Classification of Programming Languages

→ Imperitive – sequence of execution

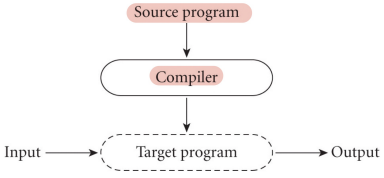
1. Von Neumann (old) – memory store both data & instructions
2. Scripting – smaller chunks of codes Python JS Rb PHP
3. Object Oriented C++ Java

→ Declarative – not specify execution sequences

1. Functional – recusive definition
2. Logic – use logical rules

Program Translation

1. Complier

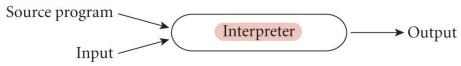


save runtime

different machine, different complier

2. Interpreter

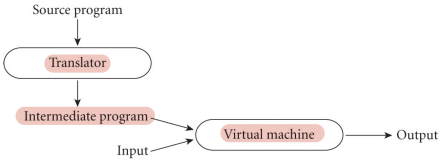
* most usage



Execution at runtime run line by line.

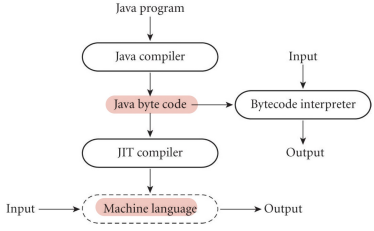
Slower but better debugger.

3. Complier + Interpreter = Virtual Machine



Create Virtual Machine then execute on it.

4. Just In Time Complier JIT



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

Names and Bindings

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

Binding - association of "Name" and "Data"

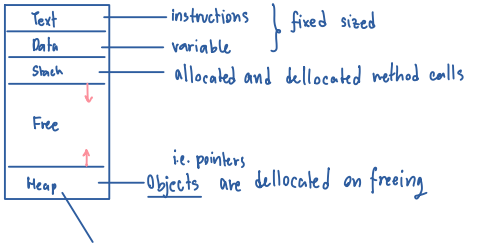
Static Binding * most commons - bound before run time, Compiled lang

Dynamic Binding - bound at runtime, Interpred Lang

Object Lifetime

└ time between creation and destruction

Memory Segment



Stack based

- Argumnts
- Return Address
- Bookeeping (reference register to other frame)
- Temporaries intermediated value storage

Stack pointer - register point at top of stack

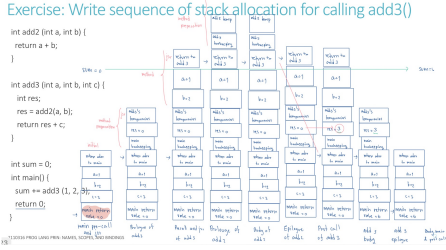
negative / positive frame offset when add/remove frame from stack.

Maintenance of Stack

Compiler generates the following:

Caller	
...	
Pre-call	Push arguments onto the stack.
jsr 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 jsr.
...	
Callee	
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()



Heap based

↳ Linked data structure or objects

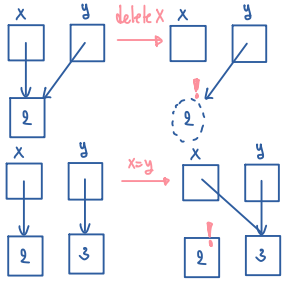
Algorithms to allocate blocks

External Fragmentation - free space scattered

Internal Fragmentation - bloch smaller than minimum reserve threshold

Problem on Manual Deallocation

Dangling References



segmentation fault → set x valve to null

Memory Leak

memory leak → garbage collection

Garbage Collection

clear unreachable objects

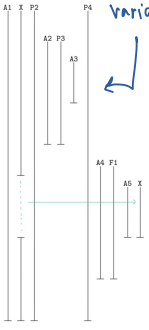
Reduce program error ∼ slow speed and complexity

