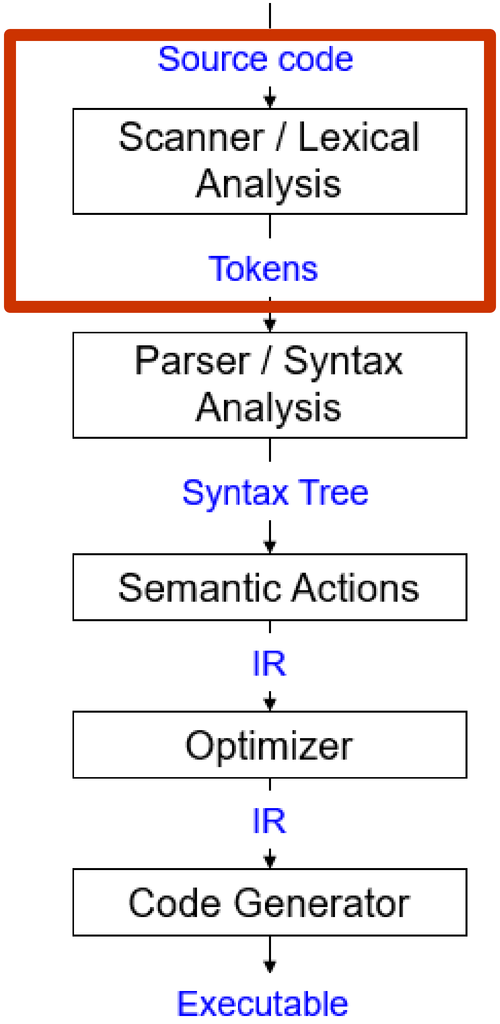
CS406: Compilers

Spring 2022

Week 3: Scanners (conclusion), Parsers

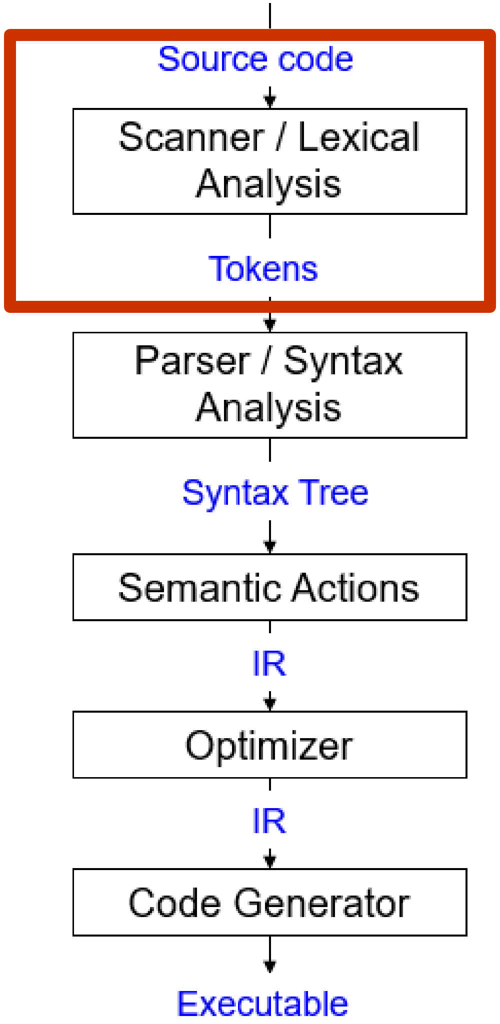
Scanners (Summary)

- Also called *Lexers* / *Lexical Analyzers*
- Input: stream of letters (program text / source code), Output: sequence / list of *tokens*
- Token: a pair <category/class, value>
  - Category defines a string *pattern*
  - Value also called *lexeme*
  - Value is a *prefix* (and hence, is a substring)
  - Value matches on of the patterns that category defines
- Scan *left-to-right* in program text, *look-ahead* to identify tokens.
  - Look-ahead buffer size determined by language design



Scanners (Summary)

- *Regular expressions* are used to formally define the patterns specified by token classes.
  - Some customization done while defining regular expressions: 1) Match the longest substring possible 2) Handle errors
- Tools such as Flex and ANTLR convert regular expressions to code. The code is your scanner implementation
  - The implementation typically converts regular expressions to *Finite Automata* (special kind of state diagram)
    - Automata are coded using efficient algorithms (E.g. Table-lookup method)
  - Efficient algorithms exist for substring matching (requiring single-pass over input program text)
    - Aho-Corasic, Knuth-Morris-Pratt (KMP)



Parsers - Overview

- Also called syntax analyzers
- Determine two things:
  1. Is a program syntactically valid?

