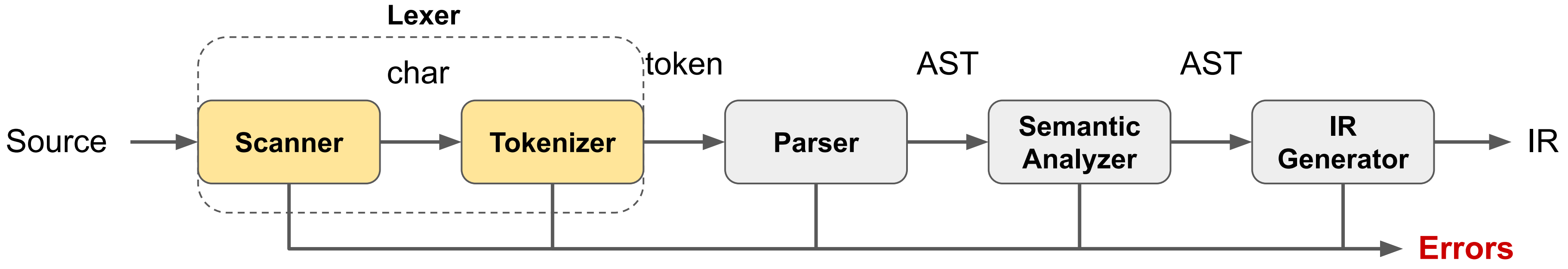
Compiling Techniques

Lecture 3: Lexical Analysis

The Lexer



- Maps character stream into words — the basic unit of syntax
- Assign a syntactic category to each work (part of speech)
    - x = x + y; becomes ID(x) EQ ID(x) PLUS ID(y) SC
    - word ~= lexeme
    - syntactic category ~= part of speech
    - In casual speech, we call the pair a token
- Typical tokens: number, identifier, +, −, new, while, if, . . .
- Scanner eliminates white space (including comments)

Context-free Language

Context-free syntax is specified with a grammar

- SheepNoise → SheepNoise baa | baa
- This grammar defines the set of noises that a sheep makes under normal circumstances

It is written in a variant of Backus Naur Form (BNF)

Formally, a grammar G = (S,N,T,P)

- S is the start symbol
- N is a set of non-terminal symbols
- T is a set of terminal symbols or words
- P is a set of productions or rewrite rules ($P: N \rightarrow N \cup T$)

Simple Expression Grammar

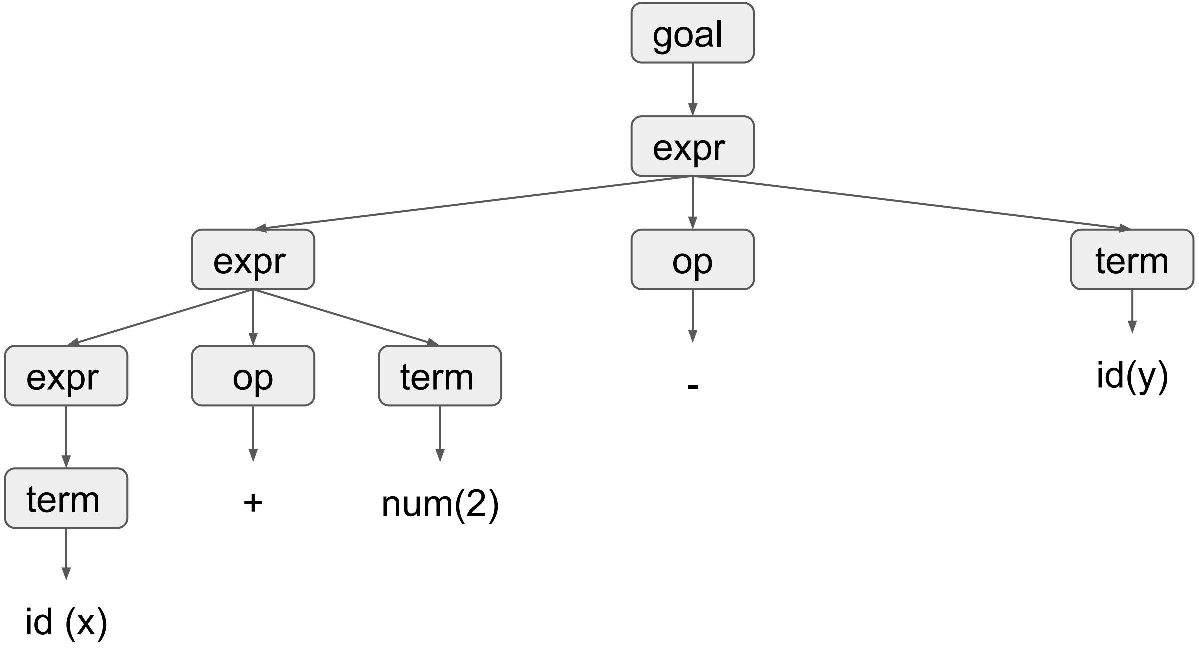
```
goal → expr
expr → expr op term | term
term → number | id
op   → + | -
```

```
S = goal
T = { number, id , +, − }
N = { goal , expr , term , op }
P = { 1, 2, 3, 4, 5, 6, 7 }
```