Scope – region for name-to-object binding

Refferencing 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
         ... X := A3;   (* body of P3 *)
      end;
      ...
   begin
      ... P3(3);        (* body of P2 *)
   end;
   ...
   procedure P4(A4 : T4);
      ...
      function F1(A5 : T5) : T6;
      var X : integer;
      ... { hides X }
      begin
         ...            (* body of F1 *)
      end;
      ...
   begin
      ... P2(2);        (* body of P4 *)
   end;
   ...
begin
   ... P4(4);           (* body of P1 *)
end
```



variable visibility

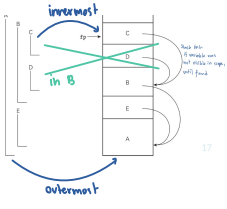
Access

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.



Static scope & Dynamic scope

determied 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	* + 1 3 2
Infix	1+2
Postfix	a++ (del increment)

Precedence – Operator priority

" + - * / "

Associativity – priority in same level (L→R)

in absence of parenthesis

Assignment

1. Value
2. Reference variable → a = 1

l-value = r-value

Schematic

every variable is l-value*

1. l-value – location
2. r-value – value

must be dereferenced, so it can be refference / get value

Short circuit

F ∧ ____ ≡ F

T ∨ ____ ≡ T

no need for execution.

Sequencing – loops, begin-end.

Selection – if-else statement.

Condition

Nested if	Switch statements
– Large range	– Generate every condition within range

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 ⟷ recursion

Types – predefined characteristics
allow valid execution

1. Discrete types
countable type
predecessor and successor

2. Scalar types
single data item

3. Composite types
combine simple → larger types

Type System – define certain language constructs

equivalent rules
two values are the same

compatibility rules
use for determine value for context

inference rules
expression based on surrounding content

Type checking
Compatibility ⟶ 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. Naive eq.
↑
cannot be casted

Compatibility – compatible with
↳ no casting
that content
↓
automatic implicit conversion

Universal References – Super types
common in Object Oriented

Inferrences – Arithmetic base
the same type as operands.

Type Conversion and Cast (2)

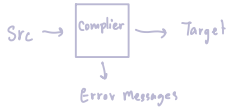
Depending on the types involved, conversion may or may not require code to be executed at run time. There are three principal cases:

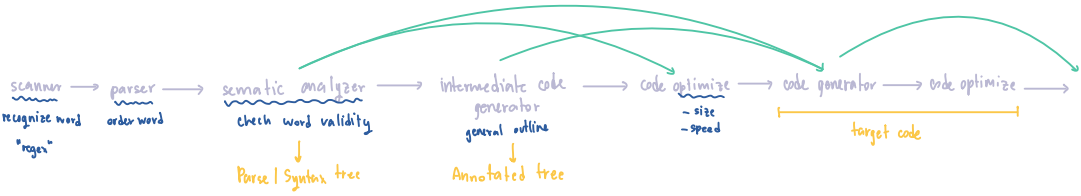
1. If two types are **structurally equivalent** (same low-level representations and set of values) but the **language uses name equivalence, no code will need to be executed at run time.**
2. The types have different sets of values, but **intersecting values** are represented in the same way (e.g. one type is a subrange of the other, one type is two's complement signed integers and the other is unsigned).
   - If 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.**
3. The types have **different** low-level representations but a **correspondence among their values** can be defined (e.g. integer to floating-point, floating-point rounded to integer). Most **processors provide a machine instruction** for this.

Complier

computer program translate source → target

higher level lower level





Some Data Structure

1. Symbol Table : Identifier names / store information of identifiers
   - access throughout
   → use hash for efficientcy.
2. Literal Table : Store constant & string
3. Parse tree : Dynamic-allocated pointer structure. Information of different data types.
   (Nodes)

Scanning : Stream of character ⟶ tokens (meaningful units) } Output : stream of tokens

Parsing : Steam of tokens ⟶ (correct grammar, context free) parse tree
"syntactic structure of program"

Sematic : parse tree ⟶ syntatic element
Program Correctness

