CZ3007
Nov/Dec 2017

1. (a)

i. Lexer Generator

Input: Token Specification Output: Lexer Implementation

Parser Generator

Input: Context Free Grammar Output: Parser

ii. Abstract Grammar

Abstract Grammar define a set of tree used to describe the structure of an AST

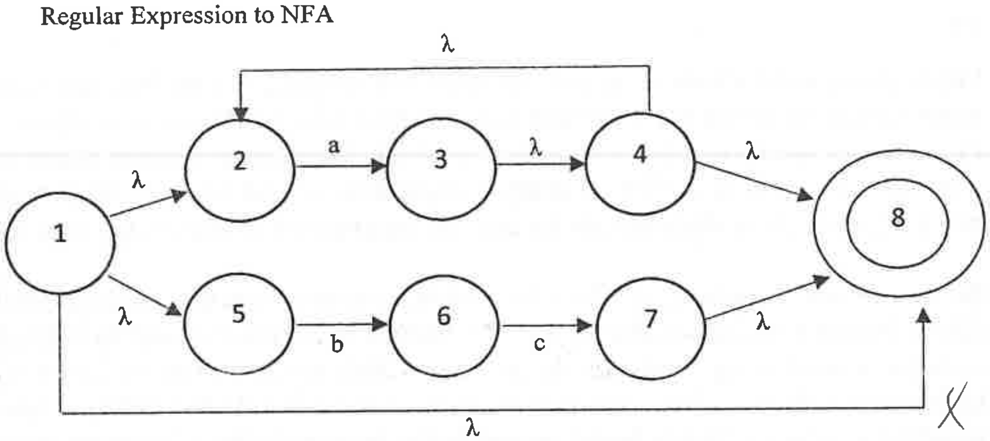
Attribute Grammar

Attribute Grammar define a set of attributes on to of an abstract grammar used to describe the computation of an AST node with common occurring pattern

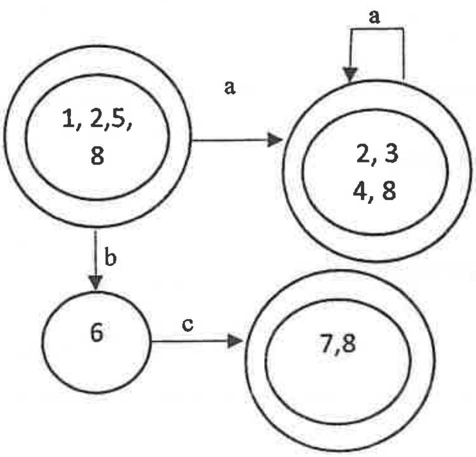
(b)

$a^* | bc$

Regular Expression to NFA



NFA to DFA



(c)

4 $[0\text{-}9]^{15}$ | 4$[0\text{-}9]^3$((" "$[0\text{-}9]^4)^3$ | ("-"$[0\text{-}9]^4)^3$)

2. (a)

P_1 return id P_2 return plus, minus, intconst
P_3 return plus
P_4 return minus
P_5 return intconst
P_6 return dot
P_7 return $
The following grammar is consider as an LL(1) grammar as it does not contain the following two categories of production rule:

1. Left Recursion
2. Common prefix

(b)

LR(0) parser used a bottom up parsing approach whereby it start from the leaves and work its way up to the roots through the reduction of a rule to a non-terminal. Whereas, LL(1) parser used a top-down approach in which it start from the root ∈ non-terminal and work its way down to the leaves using a single look-a-head token to determine how it is being expand. Thus, common prefix and left recursion will affect LL(1) but not LR(0).

Bottom-up parser parsing is driven by a table. Grammar rule that cause problems to LR(0) but not LALR(1) is due to how the table is being constructed. In LR(0)table construction for every final item that exist in a state the entire row of the table will be inserted with the reduce action of the rule whereas in LALR(1) table construction of inserting a reduce action is based on symbol in the itemFollow.Therefore reducing the chance of encountering a shift-reduce or reduce-reduce conflict.

