Mid-term Summary

Programming Language Theory

Mid-term

- **Schedule: 10/21 Wed 10AM~12PM (2 hours).**
- All Questions in English.
  - We have 2 hours considering language difficulties.
- Closed Book - No dictionaries too.
- You can answer the questions in either English or Korean, or both.
- Basically the same as Assignment 1.

Mid-term Questions

- Some questions ask you to fill in blanks.
  - Similar to Assignment1 - Q1.
    - What is the transition function for (a), (b), (c)?

state	input symbol		
	1	+	#
q_0	$(q_0, 1, R)$		(a)
q_1	(b)		
q_2			(c)
q_3			

- Other questions make you to complete the equations or code or definitions.

- You may need to expect the output of given code and explain the reasons.

Topics

- **BNF**
- **Parsing**
- **Static Memory Management**
- Dynamic Memory Managements
- Control Flow and Recursion

Backus Naur Form (BNF)

- It is a notation technique for ***context-free grammars***.
- Basically, it is a different notation of production rule we've learned.
- Grammar *G = (V, T, S, P)*
  - *Variables (V) - Nonterminals* (e.g. <expr>)
  - *Terminal Symbols (T) - Terminals* (e.g. num, 1, 2, '-')
  - *Production Rule (P)*
    - S → Ab, A → Sa, A → b
    - <S> ::= <A>b, <A> ::= <S>a, <A> ::= b

Production Rules

- We can derive strings with BNF just like grammar.
- `<S> ::= <A>b, <A> ::= <S>a, <A> ::= b`
- But, how about the Start Symbol?
- Unlike grammars in formal language, BNF notations of syntax are often missing start symbols.

In Practice

- Sometimes it doesn't even follow the specific conventions of notations.
- No '|' for choice, no '<','>' for nonterminals, they even omit '::=' symbol.
- Still we can get the idea.
- <members> ::=
  <member> |
  <member>','<members>

```
json
    element

value
    object
    array
    string
    number
    "true"
    "false"
    "null"

object
    '{' ws '}'
    '{' members '}'

members
    member
    member ',' members

member
    ws string ws ':' element
```

In Practice

- If we want to derive or check the whole JSON document,
  - start with <json>.
- If we just want to what is the object in JSON,
  - check <object> and related ones.
- Hence start symbol is often implicit and not directly mentioned.

```
json
    element

value
    object
    array
    string
    number
    "true"
    "false"
    "null"

object
    '{' ws '}'
    '{' members '}'

members
    member
    member ',' members

member
    ws string ws ':' element
```

In Practice

- Quotations - Just need to use them for clarification.
    - Symbols used for notations (::=, |, <, >) should be enclosed in quotations if they're used as terminal symbols.
- There exist many variants in BNF and EBNF - Don't need to be stuck with minor details.
- Still don't mess with { }, [ ], +, *, etc, which actually changes the meaning.

```
json
    element

value
    object
    array
    string
    number
    "true"
    "false"
    "null"

object
    '{' ws '}'
    '{' members '}'

members
    member
    member ',' members

member
    ws string ws ':' element
```

Derivation

- Starts from a nonterminal on the left, keep replacing nonterminals on the right until everything is terminal.

  - e.g.) Derive / + a b c

  - <prefix> ::= <op><prefix><prefix>|<var>
    <var> ::= "a"|"b"|"c"
    <op> ::= "/"|"+"

  - Start from <prefix>, keep replacing the leftmost or the rightmost ***nonterminals***.

Derivation

- For derivation, which expression to choose is up to you.
  - e.g.) <op> can be replaced with "/" or "+". Choose the one which can derive the given string.
- One string can be derived by more than one derivation steps.
- Bad choice may lead to wrong derivation.
  - + a b cannot be generated if <op><prefix><prefix> ⇒ / <prefix><prefix>.
- If there exists any way to derive a string, the string is syntactically correct for the BNF.

