CO20-320241

Computer Architecture and Programming Languages

CAPL

Lecture 24 & 25

Dr. Kinga Lipskoch

Fall 2019

Postfix Notation (1)

1. If `E` is a variable or constant, then the postfix notation for `E` is `E` itself
2. If `E` is an expression of the form `E1 op E2`, where `op` is any binary operator, then the postfix notation for `E` is `E'1 E'2 op`, where `E'1` and `E'2` are the postfix notations for `E1` and `E2`, respectively
3. If `E` is a parenthesized expression of the form `(E1)`, then the postfix notation for `E` is the same as the postfix notation for `E1`

The postfix notation for `(9-5)+2` is `95-2+`

Postfix Notation (2)

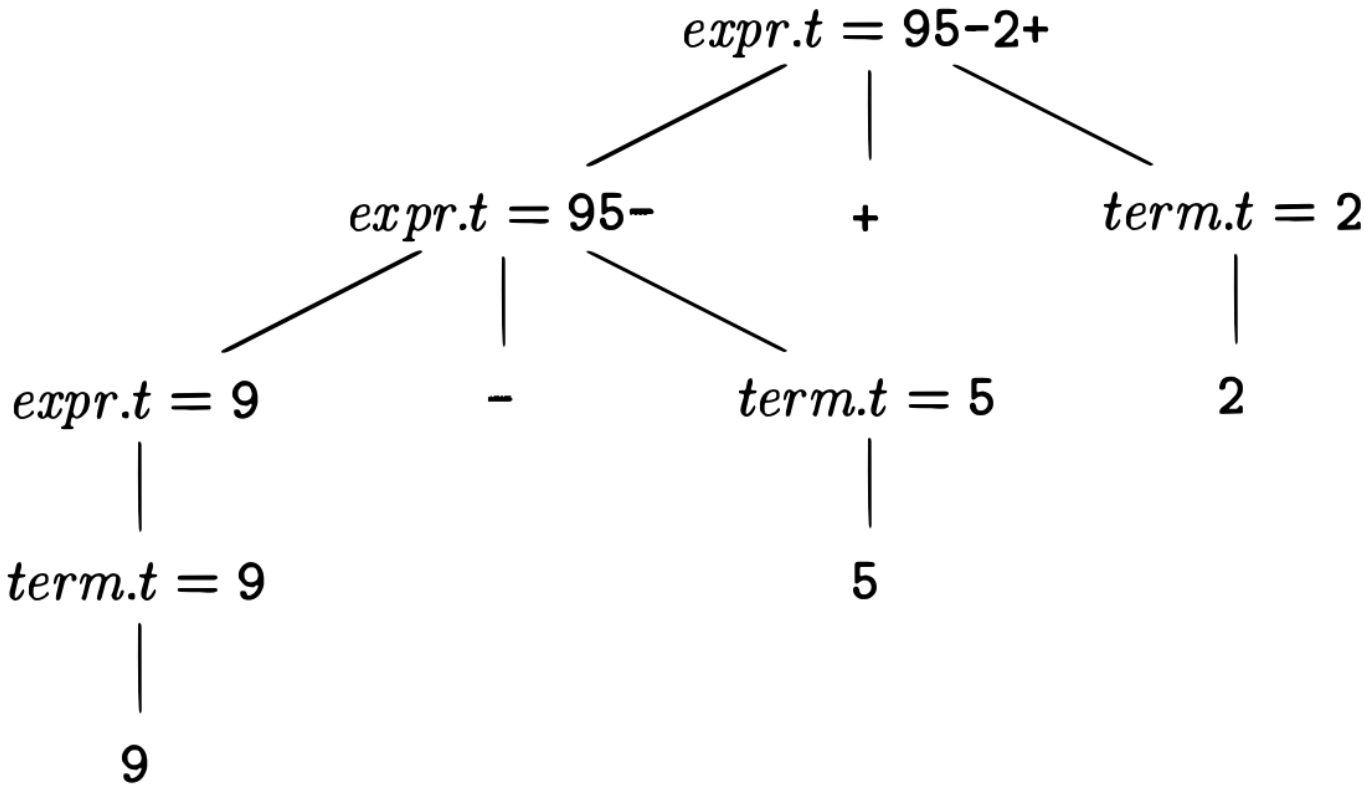
- The "trick" is to repeatedly scan the postfix string from the left, until you find an operator
- Then, look to the left for the proper number of operands, and group this operator with its operands
- Example: Consider the postfix expression `952+-3*`
- Evaluation results in: $\rightarrow$ `97-3*` $\rightarrow$ `23*` $\rightarrow$ `6`

