I.E.S. College of Engineering

2nd Internal Examination

Date : 21 April 2020

Name : Jovial Joe Jayarson

Roll No. : IES17CS016

Subject : CS304 · Compiler Design

Marks Awarded:

A1) Syntax Directed Definition (SDD)

- Syntax directed definition is a combination of context free grammar along with semantic rules.

- Also defined as a context-free grammar together with attributes and rules.

- Now attributes are associated with grammar symbols and rules with production.

eg:- Let A → A + B
A → B

- Then the corresponding SDD will be

A.value = A.value + B.value

A.value = B.value

Contrast with SDT

- SDT stands for syntax directed ~~definition~~ translation
- As the name suggest is is a translation.
- A translation scheme is embedded in fragments called semantic actions within the production.
- It follows the {} syntax or more appropriatly style of representation.

eg:- For the same production

$A \rightarrow A + B$

$\cancel{A} \rightarrow B$

The SDT will be

{A.val = A.val + B.val}

{A.val = B.val}

- Further since SDD is divided into S-attributed SDD & L-attributed SDD so also SDT's have S-attributed transations and L-attributed translations.

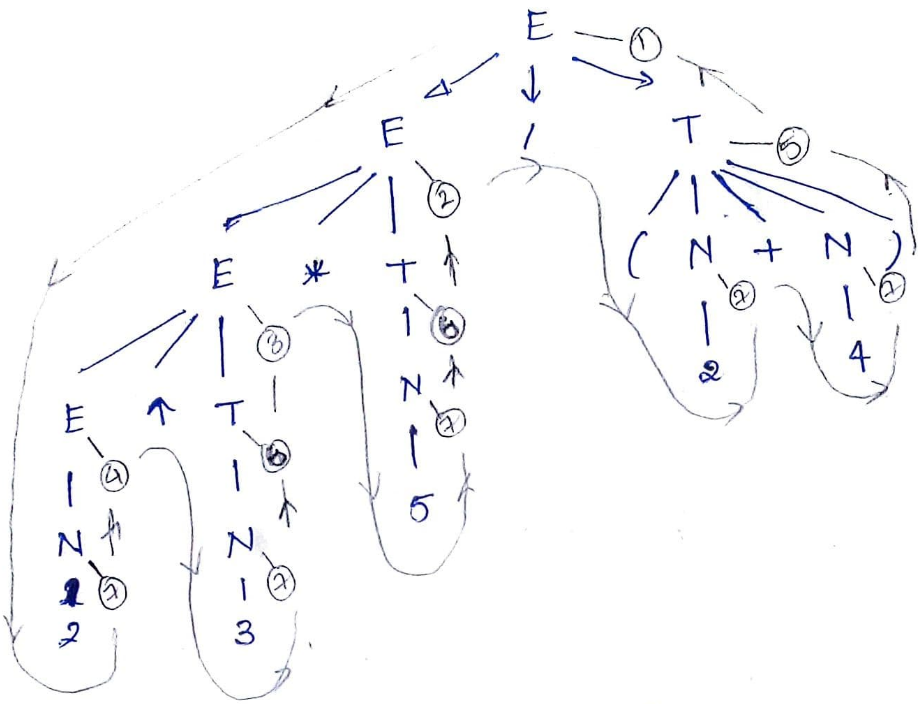
A2) Given Expression: $2 \uparrow 3 * 5 / (2+4)$

Grammar: $E \rightarrow E / T \mid E * T \mid E \uparrow T \mid N$

$T \rightarrow (N + N) \mid N$

$N \rightarrow 2 \mid 3 \mid 4 \mid 5$

- The corresponding parse tree can be generated as:



There for the postfix notation can be derived as:

$E \rightarrow E / T$	①	{ print("/") }
$E \rightarrow E * T$	②	{ print("*") }
$E \rightarrow E \uparrow T$	③	{ print ("↑") }
$E \rightarrow N$	④	{ }
$T \rightarrow (N + N)$	⑤	{ print ("+") }
$T \rightarrow N$	⑥	{ }
$N \rightarrow 2 / 3 / 4 / 5$	⑦	{ print ("No value) }

The sequen of printing will be

print(2)
print(3)
print("↑")
print(5)
print ("*")
print (2)
print ("4)
print("+")
print ("/")

