skill in defining syntax:
a. given a grammar, write set of sentences (this can be infinite)
b. given a language, write its CFG. (this is similar to write a spec. in engineering job)
practical grammar
in order to implement a parser, the grammar must be of some specific form.
it has to avoid "left recursion".

✓ ถูก Grammar ไหม
✓ ภาษานี้เขียน Grammar ได้ยังไง

เพราะคิด infinite loop

Context-Free Grammars

Using grammars in parsers

Jaruloj Chongstitvatana
Department of Mathametics and Computer Science
Chulalongkorn University

Outline

- Parsing Process
- Grammars
  - Context-free grammar
  - Backus-Naur Form (BNF)
- Parse Tree and Abstract Syntax Tree
- Ambiguous Grammar
- Extended Backus-Naur Form (EBNF)

output ภาษา parse

Parsing Process

- Call the scanner to get tokens
- Build a parse tree from the stream of tokens
  - A parse tree shows the syntactic structure of the source program.
- Add information about identifiers in the symbol table
- Report error, when found, and recover from thee error

Grammar

- a quintuple (V, T, P, S) where
  - V is a finite set of nonterminals, containing S,
  - T is a finite set of terminals,
  - P is a set of production rules in the form of $\alpha \rightarrow \beta$ where α and β are strings over $V \cup T$, and
  - S is the start symbol.
- Example

ตัวอย่าง context free grammar

$G = (\{S, A, B, C\}, \{a, b, c\}, P, S)$

$P = \{ S \rightarrow SABC,\ BA \rightarrow AB,\ CB \rightarrow BC,\ CA \rightarrow AC,$
$SA \rightarrow a,\ aA \rightarrow aa,\ aB \rightarrow ab,\ bB \rightarrow bb,$
$bC \rightarrow bc,\ cC \rightarrow cc\}$

Context-Free Grammar

- a quintuple (V, T, P, S) where
  - V is a finite set of nonterminals, containing S,
  - T is a finite set of terminals,
  - P is a set of production rules in the form of $\alpha \rightarrow \beta$ where α is in V and β is in $(V \cup T)^*$, and
  - S is the start symbol.

α (ด้านซ้าย) เป็นได้แค่ nonterminal

- Any string in (V U T)* is called a *sentential form*.

Examples

จน.ประโยคใน grammar เป็น infinity ได้

$E \rightarrow E\ O\ E$

$E \rightarrow (E)$

$E \rightarrow id$

$O \rightarrow +$

$O \rightarrow -$

$O \rightarrow *$

$O \rightarrow /$

$S \rightarrow SS$

$S \rightarrow (S)S$

$S \rightarrow ()$

$S \rightarrow \lambda$

Backus-Naur Form (BNF)

- Nonterminals are in < >.
- Terminals are any other symbols.
- ::= means $\rightarrow$.
- | means or.
- Examples:

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

  <O> ::= + | - | * | /

  <S> ::= <S><S> | (<S>)<S> | () | λ

Derivation = การแทนที่ในประโยค sentential form

- A sequence of replacement of a substring in a sentential form.

Definition

- Let $G=(V, T, P, S)$ be a CFG, α, β, γ be strings in $(V \cup T)^*$ and A is in V.

  context free grammar

  $\alpha A \beta \Rightarrow_G \alpha \gamma \beta$ if $A \rightarrow \gamma$ is in P.

  แทน

- $\Rightarrow^*_G$ denotes a derivation in zero step or more.

Examples

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

S

$\Rightarrow SS$

$\Rightarrow (S)SS$

$\Rightarrow (S)S(S)S$

$\Rightarrow (S)S(())S$

$\Rightarrow ((S)S)S(())S$

$\Rightarrow ((S)())S(())S$

$\Rightarrow ((())())S(())S$

$\Rightarrow ((())()) (())S$

$\Rightarrow ((())())(())$

$E \rightarrow E\ O\ E \mid (E) \mid id$

$O \rightarrow + \mid - \mid * \mid /$

E

$\Rightarrow E\ O\ E$

$\Rightarrow (E)\ O\ E$

$\Rightarrow (E\ O\ E)\ O\ E$

$\Rightarrow^{*} ((E\ O\ E)\ O\ E)\ O\ E$

$\Rightarrow ((id\ O\ E))\ O\ E)\ O\ E$

$\Rightarrow ((id + E))\ O\ E)\ O\ E$

$\Rightarrow ((id + id))\ O\ E)\ O\ E$

$\Rightarrow^{*} ((id + id)) * id) + id$

Leftmost Derivation / Rightmost Derivation

(ทำซ้ายก่อน) (ทำขวาก่อน)

- Each step of the derivation is a replacement of the *leftmost* nonterminals in a sentential form.

E
⇒ **E** O E
⇒ (**E**) O E
⇒ (**E** O E) O E
⇒ (id **O** E) O E
⇒ (id + **E**) O E
⇒ (id + id) **O** E
⇒ (id + id) * **E**
⇒ (id + id) * id

- Each step of the derivation is a replacement of the *rightmost* nonterminals in a sentential form.

E
⇒ E O **E**
⇒ E **O** id
⇒ **E** * id
⇒ (**E**) * id
⇒ (E O **E**) * id
⇒ (E **O** id) * id
⇒ (**E** + id) * id
⇒ (id + id) * id

Language Derived from Grammar

- Let $G = (V, T, P, S)$ be a CFG.
- A string w in T^* is derived from G if $S \overset{*}{\Rightarrow}_{\mathbf{G}} w$.
- A language generated by G, denoted by $L(G)$, is a set of strings derived from G.
  - $L(G) = \{w \mid S \overset{*}{\Rightarrow}_{\mathbf{G}} w\}$.