Synthesized Attributes

A syntax-directed definition associates

1. With each grammar symbol, a set of attributes, and
2. With each production, a set of semantic rules for computing the values of the attributes associated with the symbols appearing in the production

Attribute Values at Nodes in a Parse Tree



Syntax-Directed Definition for Infix to Postfix Translation

- Each nonterminal has a string-valued attribute `t` that represents the postfix notation for the expression generated by that nonterminal in a parse tree
- The symbol $||$ in the semantic rule is the operator for string concatenation

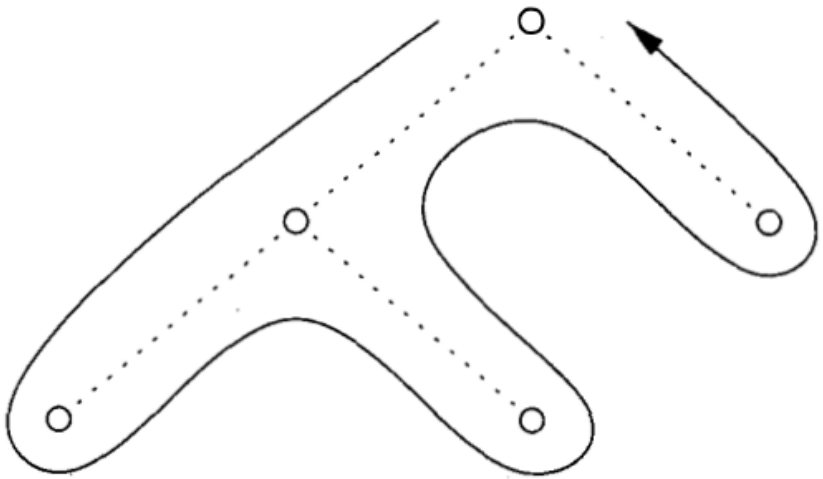
PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Tree Traversals

- Tree traversals will be used for describing attribute evaluation and for specifying the execution of code fragments in a translation scheme
- A traversal of a tree starts at the root and visits each node of the tree in some order
- A depth-first traversal starts at the root and recursively visits the children of each node in any order, not necessarily from left to right

Left-to-Right Depth-First Traversal

```
1 procedure visit(node N) {
2   for(each child C of N, from left to right) {
3     visit(C);
4   }
5   evaluate semantic rules at node N;
6 }
```



Evaluation of Attributes

- A syntax-directed definition does not impose any specific order for the evaluation of attributes on a parse tree; any evaluation order that computes an attribute `a` after all the other attributes that `a` depends on is acceptable
- Synthesized attributes can be evaluated during any bottom-up traversal, that is, a traversal that evaluates attributes at a node after having evaluated attributes at its children

Translation Schemes (1)

