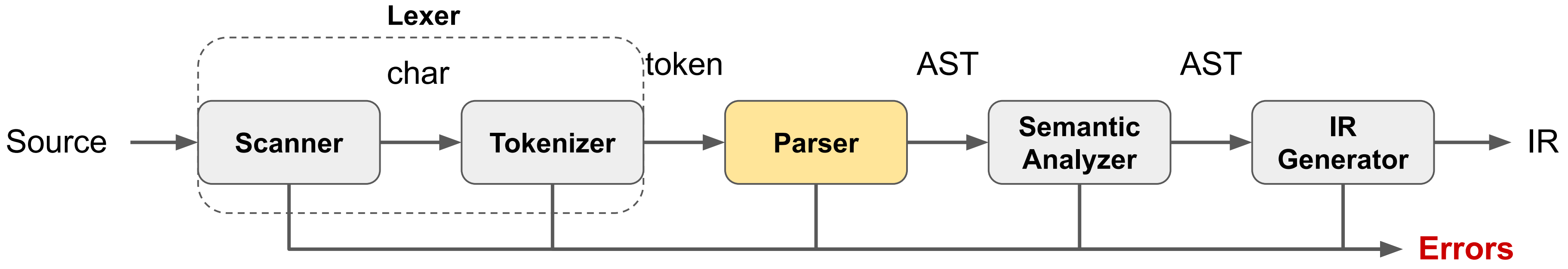
Compiling Techniques

Lecture 5: Top-Down Parsing

The Frontend



- Checks the stream of words/tokens produced by the lexer for grammatical correctness
- Determine if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Used to build an IR representation of the code

Specifying syntax with a grammar

Use Context-Free Grammar (CFG) to specify syntax

Context-Free Grammar definition

A Context-Free Grammar G is a quadruple (S, N,T, P) where:

- S is a start symbol
- N is a set of non-terminal symbols
- T is a set of terminal symbols or words
- P is a set of production or rewrite rules where only a single non-terminal is allowed on the left-hand side
  $P : N \rightarrow (N \cup T)*$

From Regular Expression to Context-Free Grammar

Kleene closure A*

- Replace ***A**** to ***Arep*** in all production rules
- Add new production rule **Arep = A Arep | ε**

Positive closure A+

- Replace ***A+*** to ***Arep*** in all production rules
- Add new production rule **Arep = A Arep | A**

Option [A]

- Replace ***[A]*** to ***Aopt*** in all production rules
- Add new production rule **Aopt = A | ε**

Example: Function Call

```
funcall ::= ID ”(” [ ID ( ”,” ID ) * ] “)”
```

Eliminate Option [ ]

```
funcall ::= ID ”(” arglist “)”
arglist ::= ID (“,” ID)* | ε
```

Remove Closure *

```
funcall ::= ID ”(” arglist “)”
arglist ::= ID argrep | ε
argrep  ::= “,” ID argrep | ε
```

Recursive-Descent Parsing

Steps to derive a syntactic analyser for a context free grammar expressed in an EBNF style:

- convert all the regular expressions as seen;
- implement a function for each non-terminal symbol A. This function recognises sentences derived from A;
- Recursion in the grammar corresponds to recursive calls of the created functions.

This technique is called recursive-descent parsing or predictive parsing.

Parser Class

```
class Parser:

    def check(self, expected : TokenKind) -> bool:
        return self.lexer.peek().kind == expected

    def match(self, expected : TokenKind) -> Token:
        if self.check(expected):
            token = self.lexer.peek()
            self.lexer.consume()
            return token

      raise Exception(f“Error: token of kind ${expected) not found”)
```

A recursive-descent parser

CFG for function call

```
funcall ::= ID ”(” arglist “)”
arglist ::= ID argrep | ε
argrep  ::= “,” ID argrep | ε
```

```
def parse_funcall():
    match(ID)
    match(LPAREN)
    parse_arglist()
    match(RPAREN)

def parse_arglist():
    if check(ID):
        match(ID)
        parse_argrep()

def parse_argrep():
    if check(COMMA):
        match(COMMA)
        match(ID)
        parse_argrep()
```

Be aware of infinite recursion!

