COMPILER CONSTRUCTION
Seminar 03 – TDDE66 2024

Adrian Pop (adrian.pop@liu.se)
Martin Sjölund (martin.sjolund@liu.se)
Mahder Gebremedhin (mahder.gebremedhin@liu.se)

Department of Computer and Information Science
Linköping University

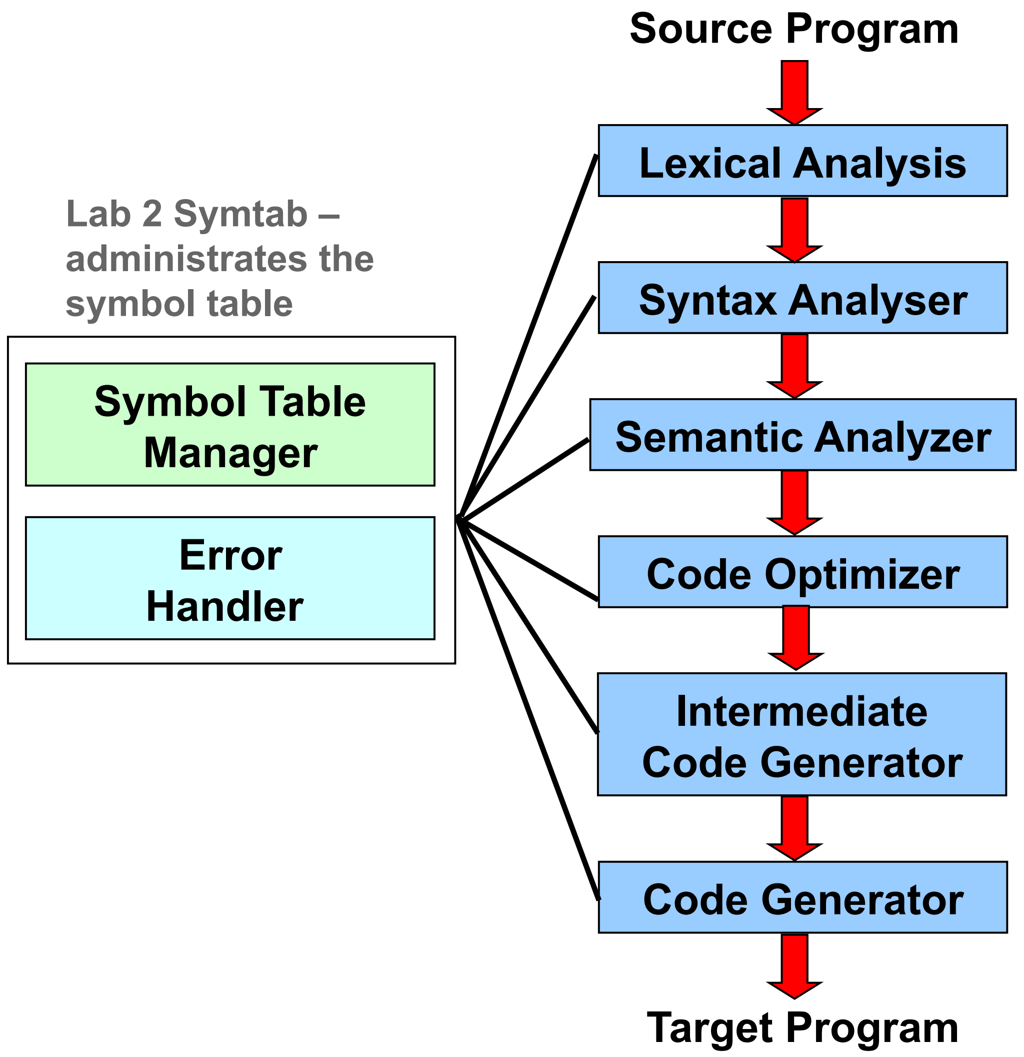
LABS

Lab 5 Optimization

Lab 6 Intermediary code generation (quadruples)

Lab 7 Code generation (assembly) and memory management

PHASES OF A COMPILER



Lab 1 Scanner – manages lexical analysis

Lab 2 Symtab – administrates the symbol table

Lab 3 Parser – manages syntactic analysis, build internal form

Lab 4 Semantics – checks static semantics

Lab 5 Optimizer – optimizes the internal form

Lab 6 Quadruples – generates quadruples from the internal form

Lab 7 Codegen – expands quadruples to assembly

LAB 5
OPTIMIZATION

COMPILER 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 the abstract syntax tree in which case machine independent optimization is being performed.
- The code in question may be the intermediate form code in which case machine independent optimization is being performed.
- The code might also be assembly/machine code in which case machine dependent optimization is done.

