

Classification of Programming Languages

→ Imperative - sequence of execution

1. Von Neumann - memory store both data & instructions
(old)

2. Scripting - smaller chunks of codes Python JS Rb PHP

3. Object Oriented C++ Java

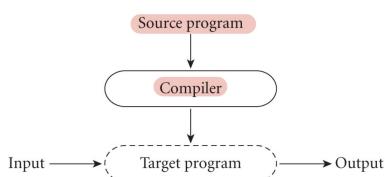
→ Declarative - not specify execution sequences

1. Functional - recursive definition

2. Logic - use logical rules

Program Translation

1. Compiler

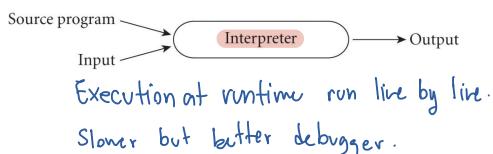


save runtime

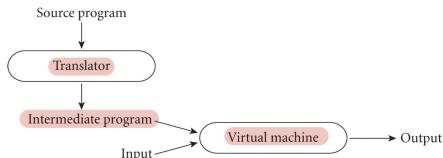
different machine, different compiler

2. Interpreter

* most usage

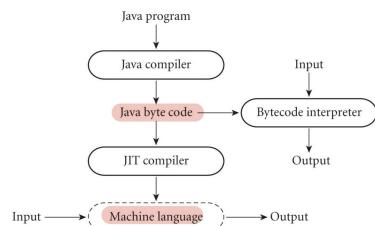


3. Compiler + Interpreter = Virtual Machine



Create Virtual Machine then execute on it.

4. Just In Time Compiler JIT



Intermediate Program got translate into machine code
before every execution ↳ JIT compiler decides

Names and Bindings

Name - identifier to variable, const, operation ...

Binding - association of "Name" and "Data"

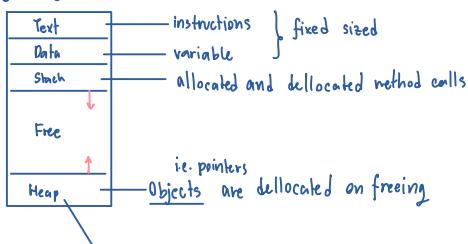
Static Binding * most common
- bound before run time, Compiled lang

Dynamic Binding - bound at runtime, Interpreted Lang

Object Lifetime

↳ time between creation and destruction

Memory Segment



Heap based

↳ Linked data structure or objects

Algorithms to allocate blocks

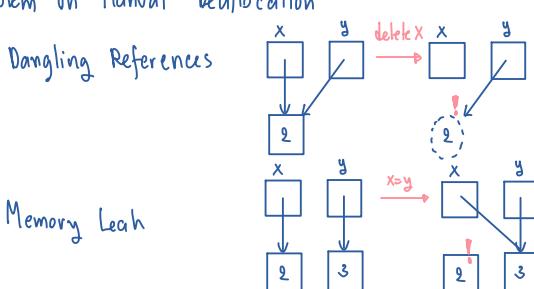


External Fragmentation - free space scattered

Internal Fragmentation - block smaller than minimum reserve threshold

Problem on Manual Deallocation

Dangling References



Stack based

- Arguments
- Return Address
- Bookkeeping (reference register to other frame)
- Temporaries intermediate value storage

Stack pointer - register points at top of stack
negative/positive frame offset when add/remove frame from stack.

Maintenance of Stack

Compiler generates the following:

| Caller | ... push... |
|------------|--|
| ... | ... |
| Pre-call | Push arguments onto the stack. |
| jr callee | Call subroutine. This also pushes the return address onto the stack. |
| Post-call | Deallocate the stack space it allocated in the pre-call (adding positive offset to sp), and continue at the address of the instruction immediately after the jr. |
| ret caller | ... |
| ... | ... |
| Callee | ... ld... |
| Prologue | Push old fp value and update it to a new frame. Push other register values that should not be changed, and push local variables (if any) onto the stack. |
| Main Body | Execute and store return value in a register (or in a location in the stack, if reserved by caller). |
| Epilogue | Restore the saved register values; and deallocate the stack space it allocated in the prologue (adding positive offset to sp). |
| rts | Return to caller, jumping back to the return address. |

