

Contents
Generating Abstract Syntax Tree
Generating Intermediate Code : TAC, SSA
Difference between Abstract Syntax tree and Parse tree
Constructing Syntax tree for Different statements
Symbol-Table

## Generating Abstract Syntax tree:

input file name : <test.c>

```
$gcc -fdump-tree-original-raw test.c
```

output: test.c.003t.original

```
$awk -f pre.awk test.c.003t.original |awk -f treeviz.awk > tree.dot
```

pre.awk and treeviz.awk transforms the file according to the dot format<graphviz package>  
dot : DDocument Template file

```
$ gedit pre.awk
```

```
#!/usr/bin/gawk -f
/^[^;]/{
gsub(/^[^@]/, "~@", $0);
gsub(/( *):( *)/, ":", $0);
print;
}
```

```
$gedit treeviz.awk
```

```
#!/usr/bin/gawk -f
BEGIN {RS = "~@"; printf "digraph G {\n node [shape = record];"}
/^[0-9]/{
s = sprintf("\n%s [label = \"{%s | {\", $1, $1);
for(i = 2; i < NF; i++)
s = s sprintf("%s | ", $i);
s = s sprintf("%s} }\"";", $i);
$0 = s;
while (/([a-zA-Z0-9]+):@[0-9]+)/{
format = sprintf("\1 \3\n %s:\1 -> \2;", $1);
$0 = gsub(/([a-zA-Z0-9]+):@[0-9]+)(.*)$/ , format, "g");
};
printf " %s", $0;
}
END {print "\n"}
```

```
$dot -Tpng tree.dot > tree.png
```

# Generating Intermediate Code:

3 forms : generic, gimple, rtl(register transfer language)

Intermediate code formats :

- a) Three address code (gimple file)
- b) SSA : static single assignment (ssa file)

**\$gcc -fdump-tree-all-graph test.c**

**\$ls**

a.out	test.c.007t.lower	test.c.015t.ssa.dot	test.c.036t.release_ssa	test.c.162t.cplxlower0.dot
test.c	test.c.007t.lower.dot	test.c.017t.inline_param1	test.c.036t.release_ssa.dot	test.c.169t.optimized
test.c.001t.tu	test.c.010t.eh	test.c.017t.inline_param1.dot	test.c.037t.inline_param2	test.c.169t.optimized.dot
test.c.003t.original	test.c.010t.eh.dot	test.c.018t.einline	test.c.037t.inline_param2.dot	test.c.249t.statistics
test.c.004t.gimple	test.c.011t.cfg	test.c.018t.einline.dot	test.c.161t.veclower	test.c.249t.statistics.dot
test.c.006t.omplower	test.c.011t.cfg.dot	test.c.033t.profile_estimate	test.c.161t.veclower.dot	
test.c.006t.omplower.dot	test.c.015t.ssa	test.c.033t.profile_estimate.dot	test.c.162t.cplxlower0	

test.c.004t.gimple : contains Three-address code

test.c.015t.ssa : contains SSA code

## Three-address Code :

- **Three-address code** (often abbreviated to TAC or 3AC) is an intermediate **code** used by optimizing compilers to aid in the implementation of **code**-improving transformations.
- Each TAC instruction has at most **three** operands and one operation (is typically a combination of assignment and a binary operator).

### Example:

$x = a + y * 10$

TAC :

$t1 = y * 10$

$t2 = a + t1$

$x = t2$

## SSA – Single Static Assignment Form:

- In compiler design, **static single assignment form** (often abbreviated as **SSA form** or simply **SSA**) is a property of an intermediate representation (IR), which requires that each variable is assigned exactly once, and every variable is defined before it is used.
- Compiler optimization algorithms which are either enabled or strongly enhanced by the use of SSA.
- Converting ordinary code into SSA form is primarily a simple matter of replacing the target of each assignment with a new variable, and replacing each use of a variable with the "version" of the variable reaching that point.

