ktunotes.in

GET IT ON Google Play

KTU NOTES

The learning companion.

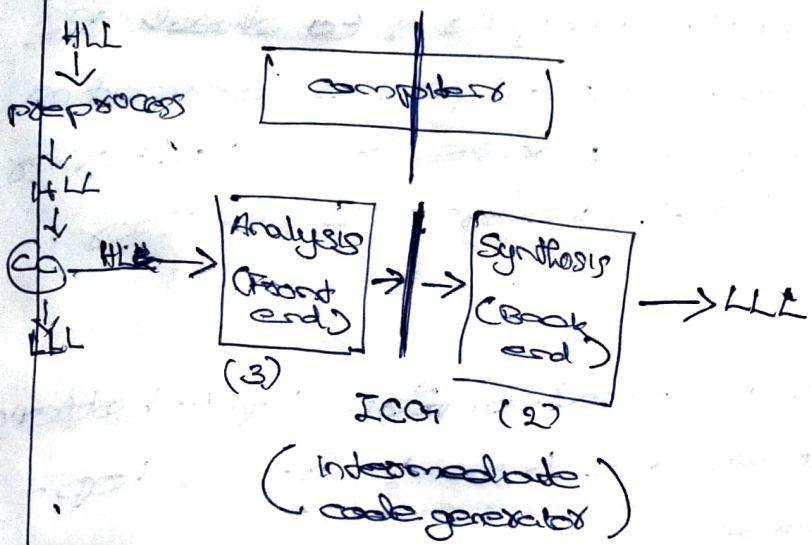
KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Website: www.ktunotes.in

11/4

Compiler Design

Phases of compiler



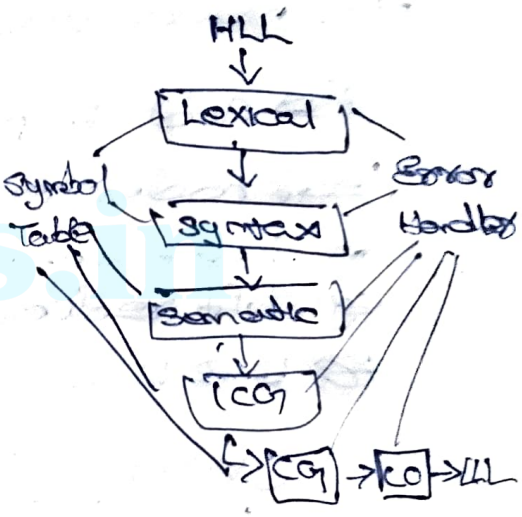
Analysis

3 phases

→ Lexical Analysis / scanner

→ Syntax Analysis / parser

→ Semantic Analysis



Lexical Analysis

It plays below 3 functionalities, they are

° It help you to identify the lexeme (well-defined token)

This is the 1° functionality

There are 2° functionality

° It will remove the comments in the pgm

° It will remove the white spaces.

There are mainly 2 tools used for lexical analysis.

- Lex
- Flex

DOWNLOADED FROM KTUNOTES.IN

Syntax Analysis

It is 2nd stage in compiler, which will receive i/p from scanner and it will tokens.

The role of Syntax Analysis is to check the whether the Grammar of prog is correct or not with the help of parse trees. Another name for syntax analysis is parser.

Ambiguity

If a parse tree generated from top to bottom those parse tree are known to as top-down parse tree

eg: Recursive descent parse tree.

And if parse tree generated from leaf node to root node, it is termed to be as bottom-up parse tree.

eg: shift reduce parser

E → E+E / E*E / id

(V, T, P, S)

{E} {+, *, id} {E}

P: E → E+E
   → E*E

The tool that we use for implementation of parse is YACC, it is elaborated as.

Yet another compiler completed.

Semantic Analysis

- The major of Semantic Analysis is to resolve the meaning of prog.
- It will help to do a type correction

ICG

Phase of compiler where code is generated without depending of system architecture like registers and addressing mode. It uses temporary variables to store data.

It can be represented in 3 ways

- Quadruples
- Triples
- Indirect triples.

eg: a = (-b+c) * (-b+c)

t1 := c
t2 := uminus b
t3 := t2+t1
t4 := t3 * t3
a := t4.

