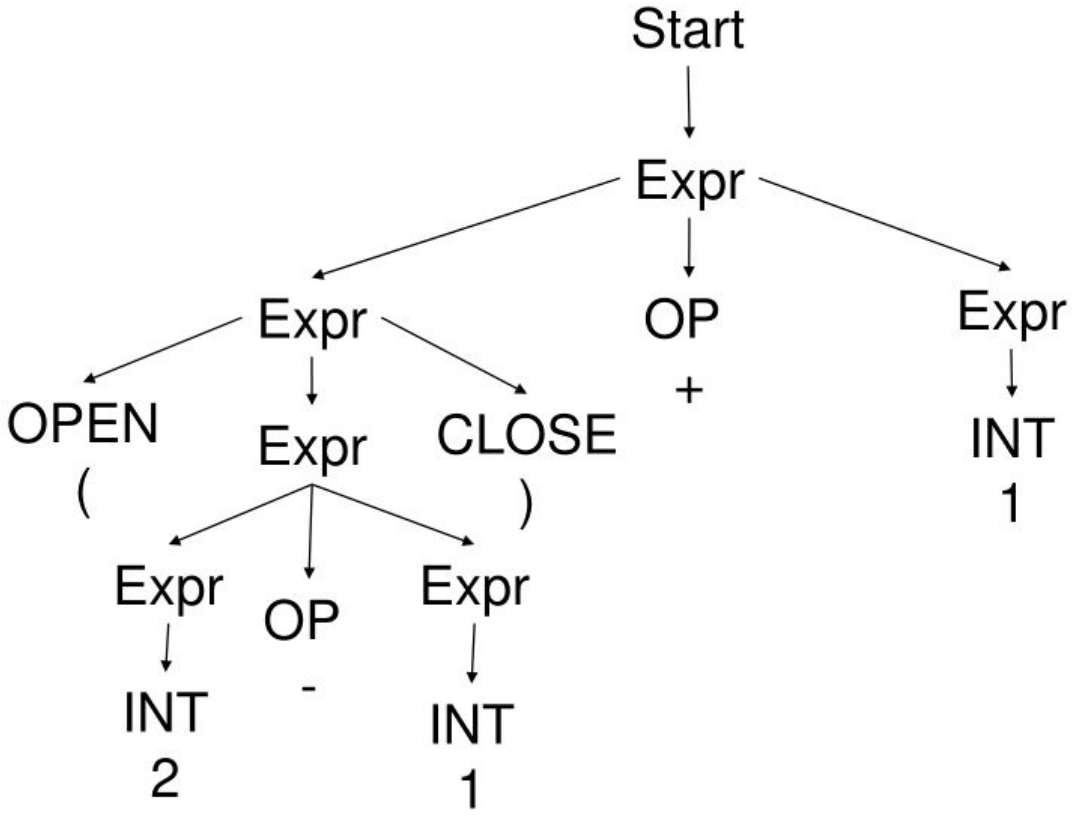
Conversion Rules



Syntax Tree for (2-1)+1

OP = +|-|*|/
INT = [0-9] [0-9]*
OPEN = (
CLOSE =)

1) Start → Expr
2) Expr → Expr OP Expr
3) Expr → INT
4) Expr → OPEN Expr CLOSE



Eliminating Ambiguity

- Solution: modify grammar
- All operators with left-associative

Original Grammar

Start → Expr

Expr → Expr OP Expr

Expr → INT

Expr → OPEN Expr CLOSE

Modified Grammar

Start → Expr

Expr → Expr OP Expr'

Expr → INT

Expr → OPEN Expr CLOSE

Expr' → OPEN Expr CLOSE

Expr' → INT

Solution to Precedence

Original Grammar

OP = + | - | * | /
INT = [0-9] [0-9]*
OPEN = (
CLOSE =)

Start → Expr
Expr → Expr OP Expr'
Expr → Expr'
Expr' → OPEN Expr CLOSE
Expr' → INT

Modified Grammar

OP1 = + | -
OP2 = * | /
INT = [0-9] [0-9]*
OPEN = (
CLOSE =)

Start → Expr
Expr → Expr OP1 Term
Expr → Term
Term → Term OP2 Final
Term → Final
Final→ INT
Final→ OPEN Expr CLOSE

Modified Grammar

Start → Stat
Stat → IF Expr THEN Stat ELSE Stat
Stat → IF Expr THEN Stat
Stat → ...

- Basic Idea: control when an IF without ELSE can occur
  - At the top level of the statements
  - Or as te last in a sequence of statements if then else if then ...

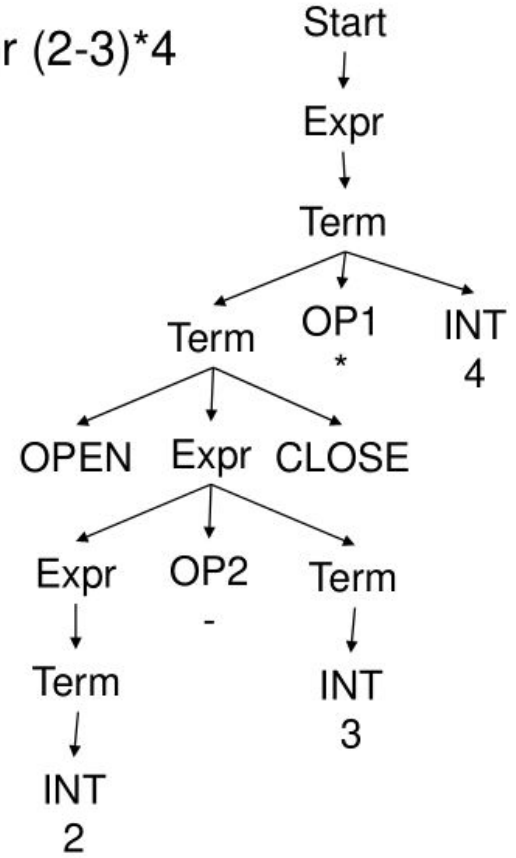
Goal → Stat
Stat → WithElse
Stat → LastElse

WithElse → IF Expr THEN WithElse ELSE WithElse
WithElse → ...

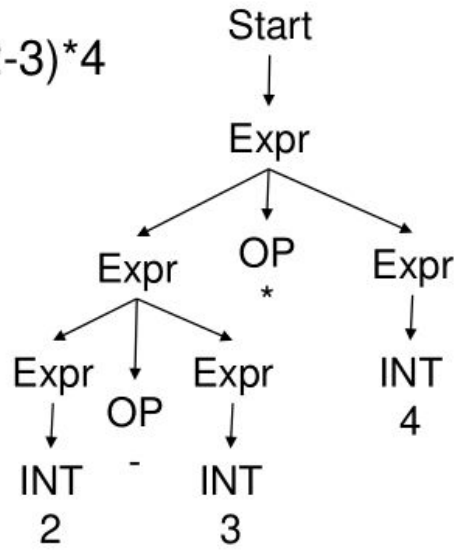
LastElse → IF Expr THEN Stat
LastElse → IF Expr THEN WithElse ELSE LastElse

Example

CST for (2-3)*4



AST for (2-3)*4



- Close to intuitive grammar
- Eliminates superfluous tokens
  - OPEN, CLOSE, etc.

31

First() Example

- First(Term')?

Grammar

Term' → * INT Term'

Term' → / INT Term'

Term' → ε

Solution

First(Term') = {*,/, ε}

First(* INT Term') = {*}

First(/ INT Term') = {/}

First(*) = {*}

First(/) = {/}

Example Follow()

- Grammar examples:
  - $S \rightarrow X\ \$$
    $X \rightarrow a$
    $X \rightarrow a\ b$
    - Follow(S) = { $ }
    - Follow(X) = { $ }
  - $S \rightarrow X\ \$$
    $X \rightarrow$ "(" X ")"
    $X \rightarrow \varepsilon$
    - Follow(S) = { $ }
    - Follow(X) = { ")", $ }

- Rules for Follow()
  - $\$ \in$ Follow(S), where S is the start symbol
  - If $A \rightarrow \alpha B \beta$ is a production then
    $\text{First}(\beta) \subseteq \text{Follow}(B)$
  - If $A \rightarrow \alpha B$ is a production then
    $\text{Follow}(A) \subseteq \text{Follow}(B)$
  - If $A \rightarrow \alpha B \beta$ is a production and β derives ε then
    $\text{Follow}(A) \subseteq \text{Follow}(B)$

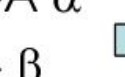
First() Set

- If two or more different productions for the same non-terminal symbol have First sets with common terminal symbols then:
  - The grammar cannot be analysed with a predictive LL(1) parser without backtracking
    - Example:
    - $S \rightarrow X\ \$$
    - $X \rightarrow a$
    - $X \rightarrow a\ b$

    - $First(X \rightarrow a) = \{ a \}$
    - $First(X \rightarrow a\ b) = \{ a \}$
    - **Which production to choose when the currrent symbol is a?**

Eliminate Left Recursion

- Productions for substitutions
  - $A \rightarrow A\,\alpha$
  - $A \rightarrow \beta$



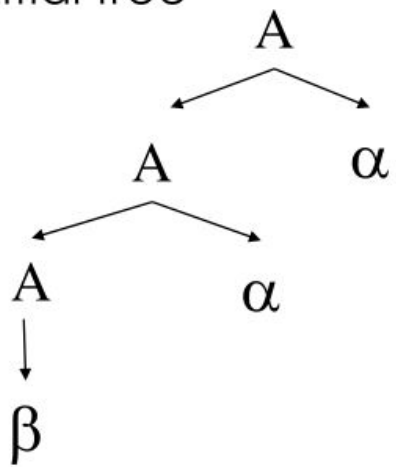
$A \rightarrow \beta\,R$

$R \rightarrow \alpha\,R$

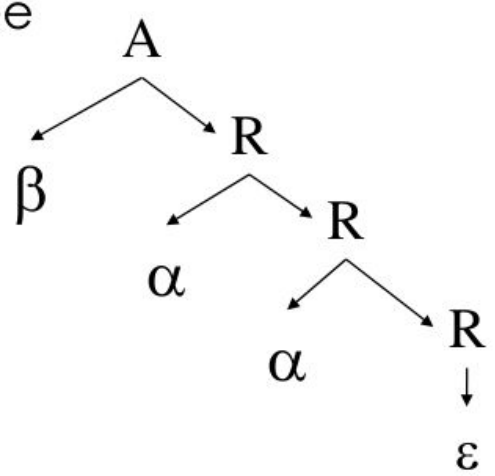
$R \rightarrow \varepsilon$

R is a new non-termnal symbol

Initial tree



New tree



Classify a Grammar as LL(1)

