Symbol Table Management

15/10/2024

Creation of symbol table

→ Identifiers & attributes are entered by analysis phases when processing the definition of an identifier.

→ Nested blocks → same variable name is used with multiple scopes.

Uses

→ To check if a variable is declared or not
→ Used by analysis & synthesis phases.
→ Type checking (semantic analysis)
→ Generate Intermediate / Target code.

Assumptions to create symbol table

→ using static scoping.
Global variable → scope 0
In nested blocks the scope increases by 1, based on the control.
Entering a block: +1
Exiting a block: −1
→ All names declared apriori
→ NO multiple declaration of a name in same scope
→ Same name in multiple nested scopes:— once per scope.

operations

→ To look up a name in current scope:
To check if it already exists (multiply declared)
→ Check usage of undeclared name
→ Link usage with symbol table entry
→ Insert a new name into the symbol table with attributes
→ Entering a new scope
Exiting a scope
} perform increase / decrease of scope variables value.

2 ops

→ ~~Insert~~ : Add a symbol [once]
→ Look up: find symbol & get its attributes. [many times]

Data Structure designed such that

→ Find & store / retrieve record quickly
→ Fast look up

① Linked List

- Linear list of records
- Add in the order of arrival
- search time complexity → O(n)

less space (adv)
additions are simple
higher access time. (disadv)

```
int c                                   scope
int main()                              0
{
    int a;                              1
    {
        int b;                          2
    }
    int d;                              1
    {
        {
            int e;                      3
        }
        int f                           2
    }
}
```

Symbol Table

Sl NO	Name	class	Datatype	Scope	Line NO
1	c	identifier	int	0	1
2	a	identifier	int	1	4,

etc. ~

Unsorted list results in higher access time

(2) Hash Table

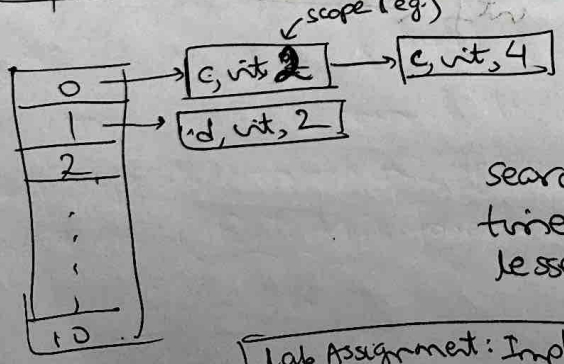
K pointers from 0 to K-1.

find hash value of name through hash function.

Name is mapped to an integer from 0 to K-1.

$H_{key} = ASCII_{cd} \bmod M$

a	b	c	d	e	f
97	98	99	100	101	102



searching trie is lesser

Lab Assignment: Implement Symbol Table

Error handling Techniques

→ stop immediately & signal an error
→ record the error but try to continue (through automatic correction routines).

4 types of errors

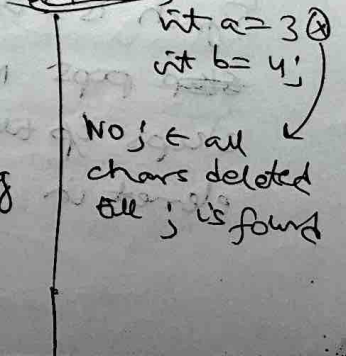
→ Lexical : missing identifier	int 23ab = 2;
→ Syntactic : unbalanced parentheses	int main()
→ Semantic : incomparable operand	int a[3.5];
→ Logical : infinite recursive call	div by 0

Error-recovery strategies

→ Panic mode
→ phrase level Recovery
→ Error Production
→ Global correction

Panic mode

Skip symbols on input till set of synchronizing tokens appears.



Phrase level

local correction

int id 5 ;
 ↑↑↑
int id = 5;

Inserting missing tokens

a = b ⊗ c

↓ Not token.

(+, −, *) ← suggestions or enter one randomly.

Error production

Erroneous constructs are augmented to the grammar. According actions would be defined in the corresponding cells of the table.