Quadruple:

addr	operator	operand	operand	Result
(0)	:=	c		t1
(1)	:=	uminus b		t2
(2)	+	t1	t2	t3
(3)	*	t3	t3	t4
(4)	:=	t4		a

Ktunotes.in

DOWNLOADED FROM KTUNOTES.IN

Triple:

addr	operator	operand	operand
(0)	:=	c	
(1)	:=	uminus b	
(2)	+	(1)	(0)
(3)	*	(2)	(2)
(4)	:=	(3)	a

Indirect triple

addr	
(11)	(0)
(12)	(1)
(13)	(2)
(14)	(3)
(15)	(4)

Code Generation.

Code generated based on architecture of system.

- Code optimization

```
MOV R1,#-b
MOV R2,#c
ADD R1,R2
MUL R1,R1
STA A,R1
```

↳ not accumulator & registers, there are only registers

Lex

Write a Lex pgm to accept strings that starts with 2 consecutive zero's.

RE = 00[0/1]*

[00][0/1]*

∧ 00[0/1]* (RE)

RE
Action
main

```
%%
[\n] printf("accept\n");
.+ printf("reject\n");
%%
main()
{
  yyin = stdin;
  yylex();
}
```

(Action)

(main)

lex filename.l
↳ file saved as corresponding program
gcc yy.lex.c
./a.out.

Write a lex pgm to accept strings where ... numbers.

RE = [0/1]*1

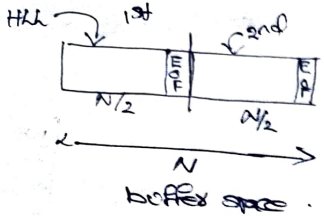
∧ [0/1]*1

```
%.%.
[\n] printf("accept");
.+ printf("reject");
%.%.
main()
{
  yyin = stdin;
```

Ktunotes.in

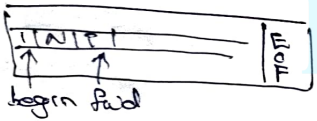
DOWNLOADED FROM KTUNOTES.IN

Working of Lexical Analyzer:

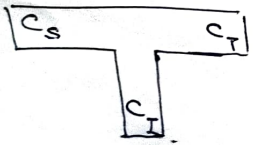


buffer space.

There is a buffer space of size N to store characters. HLL are stored in each of buffer space. They have 2 ptrs begin & fwd. Begin represent start & fwd indicate end of string. When 1st buffer space is filled next store in 2nd space and 1st space cleared for further storing.



Bootstrapping



S → source lang
I → implement
T → Target

Problem-2

(3) Find the First of the following grammars.

1) A → abB/Ce
B → d
C → f/e

FIRST(A) = { a, FIRST(C) }
= { a, f, e }

A → Ce
→ fe/ee
→ e/e

FIRST(B) = {d}
FIRST(C) = {f, e}

2) S → AB
A → a
B → C/b
C → D
D → E
E → a

FIRST(S) = { FIRST(A) } = {a}
FIRST(A) = {a}
FIRST(B) = { FIRST(C), b } = { FIRST(D), b }
{ FIRST(E), b } = { a, b }
FIRST(C) = { FIRST(D) } = { FIRST(E) } = {a}
FIRST(D) = { FIRST(E) } = {a}
FIRST(E) = {a}

3) S → ABC/ad
A → eS/Cr
C → f/P

Ktunotes.in

DOWNLOADED FROM KTUNOTES.IN

FIRST (S) = {FIRST (A), a} = {FIRST e, a, FIRST (C)}
= {e a, f, p}

FIRST (A) = {e, FIRST (C)} = {e, f, p}

FIRST (C) = {f, p}

4) S → XZY / ZbY / Ya
X → da / YZ
Y → g / e
Z → h / e

FIRST (S) = {FIRST (X), FIRST (Z), FIRST (Y)}
= {d, FIRST (Y), g, h}
= {b, d, g, h, a}

