Static Checking

(also known as "context checking" or "semantic analysis")

- type checking
- flow of control checks
- uniqueness checks
- name related checks

Static Checking by Attribute Evaluation

An attribute grammar describes static checks.

Attributes used in static checking are called *semantic attributes*.

The abstract syntax tree (AST) is traversed and decorated with attributes until it contains enough information so that static checks can be made.

Static Checking

Static checking can be performed after the AST has been constructed, or "on-the-fly", while it is being built.

In the first case, the AST is traversed, usually more than once, and attributes are evaluated each time a node is visited.

On-the-fly evaluation methods operate in conjunction with the parser (just like the tree building routines). They are used for single pass compilation.

The parsing method used limits the kind of attribute flow that can be accommodated when on-the-fly evaluation is carried out.

Intermediate Representations

The "front end" or analysis phases of a compiler produce intermediate code, which is processed by the code generator or "back end" of the compiler.

Three kinds of intermediate representation are

- abstract syntax trees,
- postfix notation and
- 3-address code.

3-Address Code

3-Address Code is like assembly language. It consists of a sequence of statements like

```
x := y op z
```

where x, y and z are names, constants, or temporary names generated by the compiler and op is an operator.

Statements can have labels, and there are statements that modify flow of control. For example:

```
label: x := y op z
goto label
if x relop y goto label
```


3-Address Code

Example: 3-address code corresponding to the syntax tree shown earlier.

```
t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a  := t5
```


Translation into 3-Address Code

Syntax directed translation of the source program into 3-address code can be defined using an attribute grammar.

Example: An attributed grammar rule for translating a 'while' loop to 3-address statements:

```
S ::= while E do S1
  {S.begin := newlabel;
   S.after := newlabel;
   S.code  :=
     S.begin <> ':' <> E.code <>
     'if' <> E.place <> '= 0' <>
     'goto' <> S.after <>
     S.code <>
     'goto' S.begin  <>
     S.after ':'
  }
```


3-Address Statements

- Basic Assignments
  x := y op z
  x := op y
  x := y
- Transfer of Control
  goto label
  if x relop y goto label
- Procedure Call
  param x1
  param x2
  ...
  param xn
  call p,n
- – and Return
  return y
- Indexing
  x[i] := y
  y := y[i]
- Operations involving pointers
  x := &y
  x := *y
  x := y

3-address code for computing a scalar product – an example from Aho, Sethi and Ullman

Assume a and b are two vectors of length 20. The following 3 address statements compute the scalar product of a and b. (It assumes that the target machine has four byte words).

```
(1)  prod := 0
(2)  i := 0
(3)  t1 := 4 * i
(4)  t3 := a[t1]
(5)  t4 := b[t1]
(6)  t5 := t3 * t4
(7)  prod := prod + t5
(8)  i := i + 1
(9)  if i <= 20 goto (3)
```


Code Generation

Code generation is the final phase in compilation. From an intermediate represention of the source program, the code generator produces a target program.

The code generator's main tasks are

- memory management,
- register allocation and
- instruction selection.

Code Generation

The code that is generated should be

- compact,
- fast and
- effective in its use of machine resources.

Memory Management

The code generated by the code generator implements a memory management strategy. This means determing

- when variables are to be bound to storage, and
- where storage is to be allocated in memory.

Variables appear as names in 3-address code. Each name in a 3-address statement refers to a symbol table entry. The entry includes information about the type of the named data item. The code generator uses this type information to produce code that allocates a suitable amount of space for the data item.

Register Allocation

Register allocation entails

- selecting the variables that will reside in registers, and
- choosing the specific register in which each variable will reside.

Register allocation must be well done if the target code is to be compact and fast.

Instruction Selection

Instruction selection means choosing suitable target machine instructions to implement the intermediate program.

For example, suppose the target machine includes an instruction

```
INC
```

to add one to a register. Then

```
INC R0
```

is a better translation of a := a + 1 than

```
MOV a, R0
ADD #1, R0
MOV R0, a
```


Flow of Control Information

The code generator uses information about the flow of control of the intermediate program to produce efficient target code.

This flow of control information is represented as a flow graph.

Nodes in the flow graph represent *basic blocks* of instructions, in which control passes sequentially from instruction to instruction.

Edges in the flow graph represent transfers of control.

Basic Blocks

In a 3-address program, a basic block is a sequence of 3-address statements in which control begins at the first instruction, and passes through to the last instruction without halting or branching.

For example,

```
t1 := a * a
t2 := a * b
a  := t1 + t2
```

is a basic block.

A basic block computes a set of expressions. These expressions are the values of the names that are *live* on exit from the basic block.

Flow Graphs for 3-address Programs

A flow graph of a 3-address program has basic blocks of 3-address statements as its nodes.