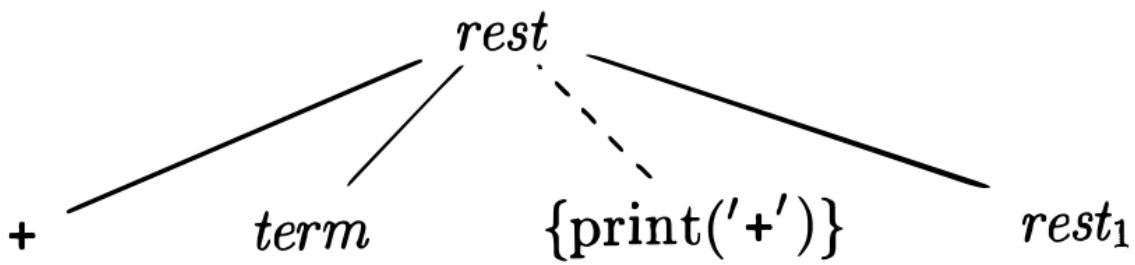
- The syntax-directed definition from previous table builds up a translation by attaching strings as attributes to the nodes in the parse tree
- Consider an alternative approach that does not need to manipulate strings; it produces the same translation incrementally, by executing program fragments
- A syntax-directed translation scheme is a notation for specifying a translation by attaching program fragments to productions in a grammar
- Compared to the syntax-directed definition the order of evaluation of the semantic rules is explicitly specified

Translation Schemes (2)

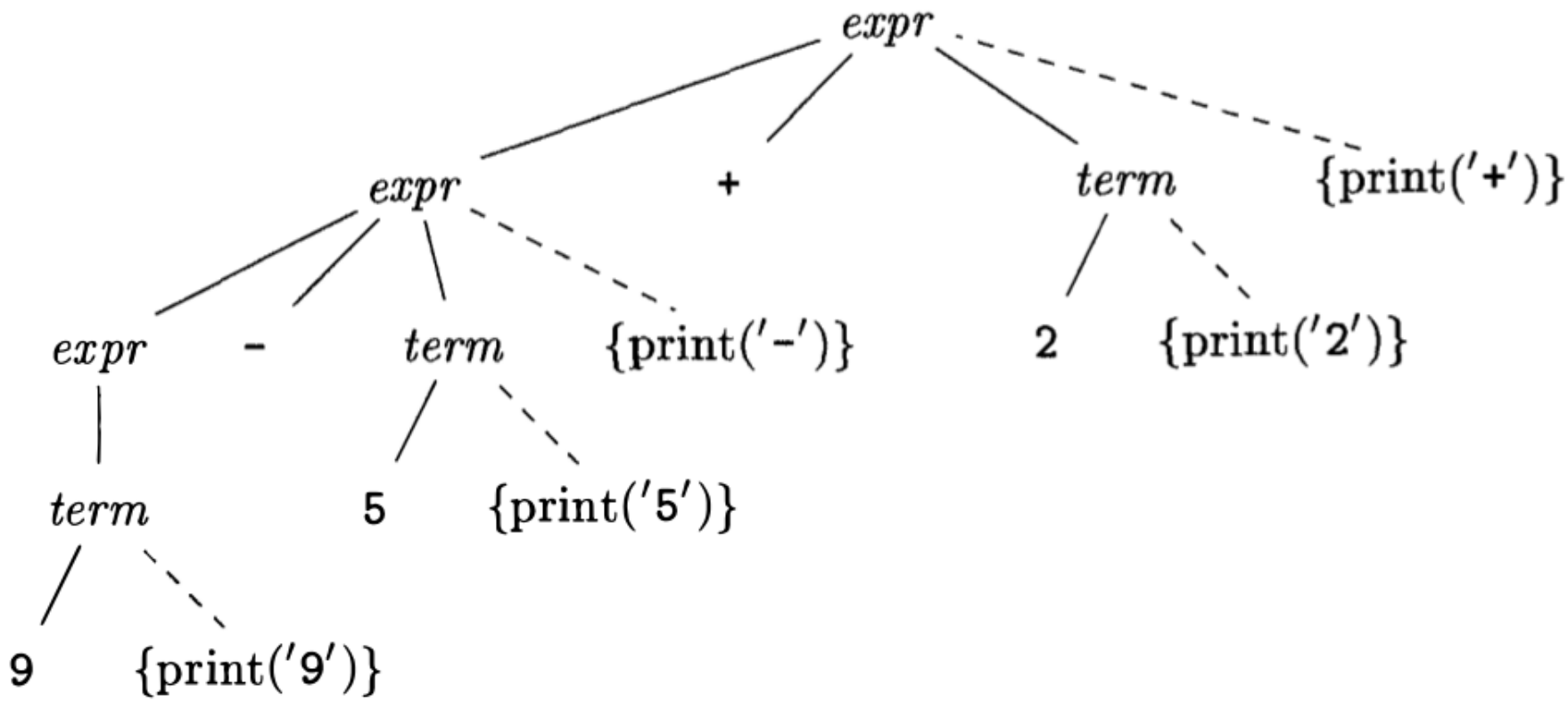
- Program fragments embedded within production bodies are called semantic actions
- The position at which an action is to be executed is shown by enclosing it between curly braces and writing it within the production body
- Example: `rest` $\rightarrow$ `+ term {print('+')} rest1`
- When drawing a parse tree for a translation scheme, we indicate an action by constructing an extra child for it, connected by a dashed line to the node that corresponds to the head of the production