Global correction

Choose minimal seq. of correction.

panic mode

on predictive parsing

Some blank cells are marked as 'synch' for synchronising tokens

'Synch' is a driver (function) that ~~skip~~ pops Non-Terminal 'A' and skips i/p till synch token or element in FIRST(A) is found.

synch elements

Phrase level

on predictive parsing

→ Each cell has a special error routine

e.g.: insert *.

a * is entered and the production is retried.

Error Production

on predictive Parsing

→ in a blank cell, an error production rule is ~~use~~ added

→ powerful but manual process

In LR parsing table

Each empty cell is assigned error procedures like e1, e2, e3, etc.

To continue parsing, a number is forcefully added to the top of the stack.

Also removal of symbols & issuance of appropriate error messages.

21/10/2024

~~Semantic Analysis~~

Semantic Analysis

Syntax Directed Translation

Attaching attributes through grammar symbols.

Semantic rules are used for:—
- Intermediate code generation
- Type checking
- Putting info. into symbol table.

Syntax Directed Definition

High-level.

Order of evaluation of semantic actions is ~~not~~ hidden.

Translation scheme

Indicates order of evaluation of semantic actions associated with a production rule.

Steps
→ Parse input
→ Build parse tree
→ Traverse tree to evaluate semantic rules

Syntax Directed Definition
- Generalisation of Context Free Grammar (CFG)
- Each grammar symbol is associated with a set of attributes.

Synthesised Inherited.

Synthesized attribute
↓
Computed from children of that node in parse tree

eg
$E \rightarrow E_1 + E_2$
$E.val = E_1.val + E_2.val$

Inherited attribute
↓
parents and Sibling

eg
$A = X\ Y\ 2$
$Y = 2 * X.val$

Attributes are mentioned after (.) operator.
e.g. E.val
↳ value.
↳ It is mentioned in YACC specifications

Uses
Dependency graph.
→ Dependencies b/w attributes
→ Evaluation order of semantic rules.
→ Semantic rules setup
→ Defines value of attribute
→ Side effects (e.g. printing a value)

> parse tree showing values at each node →
Annotated parse tree

Determination of values
→ Constant value
→ lexical analyser
→ ~~Semantic~~ semantic rules.

Annotating / Decorating :— Computing value of nodes in a parse tree.

SDD

In Syntax Directed Definition, each production $A \to \alpha$ is associated with set of semantic rules of the form

$$b = f(c_1, c_2, \ldots, c_k)$$

$f \to$ fn & b can be :—

→ Synthesised attribute of A & $c_1, c_2, \ldots c_k$ are attributes of grammar symbols in α.

→ Inherited attribute of one of grammar symbols on RHS of $A \to \alpha$, and $c_1, c_2, \ldots c_k$ are attributes of grammar symbols of $[A, \alpha]$

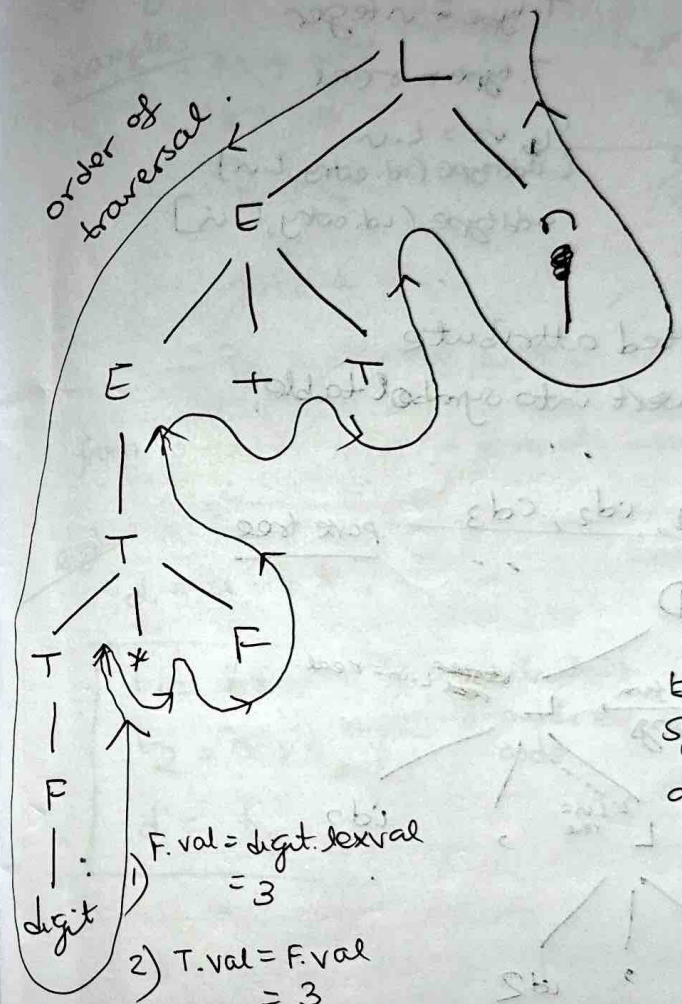
Terminal has attribute which is defined by lexical analyser.

