Kernel & non kernel items

Kernel items

The intial item $S' \to .S$ and all other items with no dot(.) at the beginning of RHS are called kernel items. for any production $A \to \alpha B\beta$

Kernels are

$A \to \alpha . B\beta$

$A \to \alpha B . \beta$

$A \to \alpha B\beta .$

Non kernel items

All items except $S' \to .S$ with dot(.) at the beginning of RHS are called non kernel items.

For example $A \to .\alpha B\beta$

---

the way of associating atributes to each of the Symbol in the grammer that provides us the order of translation is called SDT

Symbol table : A table with token + attribute of token

A syntax directed defination is a generalization of context free grammer in which.

(I) Each grammer symbol is associated with a set of attribute

(II) These set of attributes has 2 subsets called synthesized and inherited attributes

(III) Each production rule has set of semantic rules associated with it.

If the value of attribute depends upon its children then, it is Synthesized attribute

If the value of attribute depends upon it's parent or siblings then it is inherited attribute.

The passed tree showing the values of attribute at each node is called annotated parsed tree

---

Dependency graph is a directed graph that ~~show~~ has attributes as nodes and ~~their~~ depedency across the attributes as edges.

To convert syntax directed translation to syntax tree following implementation fn must be executed

① MakeNode (OP, left, right)

② Make leaf (id, entry)

③ Make leaf (num, val)

DAG → Directed acyclic graph
It is a graph it identifies common subexpression in an expression.

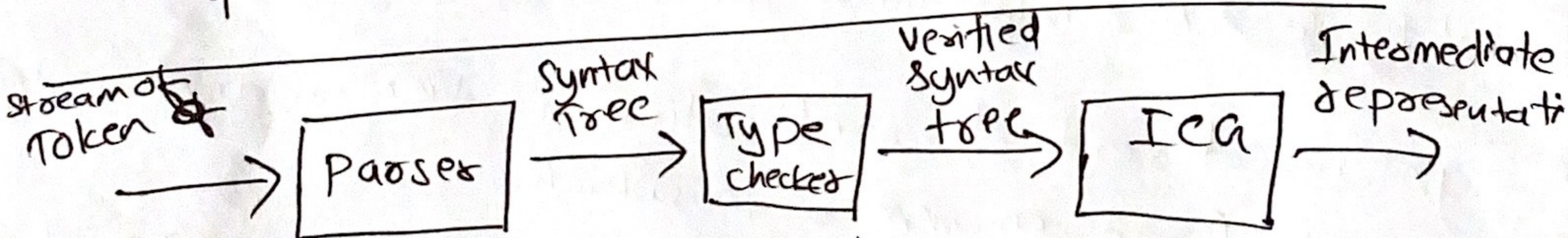

fig: Position of Type checker

---

static checking
program properties that can be checked at compile Time.

Equivalence of an Type expression.

Type checker finds if the 2 expressions are equivalent or not.

This type equivalence is of 2 categories
(1) structural equivalence
(2) Name equivalence.

(1) structural equivalence
Type expression are built from basic type and constructor, a natural concept of equivalence between two expressions. is structural equivalence.

For eg.
int is equivalent to int because same basic type
ptr(char) is equalent to ptr(char because same constructo ptr to char type.

(2) Name equivalence.
Two expressions are same equivalent if they can be represented by the same syntax tree, with same labels

for eg.
```
typedef struct node {
    int x;
} Node;

Node *f, *g;
struct node *l1, l2;
```

f & g are name equivalent
l1 & l2 are name equivalent

Transformation on basic blocks

we can apply the optimization process on a basic block. While optimization, there is no need to change the set of expressions computed by the block.

The basic block optimization is done in 2 ways.

- Structure-Preserving Transformations
- Algebraic Transformations.

1) Structure-Preserving Transformations.

The primary structure-preserving transformation on basic blocks are as follows.

a) Common sub-expression elimination
b) Dead code elimination
c) Renaming of temporary variables
d) Interchange of 2 adjacent independent statements.

a) common subexpression elimination.

suppose we have 2 expressions that compute the same values. In that case, we need to eliminate one of the expressions. This method is known as common sub expression elimination method.

Example.

```
p = a+b
q = c-d
r = x+y
s = c-d
```

Here, 2nd & 4th statements compute the same expression c-d so the basic block can be transformed as

```
p = a+b
q = c-d
r = x+y
s = q
```

b) Dead code elimination.

Whenever a programmer writes a program, there is a possibility that the program may have dead codes. The dead code may be a variable or the result of some expression computed by the programmer that may not have any further uses. By eliminating these useless things from a code, code will get optimized.

Example

A statement s=p+q appears in a block and s is a dead symbol. We can eliminate it. This elimination does not have any impact on the values of basic block.

c) Renaming of temporary variable

Consider, $x = a+b$

The given statement's value is stored in the temporary variable x that can be changed to another temporary variable y and changes all uses of x to y.
This kind of transformation is also known as a normal form block

d) Interchange of two independent adjecent statements

Suppose we have 2 statements as

$$p = a+b$$
$$q = x+y$$

The given statement can be interchanged without affecting the value of the block when the value of p does not affect the value of q.

e) Algebraic Transformations.

We can also optimize the basic blocks using algebraic identities. In this optimization, a more expensive operator is replaced by cheaper one

expensive	cheap
x^2	$x*x$
$2*x$	$x+x$
$x/2$	$x*0.5$

Peephole optimization

peephole optimization is a type of code optimization performed on a small part of the code. It is performed on a small segment of code.

Peephole is the machine dependent optimization.

The objective of peephole optimization is:

1) To improve performance

2) To reduce memory footprint

3) To reduce code size.

Peephole optimization Techniques

1) Redudent load and store elimination.

In this technique redudancy is eliminated.

2) Constant folding

The code that can be simplified by user itself, is simplified

3) Strength Reduction.

The operators that consume higher execution time are replaced by operators consuming less execution time

4) Null sequence

Useless operations are deleted

5) Combine operations

Several operations are replaced by a single equivalent operation.