Intermediate Code Generator : parce tree ⟶ intermediate language program. Many target languages
easier to understand
* having Intermediate: translate to

Code Generator : Intermediate code / Parse tree ⟶ Target Program

Errors : Feedbach 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

Construction : Regex

Define tokens —Regex, FA→ Token Pattern —Combine FA (Patterns combine) → NFA → DFA → Program simulate DFA

Regex

- character / symbol alphabet
- λ empty string
- $\emptyset$ empty set
- if r & s is Regex ⟶ union r|s, concat rs, star closure r^*, meta-symbol (r)

Extension of Regex

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

Pattern examples

(a|A) = {a, A}

$[0-9]^+$ = (0|1|...|9)(0|1|...|9)*

(0-9)? = (0|1|...|9|λ)

[a-za-z] = (A|B|...|z|a|...|z) (concat)

A. = string A follow by any one symbol

^[0-9] = [^01...9] any char not 0-9

(Define) Describe Pattern example

reserved IF = (IF, iF, If | if)

letter = [a-zA-Z]

digit = [0-9]

identifier = letter (letter | digit)*

Disambiguating Rules

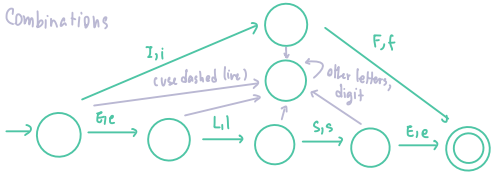
Identifier is considered reserved word.

Longest substring, single token interpolation

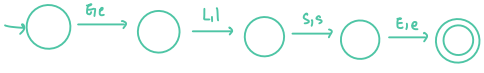
"<= is <= not < and ="

FA Recognizing Token

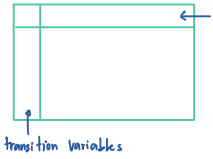
Combinations



Combine FA's



Transition Table



⟷ FA

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 (from scanner) → (add info into symbol table) Parse Tree

Grammar : quintuple where (V, T, P, S)
- V: finite non-terminals (variable)
- T: finite terminals (character)
- P: Production Rules — $\alpha \to \beta$ as $\alpha, \beta \in (V \cup T)^*$
- S: start symbol

ex. $G = (\{S, A, B, C\}, \{a, b, c\}, P, S)$ $P = \{ S \to SABC, \dots \}$

* ! Context-Free Grammar
non-terminal ⟶ non-terminal | terminal

Backus-Navr-Form (BNF)

Non-terminals in < > , Terminals are other symbols

::= means ⟶

| means or

ex

<E> ::= <E><E> | (<E>) | ID

Derivation : Replace substring in sential form.

Def $G = (V, T, P, S)$ be CFG, $\alpha, \beta, \gamma \in (V \cup T)^*$ and $A \in V$. That $\alpha A \beta \Rightarrow_G \alpha\gamma\beta \longrightarrow A \to \gamma \in P$

$\Rightarrow_G^*$: derivation in zero or more steps

ex.

$S \to SS \mid (S)S \mid ()$

starting symbol S

$\to SS$
$\to (S)SS$
$\to (S)S(S)S$
$\to_G^* (())(())$

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

$E \to EOE \mid (E) \mid ID$

$O \to + \mid *$

Leftmost

E
→ EOE
→ (E)OE
→ (EOE)OE
→ (IDOE)OE
→ (ID+E)OE
→ (ID+ID)OE
→ (ID+ID)*E
→ (ID+ID)*ID

Rightmost

E
→ EOE
→ EOID
→ E*ID
→ (E)*ID
→ (EOE)*ID
→ (EOID)*ID
→ (E+ID)*ID
→ (ID+ID)*ID

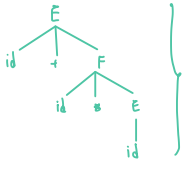
Language Derived form Grammar $L(G) = \{ w \mid S \to_G^* w \}$ w is string

Right / Left Recursive

Right Derivation : $A \to^* Ax$
Left Derivation : $A \to^* xA$

Parse Tree : Interior nodes (Non terminals) ⟶ leaf nodes (Terminals)

ex.

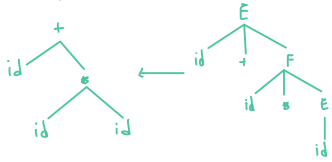


Parse Tree ⟷ Grammar

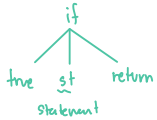
Abstract Tree ⟷ Parse Tree

Nodes : Operators

Leafs : 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→L or L→R in eq. priority)

Precedence

E → E+E
E → E*E

Fix →

E → E+E
E → F
F → F*F ← F is innermost, F have highest priority
or
F → X
X → (E) ← Expression in ( ) have higher priority

Extended Backus – Naur Form (EBNF)

Kleen's Star / Kleen's Closure seq ::= st {; st} seq ::= {st ;} st

Optional Part E ::= F[+E] | F[-E]

optional

Syntax Diagrams

Graphical Representation of EBNF

non-terminals

terminals

sequence & choices

ex X ::= (E) | id

seq ::= {st;} st

E ::= F[+E]

Assembly ⟶ Context-Free Grammar

1.
```
symbol
    sp   r10
    base 100
    dd   2
```

2.
```
code 100 ← address
    add r1 r2
    jmp exit
    ld  base
    ld  200
```

3.
```
data 200 ← address
    10  20 ← &201
end ← &200
```