     (Analogy) is an English language sentence grammatically correct?
  2. What is the structure of programming language constructs? E.g. does the sequence*

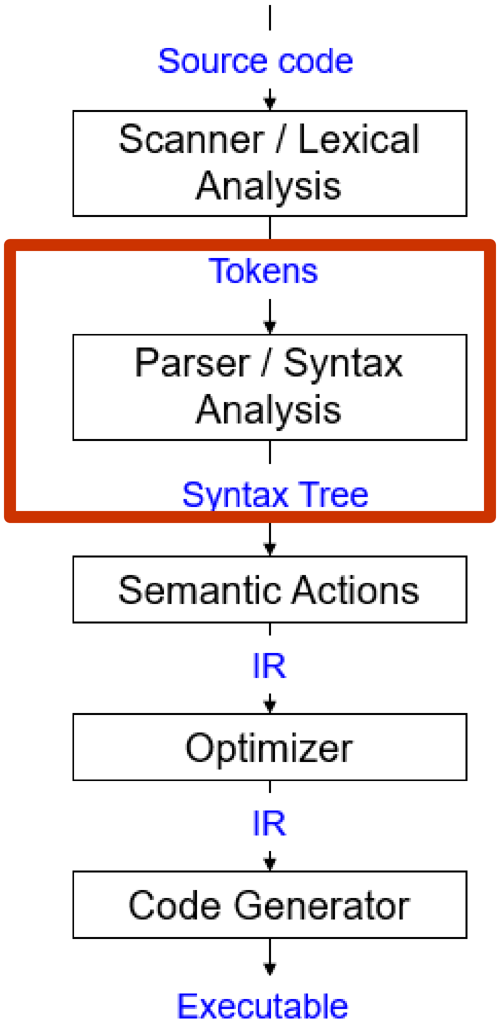
     `IF, ID(a), OP(<), ID(b), {, ID(a), ASSIGN, LIT(5), }}`

     refer to an `if` statement?

     (Analogy) diagramming English sentences

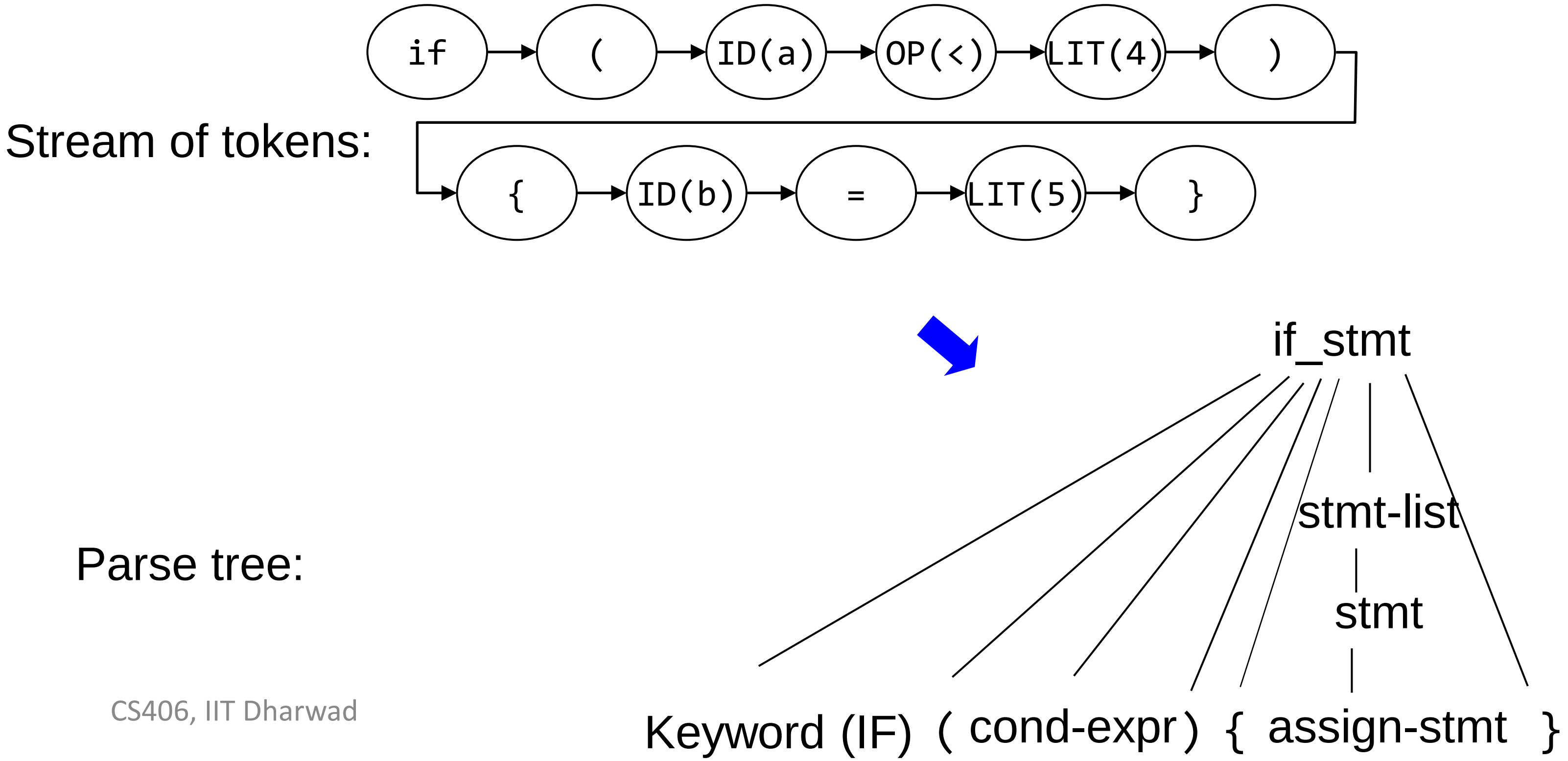
* Correponding program text:

```
if (a < 4) {
    b = 5
}
```