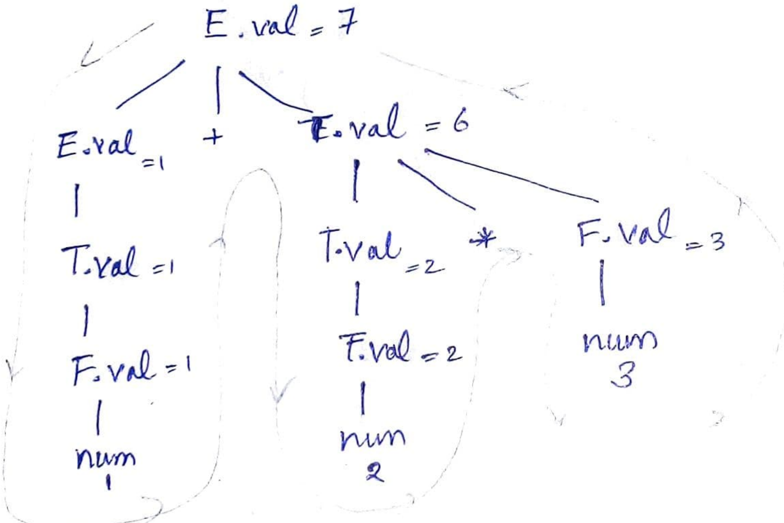
⇒ 23↑5*24+/ will be the require postfix expression.

A3.) Decorated Parse Tree

Let the ~~given~~ grammar and its correspondin SDT be:

Grammar	Semantic Actions
$E \rightarrow E + T$	{ E. value = E . value + T. value }
$E \rightarrow T$	{ E. value = T. value }
$T \rightarrow T * F$	{ T. value = T. value * F. value }
$T \rightarrow F$	{ T. value = F. value }
$F \rightarrow num$	{ F. value = num. lex . value }

Therfore the decorated parse tree would be: for (1+2*3)



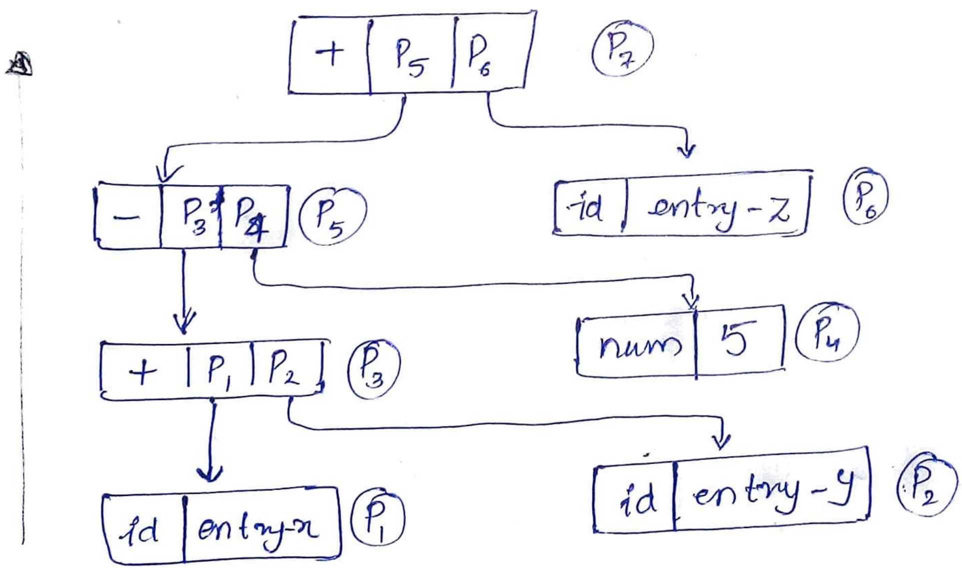
A4.) Given: $x * y - 5 + z$

- Fuctions used to create AST = mknode (op; leaf, right);
  mk leaf (id, entry) ; mk leaf (num, value)

- We would be parsing the symbols in the given expression from L to right in the postfix manner.