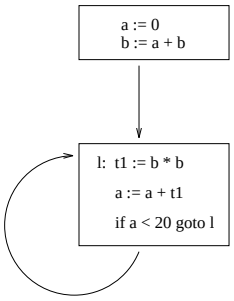
The node that contains the first statement of the program is called the initial node.

Suppose B1 and B2 are nodes. There is a directed edge from B1 to B2 if

- there is a jump from the last statement of B1 the first statement of B2, or
- B2 follows B1 in the program, and B1 does not end in an unconditional jump.

Flow Graphs for 3-address Programs

Example



Next Use Information

The code generator collects next use information to help decide how to use registers and temporary storage. The general strategy is to keep a name in storage only if it will be used again.

Next use information is collected by scanning the basic blocks backwards, and recording for each name in a block whether the name has a next use in the block, and, if not, whether it is live on exit from the block.

A Simple Code Generator – Aho, Sethi and Ullman

Input – a sequence of 3-address statements

Output – target code

Strategy – consider each of the 3-address statements in turn, "remembering" if any of the operands are currently in registers, and taking advantage of that fact if possible.

Computed results are left in registers as long as possible, and are stored only when the register is needed for another computation, or a procedure call, jump or labelled statement is encountered.

A Simple Code Generator

Important Data Structures

Register Descriptor – current contents of each register.

Address Descriptor – where current value of a name can be found at run time; e.g. in a register, on the stack, in memory.

Important Function

get reg – a procedure that returns the address of a location L in which to store the result of a 3-address assignment. Usually L will be a register, but it could be a memory location.

Code Generation Algorithm

For 3-address statements of the form `x := y op z`

- Call 'get reg' to determine the location L where the result of y op z will be stored.
- Look up y', the current location of y in the address descriptor; if y is both in memory and in a register, choose the register. Unless the value of y is already in L, generate `MOV y', L`
- Determine z', the current location of z in and generate `OP z', L`
- Update the address descriptor to indicate that L is the location of the current value of x. If L is a register, update its register descriptor to indicate that it holds the value of x, and remove x from all other register descriptors.
- If the current values of y (or z) has no next uses, and is not live on exit from the current basic block, and is in a register, then alter the register descriptor to free the register.

Code Generation Algorithm – continued

Code generation is done in a similar way for `x := op y`

For a statement like `x := y`

- If y is in a register, modify the address and register descriptors to say that x is now there. If y has no next use, modify the register descriptors to say that y is no longer there.
- If y is in memory, and x has no next use in the block, generate `MOV y, x`
- If y is in memory, and x has a next use in the block, use 'getreg' to find a register in which to load y, and make that register the location of x.

Example

Generating code for

```
d:= (a-b) + (a-c) + (a-c)
```

3-address code	target code	register descriptor	address descriptor
		all registers empty	
t := a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u := a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R1, R0	R0 contains v R1 contains u	v in R0 u in R1
d := v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 and in memory

Optimisation

Optimisation means using algorithms and heuristics with the aim of improving the code generated by the compiler.

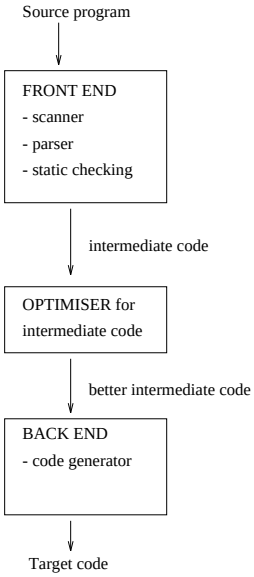
An immense number of these are known.

Some are used in almost all production compilers.

Others are only used in special *optimising compilers*.

The term "optimising compiler" is not strictly accurate. Optimisation includes some undecidable problems, and some whose solution is prohibitively expensive.

Structure of an Optimising Compiler



Criteria by which Optimisations are Judged (Fischer, LeBlanc)

Optimisations are judged by their safety and profitability.

For example suppose the loop

```
while J < I loop
    A(J) := 10/I;
    J := J+2;
end loop;
```

is transformed to

```
t1 := 10/I;
while J < I loop
    A(J) := t1;
    J := J+2;
end loop;
```

What if I=0 or J≥I initially?

Classification of Optimisations

1. Source Language Optimisations are

- performed by semantic routines,
- language specific and
- machine independent

Source Language Optimisations

Example: Loop Unscrolling

```
for I in 1..10 loop
    A(I) := 2*I;
end loop;
```

is changed to

```
A(1) := 2;
A(2) := 4;
...
A(10):= 20;
```


Classification of Optimisations

2. Target Code Optimisations

- exploit target machine architecture and
- are source language independent.