Exercise: Write sequence of stack allocation for calling add3()



segmentation fault → set x value to null

memory leak → garbage collection

Garbage Collection

clear unreachable objects

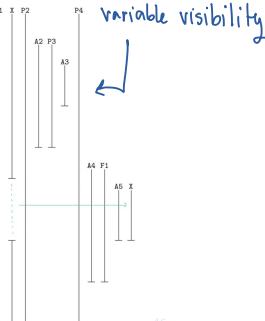
Reduce program error ~ speed and complexity

Scope - region for name-to-object binding

Referencing Environment - active binds at given point of execution

Nested subroutine

```
procedure P1(A1 : T1);  
var X : real;  
...  
procedure P2(A2 : T2);  
...  
procedure P3(A3 : T3);  
...  
begin  
    A3 := A3; (* body of P3 *)  
    ...  
end;  
...  
procedure P4(A4 : T4);  
...  
function F1(A5 : T5) : T6;  
var X : integer;  
...  
begin  
    ... (* body of F1 *)  
end;  
...  
begin  
    P2(); (* body of P4 *)  
    ...  
begin  
    P3(); (* body of P1 *)  
end;
```

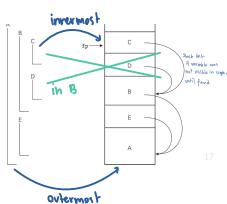


16

Access

static link points to parent frame of recent invocation

Sequence of nested calls at run time is A, E, B, D, C
C can find local objects in surrounding scope B by dereferencing static chain once and adding offset.
C can find local objects in B's surrounding scope, A, by dereferencing static chain twice and adding offset.



17

Static scope & Dynamic scope

determined at compile

determine on flow and control of subroutine

Control Flow

order of program execution

- Sequencing: implicit ordering from top to bottom
- Selection: choice is made among two or more statements
- Iteration: program fragment executed repeatedly
- Procedural abstraction: collection of control constructs encapsulated in a single unit
- Recursion: self-referential subroutines

Expression - anything that produce value

Prefix $\star + 1 \cdot 2$

Infix $1 + 2$

Postfix $a + +$ (del increment)

Iteration - loops

Enumeration - i in range init, stop, steps condition

Logical

Pre-test while

Post-test do... while

Mid-test for

Recursion

Any algorithm can be written loops \longleftrightarrow recursion

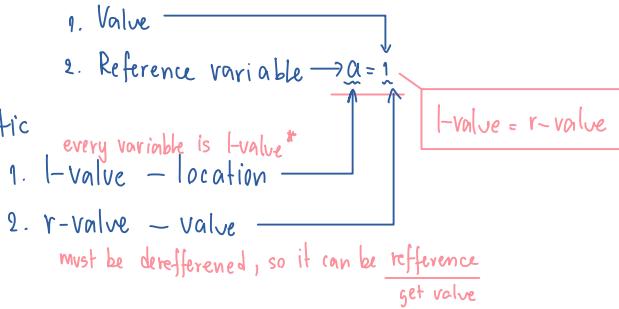
Precidence - Operator priority

$^n + - * / ^n$

Associativity - priority in same level ($L \rightarrow R$)
in absence of parenthesis

Assignment

Schematic



Short circuit

$$F \wedge \underline{\quad} = F$$

no need for execution.

$$T \vee \underline{\quad} = T$$

Sequencing - loops, begin-end.

Selection - if-else statement.

Condition

Nested if

- Large range

Switch statements

- Generate every condition within range

Types - predefined characteristics
allow valid execution

1. Discrete types
 2. Scalar types
 3. Composite types
- countable type
single data item
combine simple \rightarrow larger types
- predecessor and successor

Type System - define certain language constructs

- equivalent rules compatibility rules inference rules
- two values are the same use for determine value for context expression based on surrounding context

Type checking

Compatibility \longrightarrow violation "type clash"