```
:exit
    add r2 r3
    inc r4 1
    inc r3 dd
    jt  exit
```

symbol section, code section, data section

program = symsec codesec datasec

symsec = thsym sympair

thsym: token symbol (location) — leave it as this

sympair = thname symval sympair | nil

thname: name; symval: value

codesec = thcode thnumber codeline

codeline = thLabel oprag | oprag | nil

thLabel: token label (location)

oprag = oprags codeline

oprags: operand defined in ASM

datasec = thdata thnumber dataline

thnumber: location

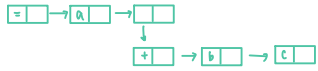
dataline = thnumber dataline | thend

thnumber: value; thend: end symbol

Recursions

Outline $a = b + c$ → Root

→ Assignment $= a + b\ c$ → in data structures (As "List")



Define basic operations:

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

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

basic identifier

number(n) string(s) isatom(x) ← smallest form

(1 2) ⟶ cons(number(1), cons(number(2), NIL))

NIL ↑ terminating symbol

Looping Iteration

```
Count(L)
    S = 0
    while L != NIL
        L = tail(L)
        S++
    return S
```

} loop through iteration

Recursion →

```
count2(L)
    if L = = NIL ← terminate case
        return 0
    return 1 + count(tail(L)) ← recursion
                ↑ 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))
                ↳ if this is atom ⟶ 1 + count3(tail(L))
                                    as same as 2.
```

Compiler & Parsing

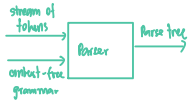
See video. So hard ! . See example in Writing Grammar from site's.

Parsing — Process of syntatic tree structures from stream of tokens

- Top down → Root → Leaves . Trace Leftmost — pre-order traversal
- Bottom up → Leaves → Root . Trace Rightmost — post_order traversal

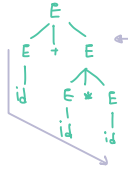
Predictive parser / Bachtraching parser

Mostly uses this !



Top Down

E → E+E



← Leftmost E is identifier / terminal

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 → MatchedSt | UnmatchedSt

   MatchedSt → if MatchedSt else MatchSt

   UnmatchedSt → if (E) St | if (E) MatchSt else UnmatchedSt

2. Empty String Production

   Next token : id Structure : par

   par → parList | ^

   parlist → exp, parList | exp

Recursive – Descent

E → EOF | F

O → + | –

F → (E) | id

Rewrite into Recursive loop → In EBNF (Extended Backus-Naur Form)

E ::= F {O F}

O ::= + | –

F ::= (E) | id

