Abstract Syntax Trees

(From Compiler Construction by W.M. McKeeman).

The Abstract Syntax Tree (AST)

An abstract syntax tree can be represented using labels on the tree arcs.

For example, the statement:

```
if a + 1 > b then x=x+1; y=x+y fi
```

has the following parse tree:

```
< S >
    /   \
   /     \
  if < E >  then < Slist > fi
    |       /   \   /
   |      /     \
    |     /       \
   |    /         \
  < E > < S > < Slist >
    |               |
   /               /
  < E > < E > < S >
      |           |
     /           /
    < T > + < T >
      |       |
     /       /
    < P > < P >
```

The corresponding AST is simpler:
Simple Code Generation

Three address representation

Source-to-source transformations may be applied to the AST.

The next step is to generate low-level code, usually three address code.

In many cases, the compound statements (e.g. for or do loops and if statements) are transformed into sequences of instructions which include three-address operations as well as cmp and jump instructions. The original structure of the program (e.g. loops) is recovered by analyzing the program flow graph.

In some cases (e.g. SUIF) information on the high-level constructs is kept as annotations to the three address representation of the program.

Program Flow Graph

A program flow graph is also necessary for compilation. The nodes are the basic blocks. There is an arc from block $B_1$ to block $B_2$ if $B_2$ can follow $B_1$ in some execution sequence.

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halts or possibility of branching except at the end.

**Algorithm BB: Basic Block Partition**

Input: A program PROG in which instructions are numbered in sequence from 1 to $|PROG|$. INST(i) denotes the ith instruction.

Output
1. the set of LEADERS of initial block instructions.
2. for all x in LEADERS, the set BLOCK(x) of all instructions in the block beginning at x
3. Method:
begin
LEADERS := \{1\}
for j := 1 to |PROG| do
    if INST(j) is a branch then
        add the index of each potential target to LEADERS
    fi
od
TODO := LEADERS
while TODO \neq \emptyset do
    x := element of TODO with smallest index
    TODO := TODO - {x}
    BLOCK(x) := \{x\}
    for i := x+1 to |PROG| while i \neq LEADERS do
        BLOCK(x) := BLOCK(x) \cup \{i\}
    od
od
end

A Simple Code Generator

Our objective is to make a reasonable use of the registers when generating code for a basic block. Consider for example:

\[
\begin{align*}
t &= a - b \\
u &= a - c \\
v &= t + u \\
d &= v + u
\end{align*}
\]

Each instruction could be treated like a macro which expand into something like:

\[
\begin{align*}
l & R1, a \\
sub & R1, b \\
st & R1, t \\
l & R1, a \\
sub & R1, c \\
st & R1, u \\
l & R1, t \\
add & R1, u \\
st & R1, v \\
l & R1, v \\
add & R1, u \\
st & R1, d
\end{align*}
\]

The resulting code is not very good. Only one register is used and there is much redundant code. A more sophisticated algorithm is needed.
Target Machine Language

We use the following target machine language:

The machine has two address instructions of the form

\[
\text{op destination,source}
\]

The destination has to be a register. The machine has several op-codes including

- \text{1} \quad \text{(move source to destination)}
- \text{add} \quad \text{(add source to destination)}
- \text{sub} \quad \text{(subtract source from destination)}

There is also a store (st) instruction.

The source (destination in the case of the store instruction) can be

1. an absolute memory address (a variable name is used),
2. a register,
3. indexed (written \(cR\), where \(c\) is a constant and \(R\) a register),
4. indirect (written \(*R\) where \(R\) is a register), and
5. immediate (denoted \(#c\) where \(c\) is a constant)

Algorithm SCG: A Simple Code Generator

\text{Input:}
1. A basic block of three address statements.
2. A symbol table SYMTAB

\text{Output:}
1. Machine code

\text{Intermediate:}
1. A register descriptor RD(R). The set variables whose values are in register \(R\)
2. An address descriptor AD(variable). The set of locations (register, stack, memory) where the value of variable can be found.
Method:

begin

for each instruction $I$ in basic block do

if $I$ is of form $(x := y \text{ op } z)$ then

$L := \text{getreg}(y, I)$;

if $L$ not in $\text{AD}(y)$ then

$y' := \text{select}(y)$

generate($L, y'$)