Static vs Dynamic typing

Static, perform at compile time
+ type check

Dynamic - check at runtime

Equivalent Rule

1. Structural eq. - have same components

↑
the order may matters

2. Name eq.

↓
cannot be casted
 \Rightarrow no casting

Compatibility - compatible with that content

↓
automatic implicit conversion

Type Conversion and Cast (2)

Depending on the types involved, conversion may or may not require code to be executed at run time. There are three principal cases:

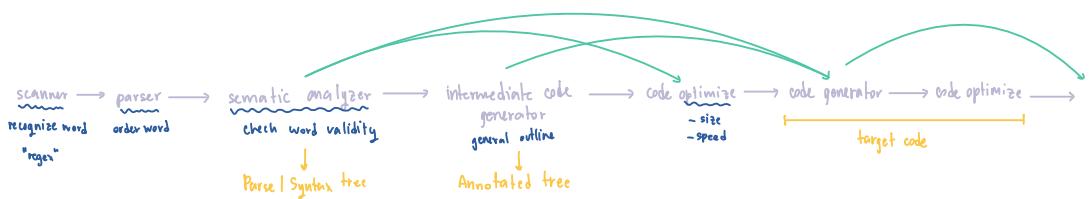
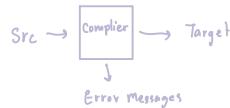
1. If two types are **structurally equivalent** (same low-level representations and set of values) but the language uses name equivalence, no code will need to be executed at run time. (same structure, but different name, e.g. If cast, just check only if it eq. No execution.)
2. The types have different sets of values, but **intersecting values** are represented in the same way (e.g. one type is a subrange of the other, one type is two's complement signed integers and the other is unsigned).
3. The provided type has some values that the expected type does not, code must be executed at run time to check if the current value is valid in the expected type. (no value in common, but different low-level representation)

Universal References - Super types
common in Object Oriented

Inferences - Arithmetic base
the same type as operands.

Compiler

computer program translates source \rightarrow target
 higher level lower level!



Some Data Structure

- Symbol Table : Identifier names / store information of identifiers
 - access throughout
 - use hash for efficiency.

- Literal Table : store constant & string

- Parse tree : Dynamic-allocated pointer structure . Information of different data types.
(Nodes)

Scanning : Stream of character \longrightarrow tokens
meaningful units } Output : stream of tokens

Parsing : Stream of tokens $\xrightarrow[\text{context free}]{\text{correct grammar}}$ parse tree
syntactic structure of program

Semantic : parse tree \longrightarrow syntactic element
Program Correctness

Intermediate Code : parse tree \longrightarrow intermediate language program . Many target languages
Generator easier to understand

Code Generator : Intermediate code / Parse tree \longrightarrow Target Program

Errors : Feedback to detect / report irregularities.

Scanners / lexical analyzers

Character streams → tokens
 ex. identifier, number, string, symbol, reserved word
 meaningful in source languages
 use in next steps



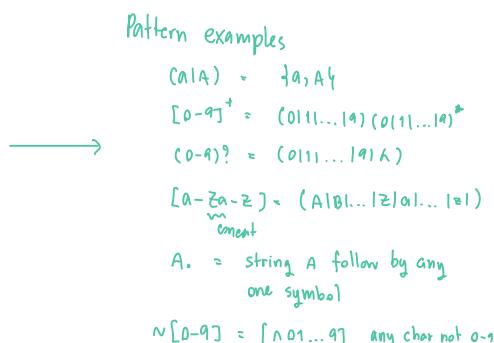
Regex

- character / symbol alphabet
- λ empty string
- \emptyset empty set
- if $r \& s$ is Regex →

| | | | |
|------------|--------|--------------|-------------|
| union | concat | star closure | meta-symbol |
| $r \cup s$ | $r s$ | r^* | (r) |

Extension of Regex

- $[a-z]$ range of a to z
- $.$ any character
- r^+ one or more repetition
- $r^?$ optional subexpression (0 or 1)
- \sim Not symbol



Disambiguating Rules

Identifier is considered reserved word.