(c)

Polygon => LCURLY Vertices RCURLY
Vertices => Vertex COMMA Vertex COMMA GT3Vertex
GT3Vertex=> Vertex COMMA GT3Vertex
| => Vertex
Vertex=>LPAR Sign INTLITERAL COMMA Sign INTLITERAL RPAR
Sign => MINUS
| =>λ

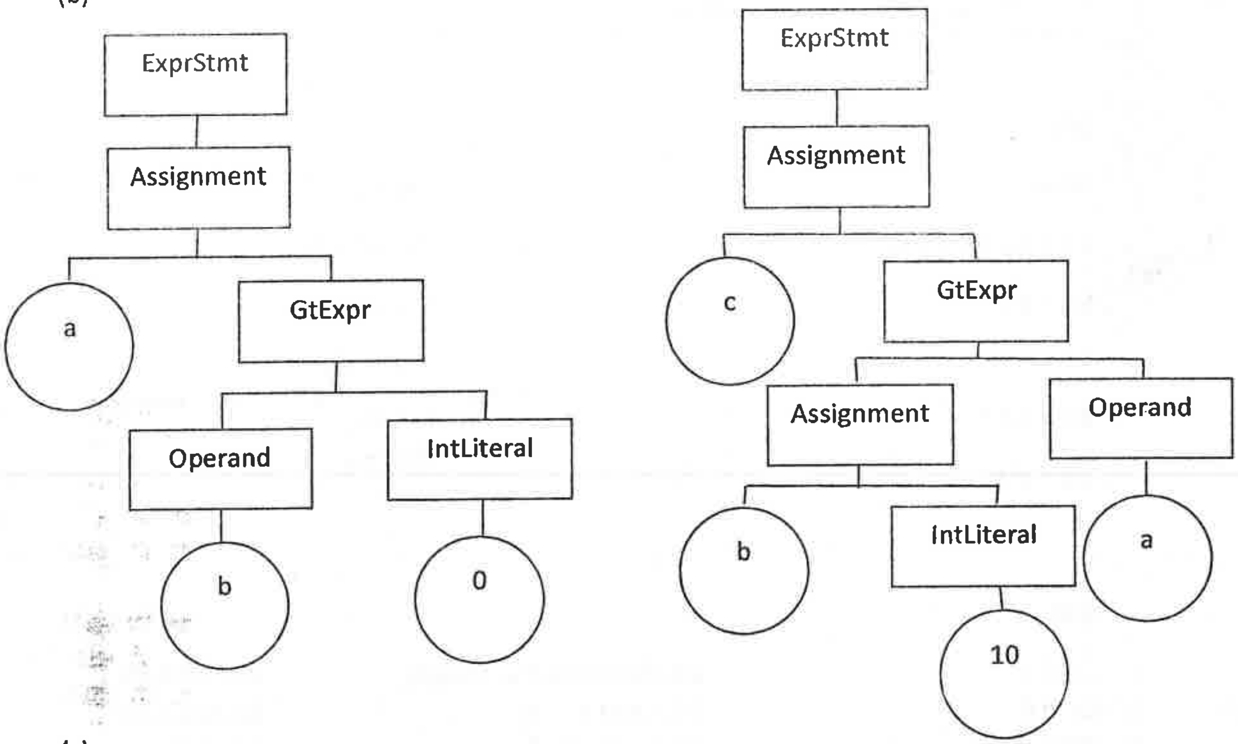
(d)

In LALR(1) vertices is use to represent the pair (state,item) while the edges are used for the transition from one state to another as well as aiding the propagating of itemFollow. In addition, in some case whereby the token that come after the next token can derive empty it shows what can follow.

3. (a)

P4: INT ID.id SEMICOLON {: return new IntDeclStmt(id);:}
P6: LCURLY Stmts.stmt RCURLY {:return new Block(stmt);:}
P7: Expr.e1 SEMICOLON {: return e1; :}
P8: ID.id EQL Expr.e1 {: (return new Assignment(id, e1)); :}
P10: AExpr.e1 GT AExpr.e2 {: return new GtExpr(e1, e2); :}

(b)



(c)

Line 3: C is not declared

Line 4: Type inconsistency

Line 7: Type inconsistency

4. (a)

Benefit:

1. Different language can be represented using the same IR. Thus, it allows sharing of optimization.
2. Platform independent optimization can be done, then the optimized IR is converted to native code. Thus, reducing the need to generate code for different platform.

