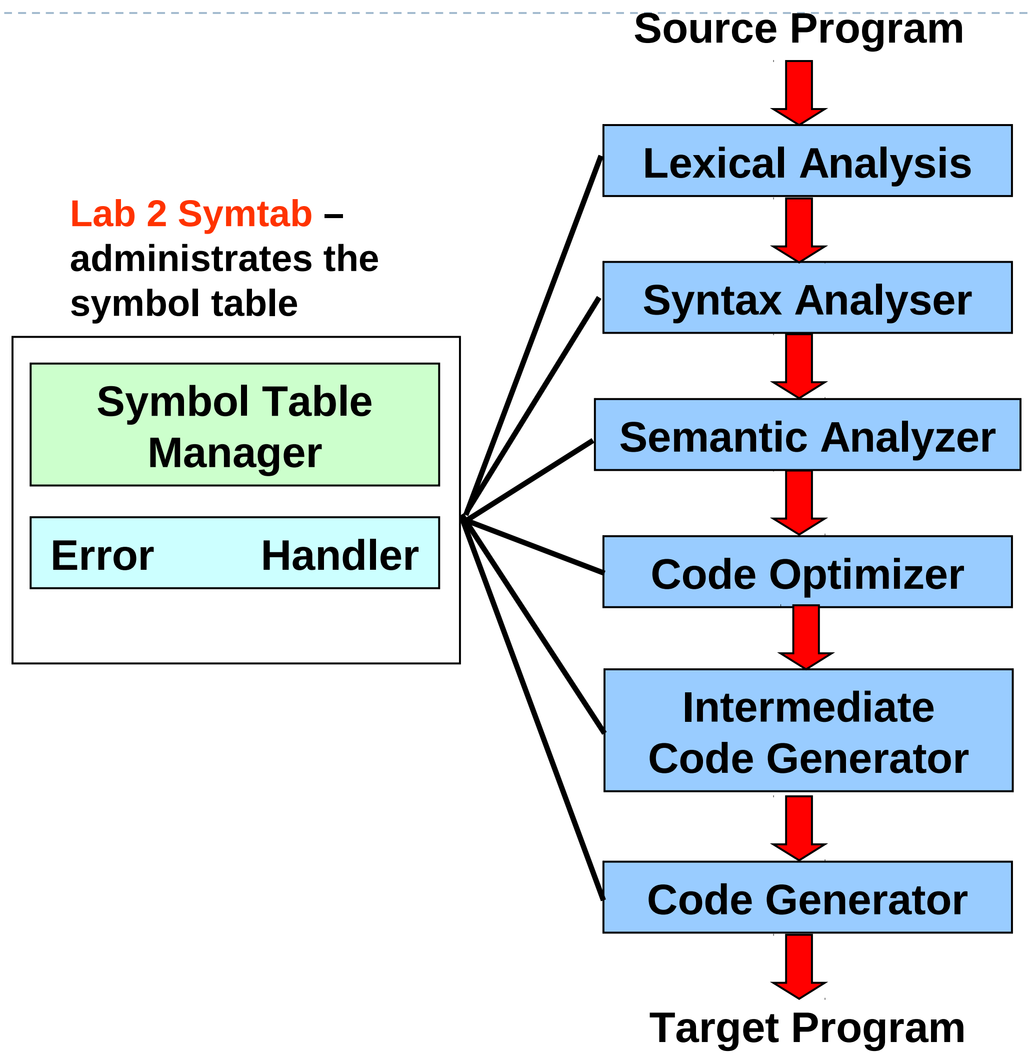
COMPILER CONSTRUCTION Lesson 2 – TDDB44

Kristian Stavåker (kristian.stavaker@liu.se)
Sergiu Rafiliu (sergiu.rafiliu@liu.se)

Department of Computer and Information Science
Linköping University

PHASES OF A COMPILER



Lab 2 Symtab – administrates the symbol table

Lab 1 Scanner – manages lexical analysis

Lab 3 Parser – manages syntactic analysis, build internal form

Lab 4 Semantics – checks static semantics

Lab 5 Optimizer – optimizes the internal form

Lab 6 Quads – generates quadruples from the internal form

Lab 7 Codegen – expands quadruples into assembly

LABORATORY ASSIGNMENTS

Assignment 4 Semantic analysis

Assignment 5 Optimization

Assignment 6 Intermediary code generation (quadruples)

Assignment 7 Code generation (assembly) and memory management

HANDING IN AND DEADLINE

- Demonstrate the working solutions to your lab assistant during scheduled time. Then send the modified files to the same assistant (put *TDDB44 <Name of the assignment>* in the topic field). One e-mail per group.