$z' := \text{select}(z)$

generate($\text{op } L, z'$)

end

AD($y$) := $\text{AD}(y) - \{L\}$

for all $R$ in REGISTERS do

RD($R$) := $\text{RD}(R) - \{x\}$

od

if NEXTUSE($y, I$) is empty and

LIVEONEXIT($y$) is false then

for all $R$ in REGISTERS do

RD($R$) := $\text{RD}(R) - \{y\}$

od

fi

...same as above for $z$...

elseif $I$ is of form $(x := \text{ op } y)$ then

... similar code as above...

elseif $I$ is of form $(x := y)$ then

if there is register $R$ in $\text{AD}(y)$ then

RD($R$) := $\text{RD}(R) + \{x\}$

end

else

$L := \text{getreg}(y, I)$

generate($L, L, y$)

RD($L$) := $\{x, y\}$

AD($x$) = $\text{AD}(x) + \{L\}$

fi

for all $R$ in REGISTERS do

forall $v$ in $\text{RD}(R)$ do

if LIVEONEXIT($v$)

and $\text{SYM}TAB.\text{loc}(v)$ not in $\text{AD}(v)$ then

(only once for each $v$)

generate($\text{st } R, \text{SYM}TAB.\text{loc}(v)$)

fi

od

od

end

The $\text{select}$ routine returns a register $R$ if the value of the parameter is in $R$

otherwise it returns the memory location containing the value of the parameter.
### NEXTUSE and LIVEONEXIT

The LIVEONEXIT(\(v\)) boolean value is true if \(v\) may be used after the basic block completes. It is computed using global flow analysis techniques to be discussed later in the course.

The NEXTUSE(\(v, I\)) is the statement number where \(v\) is used next in the basic block. It is empty if \(v\) is not used again.

NEXTUSE(\(v, I\)) can be computed as follows:

```plaintext
forall variables \(v\) in basicblock do
    USE(\(v\)) := {}
    od
forall instructions \(I\) in basic block in reverse order do
    if \(I\) is of form (\(x := y \text{ op } z\)) then
        NEXTUSE(\(x, I\)) = USE(\(x\))
        NEXTUSE(\(y, I\)) = USE(\(y\))
        NEXTUSE(\(z, I\)) = USE(\(z\))
        USE(\(x\)) = {}
        USE(\(y\)) := \(\{I\}\)
        USE(\(z\)) := \(\{I\}\)
    elseif ...
    fi
    od
```

### getreg(\(y, I\))

```plaintext
if there is register \(R\) such that RD(\(R\)) = \(\{y\}\) and
NEXTUSE(\(y, I\)) is empty
    then
        return (\(R\))
    fi

if there is \(R\) in REGISTERS such that RD(\(R\)) is empty
    then
        return (\(R\))
    fi

\(R :=\) getanyregister()
forall \(v\) in RD(\(R\)) do
    AD(\(v\)) := AD(\(v\)) - \(\{R\}\)
    if SYMTAB.loc(\(v\)) is not in AD(\(v\)) then
        generate(\(\text{st } R, SYMTAB.loc(\(v\))\))
        AD(\(v\)) := AD(\(v\)) + \{SYMTAB.loc(\(v\))\}
    fi
    od
return (\(R\))
```

Note; SYMTAB.loc(\(v\)) is the location in memory where variable \(v\) is located. loc is a component of the symbol table.
Two special operators

The [ ] operator is used to index a (one dimensional) array

\[ a := b[i] \]

can be translated as

(1)

1 \ R, b(R)

if \( i \) is in register \( R \)

(2)

1 \ R, M
1 \ R, b(R)

if \( i \) is memory location \( M \)

(3)

1 \ R, S(A)
1 \ R, b(R)

if \( i \) is in stack offset \( S \).

The * operator is similar. For example, (2) above is replaced by

1 \ R, M
1 \ R, *R

The DAG Representation of Basic Blocks

The previous algorithm aims at improving the quality of the target code, but only with respect to register utilization.

There are a number of other issues in the generation of efficient code. One of them is the elimination of redundant computation.

Thus, in the sequence

\[
\begin{align*}
  x & := b*c + b*c*2.0 \\
  b & := b*c*3.0 \\
  y & := b*c + 1.0
\end{align*}
\]

the same computation of \( b*c \) is done three times (the fourth occurrence is not the same because \( b \) is reassigned).

The following algorithm identifies and removes common subexpressions using a DAG as intermediate representation. This algorithm assumes that there are no array element or pointers in the basic block.