COMPENSATION

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

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

OTHER OPTIMIZATION TYPES

Other taxonomies of optimization divide things up differently:

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

MACHINE INDEPENDENT

Machine independent optimization is typically done using the intermediate form as a base.

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

MACHINE DEPENDENT

Machine dependent optimization performed on assembly or machine code.

- Target the specifics of the machine architecture.
- Machine dependent optimizations are extremely specific.

MACHINE DEPENDENT

- Peephole optimization of assembly code:

```
...                  ...
LD A, R0             INC A, R0
ADD 1, R0      ⇨     (removed)
ST R0, A             (removed)
LD A, R0             LD A, R0
…                    …
```

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

- In the code example, the expression '`5 + 3`' can be replaced with '`8`' at compile time.
- This makes the compiled program run faster, since fewer instructions are generated to run this piece of code.

```
function f : integer;
begin
    return 5 + 3;
end
```

```
function f : integer;
begin
    return 8;
end
```

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 FOLDING

- All **C** compilers can fold integer constant expressions that are present after macro expansion (**ANSI C** requirement).

- Most **C** compilers can fold integer constant expressions that are introduced after other optimizations.

CONSTANT FOLDING

- Some environments support several floating-point rounding modes that can be changed dynamically at run time.

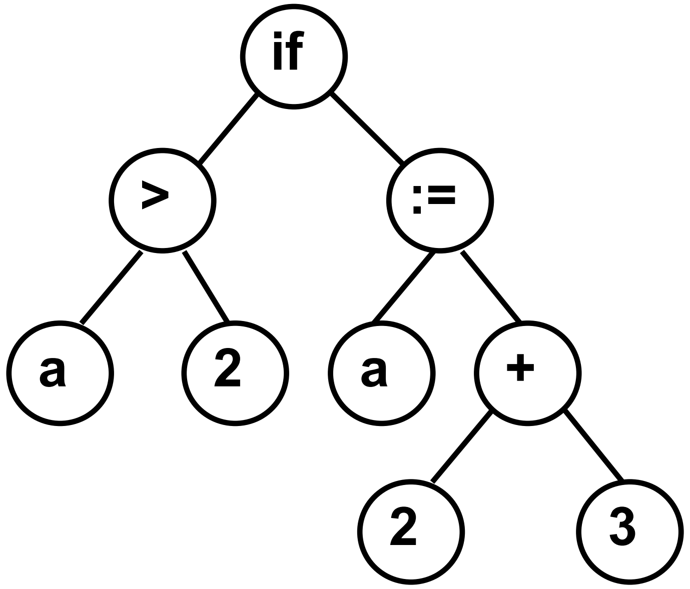
- In these environments, expressions such as '`( 1.0 / 3.0 );`' *must* be evaluated at run-time if the rounding mode is not known at compile time.

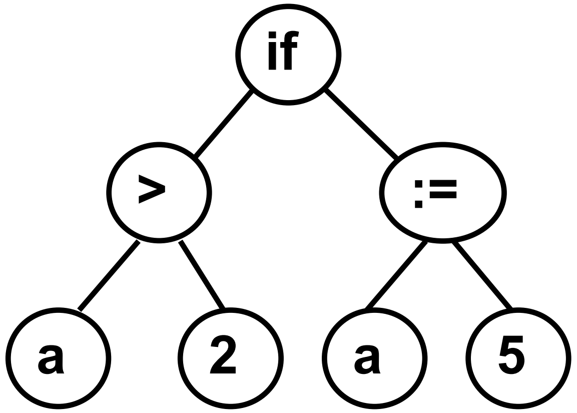
CONSTANT FOLDING

DIESEL code .... optimized DIESEL code

```
if (a > 2) then
    a := 2 + 3;
end;
```

```
if (a > 2) then
    a := 5;
end;
```





CONSTANT PROPAGATION

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

An expression is a Common Sub-expression (CS) 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.