- Deadline for all the assignments is: **December 13, 2012** (you will get 3 extra points on the final exam if you finish on time!)

ASSIGNMENT 4
SEMANTIC ANALYSIS

PURPOSE

To verify the semantic correctness of the program represented by the parse tree, reporting any errors, to produce an intermediate form and certain tables for use by later compiler phases

- Semantic correctness the program adheres to the rules of the type system defined for the language (plus some other rules)
- Error messages should be as meaningful as possible
- In this phase, there is sufficient information to be able to generate a number of tables of semantic information **identifier, type** and **literal tables**

UNIQUENESS CHECKS

In certain situations it is important that particular constructs occur only once

Declarations

- within any given scope, each identifier must be declared only once.

Case statements

- each case constant must occur only once in the “switch”.

MATHEMATICAL CHECKS

- **Divide by zero**
  - Zero must be compile-time determinable constant zero, or an expression which symbolically evaluates to zero at runtime.
- **Overflow**
  - Constant which exceeds representation of target machine language arithmetic which obviously leads to overflow.
- **Underflow**
  - Same as for overflow.

TYPE CHECKS

These checks form the bulk of semantic checking and certainly account for the majority of the overhead of this phase of compilation

In general the types across any given **operator** must be **compatible**

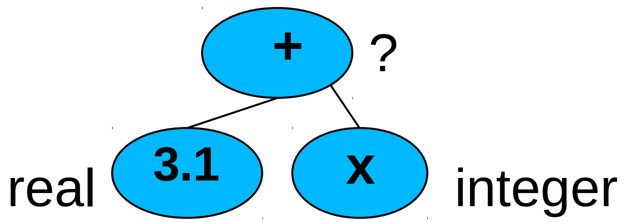
- The meaning of ***compatible*** may be:
  - *the same*
  - *two different sizes of the same basic type*

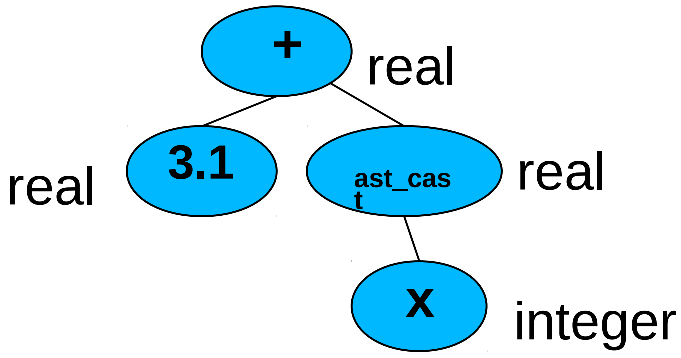
OTHER CHECKS

- All functions return something.
- The number of formal and actual parameters in a function call matches.
- ...

TYPE CONVERSION

- The semantics in DIESEL allow us, for example, to add floating-point numbers to integers.
- To make quadruple generation as simple as possible, we add a type translation node (**ast_cast**) which has a child of integer type but which is itself of floating-point type. This is known as casting the integer number to real type and is an example of type conversion.





TYPE SPECIFICATIONS FOR DIESEL

- The grammar with a description of which type restrictions apply for each production given in the laboratory compendium.
- We perform type checking one block at a time, by use of a recursive call **type_check()** which is passed from the root downwards in the AST representing the block.
- Example:
  - => 68. <term><term> AND <factor>

    Both operands must be of integer type. The result is of integer type.

FILES TO BE CHANGED

- **semantic.hh** and **semantic.cc** contains type checking code implementation for the AST nodes as well as the declaration and implementation of the semantic class. These are the files you're going to edit in this lab. They deal with type checking, type synthesizing, and parameter checking.

OTHER FILES OF INTEREST

- All these files are the same as in **lab 3**:
- **parser.y** is the input file to **bison**. This is the file you edited in the last lab, and all you should need to do now is uncomment a couple of calls to:
  - **do_typecheck()**.
- **ast.hh** contains the definitions for the AST nodes.
- **ast.cc** contains (part of) the implementations of the AST nodes.
- **optimize.hh** and **optimize.cc** contains optimizing code.
- **quads.hh** and **quads.cc** contains quad generation code.
- **codegen.hh** and **codegen.cc** contains assembler generation code.

