10 High level optimisation 1

Having constructed a parse tree and determined its order of evaluation, one approach is to generate machine code directly. This can be done by associating code generation sequences with nodes in the parse tree, as procedural attributes. Alternatively, this can be viewed as a "tree walking" phase, processing all or part of the parse tree. It is important to decide how much of the parse tree must be built before subsequent processing can begin. In some languages, such as Fortran 66, only one statement at a time needed to be handled. In more structured languages, like Pascal, where statements may be nested within one another to arbitrary depths, this may not be enough. Often the whole nested system of statements from the outermost to the innermost are first analysed. This causes problems of storage on small machines.

A more general problem occurs when we wish to produce high level optimisations before generating code. These require whole loops, for instance, to be optimised before the program structure is lost. In the case of global optimisations the entire program is needed. This led to many optimising compilers generating a high level intermediate code, which preserved much of the parse tree's information, but which could be written serially to a working file.

The use of annotated partial trees has been found to be possible as a means of encoding the necessary information. The Ada intermediate code Diana uses this approach. Today, with larger address spaces common and memory cheaper, the holding of the complete annotated parse tree is becoming more generally practical.

The most widespread high level forms, however, are probably triples, sometimes called triads, or quadruples, called quads. Although language specific information is usually included, the general approach is language independent. Tuples are written into files or working areas between passes, minimising the memory needed for them. A basic triple contains three fields:
operator field operand field1 operand field2

Not all operator values require two operands, but none are allowed to use more. Quads are similar but allow three address instructions to be encoded. The operators define the set of operations necessary to implement the semantics of the language. They are similar to the instructions of a processor's order code, but are defined to suit the language, i.e. they are like a RISC machine's order code. Typically they would include:

- logical and arithmetic operations;
- comparisons;
- conversions;
- jumps, cases and labels;
- call, return, exit etc;
- stack manipulation;
- addressing and referencing;
- language specific operations, such as Pascal's sets and Fortran's complex values;
- stop, trap, bound and unassigned checking

The language specific and other complex instructions might often be expressed in terms of sequences of simpler triples. This is not initially desirable, however, as their explicit representation allows a more compact encoding of a program and facilitates high level optimisations. Triples are not generated in their final form, but rather will be massaged repeatedly before final low level code is generated from them.

The operands might include:

- literals;
- pointers to symbol table entries for identifiers, non-simple literals and internal temporary variables;
- pointers to other tuples.

The last group allows both jump/label association (although this might be done by symbol table entries) and, most importantly, the use of a value from the evaluation of another, preceding, tuple. This use of another tuple to supply the value of an operand allows the parse tree structure to be preserved for many parts of the program. In fact,triples represent the parse tree translated into a binary tree.

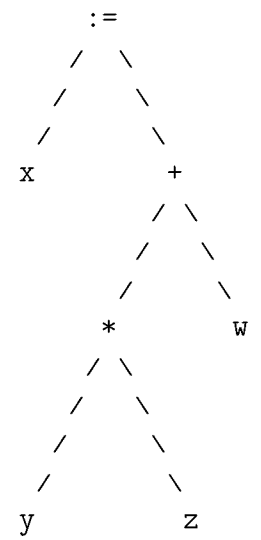
An interesting consequence is that it is fairly easy to regenerate a source program with an identical structure, although sometimes using simpler constructions. This is really useful as a program development aid, since, after optimisation, the program's structure may be radically different. Constructing a source level form of the optimised program from the tuples can help understanding and make sense of diagnostics.

Another possible use of this approach is to regenerate a source program in a different language, as long as mappings between both source and object high level languages and the set of tuples are defined.

Let us consider a simple example. Consider the statement

```
x:= y * z + w
```

which could produce the parse tree:



A corresponding set of triples would be:

	Operator	Operand1	Operand2
1	address	x.entry	null
2	multiply	y.entry	z.entry
3	add	t2	w.entry
4	assign	t1	t3

The meanings of the triples are fairly clear, as is their derivation from the parse tree. Note first the order of evaluation - left to right, depth first. This matches a translation scheme, for instance. The sequence matches the nodes of the parse tree and could be generated by a simple, recursive algorithm.

The operators used are obvious, except, perhaps, for `address`. This generates a reference to its one operand, rather than its value.

The operands are either pointers into the symbol table, such as `x.entry`, or pointers to the results of earlier triads, such as `t2`, which points to the sub-expression `y * z`.

Processing triads

The reason for generating triads or quads is to allow optimisation. They might also be used in bootstrapping, although they are not really best suited to this.

Generating code is very straightforward, as the order of evaluaion is fixed. The list of triples is processed sequentially, except that reference back to earlier triples is allowed. For this reason, once it has been processed, a triple record must be modified to describe the location of any temporary (intermediate) values generated. This typically means describing a register, stack position or memory location. Where evaluation of a triple involves only constants, the temporary value would be a literal. (In practice many compilers would "fold out" constant expressions when processing or building the parse tree.)

The backward referencing of triples must also be coupled to the register- or temporary cell-tracking mechanisms used in code generation. If an intermediate value is subsequently moved, for instance when freeing registers, the triple recording its location must be updated. There must also be a way of deciding when to "free such a value. It may be that its first use in a subsequent triple signals its future redundancy, but this may not always be sufficient.

Three address code

When writing quads we can use a more readable, high level language like form. This is usually called three address code. In fact triples can be written in a similar way, but each instruction is more limited. The advantage of three address forms is that certain common optimisations are more evident in this format.