Right/Left Recursive

- A grammar is a *left recursive* if its production rules can generate a derivation of the form $A \Rightarrow^* A\,X$.
- Examples:
  - $E \rightarrow E\ O\ id \mid (E) \mid id$
  - $E \rightarrow F + id \mid (E) \mid id$

    $F \rightarrow E * id \mid id$
    - $E \Rightarrow F + id$

      $\Rightarrow E * id + id$

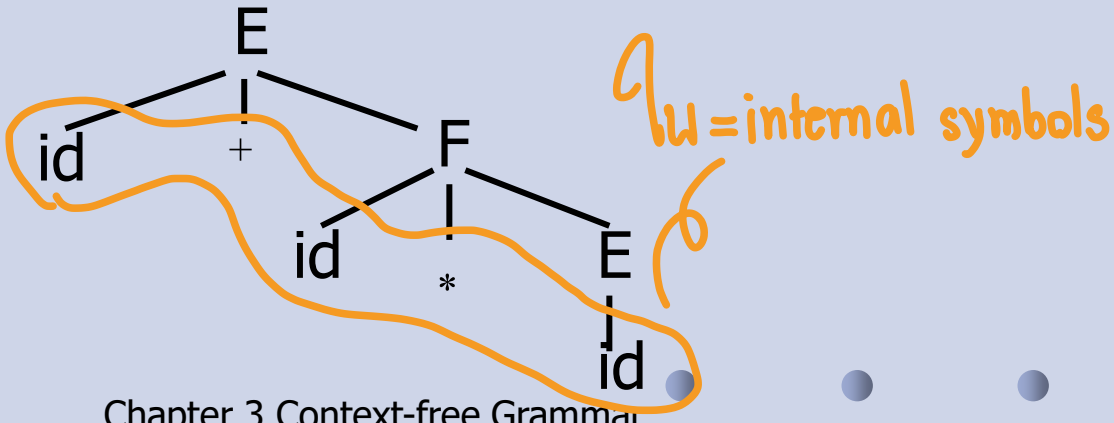
- A grammar is a *right recursive* if its production rules can generate a derivation of the form $A \Rightarrow^* X\,A$.
- Examples:
  - $E \rightarrow id\ O\ E \mid (E) \mid id$
  - $E \rightarrow id + F \mid (E) \mid id$

    $F \rightarrow id * E \mid id$
    - $E \Rightarrow id + F$

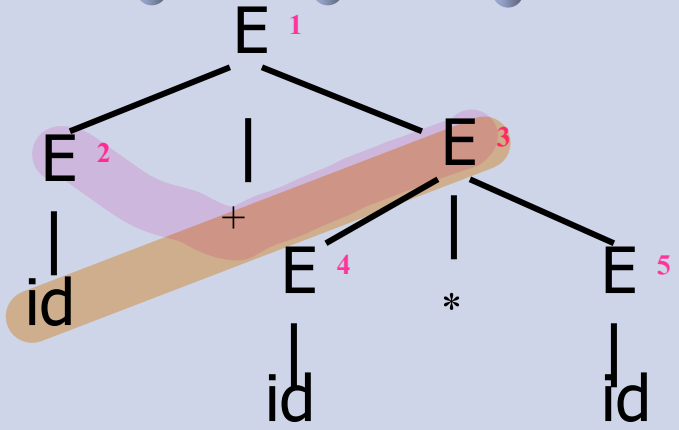
      $\Rightarrow id + id * E$

Parse Tree

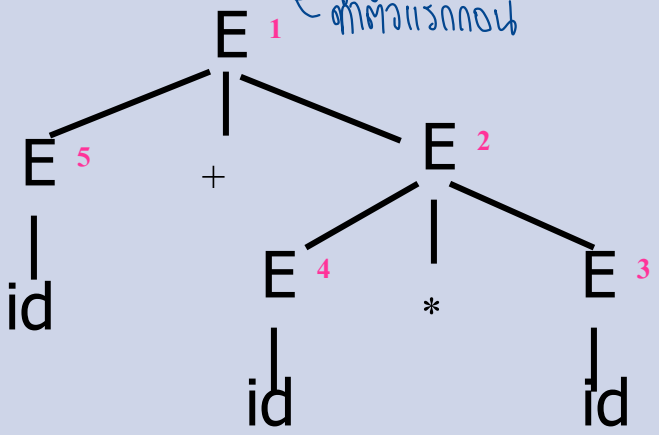
- A labeled tree in which
  - the interior nodes are labeled by nonterminals
  - leaf nodes are labeled by terminals
  - the children of an interior node represent a replacement of the associated nonterminal in a derivation
  - corresponding to a derivation



Parse Trees and Derivations



Preorder numbering



Reverse of postorder numbering

$$
\begin{aligned}
E &\Rightarrow E + E && (1)\\
&\Rightarrow id + E && (2)\\
&\Rightarrow id + E * E && (3)\\
&\Rightarrow id + id * E && (4)\\
&\Rightarrow id + id * id && (5)
\end{aligned}
$$

$$
\begin{aligned}
E &\Rightarrow E + E && (1)\\
&\Rightarrow E + E * E && (2)\\
&\Rightarrow E + E * id && (3)\\
&\Rightarrow E + id * id && (4)\\
&\Rightarrow id + id * id && (5)
\end{aligned}
$$

Example of Writing Grammar to Specify a Language

We will do one example to learn how to write a grammar of a language. Here is an assembly language source file.

```
; test xas
symbol
  sp r10    ; stack pointer
  base 100
  dd 2
  code 100
  add r1 r2
  jmp exit
  ld base
  ld 200
:exit
  add r2 r3
  inc r4 1
  inc r3 dd
  jt exit
data 200
  10 20
end
```

We will write a grammar to specify this ASM language. Assuming that the comment line that begins with ";" will be removed by the scanner. Here is the grammar.

```
program = symsec codesec datasec
symsec = tkSYMBOL sympair
sympair = tkNAME symval sympair | nil
codesec = tkCODE tkNUMBER codeline
codeline = tkLABEL oparg | oparg | nil
oparg = opargs codeline
datasec = tkDATA tkNUMBER dataline
dataline =  tkNUMBER dataline | tkEND
```

Where `tkXXX` denotes a terminal symbol; `nil` is an empty string.

These non-terminal symbols are special:

`symval` the scanner gets either: a name, a register name (r0, r1...), a constant
`opargs` parse the line: opcode operand ... (the machine instruction). We will not get into the details here.

These terminal symbols are more general:

`tkNAME` match any non-reserved word (reserved words are all terminal symbols), also keep the string in the symbol table.
`tkLABEL` match string begins with ":" (a label), also keep it in the symbol table
`tkNUMBER` match an integer

ASM grammar reads this way.
A program consists of three sections: symbol section, code section and data section.
The symbol section begins with "symbol" follows by any number of symbol pair.
A symbol pair consists of a name and a symbol-value.
The code section begins with "code" follows by a number follows by any number of code-line.
A code-line is either
1) begin with "label" follows by "opcode-arguments" or
2) "opcode-arguments" (no label)
Where "opcode-arguments" is the actual machine code. We will not get into details here.
The data section begins with "data" follows by a number follows by any number of data-line terminated by "end"
A data-line is a number.