Parsers - Overview

- Input: stream of tokens
- Output: Parse tree
  - sometimes implicit



Parsers – what do we need to know?

1. How do we define language constructs?
   - Context-free grammars
2. How do we determine: 1) valid strings in the language? 2) structure of program?
   - LL Parsers, LR Parsers
3. How do we write Parsers?
   - E.g. use a parser generator tool such as Bison

Languages

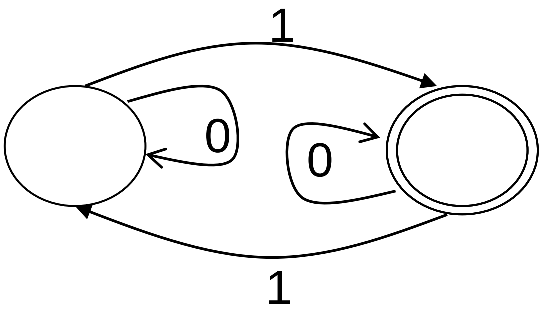
- A language is (possibly infinite) set of strings
- Regular expressions specify *regular languages.* However, regular languages are *weak formal languages* to describe the features of a practical programming language.

  What set of strings does this FA accept?

  The FA shown accepts all string with *odd number of 1s.*

  What is the regular expression for the FA?

  (0*10*)(10*10*)*



Regular expressions can describe strings specifying *parity*:

`{ mod k | k=# states in FA}`

weakness: regular expressions can't describe a string of the form: $\{ (^i\ )^i \mid i>=1\}$

Regular Languages

- Regular expressions can’t describe a string of the form:

$$\{ (^i\ )^i \mid i>=1\}$$

E.g. Parenthesized expressions

```
((2+3)*5)
```

Programming language syntax is i.e. recursive

```
((( int x; )))
```

Nested structures:

```
IF
  IF
    IF
    FI
  FI
FI
```

Context Free Grammar (CFG)

- Natural notation for describing <u>recursive structure</u> definitions. Hence, suitable for specifying language constructs.

- Consist of:
  - A set of *Terminals* (T)
  - A set of *Non-terminals* (N)
  - A *Start Symbol* (S∈N)
  - A *set of Productions* ($X \to Y_1..Y_N$) (aka. rules)

$$P: X \longrightarrow Y_1 Y_2 Y_3 .. Y_N \qquad X \in N, \qquad Y_i \in N \cup T \cup \epsilon/\lambda$$

Context Free Grammar (CFG)

- Grammar G $= (T, N, S, P)$

  E.g. $G = (\{a,b\}, \{S, A, B\}, S, \{S \rightarrow AB, A \rightarrow Aa$
  $A \rightarrow a, B \rightarrow Bb, B \rightarrow b\})$

- Implicit meanings
  - <u>First rule</u> listed in the set of productions contains <u>start symbol</u> (on the left-hand side)
  - In the set of productions, <u>you can replace the symbol X</u> (appearing on the right-hand side only) with the <u>string of symbols</u> that are on the right-hand side of a rule, which has X (on the left-hand side)

Context Free Grammar (CFG)

1. Begin with only S as the initial string
2. Replace S
   - S replaced with AB
3. Repeat 2 until the string contains only terminals
   i. AB replaced with aB
   ii. aB replaced with ab

```
G = (T, N, S, P)
P:{ S->AB,
    A->Aa,
    A->a,
    B->Bb,
    B->b }
```

Summary: we move from S to a string of terminals through a series of transformations:

$\alpha_0 \rightarrow \ldots \rightarrow \alpha_n$ where $\alpha_1 \ldots \alpha_n$ are strings

Shorthand notation: $\alpha_0 \xrightarrow{*} \alpha_n$

Language of the Grammar

- Language L(G) of the context-free grammar G
  - Set of strings that can be derived from S
  - $\{a_1a_2a_3..a_N \mid a_i \in T \ \forall i \text{ and } S \overset{*}{->} a_1a_2a_3..a_N\}$
  - Is called context-free language
    - All regular languages are context-free but not vice-versa.
    - Can have many grammars generating same language.