FIRST (X) = {d, FIRST (Y)} = {d, g, FIRST (Z)} = {d, g, h, e}

FIRST (Y) = {g, e}

FIRST (Z) = {h, e}

5) S → ABD
A → a / BBB
B → b / D
D → d / e

FIRST (S) = {FIRST (A)} = {FIRST (B), a} = {a, b, FIRST (D)}
= {a, b, d, e}

FIRST (A) = {FIRST (B), a}, {a, b, FIRST (D)}, {a, b, d, e}

FIRST (B) = {b, FIRST (D)} = {b, d, e}

FIRST (D) = {d, e}

6) S → T3 / [S]S /]S / e
T → (x)
X → TX / [X]x / e

FIRST (S) = {FIRST (T), [,], e}
= {(, [,], e}

FIRST (T) = {(}

FIRST (X) = {FIRST (T), [, e}
= {(, [, e}

7) A → BCx / y
B → yA / e
C → Ay / x

FIRST (A) = {FIRST (B), y} = {y, FIRST (C)}
= {y, FIRST (A)} = {y, x}

FIRST (B) = {y, e}

FIRST (C) {FIRST (A), x} = {x, y}

8) A → zB / Cy
B → Ax
C → zB / e

FIRST (A) = {z, FIRST (C)} = {x, z, y}

FIRST (B) = {FIRST (A)} = {x, z, y}

FIRST (C) = {z, e}

9) A → BC
B → Ax / x
C → yC / y

FIRST (A) = {FIRST (B)} = {FIRST (A), x)
= {x}

FIRST (B) = {FIRST (A), x) = {x}

FIRST (C) = {y}

ktunotes.in

DOWNLOADED FROM KTUNOTES.IN

10) $A \to BC$
$B \to Ax / x / \epsilon$
$C \to yC / y$

FIRST(A) = {FIRST(B)} = {FIRST(A), x, FIRST(C)}
= {x, y}
FIRST(B) = {FIRST(A), x, ∈}
= {x, y, ∈}
FIRST(C) = {y}

11) $S \to BAa$
$A \to \epsilon / BcA$
$B \to \epsilon / b$

~~FIRST(S) = {FIRST(B)} = {b, FIRST(A)}~~
~~= {b, FIRST(B), FIRST(A)} = {b, ∈}~~

FIRST(S) = {FIRST(B)} = {b, FIRST(A)} = {b, a, FIRST(B)}
= {b, a, c}

FIRST(A) = {FIRST(B), c} = {b, ∈, c}
FIRST(B) = {b, ∈}

9) Find follow for the following grammar.

$S \to AaAb$
$S \to Bb$
$A \to \epsilon$
$B \to \epsilon$

FIRST(S) = {FIRST(A)} = {a}
FIRST(S) = {FIRST(B)} = {b}
FIRST(A) = {∈}
FIRST(B) = {∈}

FOLLOW(S) = {$}

Golden Rule: starting symbol of the non-terminal should contain $ in its follow

Follow(A) = {a, b} ; Follow(B) = {b}

8) Find follow for following grammar:

1) $S \to AB$
$A \to a$
$B \to C / b$
$C \to D$
$D \to E$
$E \to a$

FIRST(S) = {FIRST(A)} = {a}
FIRST(A) = {a}
FIRST(B) = {a, b}
FIRST(C) = {a}
FIRST(D) = {a} = FIRST(E)

FOLLOW(S) = {$}
FOLLOW(A) = {FIRST(B)} = {a, b}
FOLLOW(B) = {FOLLOW(S)} = {$}

Golden rule: If there is no terminals than terminal to be follow, then its LHS follow value is assign to it.

FOLLOW(C) = {FOLLOW(B)} = {$}
FOLLOW(D) = {FOLLOW(C)} = {$}
FOLLOW(E) = {FOLLOW(D)} = {$}

2) $S \to A$
$A \to BC / DBC$
$B \to bB' / \epsilon$
$B' \to bB' / \epsilon$
$C \to c / \epsilon$
$D \to a / d$

FIRST(S) = {FIRST(A)} = {FIRST(B), FIRST(D)}
= {b, c, a, d}
FIRST(A) = {a, b, c, d}
FIRST(B) = {b}
FIRST(B') = {b}
FIRST(C) = {c}
FIRST(D) = {a, d}

FOLLOW(S) = {$}
FOLLOW(A) = {FOLLOW(S)} = {$}
FOLLOW(B) = {FIRST(C)} = {c, FOLLOW(A)} = {c, $}
FOLLOW(B') = F(B) = {c, $}
FOLLOW(C) = F(A) = {$}
~~FOLLOW(S) = {FIRST(B)} = {b, FIRST(C)} = {b, $}~~

Ktunotes.in
DOWNLOADED FROM KTUNOTES.IN

11:40-1:30

FOLLOW(D) = {FIRST(B)} = {b, FIRST(C)}

= {b, c, FOLLOW(A)} = {b, c, $}.