Please note how we write a right recursive grammar to represent "any number of ...". We need a "nil" to denote a finite repeat (that it will end). Here is the grammar for "any number of sympair".

```
symsec = tkSYMBOL sympair
sympair = tkNAME symval sympair | nil
```

From this grammar we can generate a parser (in any language) automatically. Here is an excerpt from the grammar of ASM language (in C).

```
int program(void){
    if( symsec() ){
        commit(codesec());
        commit(datasec());
        return 1;
    }
    return 0;
}
int symsec(void){
    if( tok == tkSYMBOL ){
        mylex();
        commit(sympair());
        return 1;
    }
    return 0;
}
int sympair(void){
loop:
    if( tok == tkNAME ){
        pusht(tokstring);
        mylex();
        commit(symval());
        goto loop;
    }
    return 1;
}
. . .
```

Where `mylex()` is the scan function. It reads the next token. `tok` is the current token. `commit()` calls a function and makes sure that it returns true. "`pusht()`" is an extra "action routine" that is defined outside the grammar. It is necessary so that the parser can generate output. In this particular example, the parser saves the string of the token in a stack to be used later.

Example of Writing Grammar to Specify a Language

We will do one example to learn how to write a grammar of a language. Here is an assembly language source file.

```
; test xas

symbol
  sp r10    ; stack pointer
  base 100
  dd 2
code 100
  add r1 r2
  jmp exit
  ld base
  ld 200
:exit
  add r2 r3
  inc r4 1
  inc r3 dd
  jt exit
data 200
  10 20
end
```

We will write a grammar to specify this ASM langauge. Assuming that the comment line that begins with ";" will be removed by the scanner. Here is the grammar.

```
program = symsec codesec datasec
symsec = tkSYMBOL sympair
sympair = tkNAME symval sympair | nil
codesec = tkCODE tkNUMBER codeline
codeline = tkLABEL oparg | oparg | nil
oparg = opargs codeline
datasec = tkDATA tkNUMBER dataline
dataline =  tkNUMBER dataline | tkEND
```

Where `tkxxx` denotes a terminal symbol; `nil` is an empty string.

♡ ลองฝึกเขียนน่อยๆเดี๋ยวก็คล่อง

Grammar: Example

List of parameters in:

- Function definition
  - function sub(a,b,c)
- Function call
  - sub(a,1,2)

<argList>
⇒ **id** , <arglist>
⇒ **id**, **id** <arglist>
<argList>
⇒ <arglist> , **id**
⇒<arglist> , **id**, **id**
⇒ … ⇒ **id** (, **id**)*

<Fdef> → **function id (** <argList> **)**
<argList> → **id** , <arglist> | **id**
<Fcall> → **id (** <parList> **)**
<parList> → <par> ,<parlist>| <par>
<par> → **id** | **const**

<Fdef> → **function id (** <argList> **)**
<argList> → <arglist> **, id** | **id**
<Fcall> → **id (** <parList> **)**
<parList> → <parlist> ,<par>| <par>
<par> → **id** | **const**

Grammar: Example

List of parameters

- If zero parameter is allowed, then ?

<Fdef> → **function id (<argList>)|**
function id ( )
<argList> → **id** , <arglist> | **id**
<Fcall> → **id (<parList>) | id ( )**
<parList> → <par> ,<parlist>| <par>
<par> → **id** | **const**

Work ?

NO!
Generate
id , id , id ,

<Fdef> → **function id (<argList>)**
<argList> → **id** , <arglist> | **id** | λ
<Fcall> → **id (<parList>)**
<parList> → <par> ,<parlist>| <par>
<par> → **id** | **const**

Grammar: Example

List of statements:

- No statement → = ว่างๆ
- One statement:
  - s;
- More than one statement:
  - s; s; s;
- A statement can be a block of statements.
  - {s; s; s;}

Is the following correct?
{ {s; {s; s;} s; {}} s; }

↳ ถูกไวยากรณ์หรือไม่

<St> ::= λ | **s;** | **s;** <St> | { <St> } <St>

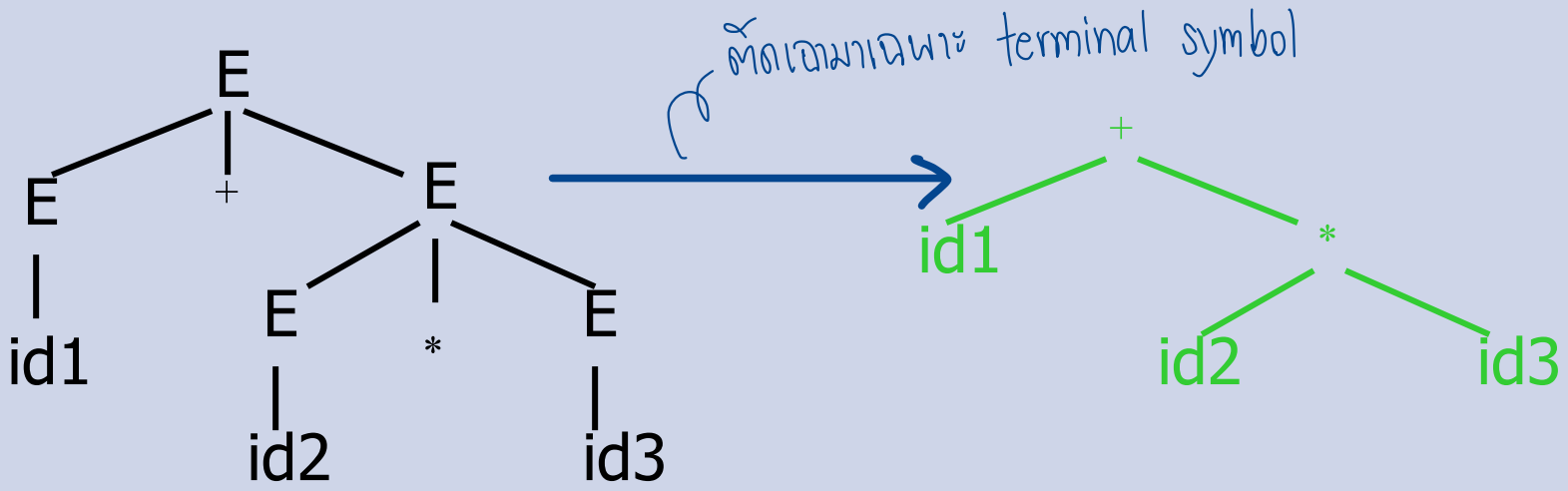
<St>
⇒ { <St> } <St>
⇒ { <St> }
⇒ { { <St> } <St>}
⇒ { { <St> } **s;** <St>}
⇒ { { <St> } **s;** }
⇒ { { **s;** <St> } **s;**}
⇒ { { **s;** { <St> } <St> } **s;**}
⇒ { { **s;** { <St> } **s;** <St> } **s;**}
⇒ { { **s;** { <St> } **s;** { <St> } <St> } **s;**}
⇒ { { **s;** { <St> } **s;** { <St> } } **s;**}
⇒ { { **s;** { <St> } **s;** {} } **s;**}
⇒ { { **s;** { **s;** <St> } **s;** {} } **s;**}
⇒ { { **s;** { **s; s;**} **s;** {} } **s;**}

เริ่มจาก start symbol แล้วไล่ไปหา (เหนื่อย!)