Table has production & semantic rules.

prodns	(Synthesised attributes)
$L \to En$	
$E \to E+T$	— eg — $E.val \Rightarrow E.val + T.val$
$E \to T$	
$T \to T*F$	
$T \to f$	
$f \to (E)$	
$f \to digit$	

3 * 5 + 4n

Parse tree based on prev rules.



1) F.val = digit.lexval = 3

2) T.val = F.val = 3

3) T.val = T.val * F.val

and so on.

Here val is a Synthesised attribute.

(int a, b, c).

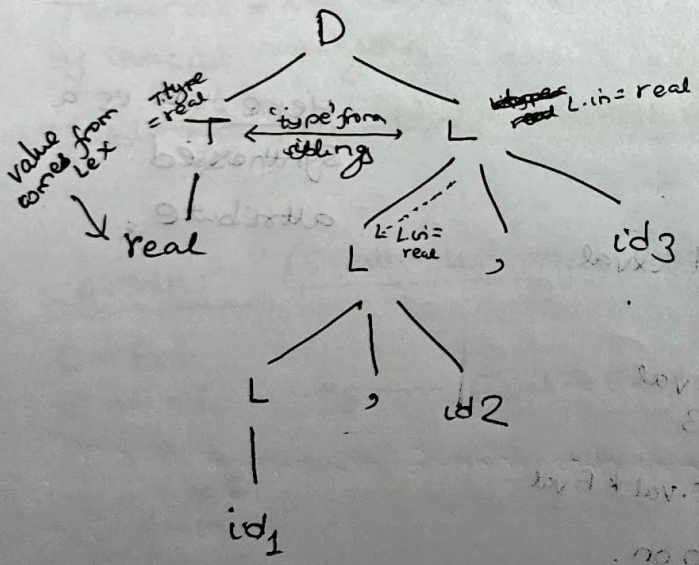
eg. Inherited attributes

$D \to TL$	$L.in = T.type$
$T \to int$	$T.type = integer$
$T \to real$	$T.type = real$
$L \to L_1, id$	$\{L_1.in = L.in$ $addtype(id.entry, L.in)\}$
$L \to id$	$addtype(id.entry, L.in)$

in ← inherited attribute

addtype ← insert into symbol table.

real id_1, id_2, id_3 — parse tree



Dep graph

29/10/2024

Three address code

eg $x = y \text{ op } z$

to be generated from directed acyclic graph (DAG).

examples

$x = y \text{ op } z$

$x = \text{op } z$

$x = y$

goto L

if x relop y goto L

$x[i] = y$

~~x = y[i]~~

$x = y[i]$

$x = \&y$

$x = *y$

from 3-addr code → generate assembly code.

e.g.

$d = a + b * c$.

