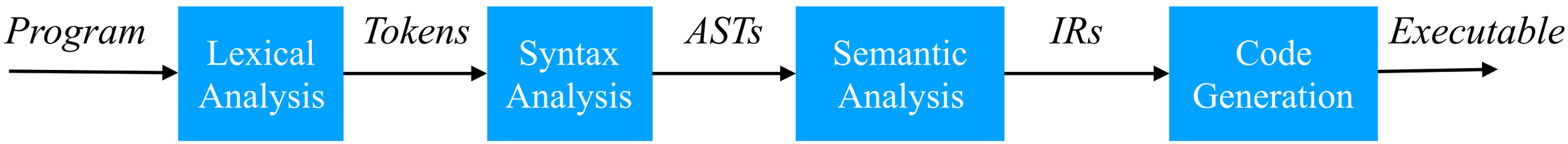
CS 162 Programming languages

Lecture 5: Introduction to Lexing and Parsing

Yu Feng

Winter 2021

A typical flow of a compiler



Lexical analysis

- Main Question: How to give structure to strings
- Analogy: Understanding an English sentence
  - First, we separate a string into words
  - Second, we understand sentence structure by diagramming the sentence
- Separating a string into words is called ***lexing***
- Note that lexing is not necessarily trivial

Lexical analysis

- Consider the following λ^+ program:

```
if x > y

then 10

else 8
```

- This program is just a string of characters

```
if x > y\nthen\t10\nelse\t8
```

- Goal: Portion the input string into substrings where the substrings are ***tokens***

What is a Token?

- Token is a syntactic category
- Example in English: noun, verbs, adjectives,...
- In a programming language: constants, identifiers, keywords, whitespaces…

Tokens in λ^+

- Tokens correspond to sets of strings
- Identifier: strings of letters, digits and '_' starting with a letter
- Integer: a non-empty string of digits
- Keywords: "let", "lambda", "if", ...
- Whitespace: a non-empty sequence of blanks, newlines, and tabs

What are tokens for?

- Classify program substrings according to their role
- Output of lexical analysis is a stream of tokens...
- ...which is input to the parser
- Parser relies on token distinction
  - An identifier is treated different than a keyword

Regular language/expressions

- We could specify tokens in many ways
- Regular Languages are the most popular
- Simple and useful theory
- Easy to understand
- Efficient to implement

The lexer in λ^+

```
rule token = parse
  | [' ' '\r' '\t' '\n'] { token lexbuf }
  | "let"               { LET }
  | "in"                { IN }
  | "fun"               { FUN }
  | "with"              { WITH }
  | "lambda"            { LAMBDA }
  | "if"                { IF }
  | "then"              { THEN }
  | "else"              { ELSE }
  | "isnil"             { ISNIL }
  | "!"                 { HEAD }
  | "#"                 { TAIL }
  | "Nil"               { NIL }
  | "@"                 { CONS }
  | "+"                 { PLUS }
  | "-"                 { SUB }
  | "*"                 { TIMES }
  | ['0'-'9']+ as n { NUMBER(int_of_string(n)) }
  | ['a'-'z' '_']['a'-'z' '0'-'9' '_']* as x { ID(x) }
  | "->"                { THINARROW }
```

The role of a parser

Phase	Input	Output
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree

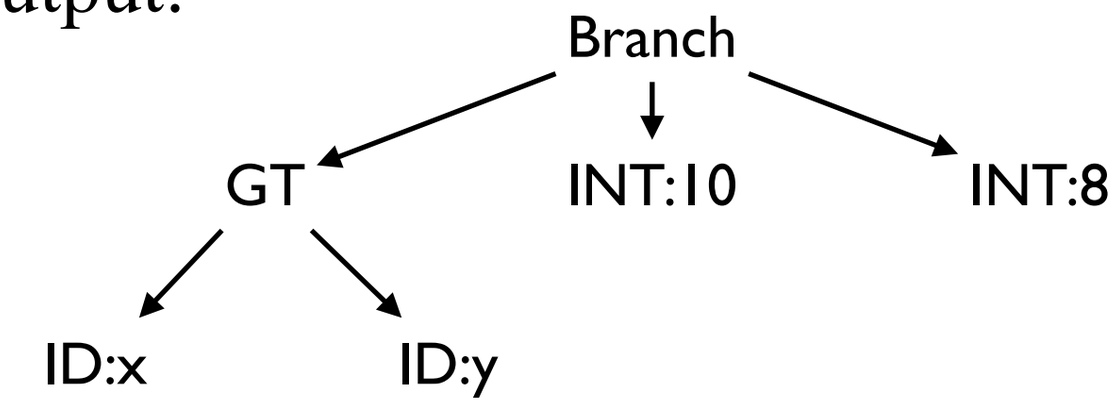
- Input: sequence of tokens from the lexer
- Output: parse tree (Abstract Syntax Tree) of the program

Example

- Input: Consider the previous λ^+ expression: **if** x>y **then** 10 **else** 8

- Parse Input: TOKEN_IF TOKEN_ID("x") TOKEN_GT TOKEN_ID("y") TOKEN_THEN TOKEN_INT(10) TOKEN_ELSE TOKEN_INT(8)

- Parser Output:



The role of a parser

- Not all strings of tokens are programs...
- Parser must distinguish between valid and invalid strings of tokens
- What we need:
  - A language for describing valid strings of tokens
  - A method for recognizing if a string of tokens is in this language or not

Context free grammar (CFGs)

