## 1. Successful Cases

```
func NewTracer(root opentracing.Span) *Tracer {
	return &Tracer{
		root:  root,
		spans: make([]opentracing.Span, 0, 32),
	}
}
```

(a) A Code Snippet $s_3$ from the Test Set of the CSN Dataset

```
func NewTracer(root opentracing.Span) *Tracer {}
return &Tracer{}
```

(b) Important Statements Selected by Our Extractor

**Reference Comment:** NewTracer returns a new tracer with the specified root span.
**CodeT5:** NewTracer creates a new tracer.
**ICS:** NewTracer creates a new tracer with the provided root span.

(c) Comments Generated by Different Techniques

Figure 1: Successful Case on Go Code Comment Generation

First, in addition to the case in our paper, we present another case in which our ICS also successfully generates a good comment. From the figure, we can observe that the comment generated by CodeT5 misses a crucial factual detail, namely “with the specified root span”. In contrast, our ICS succeeds in covering it in the comment because the method declaration is selected as an important statement, allowing ICS to pay more attention to the name and parameter of the method.

## 2. Moderate Cases

```
func (r *Render) Error(status int, message ...string) {
	r.WriteHeader(status)
	if len(message) > 0 {
		r.Write([]byte(message[0]))
	}
}
```

(a) A Code Snippet $s_4$ from the Test Set of the CSN Dataset

```
func (r *Render) Error(status int, message ...string) {}
r.WriteHeader(status)
if len(message) > 0 {}
r.Write([]byte(message[0]))
```

(b) Important Statements Selected by Our Extractor

**Reference Comment:** Error writes the given http status to the current response writer.
**CodeT5:** Error writes an error to the response.
**ICS:** Error writes a response with status code and message.

(c) Comments Generated by Different Techniques

Figure 2: Moderate Case 1 on Go Code Comment Generation

This figure shows a moderate case. In this case, both comments generated by CodeT5 and ICS contain the main contents of reference comment, but missing an important factual detail “http status”. We assume that this is because the word “http” does not appear in the code snippet. Moreover, our extractor does not work very well on this example, because it considers all statements in the code snippet to be important. This will make the role of important statements meaningless.

```
func NewDecoder(reg Registry, r io.Reader) Decoder {
    var buf []byte
    return decoder{reg, DefaultMaxSize, bytes.NewBuffer(buf), bufio.NewReader(r)}
}
```

(a) A Code Snippet $s_5$ from the Test Set of the CSN Dataset

```
func NewDecoder(reg Registry, r io.Reader) Decoder {}
return decoder{reg, DefaultMaxSize, bytes.NewBuffer(buf), bufio.NewReader(r)}
```

(b) Important Statements Selected by Our Extractor

**Reference Comment:** NewDecoder creates a new decoder using a type registry and an io . reader.
**CodeT5:** NewDecoder returns a new decoder that reads from r.
**ICS:** NewDecoder returns a new decoder using the given registry.

(c) Comments Generated by Different Techniques

Figure 3 : Moderate Case 2 on Go Code Comment Generation

This figure shows another moderate case. The reference comment can be divided into two parts: "creates a new decoder" and "using a type registry and an io . reader". Both comments generated by CodeT5 and our ICS covers the first part. However, for the second part, neither of them contains complete information. For CodeT5, its "read from r" can correspond to the "io . reader" in the reference comment, but "r" is just the name of the formal parameter of the method, which may be confusing to the readers. For ICS, it only mentions "registry" in the second part of the reference comment, which is left out by CodeT5. This case shows that there is still some room for improvement for current code comment generation models to make a high-level comprehensive comment of the code.

## 3. Weak Cases

```
func handle(w http.ResponseWriter, r *http.Request) {
    req, err := readRequest(r.Body, r.Header.Get(\"Content-Type\"))
    if err != nil {
        logrus.WithError(err).Error(\"read\")
    }
    if err := writeResponse(*req, w, onlyUpdateStatus); err != nil {
        logrus.WithError(err).Error(\"write\")
    }
}
```

(a) A Code Snippet $s_6$ from the Test Set of the CSN Dataset

```
func handle(w http.responseWriter, r *http.Request) {}
req, err := readRequest(r.Body, r.Header.Get(\"Content-Type\"))
if err := writeResponse(*req, w, onlyUpdateStatus); err != nil {}
```

(b) Important Statements Selected by Our Extractor

**Reference Comment:** Handle reads the request and writes the response.
**CodeT5:** Handle reads the request from the request body and writes the response.
**ICS:** Handle is the http . handler interface .

(c) Comments Generated by Different Techniques

Figure 4: Weak Case 1 on Go Code Comment Generation

This figure shows a weak case. In this example, the reference comment demonstrates that this function mainly contains two steps: "read the request" and "write the response". It is observed that CodeT5 covers the two steps correctly. However, although our extractor successfully extracts the important statement, the abstracter generates a comment that is unrelated to the functionality of method.

```
func (l *Lexer) errorf(format string, args ...interface{}) stateFn {
        l.handler(Token{TokenError, fmt.Sprintf(fmt.Sprintf(\"%s[%d:%d] \", l.Name,
l.LineNum(), l.Offset())+format, args...)})
        return nil
}
```

(a) A Code Snippet $s_7$ from the Test Set of the CSN Dataset

```
func (l *Lexer) errorf(format string, args ...interface{}) stateFn {}
Token{TokenError, fmt.Sprintf(fmt.Sprintf(\"%s[%d:%d] \", l.Name, l.LineNum(), l.Offset())+format,
args...)}
```

(b) Important Statements Selected by Our Extractor

**Reference Comment:** Errorf returns an error token and terminates the scan by passing back a nil pointer that will be the next state thus terminating the lexer.
**CodeT5:** Errorf is the error handler.
**ICS:** Errorf emits an error token.

(c) Comments Generated by Different Techniques

Figure 5: Weak Case 2 on Go Code Comment Generation

In this figure, the reference comment has two parts. The first part introduces the functionality of the code snippet, and the second part describes the return value of the method. From the picture we can observe that both the comments generated by CodeT5 and our ICS only cover the first part of the reference comment. What's more, while the second part does not describe the core functionality of the method, it provides valuable information for developers to understand and use the code snippet, which is difficult to summarize directly from the code snippet itself. From this case, we can see that there is still a gap between the comments produced by code comment generation models and those written by humans.