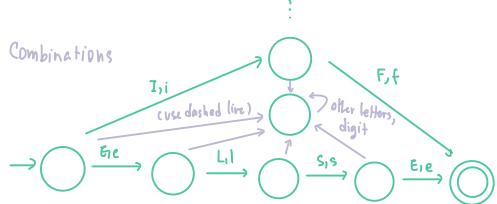
Longest substring, single token interpolation

"c= is c= not < and ="

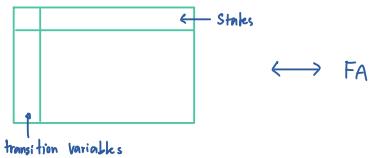
FA Recognizing Token



Combine FA's



Transition Table



Simulate DFA

```
init current_state = start      ← init
while (not final(current_state)) :   ← loop till accepting state
    next_state = dfa (current_state, next) ← next state with condition
    current_state = next_state   ← update current state
    ↴
```

Error Handling : Minor adjustment fixing the program

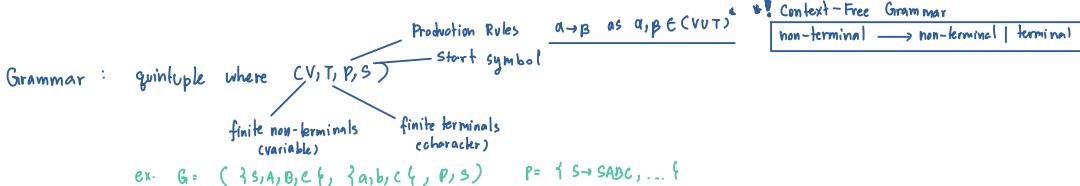
Buffering

1. Single Buffer
2. Buffer Pair
3. Sentinel

Grammar

Context-Free Grammars : Used in Parsers

Parsing Process : Tokens $\xrightarrow[\text{from scanner}]{\text{add info into symbol table}}$ Parse Tree



Bachus - Navr - Form (BNF)

Non-terminals in $<>$, Terminals are other symbols

$::=$ means \rightarrow

$|$ means or

ex. $E ::= E E | (E) | ID$

Derivation : Replace substring in sentential form.

Def. $G = (V, T, P, S)$ be CFG. $a, b, \gamma \in (V \cup T)^*$ and $A \in V$. That $aAp \Rightarrow_G a\alpha\beta p \rightarrow A \rightarrow \gamma$

\Rightarrow_G^* : derivation in zero or more steps

ex.

$S \rightarrow SS | (S)S | () | \lambda$

starting symbol \boxed{S}

$\begin{aligned} &\rightarrow SS \\ &\rightarrow (S)SS \\ &\rightarrow ((S)S)S \\ &\rightarrow ((C))S(C) \end{aligned}$

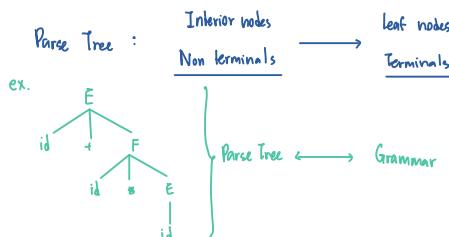
Leftmost / Rightmost Derivation : For sequential & simplicity
Replace Non-terminal at most.

Language Derived from Grammar $L(G) = \{ w \mid S \xrightarrow{*} G w \mid w \text{ is string}\}$

Right / Left Recursive

Right Derivation: $A \rightarrow^* AX$

Left Derivation: $A \rightarrow^* XA$

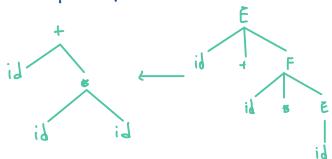


| $E \rightarrow EOE \mid (E) \mid ID$ | <u>Leftmost</u> | <u>Rightmost</u> |
|--------------------------------------|---------------------------|---------------------------|
| $\rightarrow +$ | $\rightarrow EOE$ | $\rightarrow EOE$ |
| $\rightarrow ()$ | $\rightarrow EOEOE$ | $\rightarrow EOEOE$ |
| $\rightarrow ID$ | $\rightarrow (EOE)OE$ | $\rightarrow E * ID$ |
| | $\rightarrow (IDOE)OE$ | $\rightarrow (E) * ID$ |
| | $\rightarrow (ID+ID)OE$ | $\rightarrow (EOE)* ID$ |
| | $\rightarrow (ID+ID)OE$ | $\rightarrow (EO ID)* ID$ |
| | $\rightarrow (ID+ID)* E$ | $\rightarrow (E+ID)* ID$ |
| | $\rightarrow (ID+ID)* ID$ | $\rightarrow (ID+ID)* ID$ |