Grammar:

$Z \rightarrow$ "d"

$Z \rightarrow X\ Y\ Z$

$Y \rightarrow \varepsilon$

$Y \rightarrow$ "c"

$X \rightarrow Y$

$X \rightarrow$ "a"

- Put production $X \rightarrow \gamma$ in row X, column T, ① for each $T \in First(\gamma)$
- If γ can derive ε then put production $X \rightarrow \gamma$ in row X, column T, for each $T \in Follow(X)$ ②

Non-terminals	Terminals		
	"d"	"c"	"a"
Z	$Z \rightarrow X\ Y\ Z$ ① $Z \rightarrow$ "d"	$Z \rightarrow X\ Y\ Z$ ①	$Z \rightarrow X\ Y\ Z$ ①
Y	$Y \rightarrow \varepsilon$ ②	$Y \rightarrow \varepsilon$ ② $Y \rightarrow$ "c" ①	$Y \rightarrow \varepsilon$ ②
X	$X \rightarrow Y$ ①②	$X \rightarrow Y$ ①②	$X \rightarrow Y$ ② $X \rightarrow$ "a" ①

Terminology

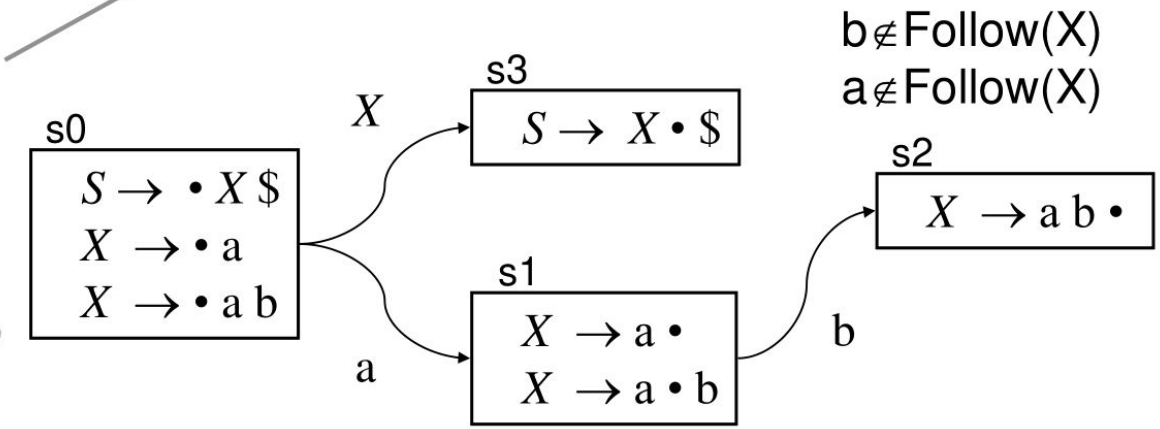
- LL(k)
  - *Top-down*, predictive
  - Leftmost derivation from top to bottom
- LR(k)
  - *Bottom-up*, *shift-reduce*
  - Rightmost derivation from bottom to top

$S \rightarrow X\,\$$ (1)
$X \rightarrow a$ (2)
$X \rightarrow a\,b$ (3)

LR(0)
New parser table

		ACTION		Goto
State	**a**	**b**	**$**	**X**
s0	shift to s1	error	error	goto s3
s1		shift to s2	reduce(2)	
s2			reduce(3)	
s3	error	error	accept	

b never follows X in the derivations: resolve conflict *shift/reduce* with *shift*



Building the parser table

- For each state
  - Transition using a terminal symbol is a shift to the destination state (*shift to sn*)
  - Transition using a non-terminal state is a goto to the destination state (*goto sn*)
  - If there exists an item $\mathbf{A \rightarrow \alpha \bullet}$ in the state do a reduction with that production for all the terminals (*reduce k*)
  - If there exists an item $\mathbf{S \rightarrow X \bullet \$}$ in the state then place accept state for terminal $

High-Level Intermediate Representation

- Known as HLIR or HIR
- It preserves the structured control flow
- Useful for optimizations at the loop level
  - Loop Unrolling, Loop Fusion, etc.
- It preserves the structure at class level
- Useful for optimizations for object-oriented languages

Low-Level Intermediate Representation

- Known as LLIR or LIR
- From an abstract data model to a flat region memory space
- Eliminates the structured control flow
  - Control flow is now represented as low-level instructions (e.g., using conditional branches and jumps)
- Useful for low-level compilation tasks
  - Register Allocation
  - Selection of Instructions
  - Scheduling

Compiler Tasks

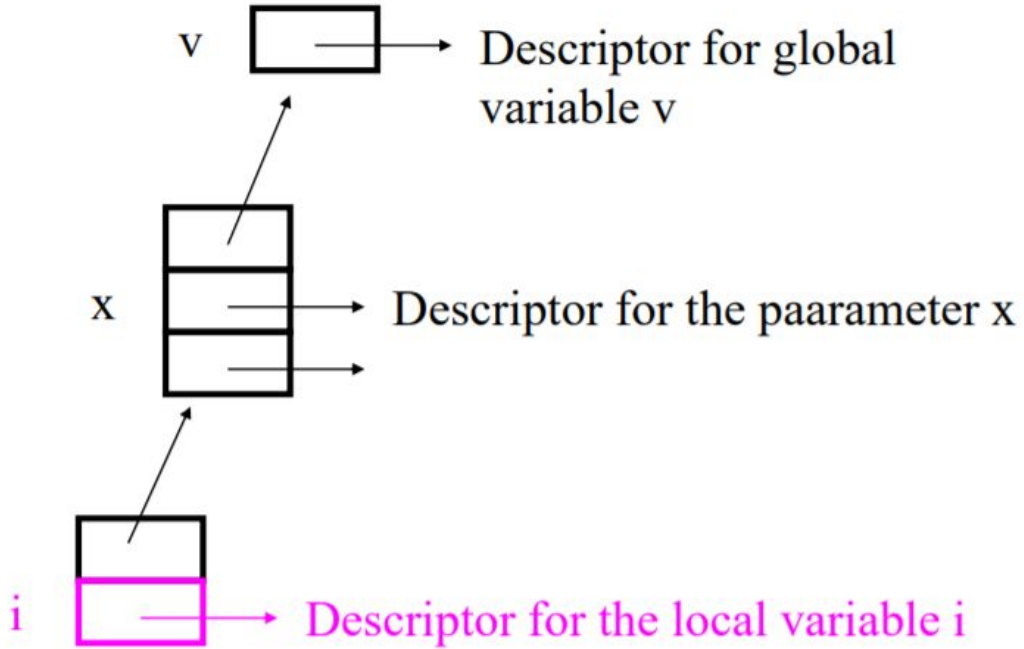
- Determine format of the structures in the memory
  - Format of the arrays and objects in the memory
  - Format of the call stack in the memory
- Generate code
  - To read values (parameters, elements of the arrays, fields, etc.)
  - To eva,uate expressions and compute new values
  - To write values
  - For control structures
- Enumerate functions and builds the symbol table
  - Invocation of a function accesses to the entry of the correspondent table of functions
- Generate code for the functions
  - Local variables and access to parameters
  - Invocations of functions

Symbol Tables

- During the creation/translation of syntax trees
- During the transation of syntax trees to intermediate representation
  - Symbol tables map identifiers (strings) to descriptors (information about the identifier)
  - Basic operatio: Lookup
    - Given a string, find its descriptor
    - Typical implementation: hash table
- Example:
  - Given the name of a variable find its descriptor (local, parameter, global)

Lookup i in an Example

- v[i] = v[i]+x;
- First it searches in the TS of the local variables
- If don't find it then goes up and searches in the next hierarchy level



Descriptors

- What they contain?
- Information used to perform semantic analysis and to generate code
  - Local descirptors: name, type, offset in the stack
  - Descriptors of functions
    - Signature (type of return, parameters)
    - Reference to the local symbol table
    - Reference to the code (IR) of the function

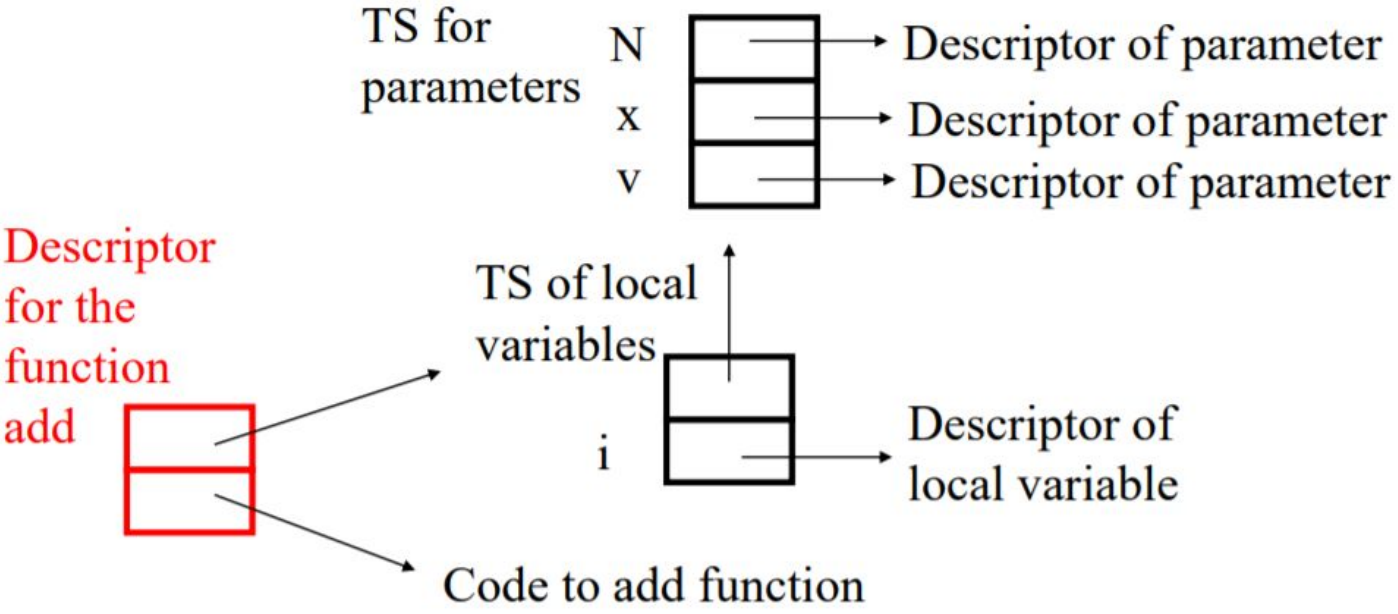
Parameters, Local, and Descriptors of Types

- Parameters and Locals refer to type descriptors
  - Descriptor of base type: int, boolean, etc.
  - Descriptor of the array type: contains reference to the descriptor of the type for the array elements
  - Descriptor of structure, etc.