```
Procedure E ;
    F;
    While (token = + || token = -)
        {O; F;}
;
```

Match procedure : Match the consume token

```
procedure match (expth) ;
    if (token = expth)
        then getToken ← Token not consumed until getToken is executed
        else error
;
```

Problems
1. Difficult Conversion to EBNF
2. Can't decide production rule
3. Can't decide ^-production : A → ^ ← break ? continue

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

- input : L → R
- leftmost Derivation
- 1: look ahead symbol

|(|n|+|(n)|)|*|n|$

↑ left most

E → TX
X → ATX | λ
A → + | -
T → FN
N → MFN | λ
M → *
F → (E) | 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

Stack

- Part sentential form in stack

Choosing Rule / Grammar (Context Free)

When non-terminal is at top of stack :

Choose Rule "N → X" that

"First set" 1. X → *tY
"Followed set" 2. X → *λ and S → *WNtY (non-terminal marked) } leads to terminal (left ≥ middle)

Given follows by terminal

Example

```
exp     → exp addop term |
          term
addop   → + | -
term    → term mulop factor |
          factor
mulop   → *
factor  → (exp) | num
```

```
First(addop) = {+, -}
First(mulop) = {*}
First(factor) = {(, num}
First(term)   = {(, num}
First(exp)    = {(, num}
```

← use Rule to find value → non-terminal

First Set

Let X = λ or ∈ V, T

First (X)

{X} : X is terminal

else

1. First(X1) - {λ} ⊂ First(X)
2. First(Xi) - {λ} ⊂ First(X) ∀(i,j) | j < i {λ} ∈ First(Xj)
3. λ ∈ First(X) ∀(j) ≤ n . λ ∈ First(Xj)

First terminal in any sentential form derived form "X"

Fills all grammar that V → *T by First set (X → *tY) all terms !

Follow Set self Reading

ends at { N N t Y }

terminal

Non terminal but can be "λ" from [X → *λ]

Construct LL(1) Parse Table : See video

Fill First, Any term that (fill λ), leads back up to follows

Example: Constructing LL(1) Parsing Table

	First	Follow
exp	{(, num}	{$,)}
exp'	{+, -, λ}	{$,)}
addop	{+,-}	{(,num}
term	{(,num}	{+,-,),$}
term'	{*, λ}	{+,-,),$}
mulop	{*}	{(,num}
factor	{(, num}	{*,+,-,),$}

Rules:
1 exp → term exp'
2 exp' → addop term exp'
3 exp' → λ
4 addop → +
5 addop → -
6 term → factor term'
7 term' → mulop factor term'
8 term' → λ
9 mulop → *
10 factor → (exp)
11 factor → num

	(	)	+	-	*	n	$
exp	1					1	
exp'		3	2	2			3
addop				4	5		
term	6					6	
term'		8	8	8	7		8
mulop					9		
factor	10					11	

① Rule