Symbol	Function/operation
x	P_1 = mk leaf (id, entry - x)
y	P_2 = mk leaf (id, entry - y)
$*$	P_3 = mknode ($*$, P_1, P_2)
5	P_4 = mk leaf (num, 5)
$-$	P_5 = mknode ($-$, P_3, P_5)
z	P_6 = mkleaf (id, entry - z)
$+$	P_7 = mknode ($+$, P_5, P_6)

$\therefore$ The Abstract Syntax Tree will be :



A5.) Runtime Storage Management

- Whenever a program is running it require some memory.

~ A compiler therefore much correctly impment the abstraction in the source language.

~ Hence the compiler requied that it must be capable to operate seamlessly with other control flow structures in the system (such as the OS and othe software)

~ This is why it is important for the compiler to manage the run-time environment in which it assumes its target programs are beign executed.

~ This enviroment deals with a huge variety of issues some of which include. global and local variables, stacks, queues etc.

~ Af the compilation the object code is generated which is brought in to the main memory for execution.

~ Here the compiler ~~uti~~ primarly utilizes two types of storage allocations

(i) Static Allocation : It is the storage that is aquired by the compiler when parsing the source code and they do not change.

(ii) Dynamic Allocation: It is the memory requirement of the program when it is run or executed.

A6.) Given expression:

$$P = a + b \times c / e \uparrow f + b \times c$$

Scanning the expession from left to right the operator precidence can be found as $\uparrow$, $\underbrace{\times\ /}_{L\ to\ R}$, $\underbrace{+\ -}_{L\ to\ R}$.

The corresponding three address code is:

$t1 = a$

$t2 = b \times c$

$t3 = e \uparrow f$

$t4 = t2 / t3$

$t5 = t1 + t4$

$P = t2 + t5$

∴ The Quadrouple representation would be:

Address ⇒

@	OPR	ARG1	ARG2	Result
(0)	=	a		t1
(1)	×	b	c	t2
(2)	↑	e	f	t3
(3)	/	t2	t3	t4
(4)	+	t1	t4	t5
(5)	+	t2	t5	P

~ Now since quadruples take more memory, the triples would be: (more optimize in space)

The Triple Representation

@	OPR	ARG1	ARG2
(0)	=	a	
(1)	x	b	c
(2)	↑	e	f
(3)	/	(1)	(2)
(4)	+	(0)	(3)
(5)	+	(1)	(4)

The triad method of representation is <u>Indirect Triples</u>

pointers

#	@
2000	(0)
2001	(1)
2002	(2)
2003	(3)
2004	(4)
2005	(5)

@	OPR	~~ARG1~~ ARG1	ARG2
(0)	=	a	
(1)	x	b	c
(2)	↑	e	f
(3)	/	(1)	(2)
(4)	+	(0)	(3)
(5)	+	(1)	(4)

A7.) Scope of Code Optimization

- Code optimization aims to improve a program. (not the algorithm within the program)

- Replacement of an algorithm is beyond the scope of code optimization (without AI).

- Code product by the compiler may not be perfect in terms of execution speed and memory consumption.

- But at the same time. highly efficient code (with maximum utilization of target machine's instruction set) is also beyond the scope of code optimization.

- Every programmer is not armed with the perfect knowledge of low level details of the target program hence a complier is employed to handle that.

Given code:

```
a, b, c = 10, 20, 30

for i in range (10):
    c = a+b
    d = a-b
    e = a*b
    g* = i
```

- This code contain a loop which performs some task 10 times.

- But if we look closely we can figure out that some of the code is loop invariant (doesnot depend upon the loop).

- The variable a, b, & c are initialized outsize the loop.
- The opeation $c=a+b$, $d=a-b$, $e=a*b$ are within the loop.
- During all the 10 iterations the value of a & b donot change.
- This implies that ~~neither~~ the value of c, d & e remain the same.
- Therefore it is highly recomened that insead of perfore state operation 10 time within a loop it must be move outside.
- Hence the optimized version of the code will be:

```
a, b, c = 10, 20, 30
c = a + b
d = a - b
e = a * b
for i in rang (10):
    s * = i
```

A8) Given expression:

$$d = (a+b) + (a-c) + (a-c)$$

Therefor the corresponding 3-address code will be:

$t1 = a + b$

$t2 = a - c$

$t3 = t1 + t2$

$\cancel{t4}d = t3 + t2$

~ Now there is a register descriptor which keeps track of what is currently in the register.