Descriptor of Functions

- Contain reference for the code (IR) of the function
- Contain reference to the local symbol table (local variables of the function)
  - Note that the existence of more than one local scope implies the existence of a subhierarchy of local symbol tables
- In the hierarchy of the symbol tables the symbol table for the parameters is parent of the symbol table for the local variables

Descriptor of the Function add



Nested Scopes

- Various forms of nesting
  - Symbol Table of the functions nested in the symbol table of the globals
  - Symbol Table of the locals nested in symbol table function
- Nesting solves ambiguity in possible conflicts
  - Same name used for a global and a local variable
  - Name refers a local variable in a function

High-Level Code Representation

- Basic idea
  - Moving towards the target language (e.g., assembly)
  - Preserve control structure
    - Format of objects
    - Structured control flow
    - Distinction between parameters, local variables, fileds, etc.
  - High-level of abstraction of the assembly language
    - load and store nodes
    - Access to abstract local storage, parameters and fields, and not memory positions directly

Representation of Expressions

- Expression trees represent the expressions
  - Internal nodes – operations such as +, -
  - Leafs – Load nodes represent access to variables
- Load nodes
  - **ldl** to access local variables – local descriptors
  - **ldp** to access parameters – parameter descriptors
  - **lda** to access array elements
    - Expression tree for the value
    - Expression tree for the index
  - For loads of class attributes, of fields of structs…

Representing Assignment Statements

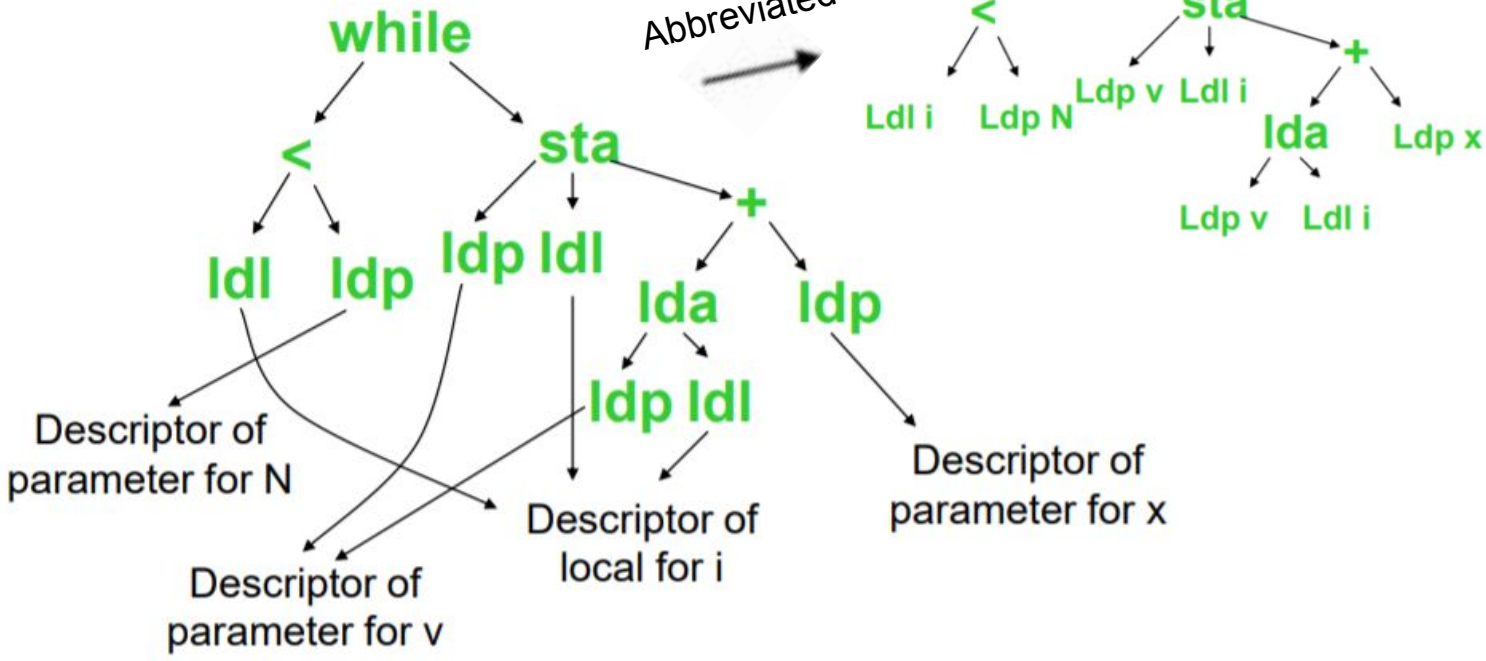
- Store Nodes
  - **stl** for *stores* of local variables
    - Local descriptor
    - Expression tree for the value to store
  - **sta** for *stores* in array elements
    - Expression tree for the array
    - Expression tree for the index
    - Expression tree for the value to store
  - For stores in class attributes, in fields of structs…

Orientation

- Intermediate representations
  - Moving in the direction of the target language (e.g., machine language)
  - Support for compiler analysis and transformations
- High-Level IR (*intermediate representation*)
  - Preserves the structure of objects, arrays, control flow,…
  - Symbol Tables
  - Descriptors

Example

```
while (i < N)
    v[i] = v[i]+x;
```



Inference of types for addition operations

- Some languages let add floats, ints, doubles
- What are the problems?
  - Type of the result of the operation
  - Conversion of the operands of the operation
- Standard rules are usually applied:
  - If addition of an **int** with a **float**
    - Convert the **int** to **float**, add the two **floats**, and the result is a **float**
  - If addition of a **float** with a **double**
    - convert **float** to **double**, add the two **doubles**, result is a **double**
- Basic principle: hierarchy of types for numbers (int, then float, then double)
- All the "forced" conversions are done in bottom-up mode in the hierarchy
  - E.g., int to float; float to double;
- Result has the type of the operand with type in the highest level of the hierarchy:
  - int + float $\rightarrow$ float,
  - int + double $\rightarrow$ double,
  - float + double $\rightarrow$ double

Verification for loads, stores, etc.

- What does the compiler have?
  - Name of variable
- What does it do?
  - *Lookup* name of variable:
    - Verifies if it is in the symbol table of locals, reference to a local descriptor
    - Verifies if it is in the symbol table of parameters, reference to a parameter descriptor
    - Verifies if it is in the symbol table of globals, reference to a global descriptor
    - If a descriptor was not found then semantic error (the variable was not declared)

Conversion to Low-Level IR

- Convert structured control flow into control flow based on jumps (non-structured)
  - Conditional and inconditional branches
- Convert structured memory model into flat memory model
  - Flat addressing for variables
  - Flat addressing for arrays
- Continues independent of the machine language, but:
  - Movement to very close to the machine, to a standard machine models (flat space address, jumps)

Inference of types for addition operations

- Some languages let add floats, ints, doubles
- What are the problems?
  - Type of the result of the operation
  - Conversion of the operands of the operation
- Standard rules are usually applied:
  - If addition of an **int** with a **float**
    - Convert the **int** to **float**, add the two **floats**, and the result is a **float**
  - If addition of a **float** with a **double**
    - convert **float** to **double**, add the two **doubles**, result is a **double**
- Basic principle: hierarchy of types for numbers (int, then float, then double)
- All the "forced" conversions are done in bottom-up mode in the hierarchy
  - E.g., int to float; float to double;
- Result has the type of the operand with type in the highest level of the hierarchy:
  - int + float $\rightarrow$ float,
  - int + double $\rightarrow$ double,
  - float + double $\rightarrow$ double

Verification for loads, stores, etc.