λ

→ mulop factor term'

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

not

1. Left-recur : A = YX* = {Y1, Y2, ..., Ym} {Y1, X2, ..., Xn}* In A → X → *AY common form
2. Left Factor

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

Ex.

$E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$

Eliminate Left Recursion →

$E \to TE'$
$E' \to +TE' \mid \varepsilon$
$T \to FT'$
$T' \to *FT' \mid \varepsilon$
$F \to (E) \mid id$

First / Follow →

$X \to \varepsilon$
$X \to y_1 y_2 \dots y_n$

First terminal: $F \to (E) \mid id$

$First(E) = First(T) = First(F) = \{ (, id \}$
$First(E') = \{ +, \varepsilon \}$
$First(T') = \{ *, \varepsilon \}$

1) $Follow(E) = \{ \$,) \}$
2) If $A \to \alpha B \beta$ ⟶ any answer in this format is acceptable
3) If $A \to \alpha B$

$F \to (E)$ — Right

$Follow(E) = \{ \$,) \}$
$Follow(E') = \{ \$,) \}$ — $E \to TE'$ (Follow(E))
$Follow(T) = \{ +, \$,) \}$ — $E' \to +TE'$
$Follow(T') = \{ +, \$,) \}$
$Follow(F) = \{ *, +, \$,) \}$ — $T \to FT'$

$T \to T'$

Predictive Parsing Table

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

⟶

Stack	Input	Output
$E	id * id + id $	
$E'T	id * id + id $	$E \to TE'$
$E'T'F	id * id + id $	$T \to FT'$
$E'T'id	id * id + id $	$F \to id$
$E'T'	* id + id $	
$E'T'F*	* id + id $	$T' \to *FT'$
$E'T'F	id + id $	
$E'T'id	id + id $	$F \to id$
$E'T'	+ id $	
$E'	+ id $	$T' \to \varepsilon$
$E'T+	+ id $	$E' \to +TE'$
$E'T	id $	
$E'T'F	id $	$T \to FT'$
$E'T'id	id $	$F \to id$
$E'T'	$	
$E'	$	$T' \to \varepsilon$
$		$E' \to \varepsilon$

output →

LL(1) Parser

First( ) — first terminal(s) of Non-Term

i.e. S → aABC
A → b
B → c
C → d

First(S): S → a A B C

First(A): A → b

First(B): B → c

First(C): C → d

i.e. S → ABC
A → a | ε
B → b
C → c

First(S): S → A B C (A → a) ; S → A B C (A → ε, B → b)

{a, b}

doesn't count

Follow( ) Terminal that follow Non-terms.

Non-Term Term

S → ABC
A → a
B → b | ε
C → c

S $ ← stack symbol

Follow(S): {$}
Follow(A): {b, c}
Follow(B): {c}
Follow(C): {$}

---

ex. E → TE'
E' → +TE' | ε (T follow by E')
T → FT'
T' → *FT' | ε
F → id | (E)

	First	Follow
E	{id, (}	{$,)}
E'	{+, ε}	{$,)}
T	{id, (}	{+, $,)}
T'	{*, ε}	{+, $,)}
F	{id, (}	{*, +, $,)}

First: Non-Terms
First(F) = {id, (}
First(T') = {*, ε}
First(T) = First(F) = {id, (}
First(E') = {+, ε}
First(E) = First(T) = First(F) = {id, (}

ε-able → Substitude ∅