Example:

```
MPY 2,Y
```

is changed to

```
SHIFTLEFT y
```


Classification of Optimisations

3. Intermediate Representation Optimisations

- depend solely on the intermediate representation,
- can be shared by many compilers for different source languages or target machines or both
- entail processing flow graphs of basic blocks

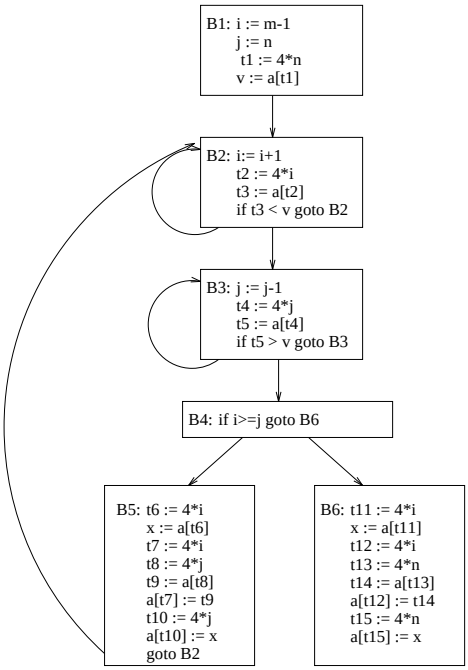
Flow Graph for a quicksort program (Aho, Sethi and Ullman)

Consider the following flow graph for a quicksort program.

The graph has been taken from Aho, Sethi and Ullman, page 591, Figure 10.5; the optimisation examples that follow have all been taken from the same source.

The code assumes that each element of the array a takes 4 bytes.

Flow Graph for a quicksort program



Some Useful Optimisations

- elimination of common subexpressions
- copy propagation
- dead code elimination
- loop optimisations
  - code motion
  - induction variable elimination
  - strength reduction

Local elimination of common subexpressions

Consider block B5 from the quicksort program

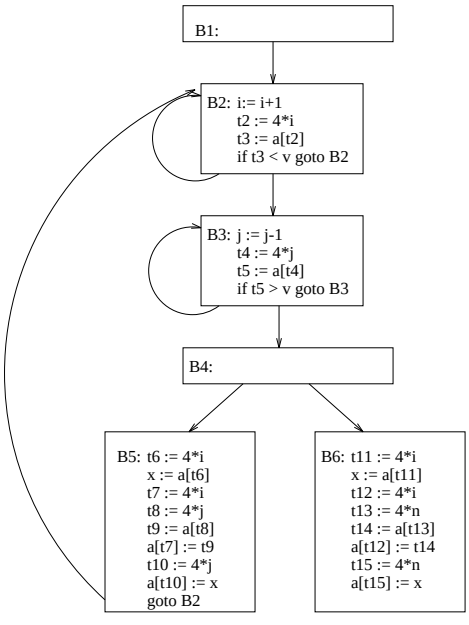
Before

```
B5: t6 := 4*i
    x  := a[t6]
    t7 := 4*i
    t8 := 4*j
    t9 := a[t8]
    a[t7] := t9
    t10 := 4*j
    a[t10] := x
    goto B2
```

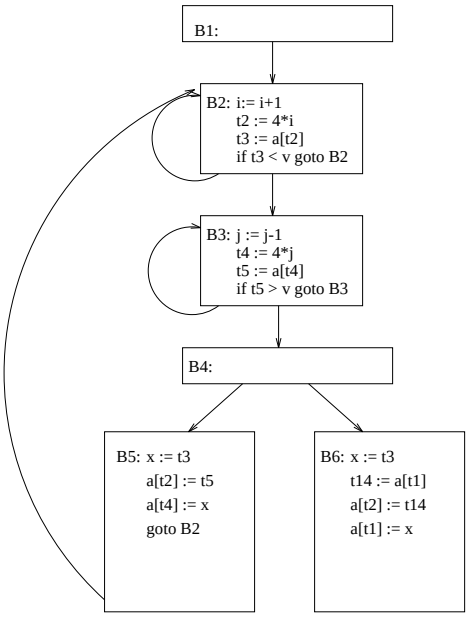
After

```
B5: t6 := 4*i
    x  := a[t6]
    t8 := 4*j
    t9 := a[t8]
    a[t6] := t9
    a[t8] := x
    goto B2
```


Quicksort flow graph before global elimination of common subexpressions



Quicksort flow graph after global elimination of common subexpressions



Copy propagation

Following a copy statement `f := g` use `g` in place of `f` wherever possible.

Consider block B5 after global elimination of common subexpressions

Before

```
B5: x := t3
    a[t2] := t5
    a[t4] := x
    goto B2
```

After

```
B5: x := t3
    a[t2] := t5
    a[t4] := t3
    goto B2
```

So what?