Left Recursion

```
E ::= E ”+” T | T
```

The parser would recurse indefinitely!
Luckily, we can transform this grammar to:

```
E ::= T (”+” T)*
```

Removing Left Recursion

You can use the following rule to remove left recursion:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_{\square} \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_{\square}$ where

β_i does not start with A and $\alpha_i \neq \varepsilon$

can be rewritten into:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_{\square} A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_{\square} A' \mid \varepsilon$

Need for lookahead

A grammar that is hard-to-predict

```
stmt     ::= assign
           | funcall
funcall ::= ID “(“ arglist “)”
assign   ::= ID “=” exp
```

If the parser picks the wrong production, it may have to backtrack. Alternative is to look ahead to pick the correct production.

```
def parse_assign():
    match(ID)
    match(EQ)
    parse_exp()

def parse_funcall():
    match(ID)
    match(LPAREN)
    parse_arglist()
    match(RPAR)

def parse_stmt():
    ???
```

How much lookahead is needed?

- In general, an arbitrary amount

Fortunately,

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are LL(1) grammars.

LL(1)

- Left-to-Right parsing;
- Leftmost derivation; (i.e. apply production for leftmost non-terminal first)
- only 1 current symbol required for making a decision.

First Sets

Basic idea: given $A \rightarrow \alpha|\beta$, the parser should be able to choose between α and β.

First Sets

For some symbol $\alpha \in N \cup T$, define First(α) as the set of symbols that appear first in some string that derives from α: $x \in$ First(α) iif $\alpha \rightarrow \ldots \rightarrow x\gamma$, for some γ.

The LL(1) property: if $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like:

$$\text{First}(\alpha) \cap \text{First}(\beta) = \varnothing$$

This would allow the parser to make the correct choice with a lookahead of exactly one symbol! (almost, see next slide!)

ε Productions

What about ε productions (the ones that consume no symbols)?

```
G ::= C b
C ::= A | B
A ::= a | ε
B ::= b
```

Input: `b`

The parser does not know whether to go down the A derivation or B derivation:

- In the case of A, we could choose the ε and consume nothing, and the b will be consumed in G (which is the only valid derivation);
- In the case of B, we could directly consume the b, but then we will have a problem later on and would need to backtrack.

Therefore, the parser *may have to backtrack* since it needs to try out different paths.

ε Productions (cont.)

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ First(α), then we need to ensure that First(β) is disjoint from Follow(α).

Follow(α) is the set of all terminal symbols in the grammar that can legally appear immediately after α. (See EaC§3.3 for details on how to build the First and Follow sets.)

Let's define First+(α) as:

- First(α) $\cup$ Follow(α), if $\varepsilon \in$ First(α)
- First(α) otherwise

LL(1) grammar

A grammar is LL(1) iff $A \rightarrow \alpha$ and $B \rightarrow \beta$ implies: First+(α) $\cap$ First+(β) = $\varnothing$

LL(1) property

Given a grammar that has the LL(1) property:

- each non-terminal symbols appearing on the left hand side is recognised by a simple routine;
- the code is both simple and fast.

Predictive Parsing

Grammar with the LL(1) property are called predictive grammars because the parser can “predict” the correct expansion at each point. Parsers that capitalise on the LL(1) property are called predictive parsers. One kind of predictive parser is the recursive descent parser.

LL(k)

Sometimes, we might need to lookahead one or more tokens.

LL(2) Grammar Example

```
stmt    ::= assign | funcall
assign  ::= IDENT ”=” exp
funcall ::= IDENT ”(” arglist ”)”
```

```
def parse_stmt():
    if check([IDENT, EQ])
        parse_assign()
    if check([IDENT, LPAREN]):
        parse_funcall()
    error()
```

We check two symbols ahead as only the 2nd symbol distinguishes the two cases!

Next Lecture

- Dealing with ambiguity
- Bottom-up parsing