- What does the compiler have?
  - Name of variable
- What does it do?
  - *Lookup* name of variable:
    - Verifies if it is in the symbol table of locals, reference to a local descriptor
    - Verifies if it is in the symbol table of parameters, reference to a parameter descriptor
    - Verifies if it is in the symbol table of globals, reference to a global descriptor
    - If a descriptor was not found then semantic error (the variable was not declared)

Conversion to Low-Level IR

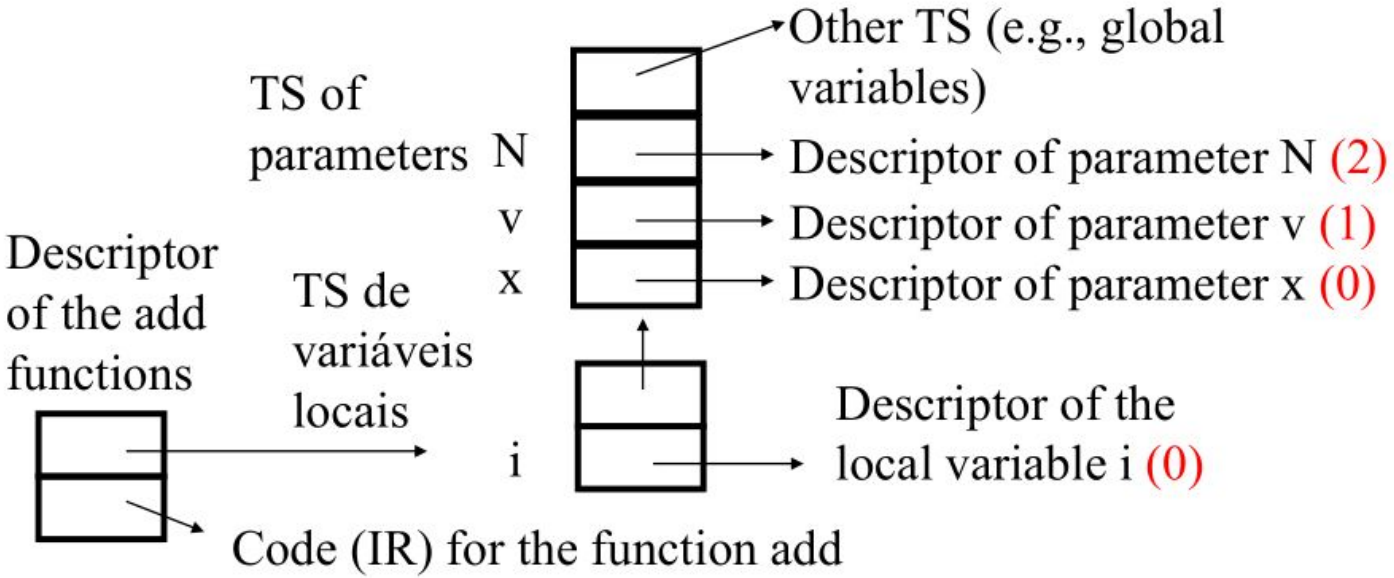
- Convert structured control flow into control flow based on jumps (non-structured)
  - Conditional and inconditional branches
- Convert structured memory model into flat memory model
  - Flat addressing for variables
  - Flat addressing for arrays
- Continues independent of the machine language, but:
  - Movement to very close to the machine, to a standard machine models (flat space address, jumps)

Management of the Stack (remember)

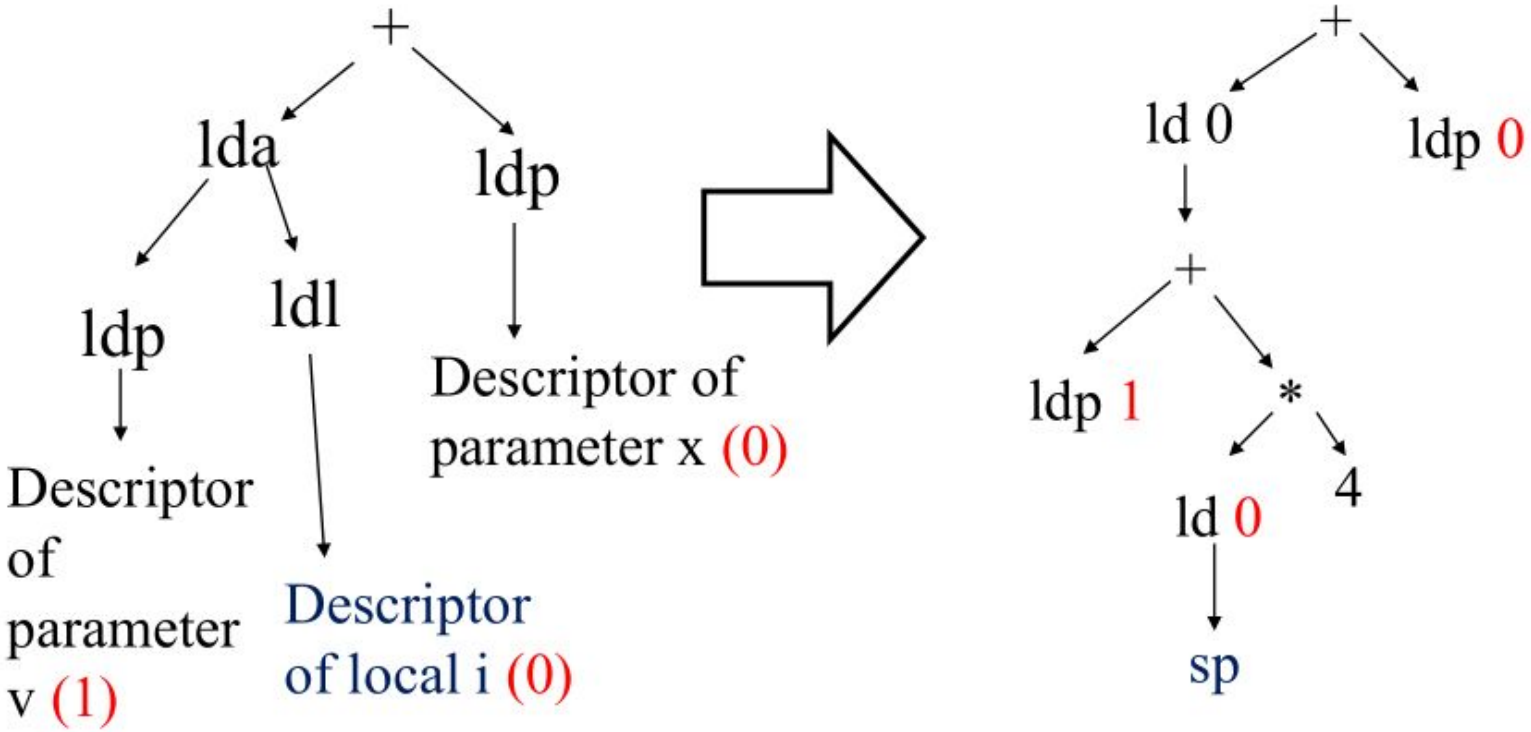
- Determine stack frame size
  - Allocate when the execution enters in the function
  - Free before returning from the function
  - Store all local variables
  - Additional space for parameters (when these surpass the number of registers assigned by convention for the arguments of the function)
- Define offsets for the local variables and parameters stored in the stack
  - Stored in the symbol tables (descriptors) of the locals and of the parameters
  - Continues to use ldp nodes to access to parameters

Elimination of ldl Nodes

- Use of *offsets* in the symbol table of locals and sp
- Replace *ldl* nodes by *ld* nodes
- Example of offsets for locals and parameters



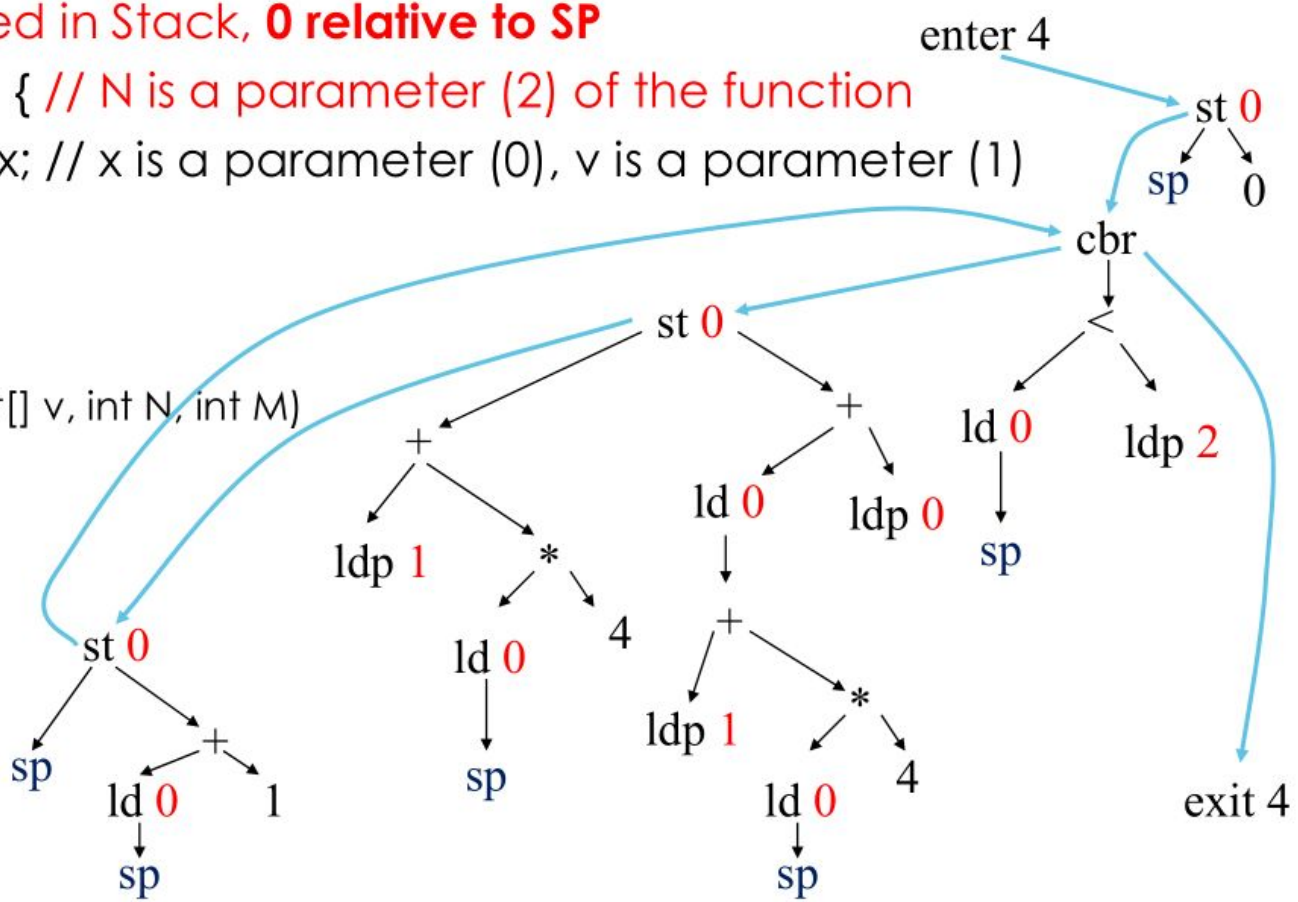
Example: v[i]+x


+
lda
ldp
ldp
ldl
Descriptor of parameter x (0)
Descriptor of parameter v (1)
Descriptor of local i (0)
+
ld 0
ldp 0
+
ldp 1
*
4
ld 0
sp