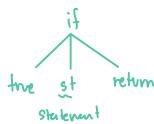
Abstract Tree \leftrightarrow Parse Tree

Nodes : Operators

Leaves : Operands



If statement in Abstract tree



Ambiguous Grammar : ≥ 2 Grammar generate certain string ex. $id + id * id$ | $id * id + id$
both valid grammar

Fix in Expressions

1. Precedence (priority of each expression)

2. Associativity (R \rightarrow L or L \rightarrow R in eg. priority)

Precedence

$$\begin{array}{l} E \Rightarrow E+E \\ E \Rightarrow E \cdot E \end{array}$$

Fix

$$\begin{array}{l} E \Rightarrow E+E \\ E \Rightarrow F \\ F \Rightarrow F+F \quad \text{F is innermost} \\ \text{or} \\ F \Rightarrow X \\ X \Rightarrow (E) \quad \text{Expression in () have higher priority} \end{array}$$

F have highest priority

Extended Backus - Naur Form (EBNF)

Kleen's Star / Kleen's Closure seq ::= st*;st* seq ::= *st;st*

Optional Part

$$E ::= F [+ E] \mid F [- E]$$

optional

Syntax Diagrams

Graphical Representation of EBNF

non-terminals

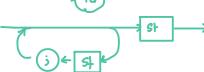
terminals

sequence & choices

ex X ::= (E) | id



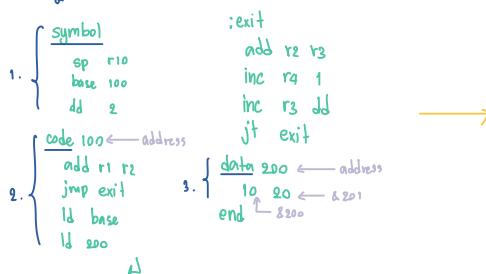
seq ::= fst;fst



E ::= F [+ E]



Assembly \rightarrow Context-Free Grammar



symbol section :exit
symbol section :exit
symbol section :exit

symbol section :exit
symbol section :exit
symbol section :exit

symbol section :exit
symbol section :exit
symbol section :exit

symbol section :exit
symbol section :exit
symbol section :exit

symbol section :exit
symbol section :exit
symbol section :exit

symbol section :exit
symbol section :exit
symbol section :exit

symbol section :exit
symbol section :exit
symbol section :exit

Recursions



Define basic operations:

head(L), tail(L), cons(x, L)

that: cons(head(L), tail(CL))

basic identifier

number(n) string(s) isatom(x) \leftarrow smallest form

$(1\ 2) \rightarrow \text{cons}(\text{number}(1), \text{cons}(\text{number}(2), \text{NIL}))$

\uparrow
terminating symbol

Looping Iteration

count(L)

```
s = 0
while L != NIL
    L = tail(L)
    s += 1
return s
```

loop through iteration

count(L)

Recursion \rightarrow

```
if L == NIL  $\leftarrow$  terminate case
    return 0
return 1 + count(tail(L))  $\leftarrow$  recursion
     $\uparrow$  add 1 each recursion
```

count3(L)

```
if L == NIL
    return 0
if isatom(head(L)) return 1 + count3(tail(L)) (2.)
return count3(head(L)) + count3(tail(L))
     $\downarrow$  if this is atom  $\rightarrow$  1 + count3(tail(CL))
         $\underbrace{\quad}$  as same as 2.
```

Compiler & Parsing See video. So hard! . See example in Writing Grammar from site's.

Parsing — Process of syntactic tree structures from stream of tokens

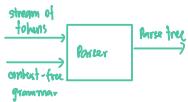
Top down \rightarrow Root \rightarrow Leaves . Trace Leftmost pre-order traversal

Bottom up \rightarrow Leaves \rightarrow Root . Trace Rightmost post-order traversal

Predictive parser

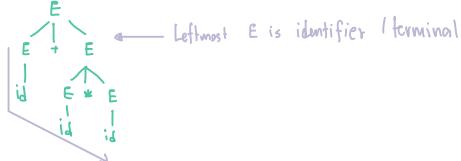
Backtracking parser

Mostly uses this!