Example

- The node for a semantic action has no children, so the action is performed when that node is first seen
- The portion of the parse tree for the above production and action is below



Actions Translating 9–5+2 into 95–2+



Actions for Translating into Postfix Notation

In a postorder traversal, we first perform all the actions in the leftmost subtree of the root, also labeled `expr` like the root. We then visit the leaf + at which there is no action. We next perform the actions in the subtree for the right operand `term` and, finally, the semantic action `{print('+')}` at the extra node.

$$
\begin{array}{lcll}
expr & \rightarrow & expr_1 \texttt{ + } term & \{\text{print}('+')\} \\
expr & \rightarrow & expr_1 \texttt{ - } term & \{\text{print}('-')\} \\
expr & \rightarrow & term & \\
term & \rightarrow & 0 & \{\text{print}('0')\} \\
term & \rightarrow & 1 & \{\text{print}('1')\} \\
 & & \cdots & \\
term & \rightarrow & 9 & \{\text{print}('9')\}
\end{array}
$$

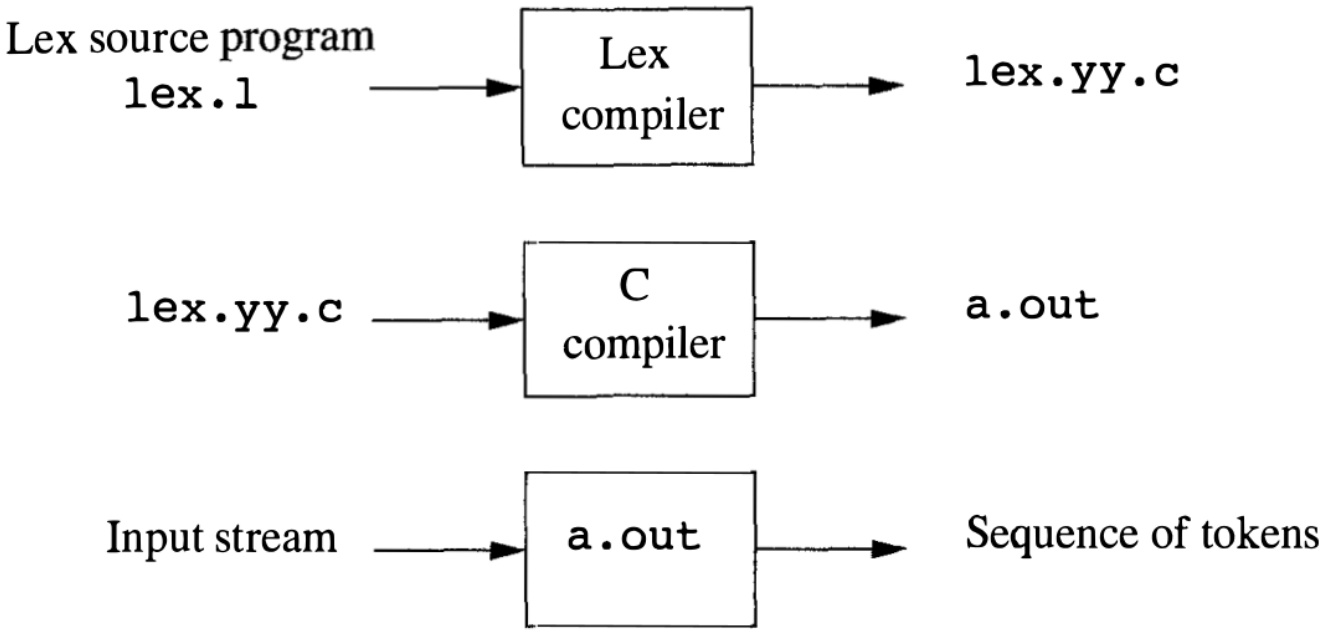
The Lexical-Analyzer Generator `Lex`

- Introduce a tool called `Lex`, or in a more recent implementation `Flex`, that allows one to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens
- Input notation for the Lex tool is referred to as the Lex language and the tool itself is the Lex compiler
- Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called `lex.yy.c`, that simulates this transition diagram

Use of Lex (1)

Lex compiler transforms `lex.l` to a C program, in a file that is always named `lex.yy.c`

The C-compiler output is a working lexical analyzer that can take a stream of input characters and produce a stream of tokens



Use of `Lex` (2)

- The normal use of the compiled C program is as a subroutine of the parser
- It is a C function that returns an integer, which is a code for one of the possible token names
- The attribute value, whether it be another numeric code, a pointer to the symbol table, or nothing, is placed in a global variable `yylval`, which is shared between the lexical analyzer and parser, thereby making it simple to return both the name and an attribute value of a token

Structure of Lex Programs

A Lex program has the following form:

```
declarations
%%
translation rules
%%
auxiliary functions
```

The translation rules each have the form:

```
Pattern { Action }
```

Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C.