Derivation

- Leftmost Derivation of
  / + a b c

```
<prefix> ::= <op><prefix><prefix>|<var>
<var> ::= "a"|"b"|"c"
<op> ::= "/"|"+"
```

- <prefix> ⇒
  <op><prefix><prefix> ⇒
  /<prefix><prefix> ⇒
  /<op><prefix><prefix><prefix> ⇒
  / +<prefix><prefix><prefix> ⇒
  / +<var><prefix><prefix> ⇒
  / + a<prefix><prefix> ⇒*
  / + a b<prefix> ⇒*
  / + a b c

EBNF Notations

- { X } : repeat X 0 or more times.
  - <statements> ::= {<statement>;}
  - <statements> ⇒* <statement>; ... <statement>;
  - <statements> ⇒* <statement>;
  - <statements> ⇒* ε

EBNF Notations

- [X] : X is optional. You can also use ‘?’ like regular expression style.
  - <signed> ::= [‘-’]<num>
  - <signed> ::= ‘-’?<num>
  - <num> ::= 1|2|3|4
  - <signed> ⇒* 1 <signed> ⇒* -1
    <signed> ⇒* 2 <signed> ⇒* -4

EBNF Notations

- <digits> ::= <digit>* repeat 0 or more
  - <digits> ::= {<digit>}
  - <digits> ⇒* ε
  - <digits> ⇒* <digit><digit>...<digit>
- <digits> ::= <digit>+ repeat at least once
  - <digits> ::= <digit>{<digit>}

EBNF Notations

- (X) : for grouping. Symbols are applied to the whole grouped terminals, nonterminals.
  - <nums> ::= (+|-)*<num>(,<num>)+
- +, - are repeated 0 or more times.
- For every repetition, we can choose + or -.
  - e.g.) ++--+-<num>,<num>,<num>
- After one <num>, ",<num>" will be added at least once, or more.
  - e.g.) +<num>,<num> or <num>,<num> or <num>,<num>,<num> . . .

Parsing

- When we were talked about derivation of a string based on a grammar or BNF,
  - We simply choose one of the production rules which might derive the given string.
  - e.g.) / +a<var><var> ⇒* / + a b c **Selected** b, c **here.**
- Then how does a compiler make such decisions?
  - Top-down, Bottom-up Parsing.

Top-down Parsing

- **Top-down parsing** starts from the start nonterminal (i.e., root).
- For each round of parsing, ***it checks all possible productions*** to be applied to nonterminals.
- Hence it is also called **exhaustive search parsing**.

- <int-part> ::= <digit>|<int-part><digit>
  - Derive 3.14
  - <int-part>.<frac-part> ⇒ <digit>.<frac-part>
  - <int-part>.<frac-part>
    ⇒ <int-part><digit>.<frac-part>

Flaws in Top-down Parsing

- It’s very tedious.
  - We simply ask the compiler to try every possibility until it finds the right one.
    - This is not efficient way of parsing.
- Non-termination.
  - If a given string cannot be derived by given BNF, parsing will never end.

Bottom-up Parsing

- Conversely, we can ***reduce terminals*** of given string w to a nonterminal using BNF.
  - e.g.) `3.14` ⇒ `<digit>.14`
- Usually it reads the input text from left to right, and finds nonterminal to replace terminals in the text.
  - This is the instruction for the compiler.

Ambiguity

- If there exist more than one production, which one should be applied?
  - For <digit>.14, we can reduce <digit> into two different nonterminals.
- <int-part> ::= <digit>|<int-part><digit>
- <frac-part> ::= <digit>|<digit><frac-part>
- For <int-part>.<digit>4, we can reduce <digit> further, or just move onto the next.