### Example :

$y := 1$

$y := 2$

$x := y$

SSA form:

$y_1 := 1$

$y_2 := 2$

$x_1 := y_2$

## Displaying Control Flow graph:

```
$dot -Tpng test.c.015t.ssa.dot > ssa.png
```

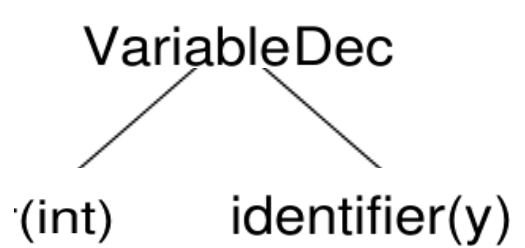
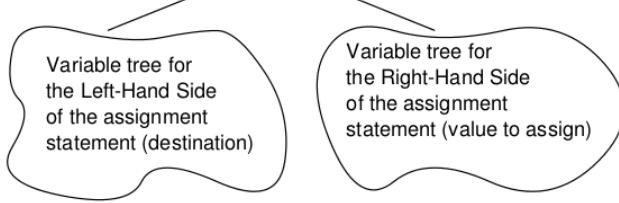
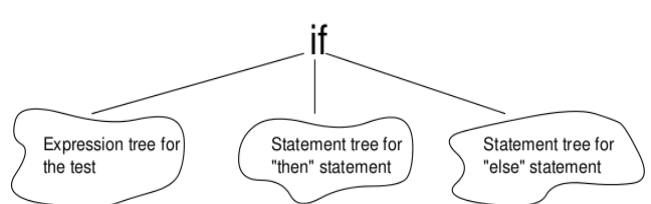
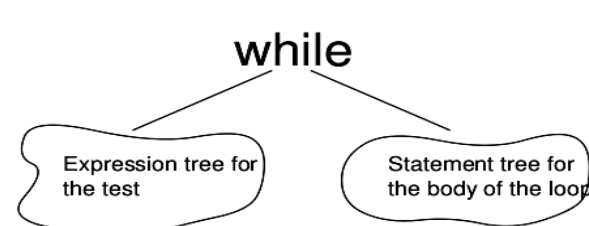
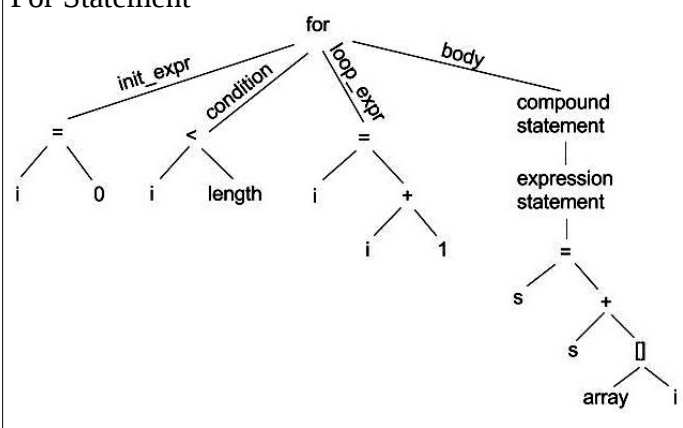
**Control flow graph** : is obtained by dividing the entire program into multiple blocks. Blocks are constructed in such a way that a control can enter the block only from the first instruction of the block and leave the block from the last instruction of the block only.

- The Structure is desired for optimizing the code. The optimized code is available in test.c.169t.optimized .
- Optimization : local (within a block) or global(for entire program).
- One of the ways of performing local optimization is : DAG optimization.