The third section holds whatever additional functions are used in the actions.

How Does it Work?

- Called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of input that matches one of the patterns P_i
- It then executes the associated action A_i
- Typically, A_i will return to the parser, but if it does not (e.g., because P_i describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser
- The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable `yylval` to pass additional information about the lexeme found, if needed

Lex Regular Expressions

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	`a`
\c	character c literally	`\*`
`"`s`"`	string s literally	`"**"`
`.`	any character but newline	`a.*b`
`^`	beginning of a line	`^abc`
`$`	end of a line	`abc$`
[s]	any one of the characters in string s	`[abc]`
[^s]	any one character not in string s	`[^abc]`
r*	zero or more strings matching r	`a*`
r+	one or more strings matching r	`a+`
r?	zero or one r	`a?`
$r\{m,n\}$	between m and n occurrences of r	`a{1,5}`
$r_1 r_2$	an r_1 followed by an r_2	`ab`
r_1 | r_2	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`

Simple Example

```
%{
#include <stdio.h>
%}
%option noyywrap /* Read only one input file */
%%
/* [0-9]+ matches a string of one or more digits */
[0-9]+  {
/* yytext is a string containing the matched text. */
                printf("Saw an integer: %s\n", yytext);
            }
.|\n    {   /* Ignore all other characters. */   }
%%
int main(void) {
    /* Call the lexer, then quit. */
    yylex();
    return 0;
}
```

Another Example (1)

```
%{
  / * definitions of manifest constants
    LT , LE , EQ , NE , GT , GE ,
    IF , THEN , ELSE , ID , NUMBER , RELOP */
%}
/ * regular definitions */
delim    [ \t\n]
ws       {delim}+
letter   [A-Za-z]
digit    [0-9]
id       {letter}({letter}|{digit})*
number   {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
```

Another Example (2)

```
1  %%
2  {ws}        { /* no action and no return */ }
3  if          { return(IF); }
4  then        { return(THEN); }
5  else        { return(ELSE); }
6  {id}        { yylval=(int)installID(); return(ID); }
7  {number}    { yylval=(int)installNum(); return(NUMBER);}
8  "<"         { yylval=LT; return(RELOP); }
9  "<="        { yylval=LE; return(RELOP); }
10 "="         { yylval=EQ; return(RELOP); }
11 "<>"        { yylval=NE; return(RELOP); }
12 ">"         { yylval=GT; return(RELOP); }
13 ">="        { yylval=GE; return(RELOP); }
```