Traditional compiler optimizations do not deal naturally with array references and pointers.
Algorithm DAG: Constructing a DAG

Input: A basic block of three address statements. No pointers or array references.

Output: A DAG where each node \( n \) has a value, \( \text{VALUE}(n) \), which is an operator in the case of an interior node or a variable name if the node is a leaf. Also, each node \( n \) has a (possibly empty) list of identifiers attached, \( \text{ID}(n) \).

Method:

\[
\begin{align*}
\text{begin} & \quad \text{for each instruction} \ I \ \text{in basic block do} \\
& \quad \text{if} \ I \ \text{is of form} \ (x := y \ op \ z) \ \text{then} \\
& \quad \quad \text{Find a node,} \ ny, \ \text{such that} \ y \ \epsilon \ \text{ID}(ny) \ (\text{only one can exist}). \ \text{If it cannot be found, create a leaf with} \ \text{VALUE}(ny) = \{y\}. \\
& \quad \quad \text{... same for} \ z, \ \text{node is} \ nz \ ... \\
& \quad \quad \text{Find or, if not found, create node} \ m \ \text{such that} \ \text{VALUE}(m) = \text{op} \ \text{and} \ ny \ \text{and} \ nz \ \text{are resp. its left and right child.} \\
& \quad \quad \text{if} \ \text{there is} \ p \ \text{such that} \ x \ \epsilon \ \text{ID}(p) \ \text{then} \\
& \quad \quad \quad \text{ID}(p) := \text{ID}(p) - \{x\} \\
& \quad \quad \text{fi} \\
& \quad \quad \text{ID}(m) := \text{ID}(m) + \{x\} \\
& \quad \text{elseif} \ I \ \text{is of form} \ (x := \text{op} \ y) \ \text{then} \\
& \quad \quad \text{... similar code as above...} \\
& \quad \text{elseif} \ I \ \text{is of form} \ (x := y) \ \text{then} \\
& \quad \quad \text{Find a node,} \ ny, \ \text{such that} \ y \ \epsilon \ \text{ID}(ny) \ (\text{only one can exist}). \ \text{If it cannot be found, create a leaf with} \ \text{VALUE}(ny) = \{y\}. \\
& \quad \quad m := ny \\
& \quad \quad \text{if} \ \text{there is} \ p \ \text{such that} \ x \ \epsilon \ \text{ID}(p) \ \text{then} \\
& \quad \quad \quad \text{ID}(p) := \text{ID}(p) - \{x\} \\
& \quad \quad \text{fi} \\
& \quad \quad \text{ID}(m) := \text{ID}(m) + \{x\} \\
& \quad \text{fi} \\
& \text{od} \\
\text{end}
\end{align*}
\]
With the DAG it is easy to determine which identifiers have their values used in the block. These are the identifiers for which a leaf is created.

Also, it is easy to determine the statements that compute values that could be used outside the block. These are the statements whose associated node, \( m \), still has its left hand side, \( x \), in \( \text{ID}(m) \) at the end of the algorithm.

To improve the chances of finding common subexpressions, commutative operations should be normalized. For example, when both operands are variables, alphabetical order could be used. Also, if one of the operands is a leaf and the other an internal node, the leaf could be placed on the left.

Constant folding can be applied by replacing the nodes that evaluate to a constant, say \( c \), with a node \( m \) such that \( \text{VALUE}(m) = c \).

The previous algorithm was introduced by Cocke and Schwartz and is known as “Value Numbering of Basic Blocks”.

When there are references to array elements and to pointers, we need to make sure that:

1. Common subexpressions are properly identified
2. The order of instructions generated from the DAG is correct.

To make sure that common subexpressions are correctly identified an extra bit is added to each node. Every time there is an assignment to an element of an array, all nodes representing elements of that array are killed by setting the bit. The ID of a killed node cannot be extended with new variable names. That is, they cannot be recognized as common subexpressions.

Also, when there is an assignment of the form \( *p := a \) all nodes in the DAG must be killed if we don’t know what \( p \) might point to.

Similar observations could be made about formal parameters when the language allows aliasing.

To guarantee correct order of generated instructions, the DAG could be extended with arcs that enforce the following rules:

1. Any two references to an array one of which is a write, must be performed in the orginal order.
2. Any two references, if one is a write and at least one of the references is through a pointer must be performed in the original order.
A Heuristic Ordering for DAGs

A simplified list of quadruples can be generated from the DAG. This list can be generated in any order that is a topological sort of the DAG.