F → id | (E)

T' → * F T' | ε

T → F T'

First(F)

Follow

* Start Symbol always has $ as follow

Follow(E) = {$,)} ← start symbol

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

Nothing Follow E'

1. Following Terminal = Follow( )
2. First of following Non-Term = Follow( )
3. If NT is Right Most in RHS, Follow of LHS is selected.

+ if following it is ε-able

Follow(T) = First(E') = {+, ε} + Follow(E')

* No ε in Follow

TE' so = First(E')

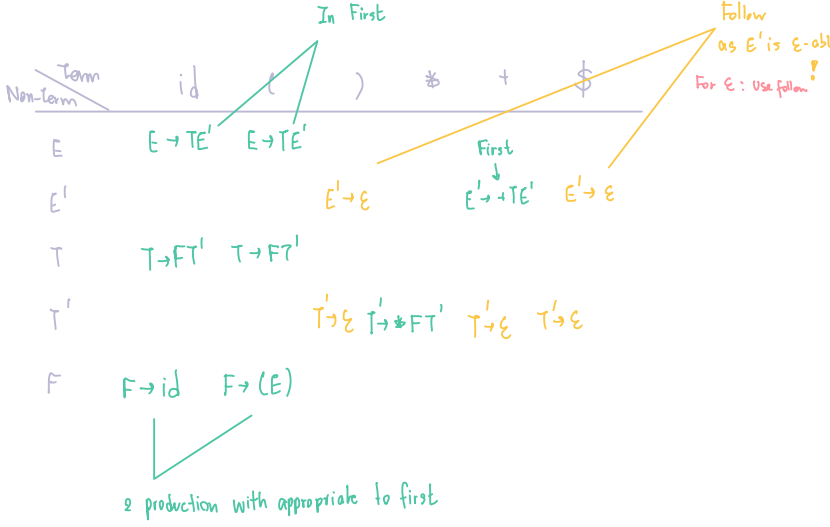
Follow(T') = Follow(T)

Follow(F) = First(T') = {*, ε} + Follow(T) + Follow(T')
= {*, +, $,)}

ex.

	First	Follow
$E \to TE'$ (T followed by E')	$\{id, (\}$	$\{\$,)\}$
$E' \to +TE' \mid \varepsilon$	$\{+, \varepsilon\}$	$\{\$,)\}$
$T \to FT'$	$\{id, (\}$	$\{+, \$,)\}$
$T' \to *FT' \mid \varepsilon$	$\{*, \varepsilon\}$	$\{+, \$,)\}$
$F \to id \mid (E)$	$\{id, (\}$	$\{*, +, \$,)\}$

Parsing Table



Rules

1. All ε-productions use FOLLOW
2. Remaining productions are placed in FIRST

S → ABCDE
A → a | ε
B → b | ε
C → c
D → d | ε
E → e | ε

First	Follow
{a, b, c}	{$}
{a, ε}	{b, c}
{b, ε}	{c}
{c}	{d, e, $}
{d, ε}	{e, $}
{e, ε}	{$}

$C_{(a)(c)}$ $

First(E) = {e, ε}

First(D) = {d, ε}

First(C) = {c}

First(B) = {b, ε}

First(A) = {a, ε}

First(S) = First(A) = {a, b, c}

↑ null above !

First b · First c

null above

Follow(S) = {$}

Follow(A) = First(B) = {b, g}

↳ subsitude in S : S → A C (D)

↓

→ First(C)

= {b, c}

test

pgm → line | EOF

line → LNUM stmt line | empty

stmt → asgmt | goto

asgmt → ID = exp

exp → ID exps

exps → + exp | empty

goto → GOTO LNUM

1. pgm → line
2. pgm → EOF
3. line → LNUM stmt line
4. line → empty
5. stmt → asgmt
6. stmt → goto
7. empty → ID exps
8. exps → + exp
9. exps → empty
10. goto → GOTO LNUM

First

FIRST(pgm) = {LNUM, EOF}
FIRST(line) = {LNUM, empty}
FIRST(stmt) = {ID, GOTO}
FIRST(asgmt) = {ID}
FIRST(exp) = {ID}
FIRST(exps) = {+, empty}
FIRST(goto) = {GOTO}

Follow

FOLLOW(pgm) = {$}
FOLLOW(line) = {$, LNUM}
FOLLOW(stmt) = {$, LNUM}
FOLLOW(asgmt) = {LNUM, $}
FOLLOW(exp) = {+, LNUM, $}
FOLLOW(exps) = {LNUM, $}
FOLLOW(goto) = {$}

Parsing Table

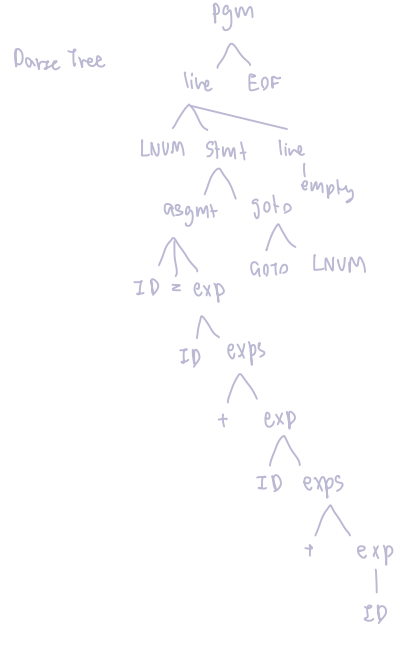
	LNUM	ID	GOTO	+	=	empty	EOF
pgm	1						2
line	3					4	4
stmt		5	6				
asgmt		7			8		
exp		9					
exps				10		11	11
goto			12				

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



Yamaguti P.

6430022021