Another Example (3)

```
%%
int installID() {/* function to install the
                    lexeme, whose first
                    character is pointed to
                    by yytext, and whose
                    length is yyleng, into
                    the symbol table and return
                    a pointer thereto */
}
int installNum() {/* similar to installID, but
                     puts numerical constants
                     into a separate table */
}
```

Example Explanation (1)

- If we find whitespace (i.e., `ws`), we do not return to the parser, but look for another lexeme
- Should we see the two letters `if` on the input, and they are not followed by another letter or digit, then the lexical analyzer consumes these two letters from the input and returns the token name `IF`
- The fifth token has the pattern defined by `id`, although keywords like `if` match this pattern as well as an earlier pattern, Lex chooses whichever pattern is listed first in situations where the longest matching prefix matches two or more patterns

Example Explanation (2)

The action taken when `id` is matched is threefold:

1. Function `installID()` is called to place the lexeme found in the symbol table
2. This function returns a pointer to the symbol table, which is placed in global variable `yylval`, where it can be used by the parser or a later component of the compiler. Note that `installID()` has available to it two variables that are set automatically by the lexical analyzer that `Lex` generates:
   - 2.1 `yytext` is a pointer to the beginning of the lexeme
   - 2.2 `yyleng` is the length of the lexeme found
3. The token name `ID` is returned to the parser

Conflict Resolution in Lex

There are two rules that Lex uses to decide on the proper lexeme to select, when several prefixes of the input match one or more patterns:

1. Always prefer a longer prefix to a shorter prefix
2. If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Lex program

Examples:

1. Treat <= as a single lexeme, rather than selecting < as one lexeme and = as the next lexeme
2. For `then` the token `THEN` is returned, rather than `ID`

The Lookahead Operator

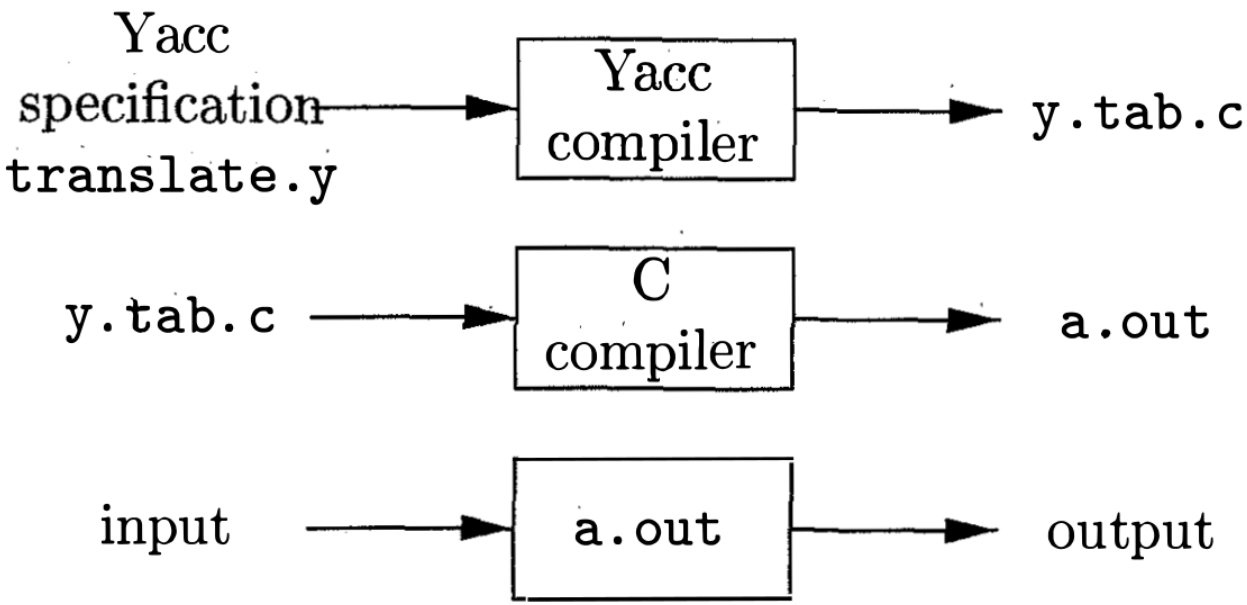
- `Lex` automatically reads one character ahead of the last character that forms the selected lexeme, and then retracts the input so only the lexeme itself is consumed from the input
- Sometimes, we want a certain pattern to be matched to the input only when it is followed by a certain other characters
- We may use the slash in a pattern to indicate the end of the part of the pattern that matches the lexeme
- Example: In some language like Fortran keywords are not reserved, it that case you should use more sophisticated regular expression for detecting `if` instructions

```
IF / (.* ) letter
```