CONSTANT PROPAGATION

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;
```

EXPRESSION SIMPLIFICATION

Some expressions can be simplified by replacing them with equivalent expressions that are more efficient.

EXPRESSION SIMPLIFICATION

The code:

```
i      := { ... } ;
a[0]  := i + 0;
a[1]  := i * 0;
a[2]  := i - i;
a[3]  := 1 + i + 1;
```

can be simplified to:

```
i      := { ... } ;
a[0]  := i;
a[1]  := 0;
a[2]  := 0;
a[3]  := 2 + i;
```

EXPRESSION SIMPLIFICATION

Programmers generally do not write expressions like '`i + 0`' directly, but these expressions can appear after optimizations.

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

IMPLEMENTATION

- In this lab you are to implement the **constant folding algorithm** as described earlier.
- You will optimize the **abstract syntax tree (AST)**.
- 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 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 need 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

(All these files are the same as in the last lab, except that you need to activate the do_optimize() call in parser.y.)

  - **`ast.hh`**: contains (part of) the implementations of the AST nodes.
  - **`ast.cc`**: contains (part of) 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`**: use your versions from earlier labs.
  - **`Makefile`** and **`diesel`** use the same files as in the last lab.

LAB 6
QUADRUPLES

INTERMEDIATE CODE

- Is closer to machine code without being machine dependent

- Can handle temporary variables

- Means higher portability, intermediary code can easily be expanded to assembler

INTERMEDIATE CODE

Various types of intermediary code are:

- Infix notation
- Postfix notation
- Three address code
  - Triples
  - **Quadruples**

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
- **Optimization** - reuse intermediate code optimizers in compilers for different languages and different machines
- **Code generation** for different source languages can be combined

INTERMEDIATE LANGUAGE

Syntax Tree

Graphical representation.

DAG

Common sub-expressions eliminated from syntax tree.

Three-address code

Close to target assembly language.

THREE-ADDRESS SYSTEM

A popular form of intermediate code used in optimizing compilers is three-address statements (or variations, such as quadruples)

Advantages of using Three-Address Code:

- Three-address operands should be simple to implement on the target machine.
- Rich enough to allow compact representation of source statements.
- Statements should be easy to rearrange for optimization.