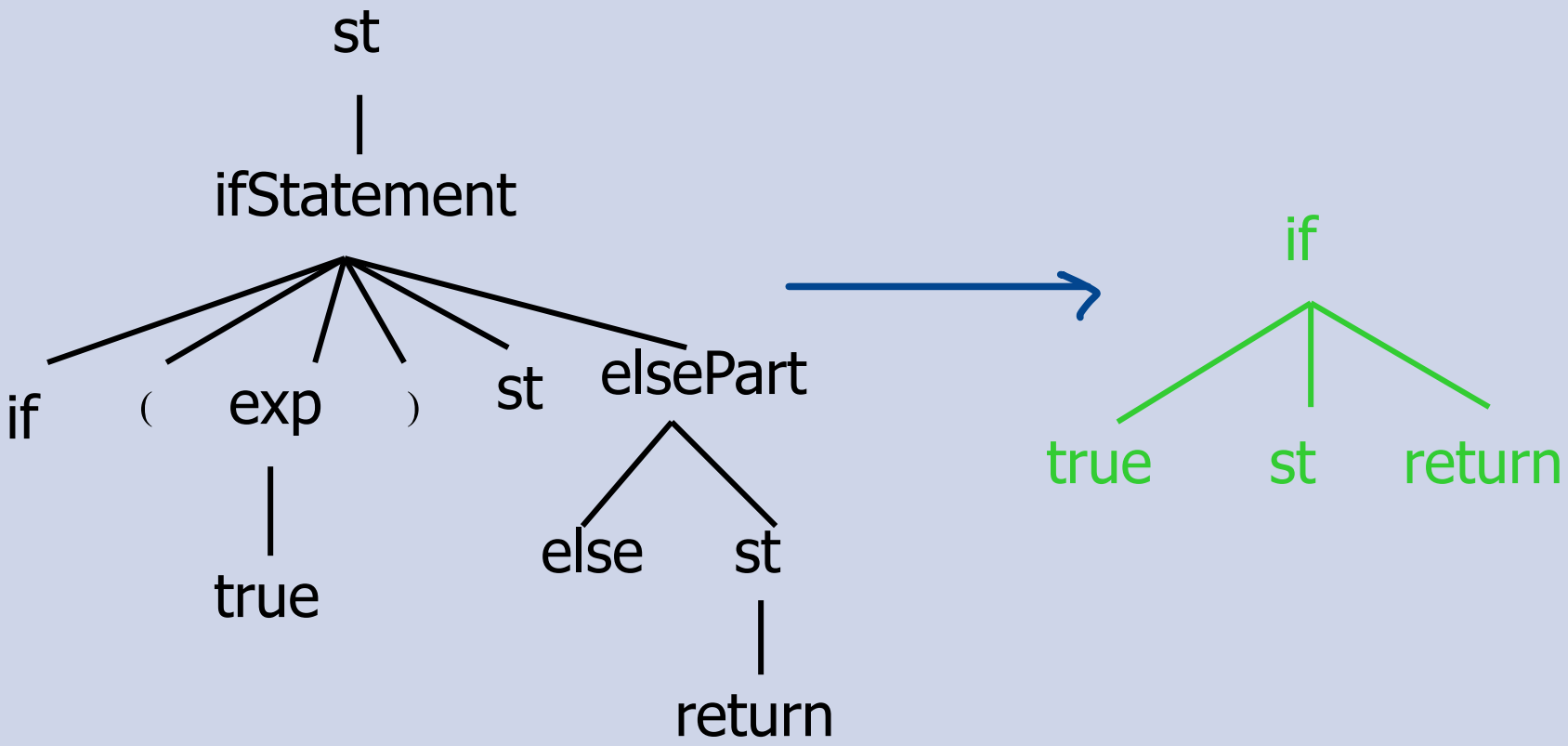
Abstract Syntax Tree

- Representation of actual source tokens
- Interior nodes represent operators.
- Leaf nodes represent operands.

Abstract Syntax Tree for Expression



Abstract Syntax Tree for If Statement



ความกำกวม

Ambiguous Grammar

- A grammar is ambiguous if it can generate two different parse trees for one string.
- Ambiguous grammars can cause inconsistency in parsing.

Example: Ambiguous Grammar

$E \rightarrow E + E$

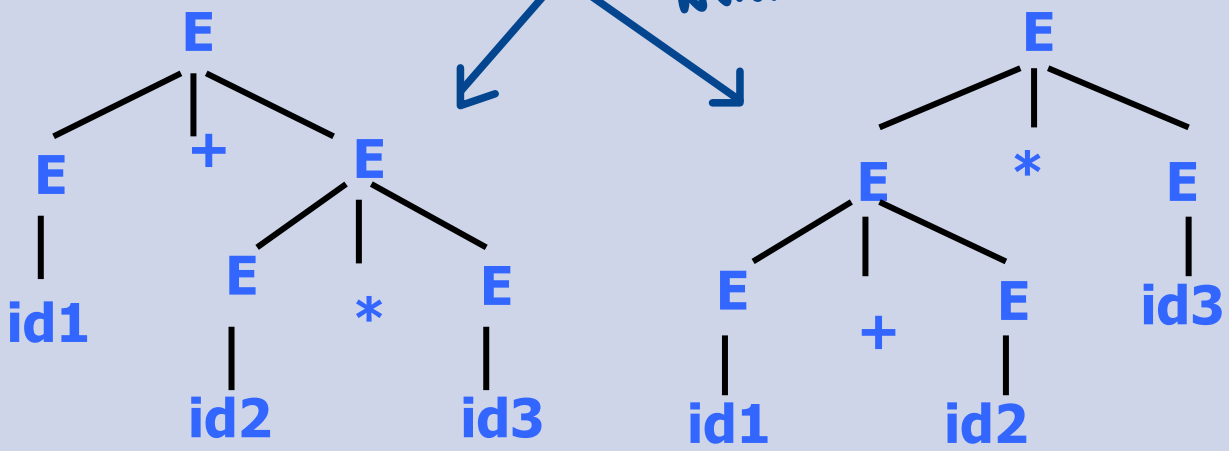
$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E / E$