Top Down

$E \rightarrow E + E$



Parser need to decide production rule at each point.

Guess based on

- Next word token
- Structure being built

Difficult task:

1. Cannot decide until later :

if - else block , Next token : if Structure : st (statement)

st \rightarrow MatchedSt | UnmatchedSt

MatchedSt \rightarrow if MatchedSt else MatchSt

UnmatchedSt \rightarrow if (E) St | if (E) MatchSt else UnmatchSt

2. Empty String Production

Next token : id Structure : par

par \rightarrow parList | λ

parList \rightarrow exp, parList | exp

Recursive - Descent

$E \rightarrow EOF \mid F$

$O \rightarrow + \mid -$

$F \rightarrow (E) \mid id$

Rewrite into
Recursive loop

In EBNF

(Extended Backus-
Naur Form)

$E ::= F \mid O F \mid$

$O ::= + \mid -$

$F ::= (E) \mid id$

Procedure E ?
F;
while (token = + || token = -)
{ O; F; }
}

Match procedure : Match the consume token

procedure match (expt) {

if (token = expt)

then getToken \leftarrow Token not consumed until getToken is executed

else error

}

1. Difficult Conversion to EBNF

2. Can't decide production rule

3. Can't decide λ -production : $A \rightarrow \lambda$ break ? continue

Problems

LL(1) Parsing : Stack, simulate left-most derivation

- ↳ input : $L \rightarrow R$
- ↳ Leftmost Derivation
- ↳ lookahead symbol

$(C[n] \cup (Ch[n]) \mid n \mid \$)$

↑ left most

$E \rightarrow tX$
 $X \rightarrow ATX \mid \lambda$
 $A \rightarrow + \mid -$
 $T \rightarrow FN$
 $N \rightarrow MFN \mid \lambda$
 $M \rightarrow *$
 $F \rightarrow (E) \mid n$

Select rules that
leads | matches
terminal symbol.



Left go in Last. Right in First! top : Non-terminal else pop-terminal

Repeat stack procedure until → match desired input

• Part sentential form in stack

Choosing Rule / Grammar (Context Free)

When non-terminal is at top of stack :

Choose Rule " $N \rightarrow X$ " that

"First set" 1. $X \rightarrow *tY$ } leads to terminal (left \sqsubset middle)
"Followed set" 2. $X \rightarrow * \lambda$ and $S \rightarrow *tWtY$

Example

$exp \rightarrow exp \text{ addop term} \mid$

term

addop $\rightarrow + \mid -$

term $\rightarrow \text{term mulop factor} \mid$

factor

mulop $\rightarrow *$

factor $\rightarrow (exp) \mid \text{num}$

First(addop) = {+, -}

First(mulop) = {*}

First(factor) = {, num}

First(term) = {, num}

First(exp) = {, num}

Given follows by terminal

Use Rule to find value → non-terminal!

First set

let $X = \lambda$ or $\in V_T$

$\text{First}(x) \quad \{x\} : x \text{ is terminal}$
 First terminal in any sentential form derived from "X"
 $\text{First}(x_i) - \{ \lambda \} \subset \text{First}(x) \quad \forall i, j \mid j < i \quad \{ \lambda \} \in \text{First}(x_j)$
 $\exists \lambda \in \text{First}(x) \quad \forall j \leq n \quad \lambda \in \text{First}(x_j)$

Fills all grammar that $V \rightarrow * \text{term}$ by First set ($X \rightarrow *tY$) all terms !