Dead code elimination

A variable is live at a point in a program if its value is subsequently used; otherwise it is dead.

A piece of code is live if the value(s) it computes is (are) used subsequently; otherwise it is dead.

Copy propagation often turns the copy statement into dead code, which can be eliminated.

Block B5 again

Before

```
B5: x := t3
    a[t2] := t5
    a[t4] := t3
    goto B2
```

After

```
B5: a[t2] := t5
    a[t4] := t3
    goto B2
```


Code Motion

Move code out of a loop to a point before the loop; this avoids recomputing values.

Before

```
Start: m := 10
       n := 50
Loop:  a[n] := m*4
       b[n] := m+15
       n := n-1
       if n >= 1 goto Loop
```

After

```
Start: m := 10
       n := 50
       t1 := m*4
       t2 := m+15
Loop:  a[n] := t1
       b[n] := t2
       n := n-1
       if n >= 1 goto Loop
```

Note: this is not part of the quicksort example!

Induction Variable Elimination

In the loop round B2 in the quicksort flow graph, every time `i` is incremented

```
t2 := 4*i
```

is recalculated. This means that the relationship

```
t2 = 4*i
```

is maintained.

`i` and `t2` are called induction variables

It is often possible to eliminate all but one of the induction variables in a loop.

However, before this can be done in the quicksort example, some further manipulation is needed.

Strength Reduction

Replace slower operations (like multiplication) by faster ones (like addition) in a loop.

In block B2, provided

```
t2 = 4*i
```

the statements

```
i := i+1
t2 := 4*i
```

can be replaced by

```
i := i+1
t2 := t2+4
```

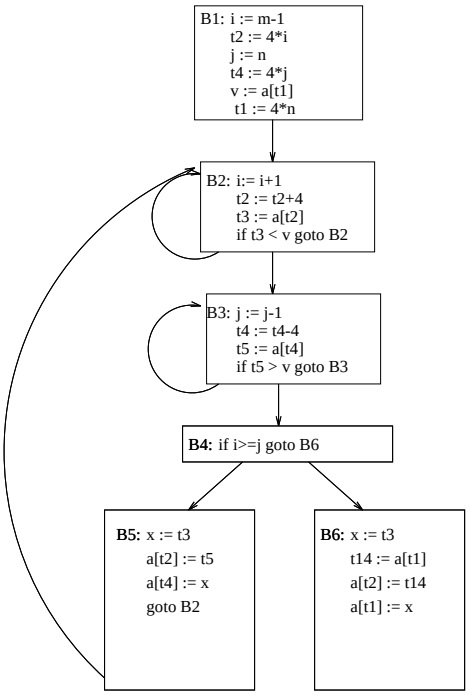
It remains to establish

```
t2 = 4*i
```

at the point of entry to B2. This is done by modifying B1, where i is initialised.

A similar transformation can be carried out in B3.

Strength Reduction in blocks B2 and B3



Back to induction variable elimination

Consider `i` and `j` in B2, B3, B4 and B5

After strength reduction in B2 and B3, `i` and `j` are only used in the test in B4.

Because

```
t2 = 4*i
```

and

```
t4 = 4*j
```

this test can be replaced by

```
t2 >= t4
```

Also, the statements

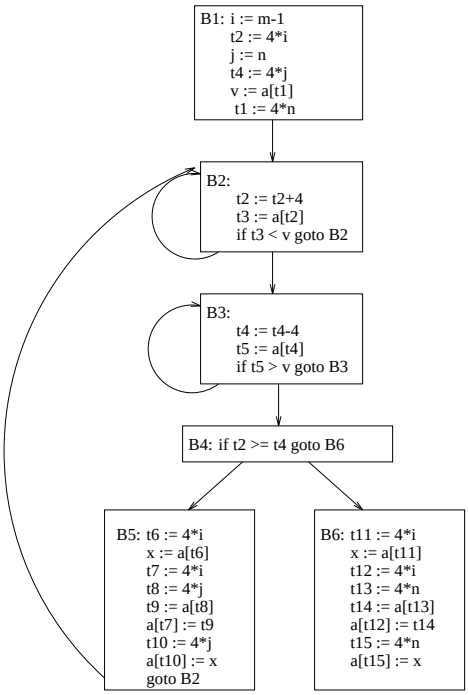
```
i := i+1
```

in B2 and

```
j := j-1
```

in B3 are now dead code, and can be eliminated.

The final flow graph



Compilation Systems - Summary

- Structure of a compiler
- Lexical analysis (scanning)
- Syntactic analysis (parsing)
- Semantic analysis (context sensitive checking)
- Code Generation
- Optimisation