Three address codes have two operands, one operator and a destination field. This corresponds to the VAX 3-address form of many instructions, to extended instruction forms in Motorola 68000 and Intel 8086 families of microprocessors and to some RISC formats, such as MIPS. They were originally used to correspond to IBM XA and other mainframe forms.

Source reconstruction from three address instructions is also much more direct, although there is the restriction that only one operator can appear in each instruction. Thus reconstructed source may have to contain additional variables, even if the regenerator elides many of these.

We shall use three address form instructions in dealing with optimisation, but the techniques described will be applicable to triples as well.

```
A := B + C * D
```

Would become:

```
T1 := C mul D
A  := T1 + B
```

Control Flow Analysis

Naïve code generation schemes which work directly from the parse tree assume that the order of execution is fixed by the high level language source code. In practice there are many ways in which this may be sub-optimal. Before considering in detail the types of optimisation which can be performed, it is worth looking at how the tuples in high level intermediate code can be analysed.

Essentially there are two forms of analysis, which can be used together to optimise code at a high level. These are:

- control flow analysis
- data flow analysis.

Initially we consider control flow, as this is relatively simple. Later data flow will need to be tackled, to support some of the optimisations. To understand control flow analysis, consider the following three address code program.

```
(1) PROD := 0
(2) I := 1
(3) T1 := 4*I
(4) T2 := addr(A) - 4
(5) T3 := T2[T1]
(6) T4 := addr(B) - 4
(7) T5 := T4[T1]
(8) T6 := T3 * T5
(9) Prod := Prod + T6
(10) I := I + 1
(11) if I  20 goto (3)
```

This represents

```
begin
   PROD := 0;
   I := 1;
   do
   begin
      PROD := PROD + A[I] * B[I];
      I := I + 1
   end
   while I  20
end
```

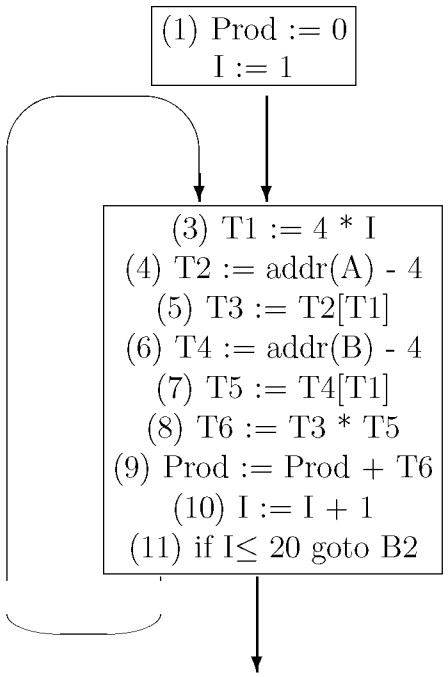
The control flow analysis of such a sequence operates in terms of basic blocks, which are connected to form a flow graph. A basic block is a sequence which has only one entry point, which is at its start.

Following Aho and Ullman we define the following algorithm:

1. The first statement is a leader;
2. Any statement which is the destination of a jump is a leader;
3. Any statement following a conditional jump is a leader.

A basic block is a leader and all statements up to but not including the next leader or the end of the program, whichever is first. Any statements not in a block are *dead code* and can be removed without altering the program.

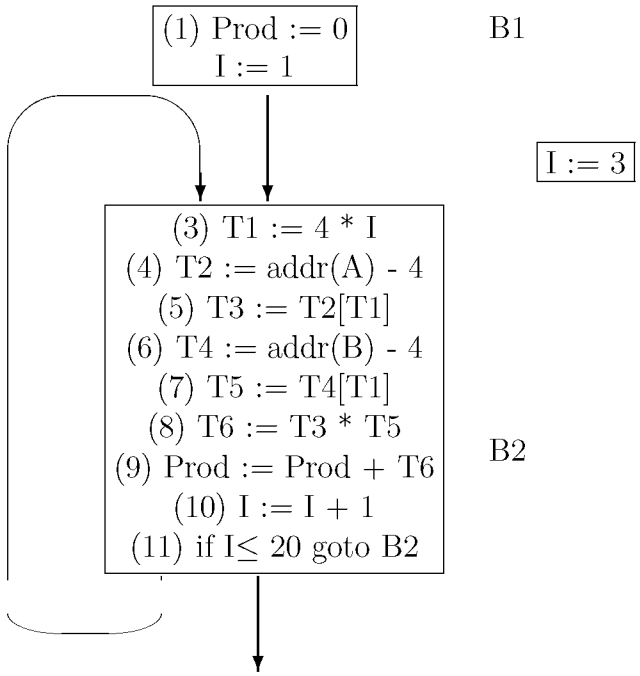
A flow graph takes these blocks as its nodes and connects them according to whether one block could follow from another in the execution of the program. Any block not reachable from the starting block of the program can be removed.



Thus the program above is:

Taking the following, slightly modified version, we discover some redundant statements:

```
begin
   PROD := 0;
   I := 1;
   goto Loop;
   I := 3;
Loop:
   do
   begin
      PROD := PROD + A[I] * B[I];
      I := I + 1
   end
   While I  20
end
```



An example of an optimisation - code motion

One of the commonest optimisations is to move code which is not affected by the outcome of loops from within those loops to the preceding code. Ignoring for the moment the issue of how this can be determined, the following is an example of such code motion in our example. This makes two assumptions:

1. The addresses involved cannot alter between loop iterations (i.e. are loop invariant), yet are not necessarily constant (i.e. cannot be evaluated at compile time);

2. The loop is executed on average at least once each time it is reached (otherwise the movement will result in worse average performance).