The order has clearly an effect on the quality of the code. Consider, the example on the following two transparencies. By evaluating the value used as the left operand just before it is needed, register allocation is better and therefore no spill code is needed.

A possible strategy is to try to make the evaluation of a node immediately follow the evaluation of its left argument.

\[
\text{while unlisted interior nodes remain do} \\
\quad \text{select an unlisted node } n, \text{ all of whose parents have been listed} \\
\quad \text{list } n \\
\quad \text{while the left child } m \text{ of } n \text{ has no unlisted parents and is not a leaf do} \\
\quad \quad \text{list } m \\
\quad \quad n := m \\
\quad \text{od} \\
\text{od}
\]

The order of evaluation is the reverse of the list produced by this algorithm.

Register Allocation Example

Consider the following assignment statement:

\[x = (a*b) + ((c*d) + (e*f));\]

Which can be represented as:

\[
\begin{align*}
& x \\
& \quad + \\
& \quad \quad + \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
\end{align*}
\]

Assume that two registers are available. Starting from the left a compiler would generate the following sequence:

\[
\begin{align*}
1 & \quad R1, a \\
\text{mpy} & \quad R1, b \\
1 & \quad R2, c \\
\text{mpy} & \quad R2, d
\end{align*}
\]

For the next operation \((e*f)\) no register is available and spill is needed. There are two possibilities for the rest of the code:
st R2, temp
l R2, e
mpy R2, f
add R2, temp
add R2, R1
st R1, x

or

st R1, temp
l R1, e
mpy R1, f
add R1, R2
add R1, temp
st R1, x

However, the spill can be avoided if the expression is evaluated in a different order:

l R1, c
mpy R1, d
l R2, e
mpy R2, f
add R1, R2
l R2, a
mpy R2, b
add R2, R1
st R2, x

Labeling Algorithm

After the DAG is generated, it can be transformed into a forest by creating a tree out of each common subexpression:

Let us now discuss how to generate code from a tree.

First, the expression represented by the tree can be transformed using associativity and commutativity. The Fortran 77 standard allows the compiler to apply these rules as long as the order specified by the parentheses is followed (that is a+b+c can be evaluated in any order, but b+c has to be evaluated first in a+(b+c)). These transformations are sometimes useful to improve the register utilization. Also, parallelism can be increased by applying these transformations.
Another technique to improve register utilization is to change the order of evaluation of the expression *without applying any algebraic laws* (see example on register allocation from first class).

To this end we use an algorithm that labels each node $n$ of a tree with the minimum number of registers needed to compute the subtree rooted at $n$.

Let $n1$ and $n2$ be the two children of $n$. Then the label of $n$ is computed as follows:

$$\text{label}(n) = \begin{cases} \text{label}(n) & \text{if label}(n1)=\text{label}(n2) \\ \text{label}(n1)+1 & \text{else} \end{cases} \text{max}(\text{label}(n1),\text{label}(n2))$$

The label of a leaf is 1 if it is the left leaf of its operator and 0 otherwise.

With this labeling, an optimal sequence of instructions (shortest instruction sequence) can be generated for the tree using the following algorithm:

```plaintext
gencode(n):
if $n$ is a leaf then
generate( 1 top(reg.stack), VALUE(n))
elseif $n$ is an interior node with left child $n1$ and right child $n2$ then
  if Label(n2)=0 then
gencode(n1)
generate( ⊕ top(reg.stack), VALUE(n2))
  elseif 1 ≤ Label(n1) < Label(n2) and Label(n1) < r then
swap(reg.stack)
R := pop(reg.stack)
gencode(n1)
generate( ⊕ top(reg.stack), R)
push(reg.stack, R)
swap(reg.stack)
  elseif 1 ≤ Label(n2) ≤ Label(n1) and Label(n2) < r then
ngenencode(n1)
R := pop(reg.stack)
gencode(n2)
generate( ⊕ R, top(reg.stack))
push(reg.stack, R)
  else
ngenencode(n2)
T := pop(temp.stack)
generate( ± top(reg.stack), T)
gencode(n1)
push(temp.stack, T)
genencode(n))
generate( ⊕ top(reg.stack), T)
fi
fi
```