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 intermediary code where each instructions has four fields:

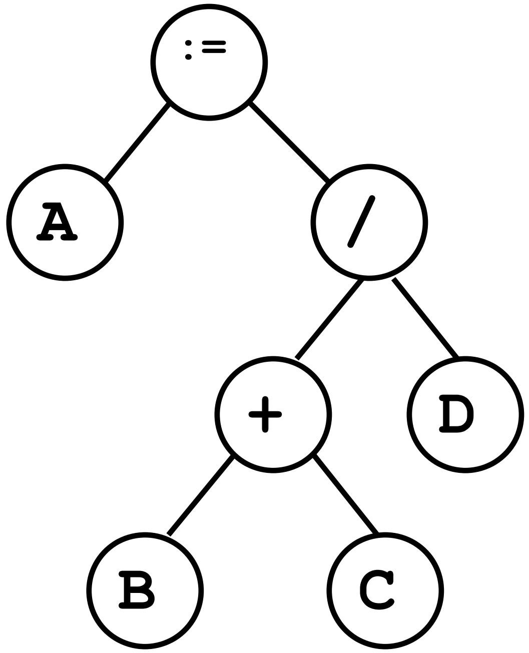
```
operator     operand1   operand2   result
```

QUADRUPLES

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

operator	operand1	operand2	result
+	A	B	T1
+	C	D	T2
*	T1	T2	T3
-	T2	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
q_iload      1      0     12
q_irindex    9     12     13
q_lindex     9     13     14
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

- In the lab temporary variables will be stored together with the local variables on the stack. (This is to avoid register allocation. The goal is not to create a good code generator, but something that works. Temporary variables regardless if they are needed or not)

- 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 64 bits.
- We do this by storing real numbers as integers in the *IEEE* format.
- Use the symbol table method **`ieee()`**. It takes a double number and returns an integer representation in the 64-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/quads.
- 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. This is done one block at a time.
- 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`**.

- In file **`symtab.cc,`** complete the empty method body **`gen_temp_var()`**. 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 variable names. (Hint: if you want to match the traces generated by the master tool, generate names that match this, including spaces)

FILES OF INTEREST

- Files you will need 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, optimize.hh, optimize.cc`** : use your versions from earlier labs.

LAB 7
ASSEMBLER

CODE GENERATION

Once the source code has been

1) scanned
2) parsed
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 assembler 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 INTEL

- We are going to use a simple method which expands each quadruple to one or more assembler instructions.

- Intel has a number of general purpose 64-bit registers. Only some will be used.

- For the real number operations we will use the floating point unit (FPU), which is a stack.

- More about this in the lab compendium.

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.

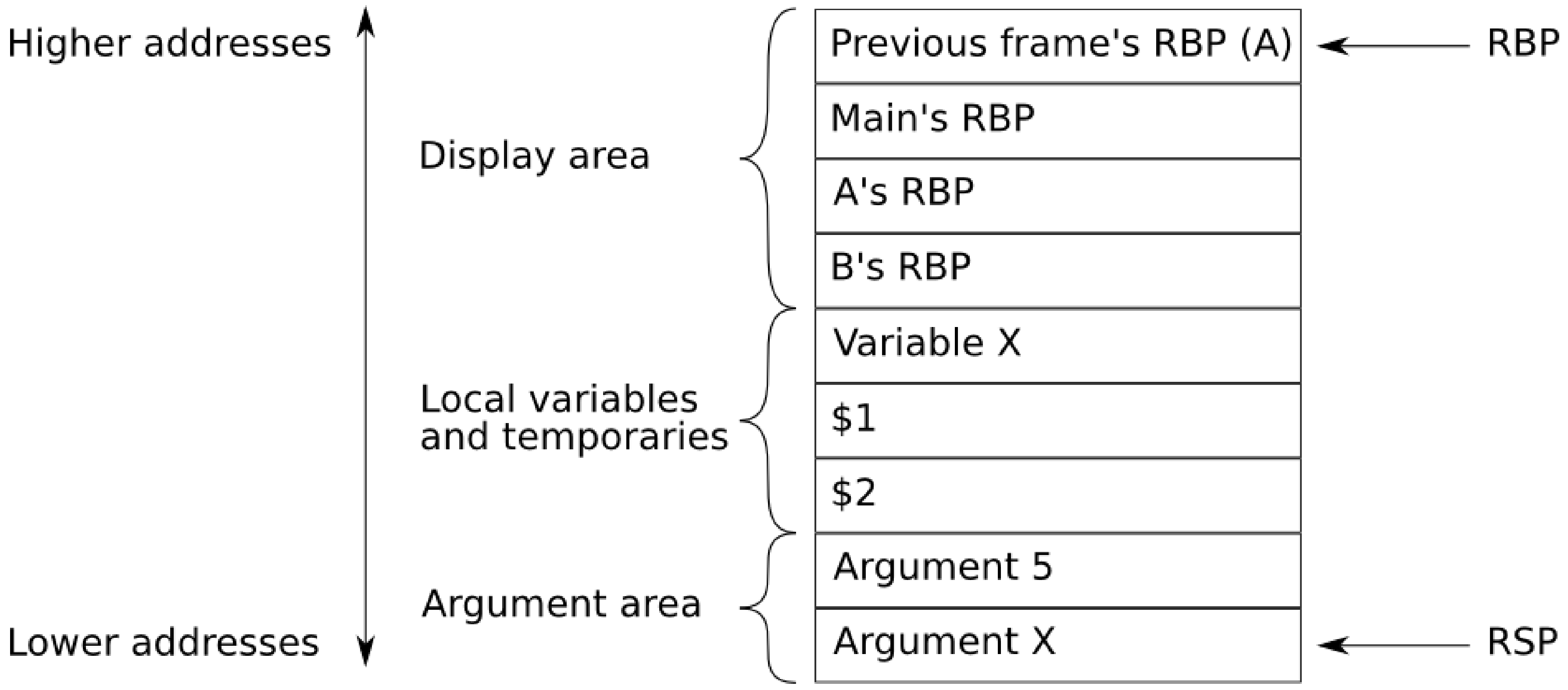
DYNAMIC MEMORY MANAGEMENT

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.

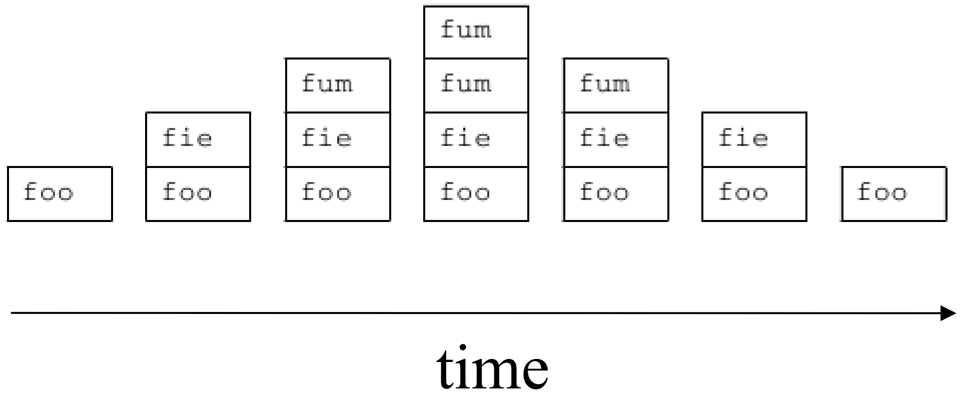
ACTIVATION RECORD



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;
…
```