OTHER FILES OF INTEREST

- **error.hh, error.cc, symtab.hh, symbol.cc, symtab.cc, scanner.l** use your versions from the earlier labs.
- **main.cc** this is the compiler wrapper, parsing flags and the like.
- **Makefile** and **diesel** use the same files as in the last lab.

ASSIGNMENT 5
OPTIMIZATION

CODE OPTIMIZATION

Optimization is the process of improving the code produced by the compiler.

The resulting code is “seldom” optimal but is rather better than it would be without the applied “improvements”.

Many different kind of optimizations are possible and they range from the simple to the extremely complex.

TYPES OF OPTIMIZATION

Three basic types of optimization:

- The “code” in question might be abstract syntax tree in which case machine independent optimization may be performed.
- The code in question may be intermediate form code in which case machine independent optimization may be performed.
- The code might also be assembly/machine code in which case machine dependent optimization may be performed.

OTHER OPTIMIZATION TYPES

Other taxonomies of optimization divide things up differently:

- Global optimization considering the whole program as a routine.
- Local optimizations within a basic block.
- Peephole optimizations considering only a small sequence of instructions or statements.

COMPENSATION

Many of the optimizations are done to *compensate* for compiler rather than programmer deficiencies.

It is simply convenient to let the compiler do “stupid” things early on and then fix them later.

MACHINE INDEPENDENT OPTIMIZATION

- Don’t consider any details of the target architecture when making optimization decisions.
- This optimization tend to be very general in nature.

MACHINE DEPENDENT OPTIMIZATION

- *Machine dependent optimization* on assembly or machine code.
- Target machine architecture specific.

CONSTANT FOLDING

Expressions with *constant operands* can be evaluated at compile time, thus improving run-time performance and reducing code size by avoiding evaluation at compile-time.

CONSTANT FOLDING

- Constant folding is a relatively simple optimization.
- Programmers generally do not write expressions such as '**5** + **3**' directly, but these expressions are relatively common after macro expansion; or other optimization such as constant propagation.

CONSTANT PROPAGATION

- Constants assigned to a variable can be propagated through the flow graph and substituted at the use of the variable.

COMMON SUB-EXPRESSION ELIMINATION

An expression is a Common Sub-Expression (CSE) if the expression is:

1) previously computed

2) the values of the operands have not changed since the previous computation

Re-computing can then be avoided by using the previous value.

COMMON SUB-EXPRESSION ELIMINATION

Below, the second computation of the expression '**x + y**' can be eliminated:

```
i := x + y + 1;
j := x + y;
```

After CSE Elimination, the code fragment is rewritten as follows:

```
t1 := x + y;
i  := t1 + 1;
j  := t1;
```

DEAD CODE ELIMINATION

- Code that is unreachable or that does not affect the program (e.g. dead stores) can be eliminated directly.

DEAD CODE ELIMINATION

```
var
   global : integer;
procedure f;
var
   i : integer;
begin
   i      := 1;     { dead store }
   global := 1;     { dead store }
   global := 2;
   return;
   global := 3;     { unreachable }
end;
```

- The value assigned to **i** is never used
- The first assignment to **global** is dead
- The third assignment to **global** is unreachable

DEAD CODE ELIMINATION

After elimination of dead code the fragment is reduced to:

```
var
   global : integer;
procedure f;
begin
   global := 2;
   return;
end;
```

FORWARD STORES

- Stores to global variables in loops can be moved out of the loop to reduce memory bandwidth requirements.

FORWARD STORES

Below the *load* and *store* to the global variable *sum* can be moved out of the loop by computing the summation in a register and then storing the result to sum outside the loop:

```
int sum;
void f (void)
{
  int i;

  sum = 0;
  for (i = 0; i < 100; i++)
    sum += a[i];
}
```

FORWARD STORES

After forward store optimization the code looks like this:

```
int sum;
void f (void)
{
  int i;
  register int t;
  sum = 0;
  for (i = 0; i < 100; i++)
    t += a[i];
  sum = t;
}
```

IMPLEMENTATION

- In this lab you are to implement **constant folding** as described earlier.
- You will optimize the **abstract syntax tree**.
- The tree traversal will be done using recursive method calls, similar to the type checking in the last lab.
- You will start from the root and then make *optimize()* calls, that will propagate down the AST, and try to identify sub-trees eligible for optimization.

