Parsing

Parsing is a process that constructs a syntactic structure(i.e. parse tree) from the stream of tokens

Top-down Parsing

> **Info**
>
> created order : created from root to leaves
> traversal of parse tree : preorder traversal
> Derivation : left-most derivation
>
> Two Types :
> 1.Backtracking Parser :
> Try different structures and backtrack if it does not matched the input
> 2.Predictive Parser :
> Guess the structure of the parse tree from the next input
>
> Parser have to decide which production rule should be used at each point of time
> If using predictive parser how should we guess?
> 1.next token like Reserved word if,(,etc.
> 2.structure to be built like if statement,etc.

LL(1) Parsing

Info

L = read input from left to right
L = simulate left most derivation
(1) = 1 lookahead symbol
ใช้stackที่ล่างสุดเป็น$ ถ้าสุดท้ายสามารถอ่านตัวนี้ได้แปลว่าaccepted
start the process by adding start symbol
If top is terminal, check if match pop it out
else, replace with P that associated with lookahead
Aware : A⇒XY top is X cuz leftmost derivation

First set

Info

First set care λ
First(A) = set ของ terminal ตัวแรกที่ derived ได้จาก A

- Let X be λ or be in V or T.
- First(X) is the set of the first terminal in any sentential form derived from X.
- For example: S → (S) S, S → λ
  The strings derived from S are λ, (), ()(), (()), (()()), ...
  So, the symbols λ or (must be in First(S).

Some properties:

$A \Rightarrow XY$

First set of A will be subset of leftmost ones

$First(A) \subset First(X)$

3 exp → term exp'

2 exp' → $\overset{\text{use}}{\text{addop}}$ term exp' | λ

1 addop → $\boxed{+ \mid -}$ all Terminal

2 term → factor term'

2 term' → $\overset{\text{use}}{\text{mulop}}$ factor term' | λ

2 mulop → $\boxed{*}$ all Terminal

1 factor → $\boxed{(}$ exp) | $\boxed{\text{num}}$ all Terminal

	First
exp	(, num
exp'	+, -, λ
addop	+, -
term	(, num
term'	λ, *
mulop	*
factor	(, num

Follow set

Info

Follow set dont care λ

Follow(A) = set ของ terminal ตัวที่จะตามมาเมื่อ A ถูก pop

- Let $ denote the end of input tokens.
- If A is the start symbol, then $ is in Follow(A).
- If there is a rule B → X A Y, then First(Y) - {λ} is in Follow(A).
- For example: S → (S) S, S → λ
  The strings derived from S are λ, (), ()(), (()), (()()), ...
  So, the symbols) must be in follow S because there is a rule S → (S).
- If there is production B → X A Y and λ is in First(Y), then Follow(A) contains Follow(B).

Some properties:

$A \Rightarrow XY$

Follow(leader) will be First(follower) - {λ}

Follow(X) = First(Y)-{λ}

Follow set of A will be subset of rightmost ones

Follow(A) $\subset$ Follow(Y)

but if First(Y) have λ

Follow(A) $\subset$ Follow(X) as well

1. exp → term exp'
2. exp' → addop term exp' | λ
3. addop → + | -
4. term → factor term'
5. term' → mulop factor term' | λ
6. mulop → *
7. factor → (exp) | num

follow so follow(exp) have)

care chain reaction

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

Constructing Parsing Table

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

จะถูกสนใจ เมื่อ Firstset มี λ

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

current derivation \ อยากได้ Token	(	)	+	-	*	num	$
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	

LL(1) Grammar

Info

A grammar is an LL(1) grammar if its LL(1) parsing table has at most one production in each table entry example of non-LL(1) grammar parsing table

	(	)	+	-	*	num	$
exp	1,2					1,2	
term	3,4					3,4	
factor	5					6	
addop			7	8			
mulop					9		

What causes grammar being non-LL(1)

1. Left-recursion
2. Left factor

Left Recursion

Info

General left recursion : A $\Rightarrow^*$ AY

Immediate left recursion :

basic : A $\rightarrow$ AX | Y

multi : A $\rightarrow$ AX_1 |...| AX_n | Y_1 |...| Y_m

regular expression : A = YX*

Immediate left recursion can be easily removed when no empty-string production and no cycle

basic :

A $\rightarrow$ YA'

```
A' → XA' | λ
multi :
A → Y₁A' |...| YₘA'
A'→ X₁A' |...| XₙA' | λ
regular expression : A = YX*
```

exp → exp + term | exp - term | term เพราะสุดท้ายต้องจบด้วย term

term → term * factor | factor

have finish with factor

factor → (exp) | num

Not recursion no need change

Remove left recursion

exp → $\boxed{\text{term}}$ exp'

exp = term (± term)*

exp' → + term exp' | - term exp' | λ

term → $\boxed{\text{factor}}$ term'

term = factor (* factor)*

term' → * factor term' | λ

factor → (exp) | num

Left-Factor

Info

Left same right diff so Separated

- Left factor causes non-LL(1)
  - Given $A \to X\ Y \mid X\ Z$. Both $A \to X\ Y$ and $A \to X\ Z$ can be chosen when A is on top of stack and a token in First(X) is the next token.

$A \to XY \mid XZ$
$A \to XB$
$B \to Y \mid Z$

$A \to X\ Y \mid X\ Z$

can be left-factored as

$A \to X\ A'$ and $A' \to Y \mid Z$

ifSt → **if (** exp **)** st **else** st | **if (** exp **)** st

can be left-factored as

ifSt → **if (** exp **)** st elsePart

elsePart → **else** st | λ

seq → st ; seq | st

can be left-factored as

seq → st seq'

seq' → ; seq | λ