- This grammar defines simple expressions with addition & subtraction over “number” and “id”
- This grammar, like many, falls in a class called “context-free grammars”, abbreviated CFG

Parse Tree

x + 2 - y



Regular Expression

Grammars can often be simplified and shortened using an augmented BNF notation where:

- x* is the Kleene closure : zero or more occurrences of x
- x+ is the positive closure : one or more occurrences of x
- [x] is an option: zero or one occurrence of x

Example: identifier syntax

```
identifier ::= letter ( letter | digit)*
digit ::= "0" | … | "9"
letter ::= "a" | … | "z" | "A" | … | "Z"
```

Exercise: Signed Numbers

Task: Write the grammar of signed integers

Use pen and paper to write down such a grammar!

Regular Language

Definition

A language is regular if it can be expressed with a single regular expression or with multiple non-recursive regular expressions.

- Regular languages can be used to specify the words to be translated to tokens by the lexer.
- Regular languages can be recognised with finite state machine.
- Using results from automata theory and theory of algorithms, we can automatically build recognisers from regular expressions.

Regular Language to Program

Given the following:

- c is a lookahead character;
- next() consumes the next character;
- error () quits with an error message; and
- first (exp) is the set of initial characters of exp.

Regular Language to Program

<table>
<tr><th>RE</th><th>pr(RE)</th></tr>
<tr><td>“x”</td><td><pre>if c == x:
  next()
else:
  error()</pre></td></tr>
<tr><td><code>(exp)</code></td><td><pre>pr(exp)</pre></td></tr>
<tr><td><code>[exp]</code></td><td><pre>if c in first(exp):
  pr(exp)</pre></td></tr>
<tr><td><code>exp*</code></td><td><pre>while c in first(exp):
  pr(exp)</pre></td></tr>
</table>

Regular Language to Program

RE	pr(RE)
exp+	pr(exp) while c in first(exp): pr(exp)
fact_1 … fact_n	pr(fact_1); … ; pr(fact_n)

term_1 | … | term_n	if c in first(term_1):
	  pr(term_1)
	elif …
	  …
	elif c in first(term_n):
	  pr(term_n)
	else
	  error()

Left Parsable

Definition: left-parsable

A grammar is left-parsable if:

```
term_1 | … | term_n    | The terms do not share any initial symbols.
fact_1 … fact_n        | If fact_i contains the empty symbol then fact_i and
                       | fact_i + 1 do not share any common initial symbols.
[exp], exp*            | The initial symbols of exp cannot contain a symbol
                       | which belong to the first set of an expression
                       | following exp.
```