IMPLEMENTATION

- Requirements:
  - Must be able to handle optimizations of all operations derived from *ast_binaryoperation*.
  - Need only optimize subtrees whose leaf nodes are instances of *ast_real*, *ast_integer* or *ast_id* (constant).
  - No need to optimize *ast_cast* nodes, but feel free to implement this.
  - No need to optimize optimization of binary relations, but feel free to implement this.
  - Your program must preserve the code structure, i.e. the destructive updates must not change the final result of running the compiled program in any way.
  - Optimization should be done one block at a time (local optimization).

FILES OF INTEREST

- Files you will have to modify
  - **optimize.hh** and **optimize.cc** contains optimizing code for the AST nodes as well as the declaration and implementation of the ast_optimizer class. These are the files you will edit in this lab.
- Other files of interest
  - **ast.hh** : contains the implementations of the AST nodes.
  - **ast.cc** : contains the implementations of the AST nodes.
  - **parser.y** : the function do_optimize() is called from here.
  - **error.hh**, **error.cc**, **symtab.hh**, **symbol.cc**, **symtab.cc**, **scanner.l**, **semantic.hh**, **semantic.cc** : use your versions from earlier labs.
  - **Makefile** and **diesel** use the same files as in the last lab.

ASSIGNMENT 6
QUADRUPLES

INTERMEDIATE CODE

- Is closer to machine code without being machine dependent.
- Can handle temporary variables.
- Means higher portability, intermediate code can easier be expanded to assembly code.
- Offers the possibility of performing code optimizations such as register allocation.

INTERMEDIATE LANGUAGE

- Why use intermediate languages?
  - **Retargeting** - build a compiler for a new machine by attaching a new code generator to an existing front-end and middle-part
  - **Optimization** - reuse intermediate code optimizers in compilers for different languages and different machines
  - **Code generation** - for different source languages can be combined

THREE-ADDRESS SYSTEM

- Source statement:

```
x := a + b * c + d;
```

Three address statements with temporaries **t1** and **t2**:

```
t1 := b * c;
t2 := a + t1;
x  := t2 + d;
```

QUADRUPLES

You will use **Quadruples** as intermediate code where each instruction has four fields:

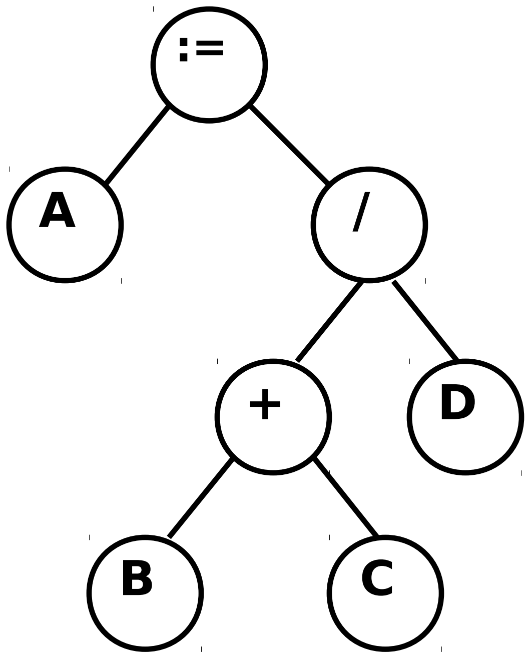
operator	**operand1**	**operand2**	**result**

QUADRUPLES

(A + B) * (C + D) - E

operator	**operand1**	**operand2**	**result**
+	A	B	T1
+	C	D	T2
*	T1	T2	T3
-	T3	E	T4

QUADRUPLES



A := (B + C) / D;

q_iplus	**10**	**11**	**13**
q_idiv	**13**	**12**	**14**
q_assign	**14**	**0**	**9**

The numbers are indexes in the symbol table

9	**10**	**11**	**12**	**13**	**14**
A	**B**	**C**	**D**	**T1**	**T2**

QUADRUPLES

- Another example:

The DIESEL statement ***a[a[1]] := a[2];*** **will generate:**

q_iload	**2**	**0**	**10**	
q_irindex	**9**	**10**	**11**	**(retrieves a value)**
q_iload	**1**	**0**	**12**	
q_irindex	**9**	**12**	**13**	
q_lindex	**9**	**13**	**14**	**(calculates an address)**
q_istore	**11**	**0**	**14**	

The numbers are indexes in the symbol table

9	**10**	**11**	**12**	**13**	**14**
A	**T1**	**T2**	**T3**	**T4**	**T5**

QUADRUPLES

- Another example:

The DIESEL statement *foo(a, bar(b), c);* will generate:

q_param	**11**	**0**	**0**
q_param	**10**	**0**	**0**
q_call	**13**	**1**	**14**
q_param	**14**	**0**	**0**
q_param	**9**	**0**	**0**
q_call	**12**	**3**	**0**

The numbers are indexes in the symbol table

9	**10**	**11**	**12**	**13**	**14**
A	**B**	**C**	**FOO**	**BAR**	**T1**

QUADRUPLES

- Operations are typed. There are both *q_rdivide* and *q_idivide*. The operation to select depends on the node type if it is an arithmetic operation but on the children's types if it is a relational operation.

HANDLING REAL NUMBERS

- When generating assembly code all real numbers are stored in 32 bits.
- We do this by storing real numbers as integers in the *IEEE* format.
- Use the symbol table method *ieee()*. It takes a float number and returns an integer representation in the 32-bit IEEE format.
- So when you are generating a quadruple representing or treating a real number call: *sym_tab->ieee(value);*

IMPLEMENTATION

- In this lab, you will write the routines for converting the internal form we have been working with so far into quadruples.
- The quadruple generation is started from *parser.y* with a call to *do_quads()*. This function will call *generate_quads()* which propagates down the AST.
- The final result is a *quad_list* containing the quadruples generated while traversing the AST.

IMPLEMENTATION

- Complete the empty generate method bodies in **quads.cc**.
- Complete the empty method body *gen_temp_var()* in the file **symtab.cc**. It takes a sym_index to a type as argument. It should create and install a temporary variable (of the given type) in the symbol table. Give your temporary variables “unique” names that are not likely to collide with the user variables.

FILES OF INTEREST

- Files you will have to modify
  - **quads.cc, quads.hh** : contains quad generation code for the AST nodes as well as the declaration and implementation of the quadruple, quad_list, quad_list_element and quad_list_iterator classes. These are the files you will edit in this lab.
  - **symtab.cc** : You will need to complete one more method in this lab.
- Other files of interest
  - **ast.hh** : contains the definitions of the AST nodes.
  - **ast.cc** : contains (part of) the implementations of the AST nodes.
  - **parser.y** : the function *do_quads()* is called from here.
  - **error.hh**, **error.cc**, **symtab.hh**, **symbol.cc**, **symtab.cc**,
  - **scanner.l**, **semantic.hh**, **semantic.cc**, **optimize.hh**, **optimize.cc** : use your versions from earlier labs.

ASSIGNMENT 7
CODE GENERATION

CODE GENERATION

Once the source code has been

1) scanned
2) parsed and transformed into internal form
3) semantically analyzed

code generation might be performed.

CODE GENERATION

- Code generation is the process of creating assembly/machine language statements which will perform the operations specified by the source program when they run.

CODE GENERATION

In addition other code is also produced:

- Typically assembly directives are produced, e.g. storage allocation statements for each variable and literal in the program.

CODE GENERATION

Un-optimized code generation is relatively straightforward:

- Simple mapping of intermediate code constructs to assembly/machine code sequences.
- Resulting code is quite poor though compared to manual coding.

CODE GENERATION FOR SPARC

- We are going to use a simple method which expands each quadruple to one or more assembly instructions.
- SPARC has 32 general 32-bit registers, and 32 floating-point registers.

global registers	**%g0, ..., %g7**	
“in”-register	**%i0, ..., %i7**	
local register	**%l0, ..., %l7**	
“out”-register	**%o0, ..., %o7**	**(%o6 stack pointer ...)**
floating-point register	**%f0, ..., %f31**	

MEMORY MANAGEMENT

- *Static memory management*: In certain programming languages recursion and dynamic data allocation is forbidden and the size must be known at compile time. No run-time support needed and all data can be referenced using absolute addresses. (FORTRAN).
- *Dynamic memory management:* Other languages such as Pascal, C++ and Java allow recursion and dynamic memory allocation.