Q A → BCD
B → hB/e
C → Cg/g/Ch/i
D → iB/e

FIRST(A) = {FIRST(B)} = {h, FIRST(C)}
= {h, g, i}
FIRST(B) = {h, e}
FIRST(C) = {g, i}
FIRST(D) = {FIRST(A), e}
= {h, g, i, e}

FOLLOW(A) = {$, FIRST(B)} = {$, h, FOLLOW(D)}

FOLLOW(B) = {FIRST(C), FOLLOW(B), FOLLOW(D)}
= {g, i, FOLLOW(D)} = {g, i, h, $}

FOLLOW(C) = {FIRST(D), g, h} = {h, g, i, FOLLOW(A)}
= {h, i, g, $, FOLLOW(D)} = {h, g, i, $}

FOLLOW(D) = {FOLLOW(A)} = {$, h}

Q A → BCx/y
B → yA/e
C → Ay/x

FIRST(A) = {FIRST(B), y} =
{y, FIRST(C)}
= {y, x}

FIRST(B) = {y, e} FIRST(C) = {y, x}

FOLLOW(A) = {$, FOLLOW(B), y} = {$, x, y}
FOLLOW(B) = {FIRST(C)} = {x, y}
FOLLOW(C) = {x}

Q A → aB/Cy
B → AbC
C → bB/e

FIRST(A) = {x, y, z}
FIRST(B) → {x, y, z}
FIRST(C) → {b, e}

FOLLOW(A) = {$, b}
FOLLOW(B) = {FOLLOW(A), FOLLOW(C)} = {$, b, y}
FOLLOW(C) = {FOLLOW(B), y} = {$, b, y}

Role of Syntax Analysis:

It check the grammar of the string. It's implemented using YACC and its output is a parse tree.

Lex
Lexere → Parser/syntax.

Ambiguity: If there is more than method to derive a string.

Elimination of Left Factoring

Assume the grammar to be
A → Aα/β [Left hand side of grammar will not get terminated]

The above grammar is need to be rewritten as
A → βA'
A' → αA'/e

For the given grammar E → E+T/T
T → T*F/F
F → (E)/id

ktunotes.in

DOWNLOADED FROM KTUNOTES.IN

left recursion

$E \to E+T / T \Rightarrow E \to TE'$, $E' \to +TE' / \epsilon$

A A α β

$T \to T*F / F \Rightarrow T \to FT'$, $T' \to *FT' / \epsilon$

A A α β

$F \to (E) / id$

FIRST(E) = {FIRST(T)} = {FIRST(F} = {(, id}

FIRST(E') = {+, ∈}

FIRST(T) = {FIRST(F} = {(, id}

FIRST(T') = {*, ∈}

FIRST(F) = {(, id}.

FOLLOW(E) = {$,)}

FOLLOW(E') = {FOLLOW(E'), FOLLOW(E)}

= {$,)}

FOLLOW(T) = {FIRST(E')} = {+, FOLLOW(E)} = {$, +,)}

FOLLOW(T') = {FOLLOW(T), } = {$, +,)}

FOLLOW(F) = {FIRST(T')} = {*,), +, $}

Assume that grammar G_1 is given as

$E \to E+T / T$

$T \to T*F / F$

$F \to (E) / id$

Above grammar G_1 is left recursive nature. So it can be written as G_1'. The above grammar G_1 is of the form $A \to A\alpha / \beta$, which has to converted into $A \to \beta A'$, $A' \to \alpha A' / \epsilon$

$E \to TE'$

$E' \to +TE' / \epsilon$

$T \to FT'$

$T' \to *FT' / \epsilon$

$F \to (E) / id$.

FIRST(E) = {(, id} FIRST(T) = {(, id}

FIRST(E') = {+, ∈} FIRST(T') = {*, ∈}

FIRST(F) = {(, id},

FOLLOW(E) = {$,)} FOLLOW(E') = {$,)}

FOLLOW(T) = {$, +,)} FOLLOW(T') = {$, +,)}

FOLLOW(F) = {*,), +, $}

Ktunotes.in

DOWNLOADED FROM KTUNOTES.IN

Parsing table

	(	)	id	α	+	$
E	$E \to TE'$		$E \to TE'$			
E		$E' \to \epsilon$			$E' \to +TE'$	$E' \to \epsilon$
T	$T \to FT'$		$T \to FT'$			
T'		$T' \to \epsilon$		$T' \to *FT'$	$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to (E)$		$F \to id$			

Q $S \to aXYb$

$X \to c/\epsilon$

$Y \to d/\epsilon$

FIRST(S) = {a} FIRST(Y) = {d, ε}

FIRST(X) = {c, ε}

FOLLOW(S) = {$}

FOLLOW(X) = {FIRST(Y)} = {d, b}

FOLLOW(Y) = {b}

	a	b	c	d	$
S	$S \to aXYb$				
X		$X \to \epsilon$	$X \to c$	$X \to \epsilon$	
Y		$Y \to \epsilon$		$Y \to d$	