Follow Set self reading

ends at $\{NNT^k\}$
Non-terminal but can be " λ "
from $X \rightarrow * \lambda$

Construct LL(1) Parse Table : See video

Example: Constructing LL(1) Parsing Table

Fill First, Any item that (fill A) looks back up to follows

| | First | Follow | |
|-----------------------------|---------|---------|--|
| exp | {, num} | {\\$} | |
| exp' | {+, -} | {\\$} | |
| addop | {+} | {, num} | |
| term | {, num} | {, num} | |
| term' | {, num} | {, num} | |
| mulop | {*} | {, num} | |
| factor | {, num} | {, num} | |
| num | {, num} | {, num} | |
| else | | | |
| 2 exp | | | |
| 2 exp → addop term exp' | | | |
| 4 addop → + | | | |
| 5 addop → - | | | |
| 6 term → factor term' | | | |
| 7 term → mulop factor term' | | | |
| 8 term → * | | | |
| 9 num → 1 exp | | | |
| 10 factor → (exp) | | | |
| 11 factor → num | | | |

Rule

mulop factor term'

LL(1) ← parsing table has at most one production in each entry.

not

1. Left-recurr : $A \Rightarrow Y^k = \{Y_1, Y_2, \dots, Y_m\} \{X_1, X_2, \dots, X_m\}^*$
In $A \rightarrow X \rightarrow AY$ common form

Remove left Recur
1. No empty string
2. no cycle

Ex.

$$\begin{array}{l}
 E \rightarrow E + T \mid T \\
 T \rightarrow T * F \mid F \\
 F \rightarrow (E) \mid id
 \end{array}
 \xrightarrow{\text{Left Recursion}}
 \begin{array}{l}
 E \rightarrow TE' \\
 E' \rightarrow +TE' \mid \epsilon \\
 T \rightarrow FT' \\
 T' \rightarrow *FT' \mid \epsilon \\
 F \rightarrow (E) \mid id
 \end{array}$$

First
Follow

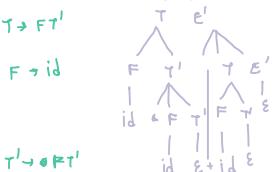
$$\begin{array}{l}
 x \rightarrow E \\
 x \rightarrow y_1 y_2 \dots y_n \\
 \text{First}(E) = \text{First}(T) = \text{First}(F) = \{ C, id \} \\
 \text{First}(E') = \{ +, \epsilon \} \\
 \text{First}(T') = \{ *, \epsilon \}
 \end{array}$$



$$\begin{array}{l}
 1) \$ \\
 2) \text{If } A \rightarrow a \underline{B} \beta \rightarrow \text{any answer in this format is acceptable} \\
 3) \text{If } A \rightarrow a B \\
 \text{Follow}(E) = \{ \$, + \} \downarrow \text{Follow}(E) \\
 \text{Follow}(E') = \{ \$ \} \downarrow \text{Follow}(E') \\
 \text{Follow}(T) = \{ +, \$ \} \downarrow \text{Follow}(T) \\
 \text{Follow}(T') = \{ *, +, \$ \} \downarrow \text{Follow}(T') \\
 \text{Follow}(F) = \{ *, +, \$ \} \downarrow \text{Follow}(F) \\
 \hline
 \text{Follow}(T) \cap \text{Follow}(T') = \{ +, \$ \}
 \end{array}$$