Example

```
i=0; // i stored in Stack, 0 relative to SP
while (i < N) { // N is a parameter (2) of the function
   v[i] = v[i]+x; // x is a parameter (0), v is a parameter (1)
   i++;
}
```

Void f(int x, int[] v, int N, int M)
X: $a0
V: $a1
N: $a2



HIR
stl i
0

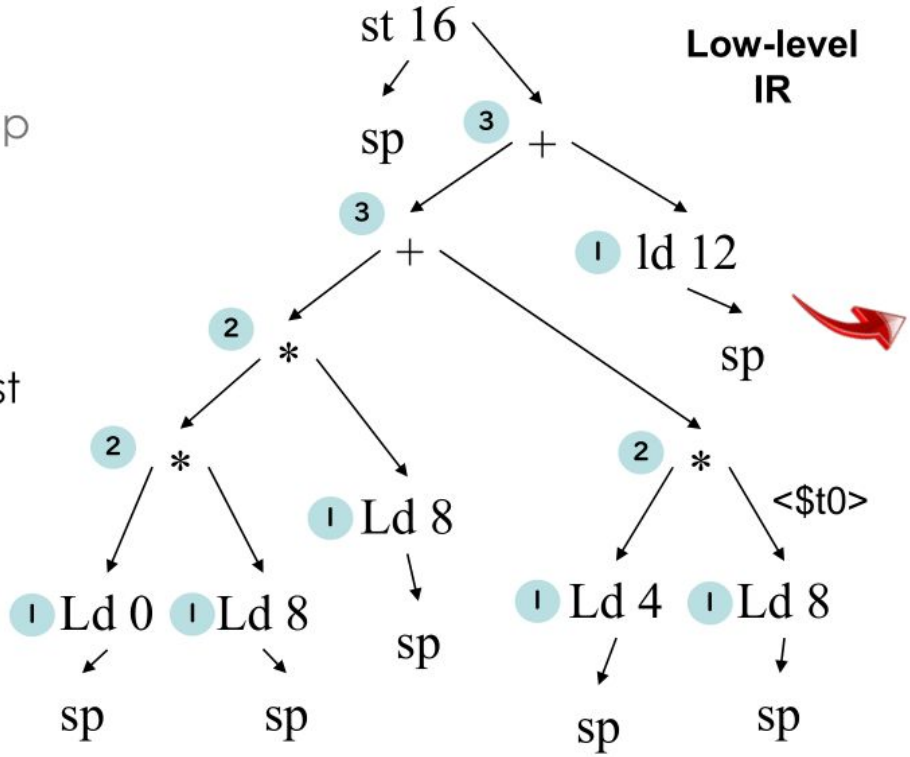
Summary of the Low-Level IR

- Array accesses translated to *ld* or *st* nodes
  - Address is the base address of the array + index * element size
- Local accesses translated to *ld* or *st* nodes
  - Address in *sp*, offset is local offset
- Access to parameters is translated to:
  - Instructions *lpd* – specify number of the parameter
- Nodes Enter and Exit of a function identify stack size used by the function

Code Generation: Sethi-Ullman Algorithm

y=a*x*x+b*x+c;

1. Label nodes bottom-up according to the register needs
2. Traverse recursively tree top-down: first child that requires most registers (left child in the case of equal no. of registers) and emit instruction per node



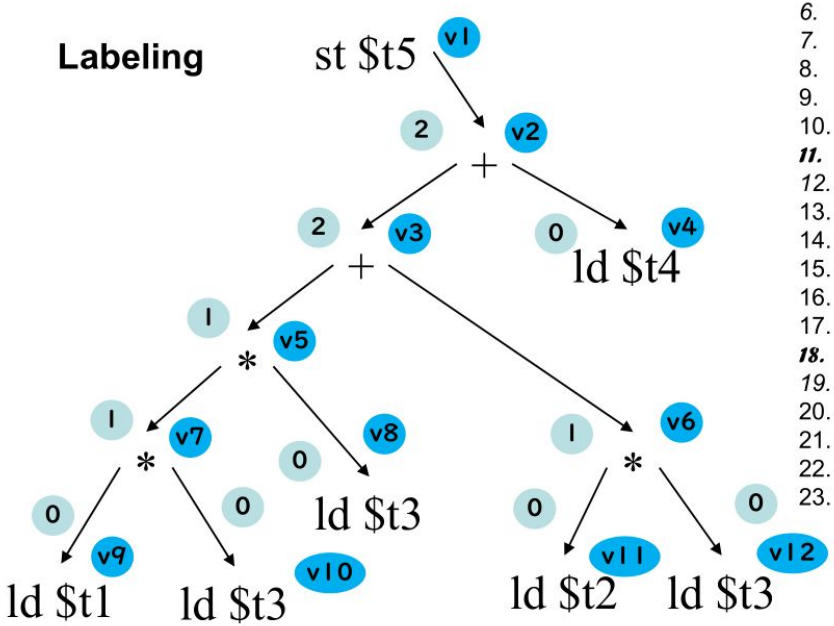
Relative position to $sp

a: 0
b: 4
x: 8
c: 12
y: 16

```
lw $t1, 0($sp)
lw $t2, 8($sp)
mul $t1, $t2, $t1
lw $t2, 8($sp)
mult $t1, $t1, $t2
lw $t2, 4($sp)
lw $t3, 8($sp)
mult $t2, $t2, $t3
add $t1, $t1, $t2
lw $t2, 12($sp)
add $t1, $t1, $t2
sw  $t1, 16($sp)
```

Code Generation: exercise 1

y=a*x*x+b*x+c;




```
1.  generate(T, R') {
2.    if T is an end_node emit("op top(), L, R");
3.    elsif T is an internal node with children l and r {
4.      if regs(r) == 0 {
5.        generate(l);
6.        emit("op top(), top(), r");
7.      }
8.     if regs(l) >= regs(r) {
9.        generate(l);
10.       R = pop();
11.       generate(r);
12.       emit("op R, R, top()");
13.       push(R);
14.    } else {  // regs(l) < regs(r)
15.       swap the top 2 stack elements;
16.       generate(r);
17.       R = pop();
18.       generate(l);
19.       emit("op top(), top(), R");
20.       push(R);
21.       swap the top 2 stack elements;
22.    }
23.}
```

Stack:

$t6	$t7

1. generate(v2 , $t5)
   5. generate(v3);
      9. generate(v5);
         5. generate(v7);
            2. emit("mul $t6, $t1, $t3");
         6. emit("mul $t6, $t6, $t3");
      10. R = pop();

$t7

      11. generate(v6);
         2. emit("mul $t7, $t2, $t3);
      12. emit("add $t7, $t7, $t6); // R is $t6
      13. push $t6;

$t6	$t7

   6. emit("add $t5, $t6, $t4");

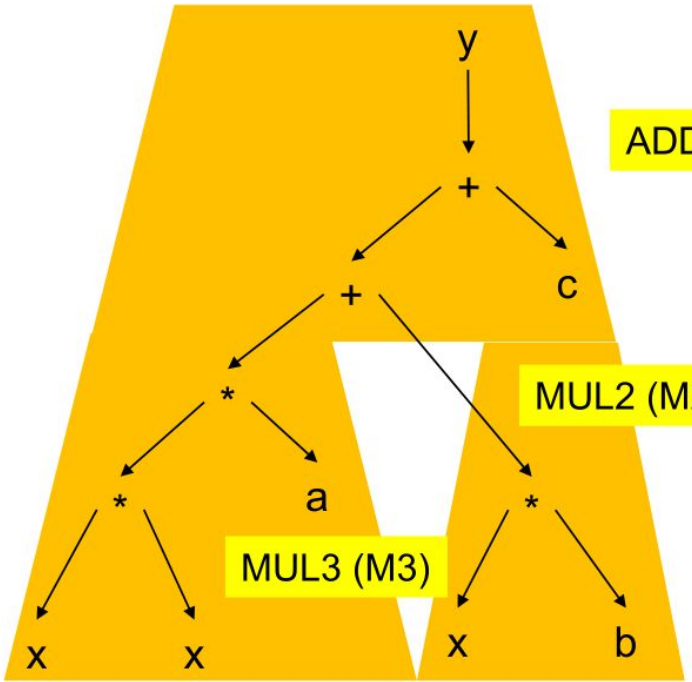
⇒ 2 registers for auxiliary values: we assume $t6 and $t7

Instruction Selection: Maximal Munch

- A simple algorithm that finds an optimal tiling: Maximal Munch (greedy, top-down pattern match)
  - Starting at the root of the tree
  - Find the largest tile that fits (the tile with most nodes)
  - Cover the root node and the possible nodes with this tile
  - Repeat the algorithm for each subtree of the tile until all the tree is tiled
  - For each tile generates the instructions of that tile
    - code generation is performed in reverse order, least instruction firts

Instruction Selection: Maximal Munch

- y=a*x*x+b*x+c;



MUL2 (M2)

MUL3 (M3)

ADD3 (A3)