~ But an address descriptor keeps track of the variable's current value can be found at run time:

~ Hence the traget code (in assembly languge) is generated as follows:

Statement	Code Generated	Register Descriptor	Address Descriptor
$t1 = a + b$	MOV R_0, a ADD R_0, b	R_0 contains t1	t1 in R_0
$t2 = a - c$	MOV R_0, a SUB R_1, c	R_0 contains t1 R_1 contains t2	t1 in R_0 t2 in R_1
$t3 = t1 + t2$	ADD R_0, R_1	R_0 contains t3 R1 contains t2	t2 in R_1 t3 in R_0
$d = t3 + t2$	ADD R_0, R_1	R_0 contains d	d in R_0

A9.) Functional Preserving Transforms

~ This is a part of "Principal Sources of optimization"

~ These are the optimization techniques which are employed to optimize the code.

~ But at the same time as the name sugges the preserve what a ~~two a~~ function performs.

~ They are: (i) Common Subexpression Elemination

(ii) Dead code Elemination

(iii) Copy Propogation

(iv) Constant Folding.

(i) Common Subexpression Elemination

~ Whenever there is a redundant subexpression it can be removed.

- Let E_1, E_2 & E_3 be 3 expression, E_3 is assigned something such that it results in E_1.

~ The E_3 is directly assigned as E_1 iff E_1 has not be altered in between.

eg:- ① a = b+c

② b = a + d

③ c = b+c

④ d = a + d

① & ③ are similar but the value of 'b' is altered in ②

② & ④ are similar & the value of 'd' is not altered. ~~then~~ in between

~ Therefor ④ can be written as $d = b$

∴ The optimized code will be: (transform)

$$a = b + c$$
$$b = a + d$$
$$c = b + c$$
$$d = a + d$$

(ii) Dead code Elemination

~ Here when some part of the code is reached it said to be working (or alive)

~ Otherwise it is called 'dead' because it never gets executed.

eg:-

```
def sum (a+b):
    sum = a+b
    return (sum)
    print ("sum: ", sum)
```

dead line - will never get execued.

⇒ Refactored code:

```
def sum(a+b):
    sum = a+b
    print("sum: ", sum)
    return (sum)
```

(iii) Copy propogation

~ Copy statemes are like $f := g$.

~ Statements like this

B: = A
C := B
D := C

can be optimized as

B := A
C := A
D := A

(since A has not be altered in between)

(iv) Constant Folding

- If an express consists of only constant operands then the expression can be evaluated and result is stored during the compile time.

eg:- b = 3 + 10 $\xrightarrow{}$ b = 13

optimised during compile-time as

10.) Design Issues of a Code Generator

- There are certain issues that are to be kept in mind during code generation.

(i) Input to the code generator

- The assumption is that the input is free of errors.
- It takes inputs of the form
  - (a) Postfix Notations
  - (b) Three Address Code
  - (c) Syntax Directed Trees.

(ii) Target Program

- The output of the code generation is the target program.
- It is mainly produced in three forms
  - (a) Assembly language : This assumes that the

code runs on multiple platforms and still retain its human readability.

ⓑ Absolute Machine language:

- Directed machine code is placed at a fixed location in the primary memory.
- It has faster execution speeds

ⓒ Relocatable Macro Language:

- When the program is large and consists of multiple modules / libraries - components like linker and loader are required.
- This make the program to be store at multiple locations to be access and executed later.

(iii) Memory Management

- Mapping of variables and corresponding address are perfomed using the symbol table.
- This table is thus utilized for memory mangement.
- Both static & dynamic managemt is taken into consideration.

(iv) Instruction Selection

- This is a complicated task to optimize the program with most suited instruction.

eg:- $a = a - 1$ may be translated as

MOV R_0, a
SUB R_0, #1
MOV a, R_0

This is syntaxilly correct but not efficient.
The efficie instuction would be $\boxed{\text{DEC a}}$.

(v) Register Allocation

- The number of register available in a system is less.
- Here crucial decision needs to be made to realized what to be stored in the registers at the current time.

- Register allocation (register is allocated): which register contain which value.

- Register assignment (register is assigned to): which value is store in which register.

- Finding the optimal solution is considered to be an NP complete problem.

vi) Evaluation Order

- Choosing the right order of exection also determines how a program is going to perform.
- Code optimization helps in solving this to some extent.