$E \rightarrow id$

สร้างเป็น tree ได้ 2 แบบ



Ambiguity in Expressions

- Which operation is to be done first?
  - solved by precedence
    - An operator with higher precedence is done before one with lower precedence.
    - An operator with higher precedence is placed in a rule (logically) further from the start symbol.
  - solved by associativity
    - If an operator is right-associative (or left-associative), an operand in between 2 operators is associated to the operator to the right (left).
    - Right-associated : W + (X + (Y + Z))
    - Left-associated : ((W + X) + Y) + Z

Precedence

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow E / E$

$E \rightarrow id$

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow F$ ***

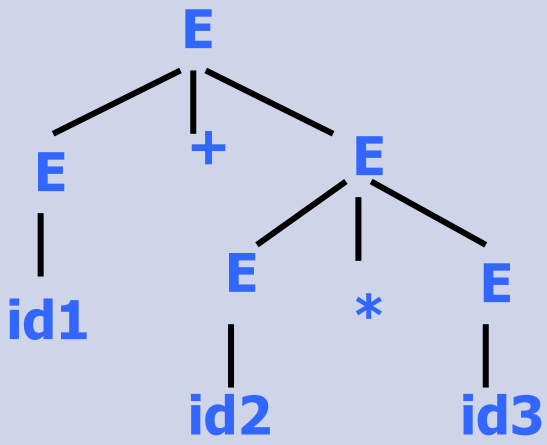
$F \rightarrow F * F$

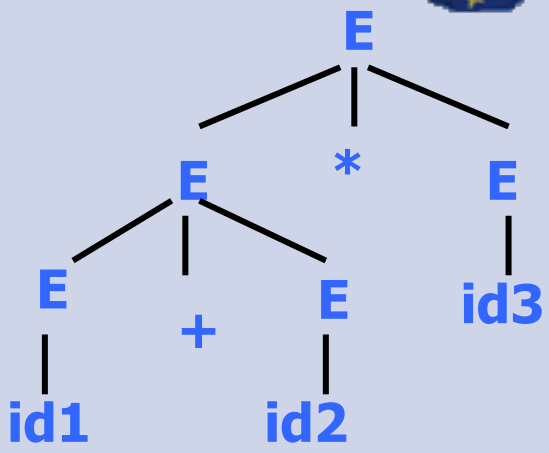
$F \rightarrow F / F$

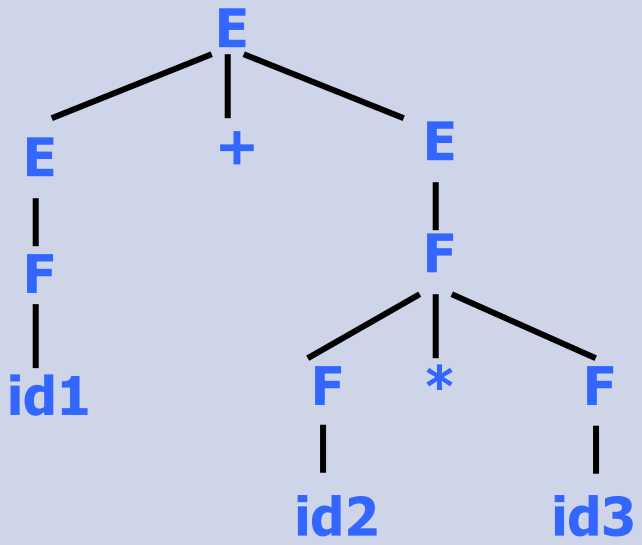
$F \rightarrow id$

แก้แล้ว

เนื่องจาก F อยู่ข้างใน (ลึกกว่า E) แสดงว่า F มีน้ำหนักมากกว่า







Precedence (cont'd)