# Difference between Abstract Syntax tree and Parse tree

Parse Tree (Concrete Syntax Tree)	Abstract Syntax Tree
<ul style="list-style-type: none"> <li>Concrete syntax: what the programmer wrote</li> <li>tree representation of grammar derivation</li> </ul>	<ul style="list-style-type: none"> <li>Abstract syntax: what the compiler needs</li> <li>condensed form of parse tree.</li> <li>Abstract tree has less information than the concrete tree.</li> <li>Operators and keywords do not appear as leaves</li> <li>Chains of single productions are collapsed</li> </ul>
<pre> graph TD     S --&gt; IF     S --&gt; B     S --&gt; THEN     S --&gt; S1     S --&gt; ELSE     S --&gt; S2         </pre>	<pre> graph TD     IfThenElse[If-then-else] --&gt; B     IfThenElse --&gt; S1     IfThenElse --&gt; S2         </pre>
<p> <math>E \rightarrow E + T</math>  <math>E \rightarrow T</math>  <math>T \rightarrow T * F</math>  <math>T \rightarrow F</math>  <math>F \rightarrow \text{num} \mid \text{id} \mid (E)</math> </p> <pre> graph TD     E1[E] --&gt; E2[E]     E1 --&gt; P1[+]     E1 --&gt; T1[T]     E2 --&gt; T2[T]     T2 --&gt; F1[F]     F1 --&gt; 3     T1 --&gt; T3[T]     T1 --&gt; M1[*]     T1 --&gt; F2[F]     T3 --&gt; F3[F]     F3 --&gt; 4     F2 --&gt; 5         </pre> <p>Parse tree for <math>3 + 4 * 5</math></p> <p><math>3 * (4 + 5)</math></p>	<pre> graph TD     Plus1[+] --&gt; 3     Plus1 --&gt; Star1[*]     Star1 --&gt; 4     Star1 --&gt; 5         </pre> <p>Abstract Syntax Tree for <math>3 + 4 * 5</math></p> <p>What about parentheses? Do we need to store them?</p> <ul style="list-style-type: none"> <li>• Parenthesis information is store in the shape of the tree</li> <li>• No extra information is necessary</li> </ul> <pre> graph TD     Star2[*] --&gt; IL3[Integer_Literal(3)]     Star2 --&gt; Plus2[+]     Plus2 --&gt; IL4[Integer_Literal(4)]     Plus2 --&gt; IL5[Integer_Literal(5)]         </pre> <pre> graph TD     Star3[*] --&gt; Plus3[+]     Star3 --&gt; Plus4[+]     Plus3 --&gt; IL3[Integer_Literal(3)]     Plus3 --&gt; Plus5[+]     Plus4 --&gt; IL4[Integer_Literal(4)]     Plus4 --&gt; Plus6[+]     Plus5 --&gt; IL5[Integer_Literal(5)]     Plus6 --&gt; IL6[Integer_Literal(6)]         </pre> <p><math>(3 + 4) * (5 + 6)</math></p>
<p><math>((4))</math></p>	<p>Integer_Literal(4)</p>

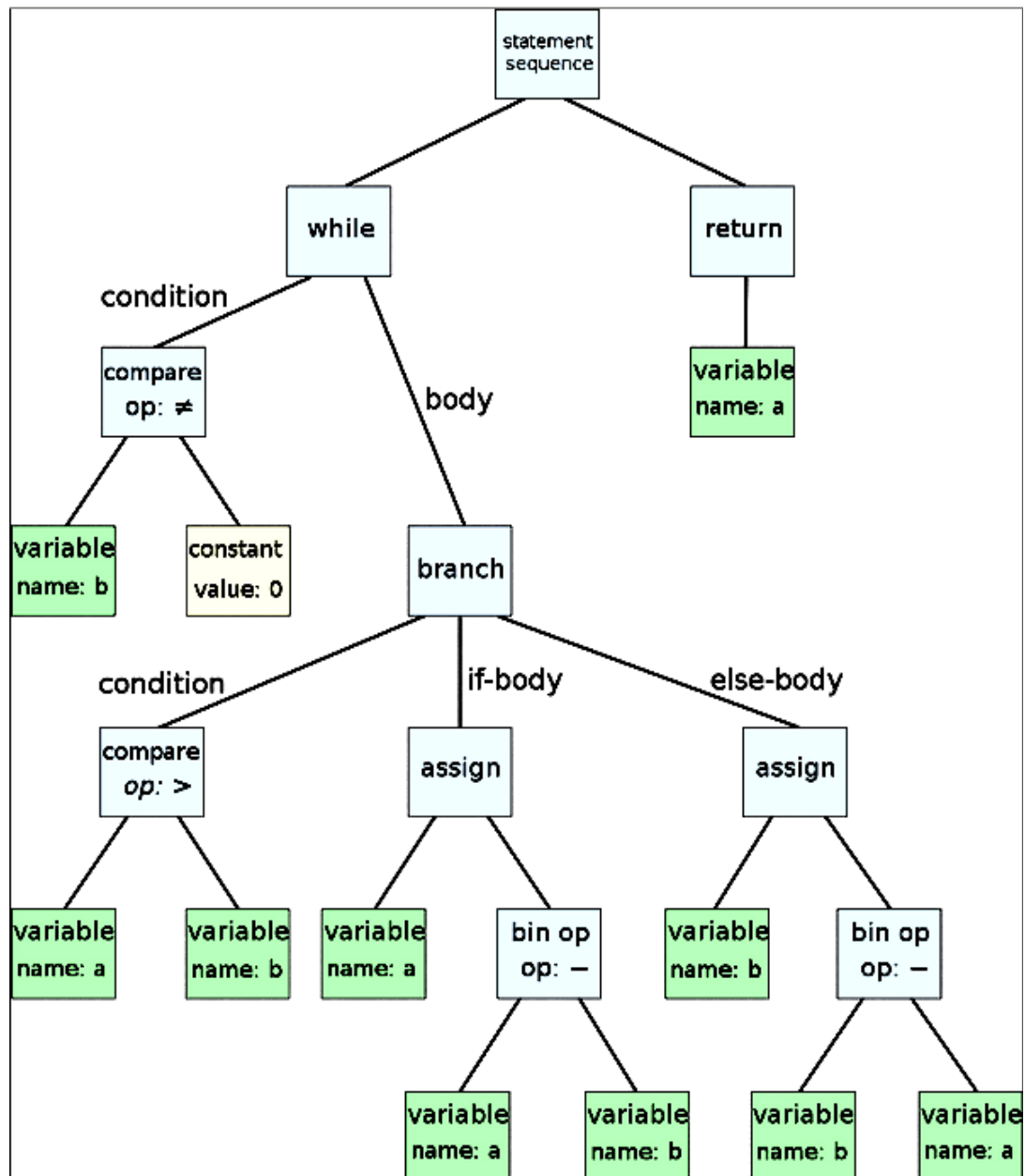
# Constructing Syntax tree for Different statements