- Programming language constructs have ***recursive*** structure
- Example: An λ^+ expression is
  - *expression* + *expression*,
  - **if** *expression* **then** *expression* **else** expression, ...
- Context free grammars are a natural notation for this recursive structure

CFGs in more detail

- A CFG consists of:
  - A set of terminals T
  - A set of non-terminals N
  - A start symbol S (non-terminal)
  - A set of productions: $X \rightarrow Y_1 Y_2 \ldots Y_n$

where $X \in N$ and $Y_i \in (T \cup N \cup \{\varepsilon\})$

CFGs example

- Recall the earlier fragment of λ^+:

EXPR → **if** *EXPR* **then** *EXPR* **else** *EXPR*

| *EXPR* + *EXPR*

| *ID*

- Some strings in this language:

ID

IF ID THEN ID ELSE ID

ID + ID

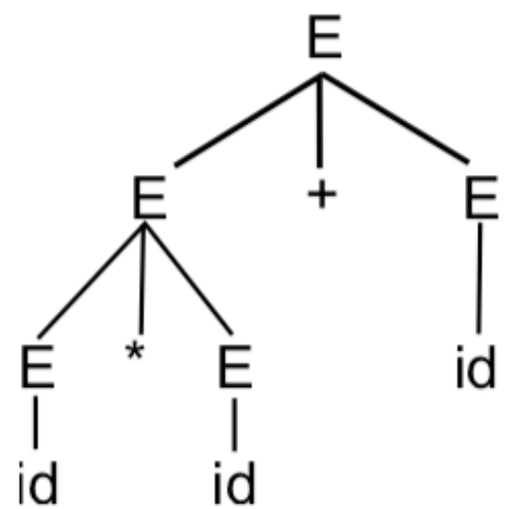
IF ID THEN ID+ID ELSE ID

IF IF ID THEN ID ELSE IF THEN ID ELSE ID

From derivations to parse trees

- A derivation is a sequence of productions: $S \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots$
- A derivation can be drawn as a tree
  - Start symbol is the tree's root
- For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X

```
   E
→ E+E
→ E*E+E
→ id*E+E
→ id*id + E
→ id*id + id
```



Ambiguity

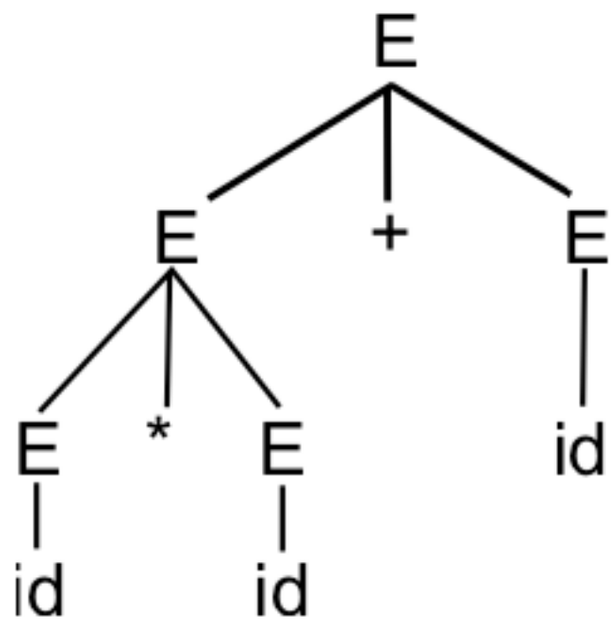
- Consider this grammar:

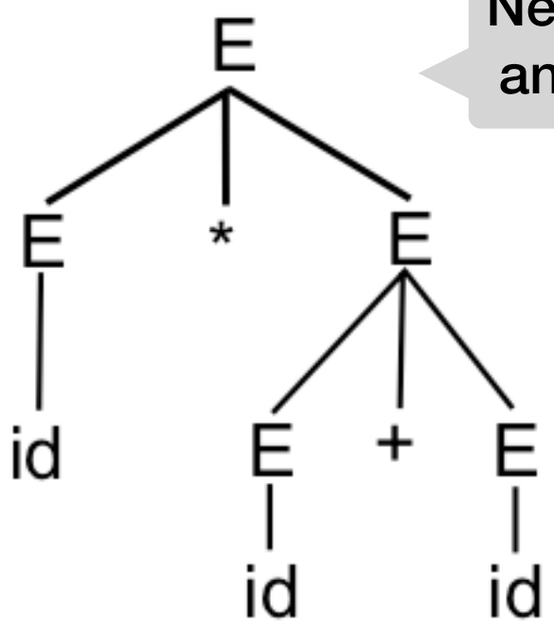
EXPR → *E* * *E*

| *E*+*E* | *(E)*

| *id*

- Now, this string *id*id*+*id* has two parse trees!





Need Precedence and Associativity

The parser in λ^+

```
expr:
    | LAMBDA idlist DOT expr %prec LAMBDA { mk_lambdas $2 $4 }
    | FUN ID WITH idlist EQ expr IN expr  { LetBind($2, mk_lambdas $4 $6, $8) }
    | LPAREN expr RPAREN                  { $2 }
    | IF expr THEN expr ELSE expr         { IfThenElse($2, $4, $6) }
    | LET ID EQ expr IN expr              { LetBind($2, $4, $6) }
    | term                                { $1 }
```

```
%nonassoc LPAREN ID NIL NUMBER TRUE FALSE
%right LAMBDA
%left AND OR
%left LT GT EQ
%left PLUS SUB
%left TIMES
%right CONS
%left APP
%nonassoc HEAD TAIL
```

TODOs by next lecture

- Hw2 is out. Please start early!
- Come to the discussion session if you have questions