(bi)

Label1

X < 0 goto Label0

$i0 = x*x

$i1 = 8*y

$i2 = $i1 * z

$i3 = z*z

$i4 = $i2 - $i3

Y = $i4 + $i0

X = x -1

Goto Label1

Label0
return y

(bii)

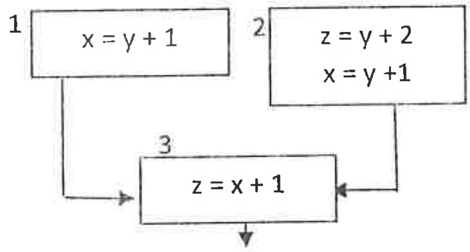
1. Load x
2. Load 0
3. Ifgt Label0
4. Load y
5. Ret
6. Label0
7. Load x
8. Load x
9. Mul
10. ~~Store $i0~~
11. Load y
12. Load 8
13. Mul
14. ~~Store $i1 (Rearrange)~~
15. Load z
16. ~~Load $i1~~
17. Mul
18. ~~Store $i2~~
19. Load z
20. Load z
21. Mul
22. ~~Store $i3~~
23. ~~Load $i2~~
24. ~~Load $i3~~
25. Sub
26. ~~Store $i4~~
27. ~~Load $i4~~
28. ~~Load $i0~~
29. Add
30. Store y
31. Load x
32. Load 1
33. Sub
34. ~~Store x~~
35. ~~Load x~~
36. Load 0
37. Ifgt Label0
38. Load y
39. ret

(biii)

Operand Stack	4
	3
	2
	1
Local Slot	z
	y
	x
	this

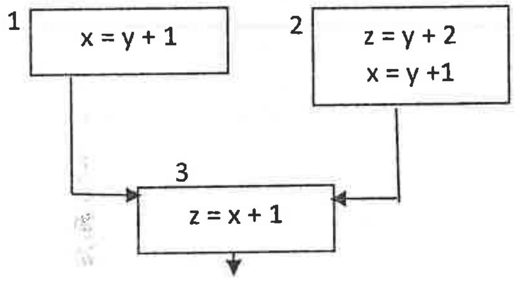
5. (a)

For the forward part it is because to compute the outgoing available expression of a node we make use of the incoming available expression of the node as well.



From the above control flow graph we are able to observe that the available expression after node 3 is z = x + 1 and x = y + 1 as even though node 3 only compute the express z = x + 1 but since x is not being written in node 3 it is still available after the execution of node 3. Thus, this show that to compute the outgoing available expression of a node we need to consider what had been computed previously as well.

For the must part it is because in order for an expression to be consider as available it must be available in all the path which precede the current node that is being computed

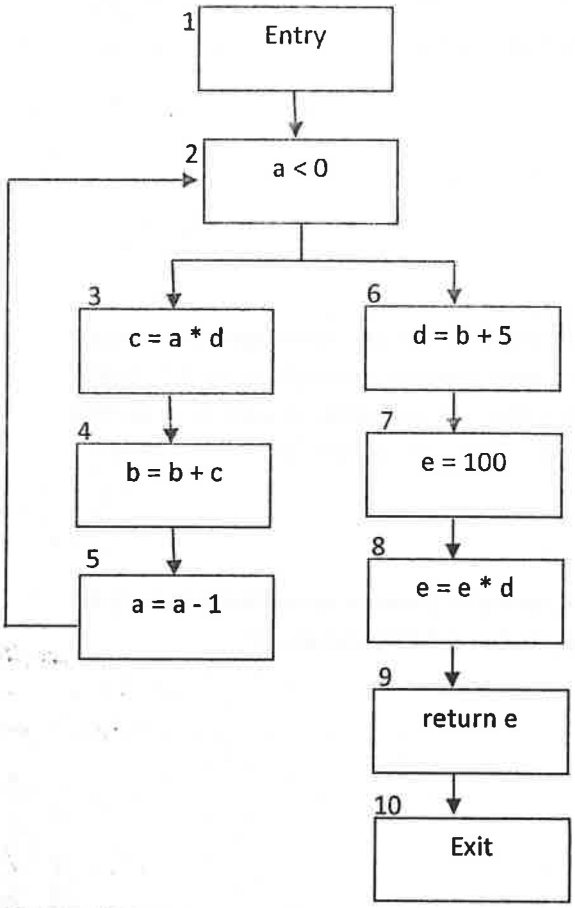


From the above control flow graph it can be observe that x = y + 1 is the only available expression that is use for the computation of the outgoing available expression of node 3. Even though node 2 compute z = y + 2 but if the flow of the program execution is coming from node 1 instead of node 2 then z = y + 2 is not computed before node 3. Thus, this explain why it is a must analysis.