DYNAMIC MEMORY MANAGEMENT

- All data belonging to a function/procedure is gathered into an *Activation Record (AR)*. An AR is created when the function/procedure is called and memory is allocated on a *stack*.

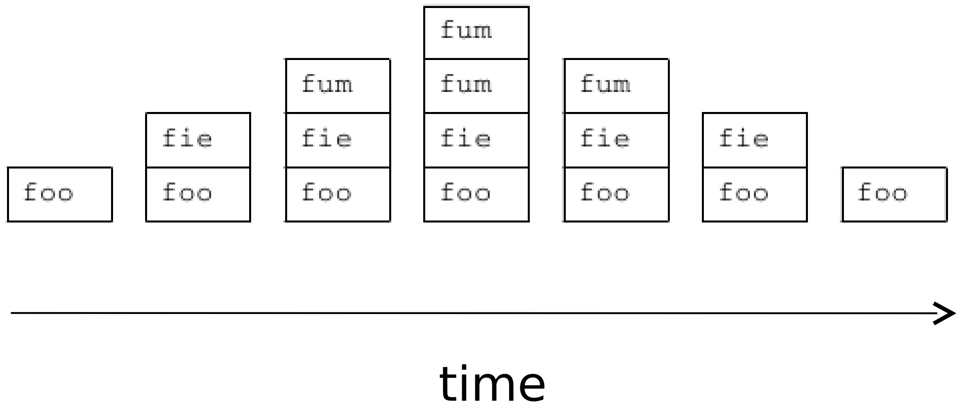
ACTIVATION RECORD

- Local data
- Temporary data
- Return address
- Parameters
- Pointers to the previous activation record (dynamic link).
- Static link or display to find the right reference to non-local variables.
- Dynamically allocated data (dope-vectors).
- Possibly space for return values (applies to functions, not procedures).
- Place to save register contents.

ACTIVATION RECORD

- An example:

```
procedure fum(i : integer);

begin

   if i <> 0 then
       fum(i - 1);
   end;

end;

procedure fie;

begin
   fum(1);
end;

procedure foo;
begin
  fie();
end;
```