The Parser Generator `Yacc`

- A parser generator can be used to facilitate the construction of the front end of a compiler
- Yacc stands for "yet another compiler-compiler"
- A file `translate.y`, containing a Yacc specification of the translator is prepared
- `yacc translate.y` transforms the file `translate.y` into a C program called `y.tab.c`
- The program `y.tab.c` is a representation of an LALR parser written in C
- Compiling `gcc y.tab.c -ly` we obtain the desired object program `a.out` that performs the translation

Use of Yacc


Yacc
specification
translate.y
Yacc
compiler
y.tab.c
y.tab.c
C
compiler
a.out
input
a.out
output

Structure of Yacc Programs

A Yacc source program has three parts:

```
1 declarations
2 %%
3 translation rules
4 %%
5 supporting C routines
```

Example grammar for a simple desk calculator:

(1) E → E + T | T

(2) T → T * F | F

(3) F → (E) | digit

Calculator Example (1)

```
1  %{
2  #include <ctype.h>
3  %}
4  %token DIGIT
5  %%
6  line   : expr '\n' { printf("%d\n", $1); }
7          ;
8  expr   : expr '+' term { $$ = $1 + $3; }
9          | term
10         ;
11 term   : term '*' factor { $$ = $1 * $3; }
12         | factor
13         ;
14 factor   : '(' expr ')' { $$ = $2; }
15         | DIGIT
16         ;
```

Calculator Example (2)

```
%%
yylex() {
  int c;
  c = getchar();
  if (isdigit(c)) {
    yylval = c - '0';
    return DIGIT;
  }
  return c;
}
```

Example Explanation

- Declarations consisting of C declarations and token declarations, both are optional
- If `Lex` is used to create the lexical analyzer that passes token to the `Yacc` parser, then these token declarations are also made available to the analyzer generated by `Lex`
- Grammar productions have associated semantic actions
- `$$` refers to the attribute value associated with the nonterminal of the head, while `$i` refers to the value associated with the $\texttt{i}^{\texttt{th}}$ grammar symbol
- A lexical analyzer by the name `yylex()` must be provided

Syntactical Details

- In the declarations portion, we can assign precedences and associativities to terminals
- Examples:
  - `%left '+', '-'`
  - `%right '^'`
  - `%prec UMINUS`

Scanner and Parser Examples

- Simple scanner
  - `simple_scanner.l`
- Simple parser
  - `simple_parser.y`
- Scanner and parser for calculator
  - `Makefile`
  - `calculation-scanner.l`
  - `calculation-parser.y`
  - `tokens.h`
  - `tokens.c`
  - `calculator.c`

Final Exam Details

- Friday, 13^{th} of December, 2019, 12:30 - 14:30 in SCC Hall 4
- Tutorial on Sunday, 8^{th} of December, 2019, 17:00 - 19:00, West Hall 4
- Written exam
- Material from parts II, III, IV of the course
- Short theoretical questions possible as well