24

3

(b)



(c)

- $in(1) = out(1)$
- $in(2) = out(2) \cup \{a\}$
- $in(3) = out(3) - \{c\} \cup \{a, d\}$
- $in(4) = out(4) - \{b\} \cup \{b, c\}$
- $in(5) = out(5) - \{a\} \cup \{a\}$
- $in(6) = out(6) - \{d\} \cup \{b\}$
- $in(7) = out(7) - \{e\}$
- $in(8) = out(8) - \{e\} \cup \{e, d\}$
- $in(9) = out(9) \cup \{e\}$
- $in(10) = \emptyset$

- $out(1) = in(2)$
- $out(2) = in(3) \cup in(6)$
- $out(3) = in(4)$
- $out(4) = in(5)$
- $out(5) = in(2)$
- $out(6) = in(7)$
- $out(7) = in(8)$
- $out(8) = in(9)$
- $out(9) = in(10)$
- $out(10) = \emptyset$

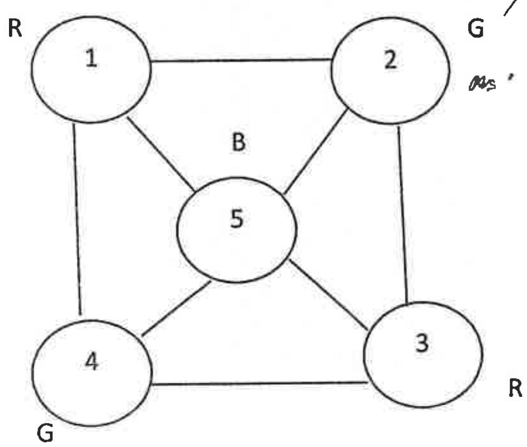
- $in(1) = in(2)$
- $in(2) = in(3) \cup in(6) \cup \{a\}$
- $in(3) = in(4) - \{c\} \cup \{a, d\}$
- $in(4) = in(5) - \{b\} \cup \{b, c\}$
- $in(5) = in(2) - \{a\} \cup \{a\}$
- $in(6) = in(7) - \{d\} \cup \{b\}$
- $in(7) = in(8) - \{e\}$
- $in(8) = in(9) - \{e\} \cup \{e, d\}$
- $in(9) = in(10) \cup \{e\}$
- $in(10) = \emptyset$

	0	1	2	3	4	5
in(1)	Ø	Ø	a	a, b, d	a, b, d	a, b, d
in(2)	Ø	a	a, b, d	a, b, d	a, b, d	a, b, d
in(3)	Ø	a, d	a, b, d	a b, d	a, b, d	a, b, d
in(4)	Ø	b, c	a, b, c	a, b, c	a, b, c, d	a, b, c, d
in(5)	Ø	a	a	a, b, d	a, b, d	a, b, d
in(6)	Ø	b	b	b	b	b
in(7)	Ø	Ø	d	d	d	d
in(8)	Ø	e, d	e, d	e, d	e, d	e, d
in(9)	Ø	e	e	e	e	e
in(10)	Ø	Ø	Ø	Ø	Ø	Ø

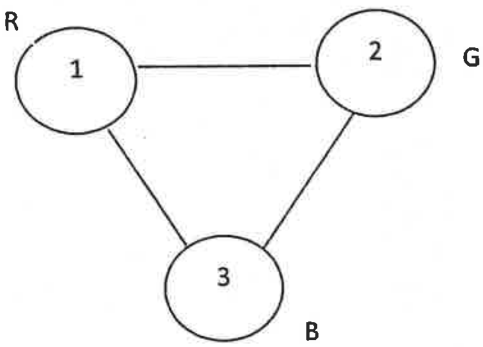
(d)

In most general case the value of n is k-1. However, there are some special case in which the value of n can be greater than k – 1 that is k+1 such as:

k=4 / n=3



k=2



Email: ZNG018@e.ntu.edu.sg

Name: Ng Zi Peng

4 25