<p>Decl -&gt; Type VarList  Type -&gt; int float char  VarList -&gt; VarList, id   id</p>	<p>AST – Variable Declaration  int y;</p>  <pre> graph TD     VariableDec --&gt; int_paren["(int)"]     VariableDec --&gt; identifier["identifier(y)"] </pre>
<p>AssignExpr -&gt; id = E  E → E + T  E → T  T → T * F  T → F  F → num   id   (E)</p>	<p>AST – assignment Statements  assign</p> 
<p><b>Sample CFG</b>  S -&gt; if(cond){S}   if(cond){S}else{S}    while(cond){S}    for(AssignExpr cond AssignExpr){S}    AssignExpr    S; S    epsilon    UnaryExpr    Decl    break ;   continue ;    return &lt;expr&gt; ;    goto &lt;id&gt; ;   epsilon</p> <p>cond -&gt; expr   expr logOp expr  expr -&gt; relexp   logexp   E  relexp -&gt; E relOp E  logexp -&gt; E logOp E  logOp -&gt;      &amp;&amp;  relOp -&gt; &lt;   &gt;   &lt;=   &gt;=   !=   ==</p> <p>AssignExpr -&gt; id = E;  E -&gt; E + T   T  T -&gt; T * F   F  F -&gt; id   num   (E)</p> <p>UnaryExpr -&gt; ++ UE   UE++   --UE   UE--  UE -&gt; id   (E)</p> <p>Decl -&gt; Type VarList;  Type -&gt; int float char  VarList -&gt; VarList, id   id</p>	<p>If Statement</p>  <p>While statement</p>  <p>For Statement</p> 

## Example

An abstract syntax tree for the following code for the Euclidean algorithm:

```
while b != 0
  if a > b
    a = a - b
  else
    b = b - a
return a
```



# Symbol Table

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table.

## Symbol Table Requirements

1. **Fast lookup.** The parser of a compiler is linear in speed. The table lookup needs to be as fast as possible.
2. **Flexible in structure.** The entries must contain all necessary information, depending on usage of identifier
3. **Efficient use of space.** This will require runtime allocation (dynamic) of the space.
4. **Handle characteristics of language** (e.g., scoping, implicit declaration)
  - Scoping requires handling entry of a local block and exit.
  - Block exit requires removal or hiding of the entries of the block.

## Symbol Table Operations

A symbol table, either linear or hash, should provide the following operations.

### **insert()**

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

## lookup()

lookup() operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.
- if the symbol declared multiple times.

## Scope Management

A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.

## Who Creates Symbol-Table Entries?

It is the semantic analysis phase which creates the symbol table because it is not until the semantic analysis phase that enough information is known about a name to describe it.

Many compilers set up a table at lexical analysis time for the various variables in the program, and fill in information about the symbol later during semantic analysis when more information about the variable is known.

With its knowledge of the syntactic structure of a program, a parser is often in a better position than the lexical analyzer to distinguish among different declarations of an identifier. In some cases, a lexical analyzer can create a symbol-table entry as soon as it sees the characters that make up a lexeme. More often, the lexical analyzer can only return to the parser a token, say `id`, along with a pointer to the lexeme. Only the parser, however, can decide whether to use a previously created symbol-table entry or create a new one for the identifier.