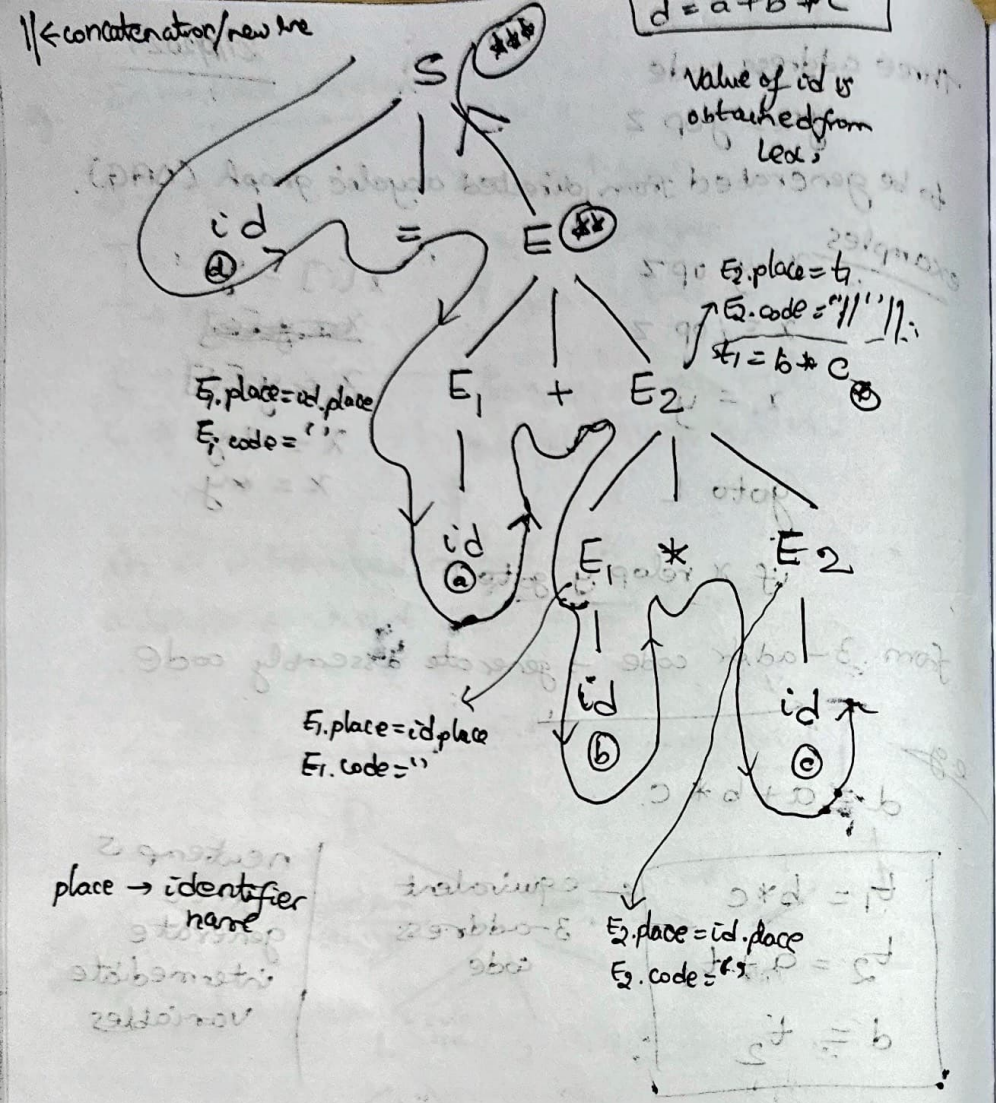
$t_1 = b * c$

$t_2 = a + t_1$

$d = t_2$

← equivalent 3-address code

newtemp is a fn to generate intermediate variables



Semantic rules to be written as per requirement.

Data structures to store 3-addr codes

1] Quadraples
2] Triples
3] Indirect Triples

1] (op, arg1, arg2, res)

$x = y$ op z.

y z → pointer to symbol table

2] (op, arg1, arg2)

uses sl no reference in place of arg1 & 2.

3] Statement: Index of triples is maintained by separate structure.

gen → fn to generate 3-addr code using

→ E.place
→ Assignment operator '='
→ G.place
→ op (*, +)
→ E2.place

* E2.code = " " || t1 = b * c