Predictive Parsing Table

| | + | * | (|) | id | \$ | Stack | Input | Output | |
|------|---------------------------|----------------------|---------------------------|---------------------------|----|----|----------------|-------------------|------------------------|---------------------------|
| E | $E \rightarrow TE'$ | | $E' \rightarrow \epsilon$ | $E' \rightarrow \epsilon$ | | | $\$ E$ | $id * id + id \$$ | $E \rightarrow TE'$ | E |
| T | | $T \rightarrow FT'$ | | $T' \rightarrow FT'$ | | | $\$ E' T$ | $id * id + id \$$ | $T \rightarrow FT'$ | T |
| T' | $T' \rightarrow \epsilon$ | $T' \rightarrow FT'$ | $T' \rightarrow \epsilon$ | $T' \rightarrow \epsilon$ | | | $\$ E' T F$ | $id * id + id \$$ | $T \rightarrow id$ | E' |
| F | | $F \rightarrow (E)$ | | $F \rightarrow id$ | | | $\$ E' T F id$ | $id * id + id \$$ | $F \rightarrow id$ | F |
| | | | | | | | $\$ E' T F *$ | $id * id \$$ | $T' \rightarrow * FT'$ | T' |
| | | | | | | | $\$ E' T F$ | $id * id \$$ | | $E' \rightarrow + TE'$ |
| | | | | | | | $\$ E' T id$ | $id + id \$$ | $F \rightarrow id$ | E |
| | | | | | | | $\$ E' T$ | $+ id \$$ | | $T \rightarrow \epsilon$ |
| | | | | | | | $\$ E'$ | $+ id \$$ | | $E' \rightarrow + TE'$ |
| | | | | | | | $\$ E' T +$ | $+ id \$$ | | $E' \rightarrow + TE'$ |
| | | | | | | | $\$ E' T$ | $id \$$ | | $T \rightarrow \epsilon$ |
| | | | | | | | $\$ E' T F$ | $id \$$ | | $T \rightarrow FT'$ |
| | | | | | | | $\$ E' T id$ | $id \$$ | | $F \rightarrow id$ |
| | | | | | | | $\$ E' T$ | $\$$ | | $E' \rightarrow + TE'$ |
| | | | | | | | $\$ E'$ | $\$$ | | $T' \rightarrow \epsilon$ |
| | | | | | | | $\$$ | $\$$ | | $E' \rightarrow \epsilon$ |



testpgm \rightarrow line EOFline \rightarrow LNUM stmt line | emptystmt \rightarrow asgmt | gotoasmgt \rightarrow ID = expexp \rightarrow ID expsexps \rightarrow + exp | emptygoto \rightarrow GOTO LNUM1. pgm \rightarrow line2. pgm \rightarrow EOF3. line \rightarrow LNUM stmt line4. line \rightarrow empty5. stmt \rightarrow asgmt6. stmt \rightarrow goto7. empty \rightarrow ID exps8. exps \rightarrow + exp9. exps \rightarrow empty10. goto \rightarrow GOTO LNUM

First

$$\begin{aligned} \text{FIRST}(\text{pgm}) &= \{\text{LNUM}, \text{EOF}\} \\ \text{FIRST}(\text{line}) &= \{\text{LNUM}, \text{empty}\} \\ \text{FIRST}(\text{stmt}) &= \{\text{ID}, \text{GOTO}\} \\ \text{FIRST}(\text{asmgt}) &= \{\text{ID}\} \\ \text{FIRST}(\text{exp}) &= \{\text{ID}\} \\ \text{FIRST}(\text{exps}) &= \{+, \text{empty}\} \\ \text{FIRST}(\text{goto}) &= \{\text{GOTO}\} \end{aligned}$$

Parsing Table

| | LNUM | ID | GOTO | + | = | empty | EOF |
|-------|------|----|------|----|---|-------|-----|
| pgm | 1 | | | | | | 2 |
| line | 3 | | | | | | 4 |
| stmt | | 5 | 6 | | | | |
| asmgt | | 7 | | | 8 | | |
| exp | | 9 | | | | | |
| exps | | | 10 | 11 | | | 11 |
| goto | | | 12 | | | | |

Program is:

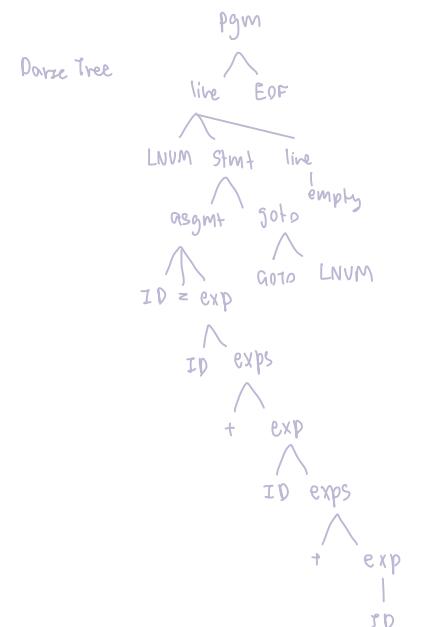
10 A = B

30 B = A + A + A

40 GOTO 10

So input is \$LNUM, ID, =, ID, +, ID, +, ID, GOTO, LNUM, EOF \$

Parse Tree



Yours truly P.

V430n72027