To Resolve Ambiguity

- One way to resolve ambiguity is to rewrite the grammar.
- Think about the `a + b * c` example again.
- ```
  <expr> ::= <expr> + <expr>
           | <expr> * <expr>
           | a | b | c
  ```
- We know that we have two parse trees for the expression, based on which operator (+, *) is considered first.
  ```

To Resolve Ambiguity

- We can introduce new nonterminals.
- ```
  <expr>  ::= <expr> + <expr*> | <expr*>
  <expr*> ::= <expr*> * <var> | <var>
  <var>   ::= a | b | c
  ```

- This example is not that difficult to resolve the ambiguity.
- But usually it is very hard to tell whether a grammar has ambiguity or not, and also to resolve it.
  ```

Static Management

- Static memory management is performed by the compiler, before program execution.
- Statically allocated objects are located in a fixed zone of memory.
- These objects stay in there for the entire program execution.
- Global variables, object code, constants, compiler generated tables.

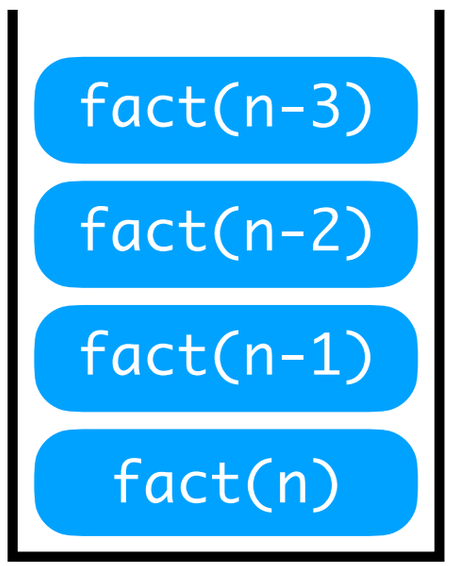
With Recursion

- For each call, a new activation record for the same function `fact()` is pushed to the stack.

- In each activation record, ***the same parameter*** n ***will have different values***.

- The value of n can be decided and changed during runtime.

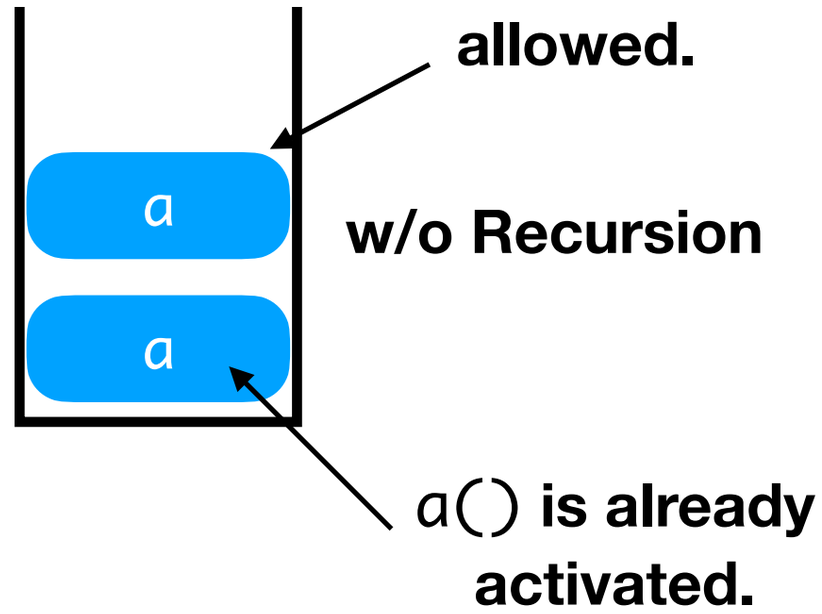
```
int fact(int n) {
    if(n == 1)
        return 1;
    else
        return n*fact(n-1);
}
```



Without Recursion

- Without recursion, more than one activation record of the same function cannot be in the stack at the same time.
- Hence it's possible to handle other components of PL statically.
  - if a() declares a local variable, we only need one memory location to store its value → it can be loaded statistically.

```
a(){
    b();
    a();
}
```



Summary

- BNF
- Parsing
- Static Memory Management