Example: Recognising identifiers

```
void ident () {
  if (c is in [ a-zA-Z ] )
    letter();
  else
    error();

  while (c is in [ a-zA-Z0-9]) {
    switch (c) {
      case c is in [ a-zA-Z ] : letter();

      case c is in [0 -9] : digit();

      default : error();
    }
  }
}

void letter( ) { … }

void digit() { … }
```

Example: Simplified Python Version

```
void ident () {
  if (Character.isLetter(c))
    next();
  else
    error();
  while (Character.isLetterOrDigit(c))
    next();
}
```

Role of Lexical Analyser

The main role of the lexical analyser (or lexer) is to read a bit of the input and return a lexeme (or token).

```
def Lexer:
  def nextToken(self) {
    // return the next token, ignoring whitespaces
  }
  …

}
```

White spaces are usually ignored by the lexer. White spaces are:

- white characters (tabulation, newline, . . .)
- comments (any character following “//” or enclosed between “/*” and “*/”

What is a token?

A token consists of a token class and other additional information.

Example: some token classes

IDENTIFIER	→	foo , main , cnt , …
NUMBER	→	0 , −12, 1000 , …
STRING_LITERAL	→	”Hello world!”, ”a”, …
EQ	→	==
ASSIGN	→	=
PLUS	→	+
LPAR	→	(
…	→	…

```
class Token:
  Kind: TokenKind
  Value: Any = None
```

Example

Given the following Python program:

```
def foo (i):
  return i+2
```

the lexer will return:

```
DEF IDENTIFIER(”foo”) LPAR IDENTIFIER (”i”) RPAR COLON
  RETURN IDENTIFIER(”i”) PLUS NUMBER(”2”)
```

A Lexer for Simple Arithmetic Expressions

Example: BNF syntax

```
identifier    ::= letter ( letter | digit )*
digit         ::= ”0” | . . . | ”9”
letter        ::= ”a ” | . . . | ” z ” | ”A” | . . . | ”Z”
number        ::= digit+
plus          ::= ”+”
minus         ::= ”−”
```

Example: token definition

```
from enum import Enum
from dataclasses import dataclass


class TokenClass(Enum):
    IDENTIFIER = 0
    NUMBER     = 1
    PLUS       = 2
    MINUS      = 3

@dataclass
class Token:
    type: TokenClass
    value: any = None

    def __repr__(self):
       return self.type.name + ((":" + str(self.value)) if self.value else "")
```

Example: scanner implementation

```
class Scanner:
    def __init__(self, stream):
        self.stream = stream
        self.buffer = None

    def peek(self):
        if not self.buffer:
            self.buffer = self.next()
        return self.buffer
```

```
    def next(self):
        if self.buffer:
            c = self.buffer
            self.buffer = None
            return c

        return self.stream.read(1)
```

Example: Tokenizer implementation

```
class Tokenizer:
    def __init__(self, scanner):
        self.scanner = scanner
        self.buffer = None

    def peek(self):
        if not self.buffer:
            self.buffer = self.next()
        return self.buffer
```

```
    def next(self):
        if self.buffer:
            c = self.buffer
            self.buffer = None
            return c
        c = self.scanner.next()

        if c.isspace():
            return self.next()

        if c == "+":
            return Token(TokenClass.PLUS)

        if c == "-":
            return Token(TokenClass.MINUS)
```

Example: Tokenizer implementation (continued)

```
if c.isalpha():
    name = c
    c = self.scanner.peek()
    while c.isalpha() or c.isdigit():
        name += c
        self.scanner.next()
        c = self.scanner.peek()

    return Token(TokenClass.IDENTIFIER, name)
```

Example: Tokenizer implementation (continued)

```
if c.isdigit():
    digits = c
    c = self.scanner.peek()
    while c.isdigit():
        digits += c
        self.scanner.next()
        c = self.scanner.peek()
    value = int(digits)
    return Token(TokenClass.NUMBER, value)
```

Next Lecture

- Automatic Lexer Generation