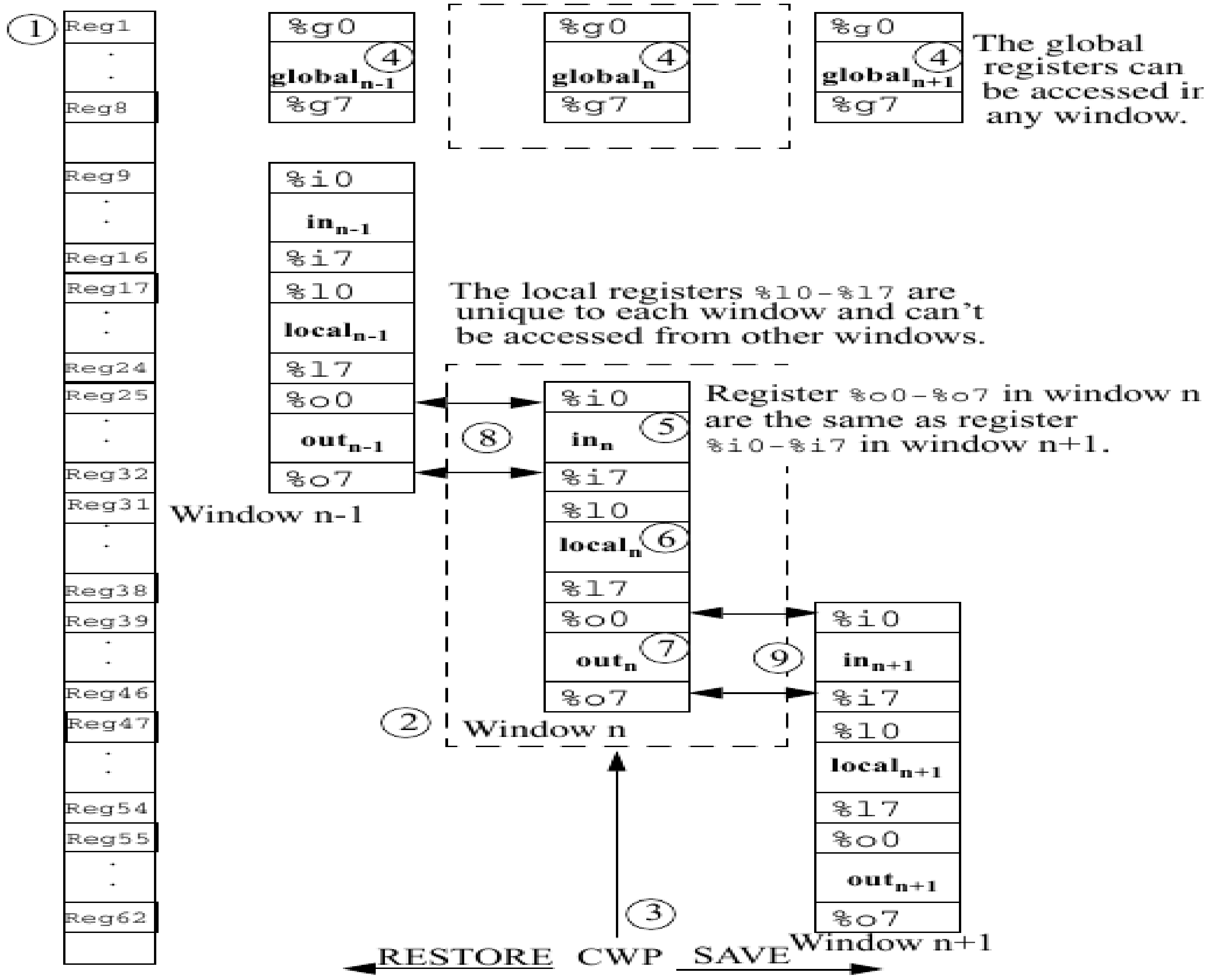
REGISTER WINDOWS

- The SPARC processor has many general registers, typically about 128 or more. But the programmer can’t access all at the same time.
- The register window mechanism only shows a subset of all the registers at a given time, for each procedure/function (in this case 32 general registers). These registers are organized in four groups: global, in, local, and out.
- The global registers can be accessed from any window.

REGISTER WINDOWS

- There are two instructions, *SAVE* and *RESTORE*, which (among other things) move the register window. *SAVE* and *RESTORE* are also used to create and release activation records.
- These instructions, together with the overlaying of out- and in-registers in adjacent windows, are used to implement parameter transfer as well as creating and releasing activation records.

REGISTER WINDOWS



IMPLEMENTATION

- In this lab, you will write certain routines that help expanding quadruples into assembly, as well as some routines for handling creating and releasing activation records.
- The assembly code generation is done by traversing a quad list, expanding each quad to assembly as we go. The expansion is started from *parser.y* with a call *generate_assembler()* to a code generator class.

IMPLEMENTATION

- Complete the *prologue()* method (used when entering a block).
- Complete the *epilogue()* method (used when leaving a block).
- Write the *find()* method which given a *sym_index* returns the display register level and offset for a variable, array or parameter to the symbol table.
- Write the *fetch()* method that retrieves the value of a variable, parameter or constant to a given register.

IMPLEMENTATION

- Write the *store()* method which stores the value of a register in a variable or parameter.
- Write the *array_address()* method which retrieves the base address of an array to a register.
- Complete the *expand()* method which translates a quad list to assembly code using the methods above. You will need to write code for expanding *q_param* and *q_call* quads.

FILES OF INTEREST

- Files you will have to modify
- **codegen.hh**, **codegen.cc** : contains assembly generation code for SPARC assembly. These are the files you will edit in this lab.
- Other files of interest
  - **parser.y** is the input file to bison.
  - **ast.hh** contains the definitions for the AST nodes.
  - **ast.cc** contains (part of) the implementations of the AST nodes.
  - **error.hh**, **error.cc**, **symtab.hh**, **symbol.cc**, **symtab.cc**, **scanner.l**, **semantic.hh**, **semantic.cc**, **optimize.hh**, **optimize.cc**, **quads.hh**, **quads.cc** use your versions from the earlier labs.
  - **main.cc** this is the compiler wrapper, parsing flags and the like. Same as in the previous labs.
  - Makefile and diesel use the same files as in the last lab.