Q Construct parsing tree

$S \to TS / [S]S /)S / \epsilon$

$T \to (X)$

$X \to TX / [X]X / \epsilon$

FIRST(S) = {FIRST(T), [,), ε} = {(, [,), ε}

FIRST(T) = {(}

FIRST(X) = {FIRST(T), [, ε} = {(, [, ε}

FOLLOW(S) = {$,]}

FOLLOW(T) = {FIRST(S), FIRST(X)} = {$,],), [, (}

~~FOLLOW(X) =~~ = {$,], (, [,)}

~~FOLLOW(X) = {),] }~~

= {(, [,], $,)}

FOLLOW(X) = {),]}

	[	]	(	)	$
S	$S \to [S]S$		$S \to TS$	$S \to)S$	
T			$T \to (X)$		
X	$X \to [X]X$	$X \to \epsilon$	$X \to TX$	$X \to \epsilon$	

Ktunotes.in

DOWNLOADED FROM KTUNOTES.IN

Stack implementation

id + id * id. accepted by G or not?

E → TE'
E' → +TE' / ∈
T → FT'
T' → *FT' / ∈
F → (E) / id.

Stack	i/p string	Remark
$E	id+id*id$	E→TE' PUSH
$E'T	id+id*id$	T→FT' PUSH
$E'T'F	id+id*id$	T→FT' PUSH
$E'T'id	id+id*id$	F→id PUSH
$E'T'	+id*id$	F→id POP.
$E'	+id*id$	T'→∈ PUSH
$E'T+	+id*id$	E'→+TE' PUSH
$E'T	id*id$	POP +
$E'T'F	id*id$	T→FT' PUSH.
$E'T'id	id*id$	F→id PUSH
$E'T'	*id$	POP id
$E'T'F*	*id$	T'→*FT' PUSH
$E'T'F	id$	* POP.
$E'T'id	id$	F→id
$E'T'	$	POP id.
$E'	$	POP T
$	$	

3) Elimination of Ambiguity

Stmt → if expr then stmt / if expr then stmt else stmt / other.

The above Grammar G is ambiguous in nature so it has to get eliminated

stmt → S else stmt / other
S → if expr then stmt.

2) check whether the given grammar is LL(1)

S → iAtS / iAtSeS / a
A → b

S → S+S / S*S / a

S → iAtS S' / a
S' → eS / ∈.
A → b

FIRST(S) → { i, a }
FIRST(S') → { e, ∈ }
FIRST(A) = { b }
FOLLOW(S) = { $, FIRST(S') } = { $, e, e }
= { $, e }
FOLLOW(S') = { FIRST(S) FOLLOW(S) } = { $, e }
FOLLOW(A) = { t }

Ktunotes.in

DOWNLOADED FROM KTUNOTES.IN

	c	d	a	e	b	$
S	S→cAdS'		S→a			
S'				S'→ε S'→eS		S'→ε
A					A→b	

Parse tree

Parse tree is generated by syntax Analyser & implemented by YACC.

There are two methods in construction of parse tree

1) Top down parse tree : RDP, LL, OPP

2) Bottom up " " : shift reduce parse tree (SRP)

RDP → Recursive Descent Parse tree

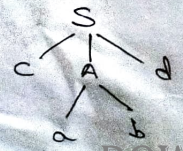
OPP → Operator precedence parse tree.

RDP

Q Construct a parse tree that will accept the i/p string w = cad$ for the above grammar G.

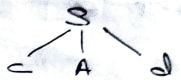
S → cAd

A → ab/a

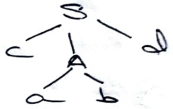


Backtracking

1: Initially create a tree with single node labelled S, and the i/p ptr pointing to the terminal symbol i.e first symbol of w and expand the tree with production of S



2: The left most 'c' matches with w, so the i/p ptr points to the 2nd symbol in w ('a') and it will consider the non-terminal A & expand it with grammar rule A→ab



3: The 2nd symbol 'a' of w also matches with the parse tree, so the advance input ptr will point to the 3rd symbol of w ('d'), but the 3rd leaf of the tree is 'b' doesn't match with the i/p symbol 'd'. Hence it will discard the chosen production & reset back to the 2nd position (i.e A). This is termed as backtracking

4: Now it will try the alternate production of a (i.e d)

DOWNLOADED FROM KTUNOTES.IN