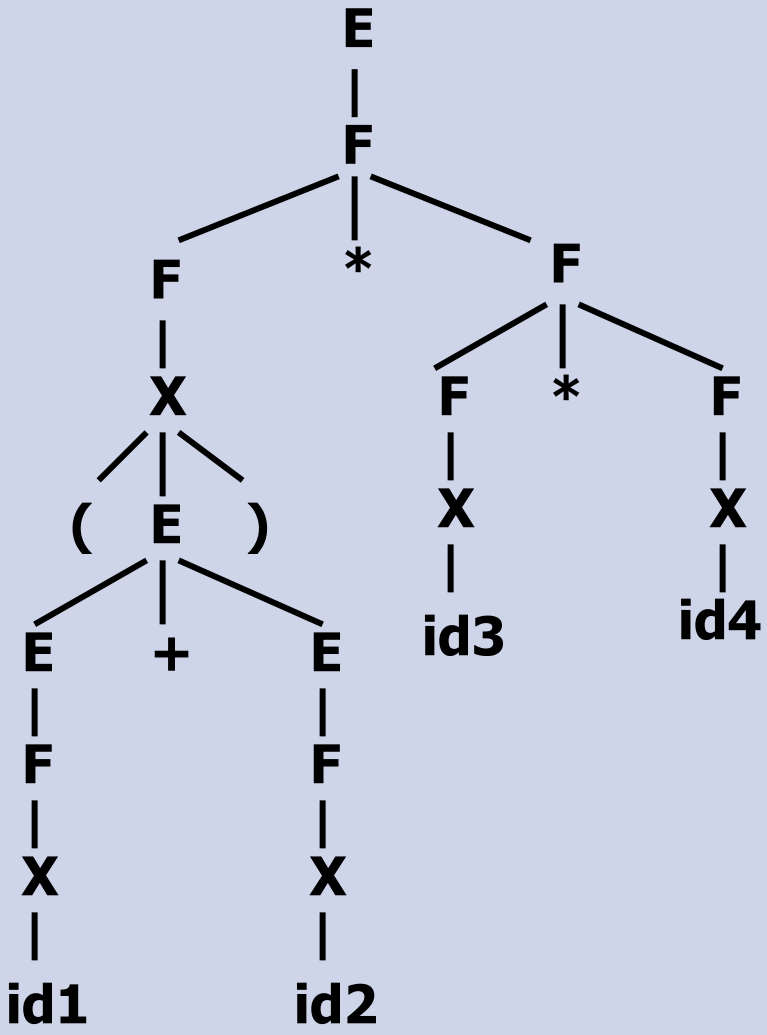
$E \rightarrow E + E \mid E - E \mid F$

$F \rightarrow F * F \mid F / F \mid X$

$X \rightarrow (E) \mid id$

ยิ่งทำให้ priority มากขึ้นสูงขึ้น

(id1 + id2) * id3 * id4



Associativity

- Left-associative operators

$E \rightarrow E + F \mid E - F \mid F$

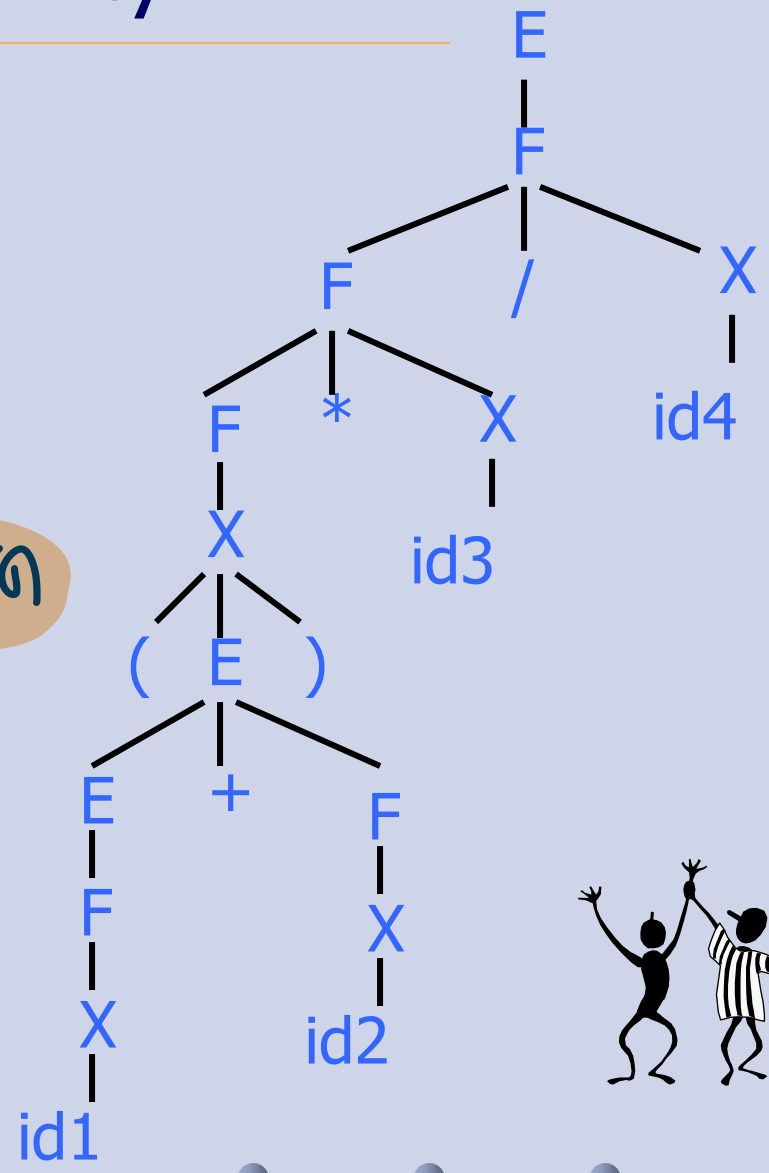
$F \rightarrow F * X \mid F / X \mid X$

$X \rightarrow (E) \mid id$

Priority สูงสุด อยู่ลึกสุด

(id1 + id2) * id3 / id4

= (((id1 + id2) * id3) / id4)

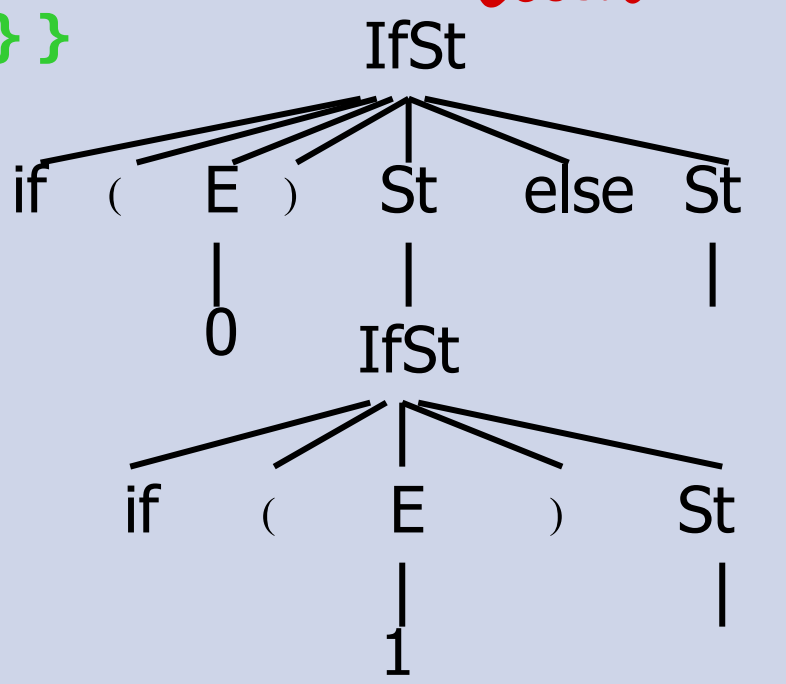


Ambiguity in Dangling Else

St → IfSt | ...

IfSt → if (E) St | if (E) St else St

E → 0 | 1 | ...



Disambiguating Rules for Dangling Else

St →
  MatchedSt | UnmatchedSt

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

MatchedSt →
  if (E) MatchedSt else MatchedSt|
  ...

E →
  0 | 1

- if (0) if (1) St else St

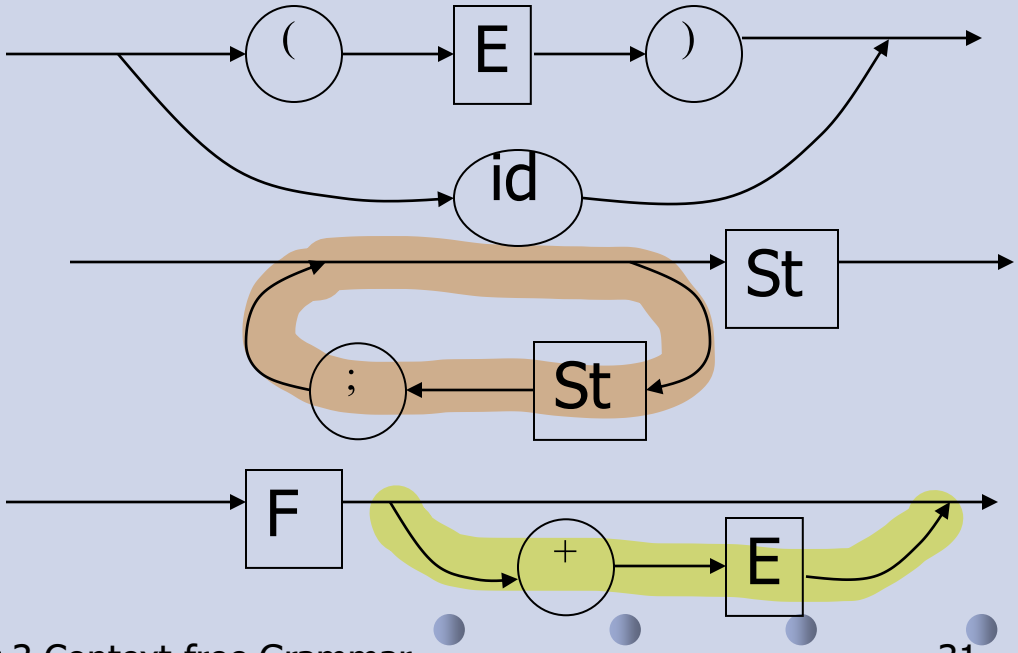
= if (0)
    if (1) St else St

Extended Backus-Naur Form (EBNF)