Cost = 4+7+2 = 13

Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

Instruction Selection: Maximal Munch

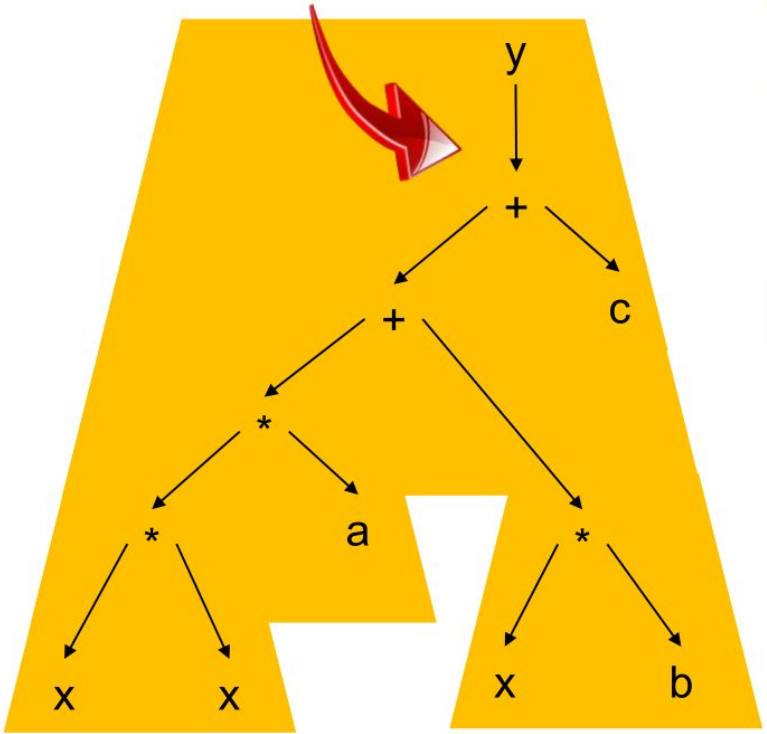
- Maximal Munch does not give the tiling with the minimum cost:
  - It decides locally about the largest pattern to fit, this might prevent the tiling of large patterns in the subtrees
- Gives optimal tiling, i.e., no adjacent tiles can form a tile with lower cost
- One possible solution to achieve minimum cost (i.e., tiling with minimum global cost)
  - Dynamic Programming

Instruction Selection: Dynamic Programming

- Bottom-Up Exhaustive Cataloging of Optimum Solutions
- Optimum Solution of Node Based on Optimum Solution of Subnodes
- Delivers the Global Optimum
- Very Efficient
  - Used in, e.g., Twig, and BURG

Dynamic Programming Example

- y=a*x*x+b*x+c;



Tiles not considered as they use non-optimal subtree tiles:

M2-M2-A2-A2, M2	A2	1	13	14
M2-M2-A3, M2	A3	2	12	14
M3-A2-A2, M2	A2	1	12	13
M2-MA-A2, M2	A2	1	12	13
M2-M2-A2-A2, M2	A2	1	13	14

Instruction	function	cost
ADD2 (A2)	a ←b+c	1
ADD3 (A3)	a ←b+c+d	2
MUL2 (M2)	a ←b*c	4
MUL3 (M3)	a ←b*c*d	7
MADD (MA)	a ←b*c+d	4

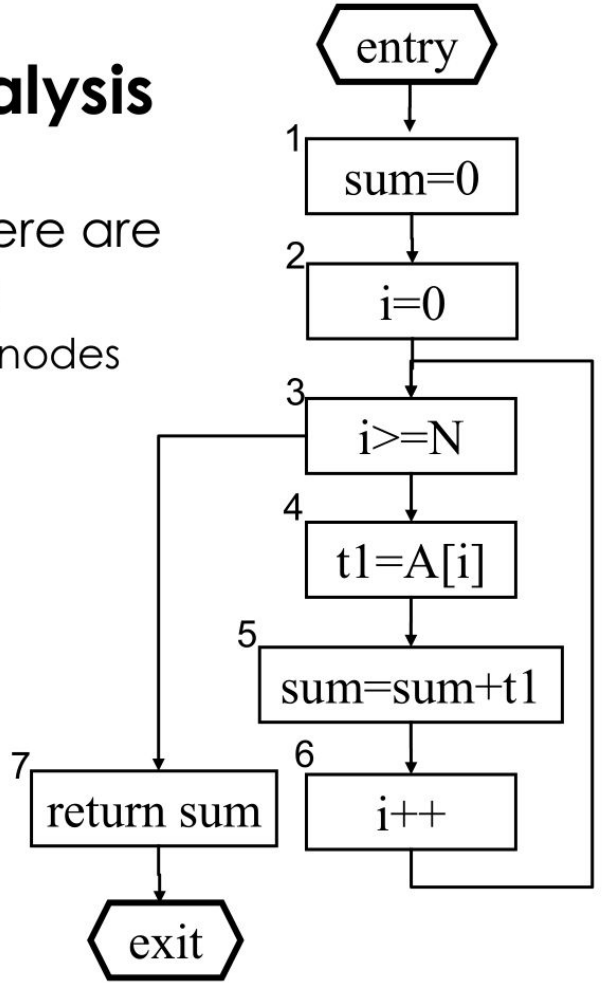
- The corresponding Instruction Tree Patterns are the following:

Instruction	Effect	IR Tree Pattern
—	r_i	TEMP r_i
add	$r_i \leftarrow r_j + r_k$	+
mul	$r_i \leftarrow r_j * r_k$	*
sub	$r_i \leftarrow r_j - r_k$	−
div	$r_i \leftarrow r_j / r_k$	/
addi	$r_i \leftarrow r_j + c$	+ (CONST c) (CONST c)
		CONST c
subi	$r_i \leftarrow r_j - c$	− CONST c

Instruction	Effect	IR Tree Pattern
load	$r_i \leftarrow M[r_j + c]$	MEM + CONST c ; MEM + CONST c ; MEM CONST c ; MEM
store	$M[r_j + c] \leftarrow r_i$	MOVE MEM + CONST c ; MOVE MEM + CONST c ; MOVE MEM CONST c ; MOVE MEM
movem	$M[r_j] \leftarrow M[r_i]$	MOVE MEM MEM

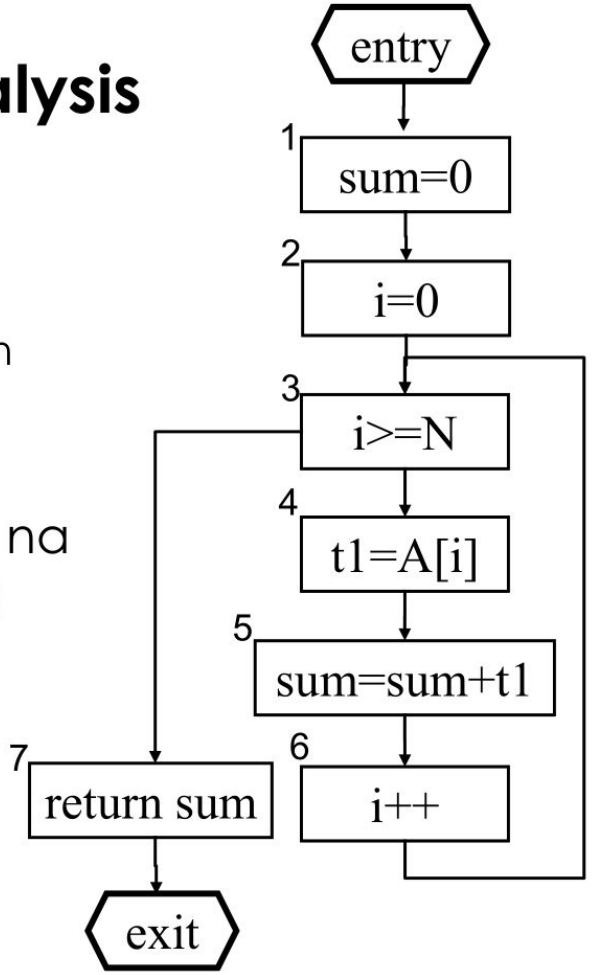
Dataflow Analysis

- Given a node n in a flow graph, there are
  - Out-edges that lead to sucessor nodes
  - In-edges that come from predecessor nodes
- Sets:
  - ***succ[n]*** is the set of sucessors
    - ***succ[3]*** = {4, 7}
  - ***pred[n]*** is the set of predecessors
    - ***pred[3]*** = {2, 6}



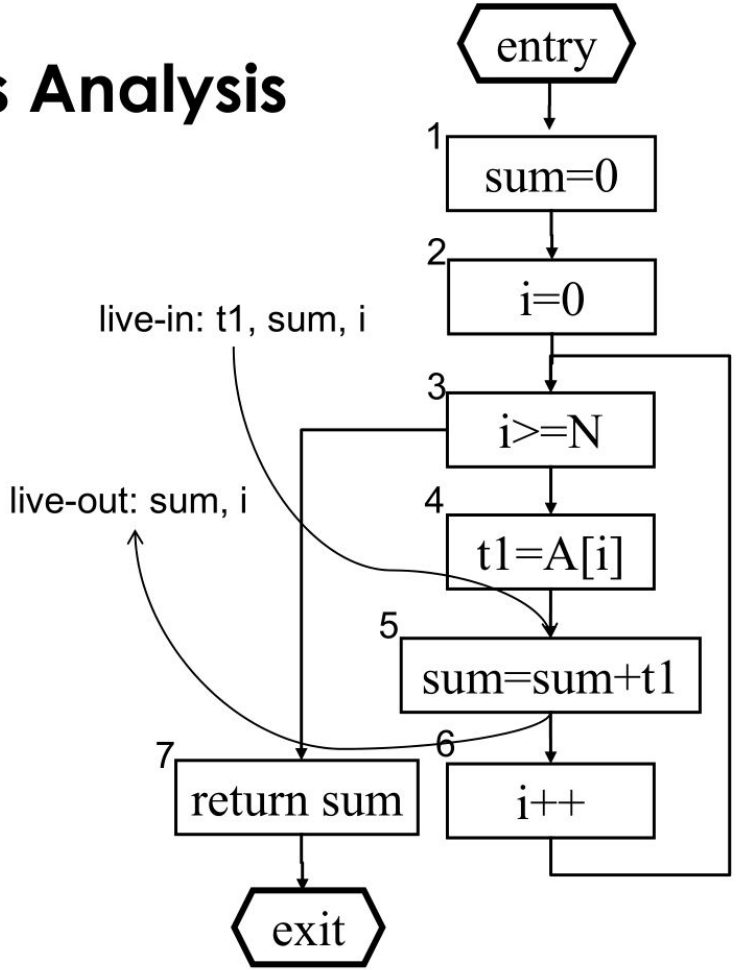
Liveness Analysis

- An assignment to a variable or temporary defines the variable
  - ***def[n]*** is the set of variables defined in n
  - ***def[5]*** = {sum}
- Na occurence of a variable or temporary in the right-hand side of na assignment (or in other expressions) uses the variable
  - ***use[n]*** is the set variables used in n
  - ***use[5]*** = {sum, t1}



Liveness Analysis

- A variable is live on an edge if there is a forward path from that edge to a *use* that does not go through any *def* of the same variable
- A variable is *live-in* at a node if it is live in any of the *in-edges* of the node
- A variable is *live-out* at a node if it is live on any of the *out-edges* of the node



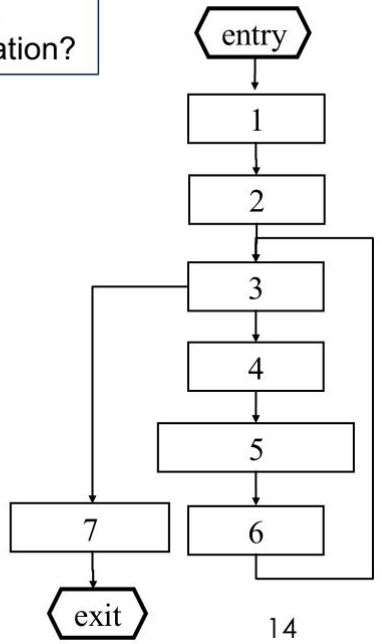
Liveness Analysis

- Computation of liveness analysis (forward):

$$in[n] \leftarrow use[n] \cup (out[n] - def[n])$$
$$out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$$

last iteration?