Context-Sensitive Grammar

- Can have context-sensitive grammar and languages (think: aB->ab)
  - Cannot replace right-hand side with left-hand side irrespective of the context.
  - E.g. aB->ab lays down a context: ‘a’ must be a prefix in order to transform the string “aB” to a string of terminals “ab”
    - ccaBb can be replaced by ccabb

Is grammar G context-free?

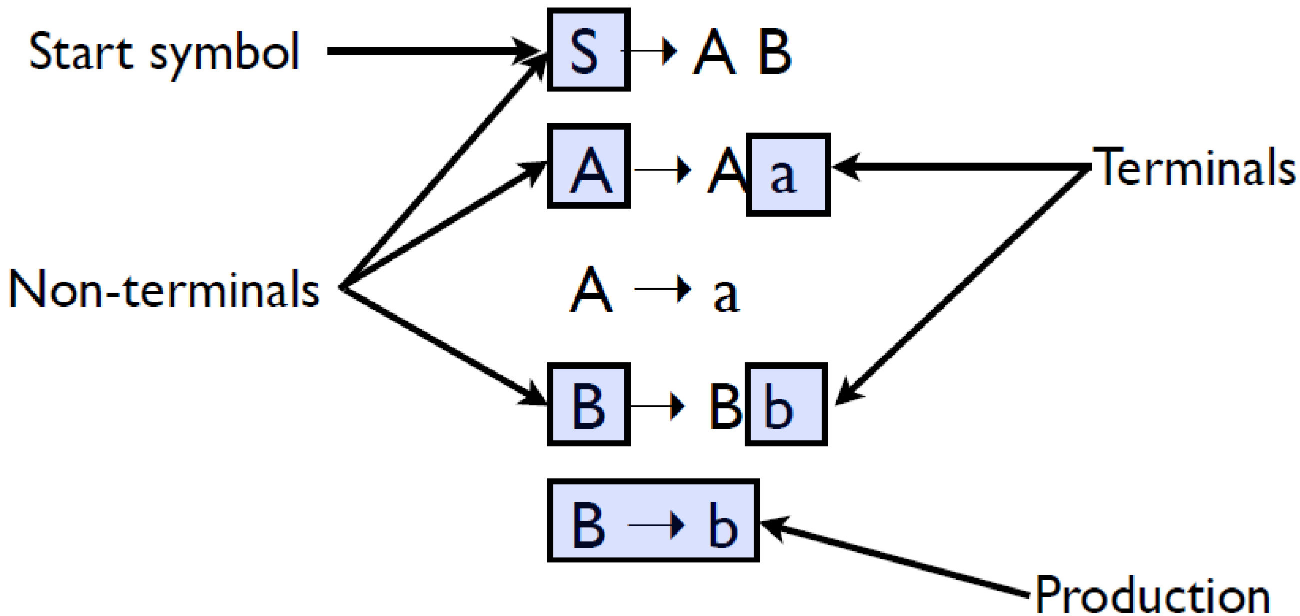
```
G = (T, N, S, P)
P:{ S->AB,
   A->Aa,
   A->a,
   B->Bb,
   B->b }
```

Does a string belong to the Language?

- How do we apply the grammar rules to determine the validity of a string? (i.e. string belongs to the language specified by the context-free grammar)
  - Begin with S
  - Replace S
  - Repeat till string contains terminals only

    L(G) must contain strings of terminals only
- Notation:
  - We will use Greek letters to denote strings containing non-terminals and terminals

Simple grammar



A -> Aa
| a

Backus Naur Form (BNF)

Generating strings

$S \rightarrow A\ B$

$A \rightarrow A\ a$

$A \rightarrow a$

$B \rightarrow B\ b$

$B \rightarrow b$

- Given a start rule, productions tell us how to rewrite a non-terminal into a different set of symbols
- Some productions may rewrite to λ. That just removes the non-terminal

To derive the string "a a b b b" we can do the following rewrites:

$$S \Rightarrow A\ B \Rightarrow A\ a\ B \Rightarrow a\ a\ B \Rightarrow a\ a\ B\ b \Rightarrow$$

$$a\ a\ B\ b\ b \Rightarrow a\ a\ b\ b\ b$$

Slide courtesy: Milind Kulkarni

Exercise

Which of the below strings are accepted by the grammar:

```
1: A ->  aAa
2: A ->  bBb
3: A ->  λ
4: B ->  cA
5: B ->  λ
```

1. abcba   1->2->4->3
2. abcbca
3. abba   1->2->5
4. abca