- Kleene's Star/ Kleene's Closure
  - Seq ::= St {; St}
  - Seq ::= {St ;} St
- Optional Part
  - IfSt ::= if (E) St [else St]
  - E ::= F [+ E] | F [- E]

Syntax Diagrams

- Graphical representation of EBNF rules
  - nonterminals: IfSt
  - terminals: id
  - sequences and choices:
- Examples
  - X ::= (E) | id
  - Seq ::= {St ;} St
  - E ::= F [+ E]



Reading Assignment

- Louden, Compiler construction
  - Chapter 3

Recursive Programming with List:

A gentle and fun introduction

In this note, I will introduce you to start writing a program in recursive style. All examples here can be written in C using my list library here.

List and operations

A list contains varying numbers of different objects, for example number (integer) and string (name). A simple object is called an atom. A list can contain another list. Here is a list.

```
(1 3 "hello")     ( (3 4) ("a" "b" "c") )
```

← List

Many kind of objects can be represented by lists. For example an expression:

```
a = b + c
```

can be drawn as a tree where the root is an operator like this,

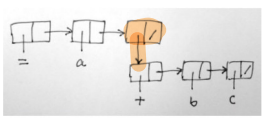
```
    =
   / \
  a   +
     / \
    b   c
```

ใช้เก็บรายการ operand

Assume we use a list to represent a tree, the first element is an operator, the rest is the operands.

```
("=" "a" ("+" "b" "c" ))
```

This data structure is built from a number of cells. A cell contains two fields, head and tail. There are two types of cell: atom and dot-pair. An atom is the end of a branch (a kind of leaf node). A dot-pair is a pointer to another list. The above example can be draw like this:



The cell that pointed to "=" and "a" are atoms. The cell pointed to the list ("+" "b" "c") is a dot-pair. How to distinguish a cell and a dot-pair depends on the choice of implementation of the data structure. A special atom called NI is used to denotes the end of list. To make it easier to write a list, I will omit ".." in the string.

Functions on List

Before we proceed further let define functions on List used in this session.

`head(L)`	returns first element of the list
`tail(L)`	returns the list without the first element
`cons(X,L)`	returns a new list with X as its head and L as its tail.

↳ Method

Therefore

```
L  is   cons( head(L), tail(L) )
```

`number(n)`	returns an atom of number n
`string(s)`	returns an atom of string s
`isatom(X)`	checks if X is an atom

To make a list we "cons" an atom with NIL.

```
(1 2)    is   cons(number(1), cons(number(2), NIL))
```

With these few functions we can now play with our little "program".

Given a list, return a number

Now that we know list, let us start by writing a simple program to count the number of elements in a list. We will start with a simple iterative version using "while".

Let input be a list of numbers such as (1 2 3 4)

```
count1(L)
  s = 0
  while L != NIL
    L = tail(L)
    s++
  return s
```

The iteration works as L is shortening until it is empty. Instead of using "while", we can do a recursion on tail(L).

```
count2(L)
1.   if L == NIL return 0
2.   return 1 + count2(tail(L))
```

The recursion on Line 2 continue on the tail(L). Line 1 tells us when to stop. The result is pending 1 + 1 + ... 0 until L is empty on the last call and return 0, then all the 1 + 1 + ..0 are executed and return. Let us see how the list L behaves.

```
count2( (1 2 3) )
   1 + count2( (2 3 ) )
         1 + count2( (3) )
               1 + count2( NIL )
                     0
```

Now count1 and count2 give the same result. count2 is a recursive version. However count2 will not count "inside" the element which is a list, for example `count2 ( (1 (2 3)) )` returns 2 instead of 3. We will refine it to do that.

```
count3(L)
1.    if L == NIL return 0
2.    if isatom(head(L)) return 1 + count3(tail(L))
3.    ... if head(L) is a list, what to do?
```

The count3 is similar to count2. Line 2 of count3 is modified from Line 2 of count2 but now it checks that the current element is not a list. So, what to do if it is a list? Let us consider what count3 is written to do. It can count "inside" a list. If we assume for now that it can do that, then we can use count3 to work in Line 3.

```
count3(L)
1.    if L == NIL return 0
2.    if isatom(head(L)) return 1 + count3(tail(L))
3.    return count3(head(L)) + count3(tail(L))
```

So, Line 3 is where the magic occurs. When you think of it, if head(L) is an atom, Line 3 will be reduce to 1 + count3(tail(L)) which is exactly like Line 2 of count2. Let us try count3 on (1 (2 3))

```
count3( (1 (2 3) ) )
2>  1 + count3( ((2 3)) )   notice that head(L) is now a list not an atom
3>      count3( (2 3) ) + count3( NIL )   (**)
2>      1 +  count3( ( 3 ) ) . . .
2>             1 + count3( NIL )
1>                   0 back to (**)  for  the last count3(NIL)
1>                           0
```

We can refine count3 a bit further. Noticing that Line 2 can be simplified.

```
count3(L)
1.    if L == NIL return 0
2.    if isatom(L) return 1
3.    return count4(head(L)) + count4(tail(L))
```

Given a number, return a list

We have learn how to write a recursive program to take apart a complex structure. Next exercise is to "build" a recursive structure. Let us start with building a unary number which is represented by a list of 1's. Given a number n, write a program to produce a list of n 1's. (1 1 ...). As before we will start with an iterative version.

```
unum1(n)
  L = NIL
  while n > 0
     L = cons(number(1),L)
     n--
  return L
```

We use iteration to cons' "1" into a list, n times.

```
unum1(2)    returns  (1 1)

unum1(3)    returns  (1 1 1)
```

A recursive version use the recursion of reducing n until it is 0.

```
unum2(n)
1.    if n == 0 return NIL
2.    return cons(number(1), unum2(n-1))
```

Line 2 did exactly that. The cons' of 1 is pending while unum2 keeps producing the next 1 ... until n == 0.

```
unum2(3)
    cons(1, unum2(2))
                cons(1, unum2(1))
                           cons(1, unum2(0))
                                       NIL

return  cons(1, cons(1, cons(1, NIL )))  == (1 1 1)
```

Let try to define some arithmetic operation on this unary number. Starting with add.