				1st iteration		2nd iteration		3rd iteration		4th iteration		6th iteration	
node	use	def	succ	in	out	in	out	in	out	in	out	in	out
1		s	2										s
2		i	3				i		i		s,i	s	s,i
3	i		4,7	i		i	s,i	s,i	s,i	s,i	s,i	s,i	s,i
4	i	t	5	i		i	s,t	s,i	s,t	s,i	s,t,i	s,i	s,t,i
5	s,t	s	6	s,t		s,t	i	s,t,i	i	s,t,i	i	s,t,i	s,i
6	i	i	3	i	i	i	i	i	s,i	s,i	s,i	s,i	s,i
7	s			s		s		s		s		s	



14

Liveness Analysis

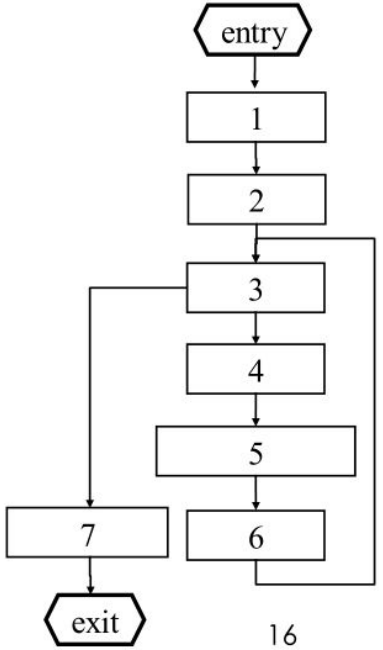
- Computation of liveness analysis (backward):

$$out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$$

$$in[n] \leftarrow use[n] \cup (out[n] - def[n])$$

beginning				1st iteration		2nd iteration		3rd iteration	
node	use	def	succ	out	in	out	in	out	in
7	s				s		s		s
6	i	i	3		i	s,i	s,i	s,i	s,i
5	s,t	s	6	i	s,t,i	s,i	s,t,i	s,i	s,t,i
4	i	t	5	s,t,i	s,i	s,t,i	s,i	s,t,i	s,i
3	i		4,7	s,i	s,i	s,i	s,i	s,i	s,i
2		i	3	s,i	s	s,i	s	s,i	s
1		s	2	s		s		s	

3 iterations



16

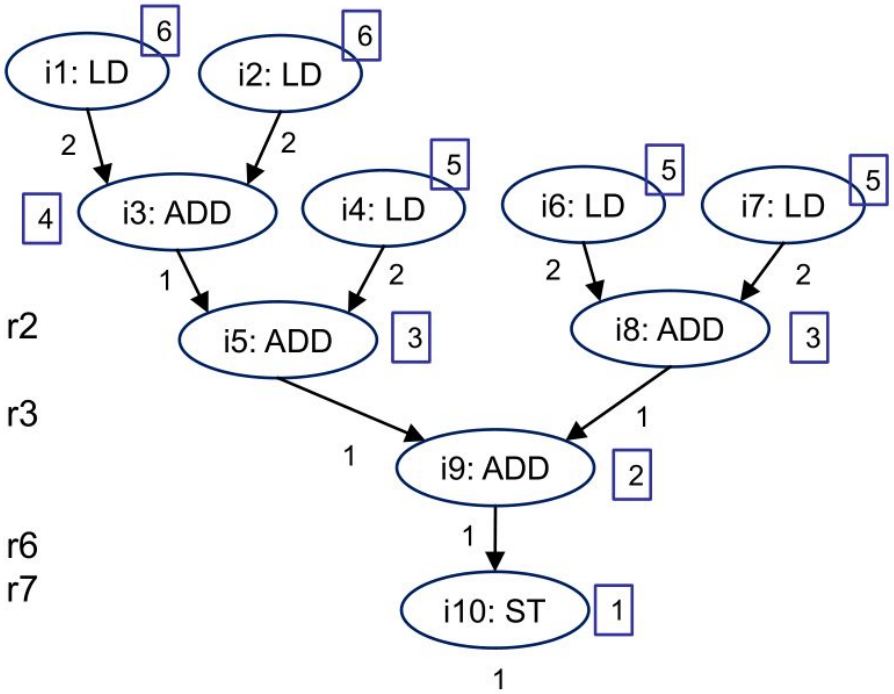
List Scheduling: Example E2

f = (a + b) + c + (d + e)

Assembly Code

i1: LD r1, a
i2: LD r2, b
i3: ADD r5, r1, r2
i4: LD r3, c
i5: ADD r8, r5, r3
i6: LD r4, d
i7: LD r6, e
i8: ADD r7, r4, r6
i9: ADD r9, r8, r7
i10: ST f, r9

Data-dependence graph (DDG)



Cycle: 7 Ready:
Active: 10

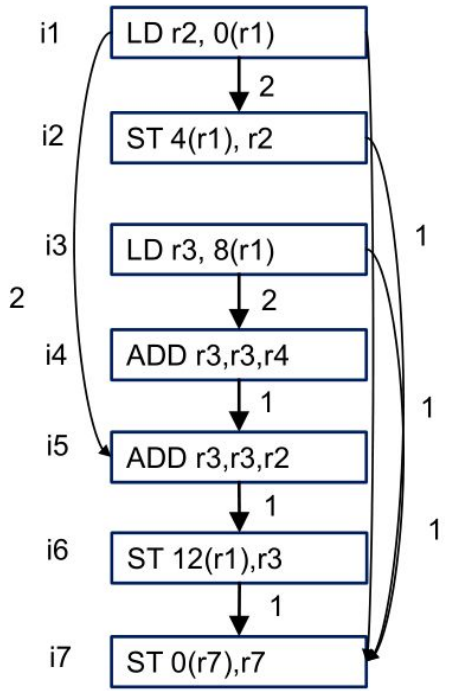
Cycle	MEM	MEM	ALU	ALU
1	1	2		
2	4	6		
3	7		3	
4			5	
5			8	
6			9	
7	10			

33

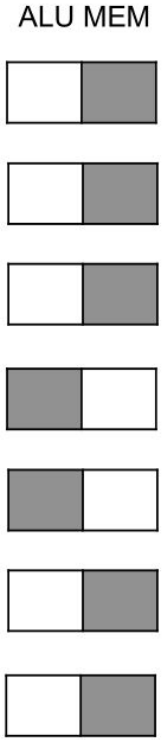
Example E3

- List scheduling result

Data-Dependences



Resource-reservation tables



Schedule

ALU	MEM
	LD r3, 8(r1)
	LD r2, 0(r1)
ADD r3, r3, r4	
ADD r3, r3, r2	ST 4(r1), r2
	ST 12(r1), r3
	ST 0(r7), r7

Resource-reservation table

ALU	MEM

Critical path: 6 clock cycles

Software Pipelining

- Important compiler optimization that overlaps (fully of partially) successive iterations (i.e., the subsequent iteration may start before the previous one is finished)
- This example shows, at source code level, the overlapping of 2 consecutive iterations

```
for(i=0;i<1000; i++)
   C[i]=A[i]+B[i];
```

Latency* = 1000*3 = 3000 cycles

```
t1 = A[0]; t2 = B[0];                          // prologue
for(i=1;i<1000; i++) {
   t3 = t1+t2; t1 = A[i]; t2 = B[i];           // kernel
   C[i-1] = t3;
}
t3 =  t1+t2;                                   // epilogue
C[999] = t3;                                   // epilogue
```