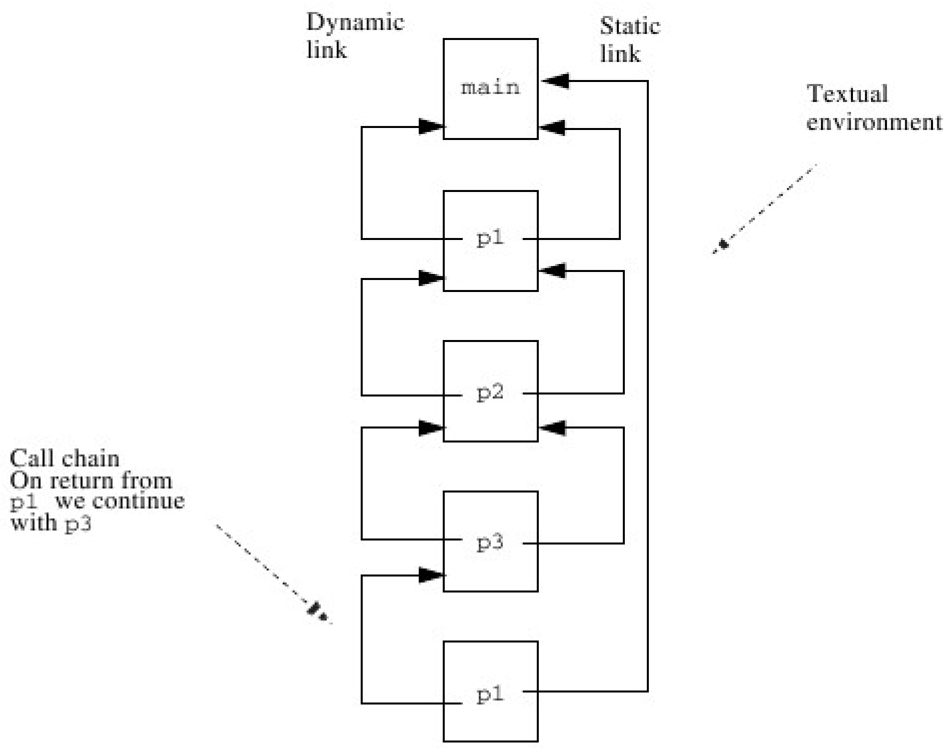
ACTIVATION RECORD

Static versus Dynamic link:

```
program main;

procedure p1;
   procedure p2;
      procedure p3;
      begin
         p1(); { (1) }
      end;
   begin
      p3();
   end;
begin
   p2();
end;

begin
   p2();
end.
```