(1 1) + (1 1 1) = (1 1 1 1 1)   that is 2 + 3 = 5

So, adding two unary numbers is just merging two lists. We start by trying to write a concatenate function for two lists. We will take one element from the fist list and "cons" it to the second list. An iterative version is this.

```
cat1(a b)
1.  while a != NIL
2.     b = cons(head(a), b)
3.     a = tail(a)
    return b
```

Now we can try a recursive version.

```
cat2(a b)
1.   if a == NIL return b
2.  return cat2( tail(a), cons(head(a),b) )
```

Line 2 of cat2 did the work similar to Line 2,3 of cat1. Building up a result is done by "cons". Reducing a (hence converge to a stop in the end) is done by tail(a). However please observe the order of "cons". For illustration, we will use lists of integer.

```
cat2( (1 2) (3 4 5) )
    cat2( (2) (1 3 4 5) )
          cat2( NIL (2 1 3 4 5) )
```

Fortunately, for unary addition, the order is not relevant. So, uadd(a b) == cat2(a b). We can refine it a bit more by recognising an arithmetic identity.

```
n + 0 =  n,   0 + m = m

uadd(n m)
  if m == NIL return n
  if n == NIL return m
  return uadd( tail(n), cons(head(n),m))
```

To complete the arithmetic operation, let we try a subtraction. We notice that we can take an element away from both number one-by-one until one list is empty. The remaining list is the difference. Let assume n >= m, so m will be empty first.

```
usub(n m)      we assume n >= m
  if m == NIL return n
  return usub( tail(n), tail(m) )
```

It turns out that subtraction is easier than addition in this unary system. (In tenary number, subtraction is more difficult than addition).

There is one more thing I want to tell you. When you want to build up a result, you can use an additional variable to hold it. This variable is usually passed as an extra parameter to your recursive function. Let me give an example of reversing a list. Write a program to reverse a list, returning a new reversed copy of the original. Where will you keep the result in progress?

```
reverse(L)
```

but if you have an extra parameter, you can keep your intermediate result there.

```
reverse2(L, L1)
   if L == NIL return L1
   return reverse2( tail(L), cons(head(L), L1)
```

ประหยัดที่เก็บ

Now you can write your reverse. The "accumulating" variable starts with NIL. Did you notice that reverse2 is like cat2? (Think why?)

```
reverse(L)
  return reverse2(L, NIL)
```

Data == Program

I will show how to represent a program with List. Use count_simple(L) as a case study.

We will represent a statement by a list with the first element as "operator" and the rest of the list are the arguments to that operation.

```
( op arg1 arg2 ... )
```

to define a function we will use the operator "def". So, to define count_simple(L):

```
program -> ( "def" "count_simple" arglist body )

arglist -> ( "L" )
```

the body of this function consists of two lines.

```
line1 -> ("if" ( "==" "L" nil) ("ret" 0))
line2 -> ("ret" ("+" 1 ("count_simple" ("tail" "L")))
```

nil is a special atom is our universe of objects. To represent a "block" of statement we use the operator "do".

```
body -> ("do" line1 line2)
```

with this representation of a program as a list (data), we can write an interpreter to evaluate (run) this program.

This is a complete picture of a list representing the count_simple function:

```
("def" "count_simple" ("L")
  ("do"
    ("if" ("==" "L" nil) ( "ret" 0))
    ("ret" ("+" 1 ("count_simple" ("tail" "L")))))
```

Exercises

Must do!

Here are some exercises for you to practice.

1) A simple one first, write a function to multiply two unary numbers.
2) A famous function. Write an append function to put an element at "the end" of a list.
3) How to clone a list? A more ambitious program should also cope with a complex structure (list in list)
4) Write a program to check "equality" of two lists. (again, a more ambitious program should cope with complex list)
5) * In my youth (many many years ago :-), I am intrigue by this puzzle. Write a function to "flatten" a complex list. Here is an example
input: ((1 2) (3 (4 5))) output: (1 2 3 4 5)
6) If you feel "oh my gosh, this is fun!", you can try this exercise. Write an interpreter for the language that use list as its main data structure. You need to be able to "execute" this kind of list `(+ 1 2)`. The needed operators are `(if cond true-act false-act)` for if..then..else and `(do ex1 ex2)` for executing ex1 follows by ex2. Defining a function will be a bit tricky, you need to be able to create a variable (called lambda in programming language theory). An example of this language (of the above reverse2 definition):

```
  (def  reverse2 (L L1) (if (eq L NIL) (L1) (reverse2 (tail L) (cons (head L) L1))))
```

* means one hour of thinking.

Enjoy programming.

No you don't need to fire up your compiler. You can write all of them down in a sheet of paper. How can you know that you got it right? Simply, prove it :-)

But if you want to try to run it, you can code it in C. Here is the C program of the above examples. This program is used in conjunction with the ailib.c (ailib.h) library.

last update 14 Aug 2018

```
1.
number(L) {
    if (L == NIL) return 0 ;
    return 1 + number(tail(L)) ;
}

multiply(L, num) {
    if (num == 0) return L ;
    return multiply(cons(1, L), num - 1) ;
    //return cons(1,  mul(num-1)) ;
}

mul(a, b) {
    int num1 = number(a) ;
    int num2 = number(b) ;
    multiply(NIL, num1 * num2) ;
}
-----------------------------------------------------------------------

2.
append(x, L) {
    if (L == NIL) return cons(x, NIL) ;
    return cons(head(L), append(x, tail(L))) ;
}
-----------------------------------------------------------------------

3.
clone(L) {
    if (L == NIL) return NIL ;
    if (isatom(head(L))) return cons(head(L), clone(tail(L))) ;
    return cons(clone(head(L)), clone(tail(L))) ;
}
//หรือ
clone(L) {
    if (L == NIL) return NIL ;
    return cons(head(L), clone(tail(L))) ;
}
```

```
4.
equal(a, b) {
    if (a == NIL && b == NIL) return TRUE ;
    if (a == NIL || b == NIL) return FALSE ;
    if (isatom(head(a)) && isatom(head(b))) {
        if (head(a) != head(b)) {
            return FALSE ;
        }
        return (tail(a), tail(b)) ;
    }
    else if (isatom(head(a)) || isatom(head(b))){
        return FALSE ;
    }
    else {
        if (equal(head(A), head(B))) {
            return equal(tail(a), tail(b)) ;
        }
        return FALSE ;
    }
}
```