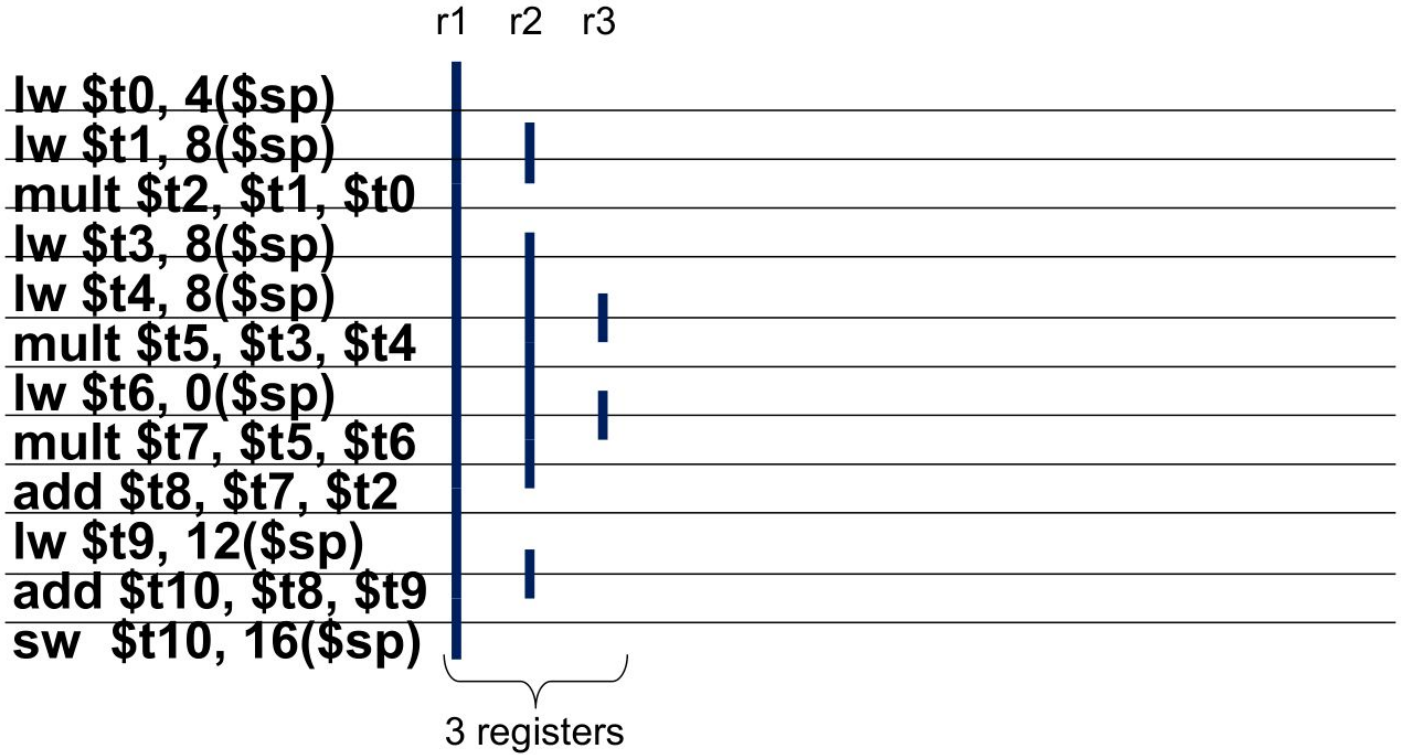
Latency** = 1+999*2+2 = 2001 cycles

*,** Rough estimations based on the high-level statements and considering 1 clock cycle per operation

** also considering that operations at the same line are executed in parallel

Register Allocation

- Let's try to reduce the number of registers t in the following MIPS code



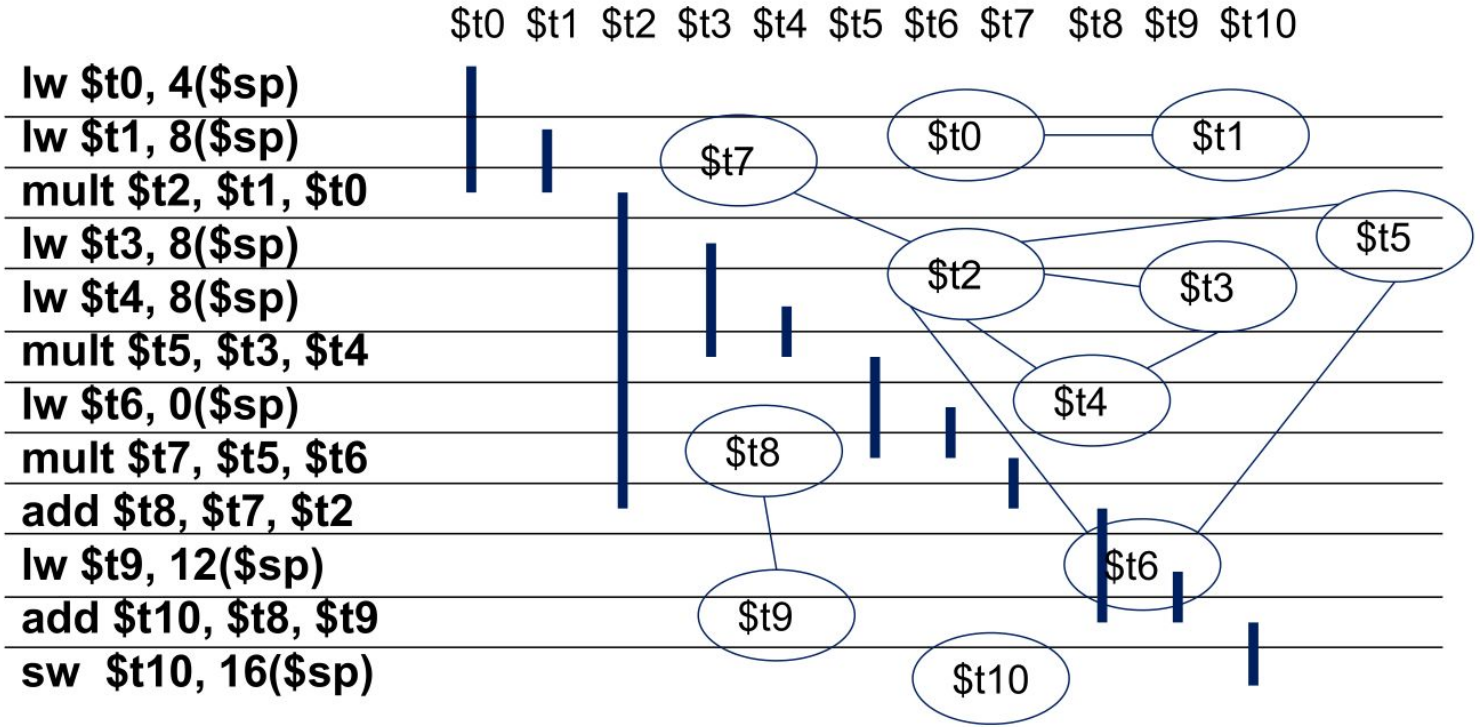
Register Allocation by Graph Coloring

- Graph Coloring
  - Calculate the live range for each variable
  - Construct the Register-Interference Graph* (there is interference when 2 variables have lifetimes with non-null intersection)
    - Edges represent interference
    - Nodes represent variables
  - Find the minimum colors or the k colors
  - Each color corresponds to a register
    - i.e., number of registers = number of colors

* Also known as Register-Conflict Graph

Register Allocation by Graph Coloring

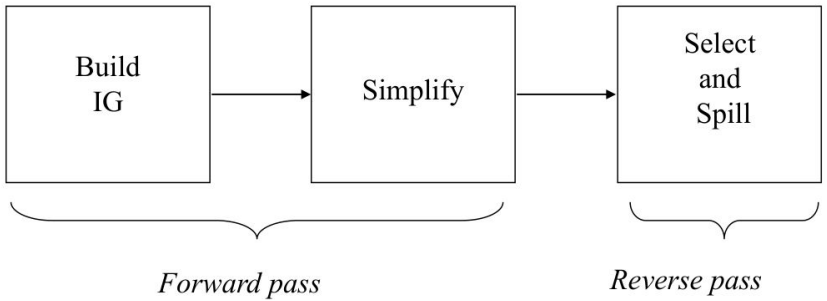
- Register-Interference Graph (IG)



- Kempe's algorithm [1879] for finding a K-coloring of a graph
- Step 1 (simplify): find a node n with degree(n)<k and cut it out of the graph (remember this node on a stack for later stages)

- Once a coloring is found for the simpler graph, we can always color the node we saved on the stack
- Step 2 (color): when the simplified subgraph has been colored, add back the node on the top of the stack and assign it a color not taken by one of the adjacent nodes

- Step 3 (spilling): once all nodes have K or more neighbors, pick a node for spilling
  - Storage on the stack



Spilling

- Consider: add t1, t2, t3
  - Suppose t3 is selected for spilling and assigned to stack location [8+$sp]
  - Invented new temporary t35 for just this instruction and rewrite:
    - **lw $t35, 8($sp)**; add t1, t2, t35
  - Advantage: t35 has a very short live range and is much less likely to interfere
  - Rerun the algorithm
    - fewer variables will spill

Spilling

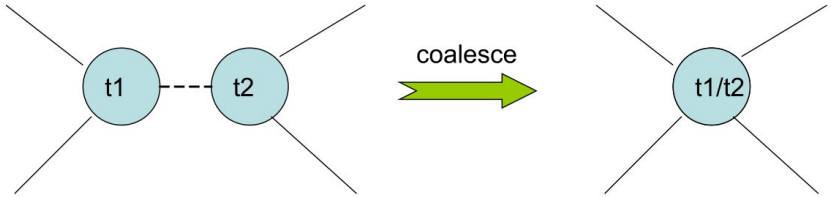
- Variables selected to Spill?
- The selection can be based on a number of properties:
  - frequencies of execution of uses/defs (based on the iteration count, profiling results)
  - number of uses/defs
  - number of adjacent nodes for the variable in the Interference Graph
  - Lifetime duration
  - etc.

Precolored Nodes

- Some variables are pre-assigned to registers
- Treat these registers as special temporaries; before beginning, add them to the graph with their colors
- Can't simplify a graph by removing a precolored node
- Precolored nodes are the starting point of the coloring process
- Once simplified down to colored nodes start adding back the other nodes as before

Coalescing

- Problem: coalescing can increase the number of interference edges and make a graph uncolorable



- Solution 1 (Briggs): avoid creation of high-degree (>= K) nodes
- Solution 2 (George): a can be coalesced with b if every neighbor t of a:
  - already interferes with b, or
  - has low-degree (< K)

- Code generation produces a lot of extra move instructions
  - mov t1, t2 (t1 ← t2)
  - If we can assign t1 and t2 to the same register, we do not have to execute the mov
  - Idea: if t1 and t2 are not connected in the interference graph, we coalesce into a single variable
  - First: Include in the register interference graph a move-related edge between two variables used in a move instruction

Simplify and Coalesce

- Step 1 (simplify): simplify as much as possible without removing nodes that are the source or destination of a move (move-related nodes)
- Step 2 (coalesce): coalesce move-related nodes provided low-degree node results
- Step 3 (freeze): if neither steps 1 or 2 apply, freeze a move instruction: registers involved are marked not move-related and try step 1 again
- Step 4 (spill): if there are no low-degree nodes, select a node for potential spilling
- Step 5 (select): pop each element of the stack assigning colors and turning potential spill into actual spill if needed
- Step 6 (rewrite the program): rewrite the program based on the register allocation, remove move operations with coalesced variables, and inserting spilling code. If there is spill build a new register-inference graph and goto Step 1