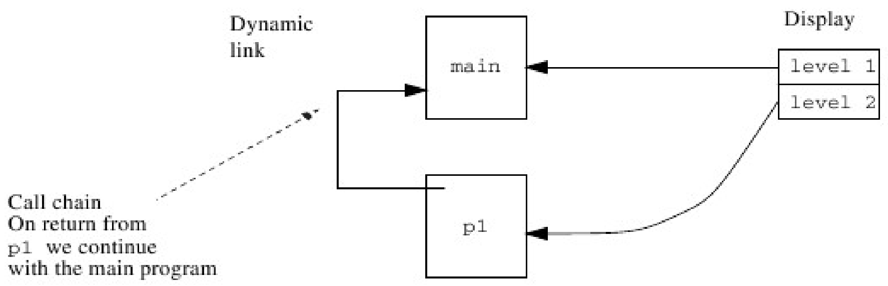
DISPLAY

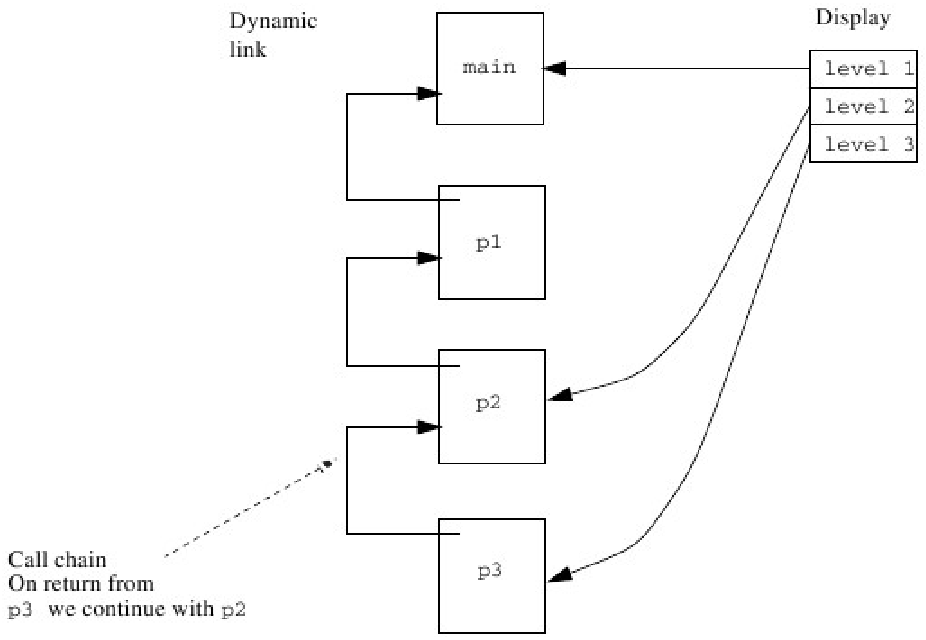
```
program main;

procedure p2;
   procedure p3;
   begin
      { (2) }
   end;
begin
   p3();
end;

procedure p1;
begin
   p2();    { (1) }
end;

begin
    p1();
end.
```





Intel x87 Floating Point Unit

Recall postfix code and stack machines

```
fld arg1 # Push arg1 on the stack
fld arg2 # Push arg2 on the stack
faddp    # Add the top 2 elements together
fstp res # Pop stack into res
```

NOTE: Floating point constants cannot be loaded into the FPU using `fld` as an immediate value, nor from a register. There exists instructions to load 0.0, 1.0, π, $\log_2 10$, etc. You need to find a way to load constants into the FPU stack in `fetch_float`.

IMPLEMENTATION

- In this lab, you will write certain routines that help expanding quadruples into assembler, 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 assembler 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 from memory and stores it into a given register.

IMPLEMENTATION

- Write the **`fetch_float()`** method that pushes the value of a variable, parameter or constant to the FPU. Note that this method will never generate code for constant integers but will for constant reals.
- Write the **`store()`** method which stores the value of a register in a variable or parameter.
- Write the **`store_float()`** method which pops the FPU stack and stores the value in a variable or parameter.

IMPLEMENTATION

- Write the **`array_address()`** method which retrieves the base address of an array to a register.
- Write the **`frame_address()`** method which, given a lexical level and a register, stores the base address of the corresponding frame from the display area.
- Complete the **`expand()`** method which translates a quad list to assembler 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 need to modify
  - **`codegen.hh, codegen.cc :`** contains assembler generation code for the Intel assembler. 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.