LECTURE 7

Lex and Intro to Parsing

LEX

Last lecture, we learned a little bit about how we can take our regular expressions (which *specify* our valid tokens) and create real programs that can *recognize* them.

We ended the lecture with a little introduction to Lex and a sample Lex file.

Today, we're going to create a Lex file together.

LEX

Let’s say I want to create a scanner which matches the following tokens:

- Integers – any whole number and its negation.
- Reals – in decimal format.
- Identifiers – any sequence of letters, digits, and underscores which start with a letter.
- Operators – any of +, -, *, /, =
- Whitespace

I’m going to print out the tokens that I match and count how many of each I encounter.

LEX

Let's review the format of a Lex file.

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

Definitions: Can include global C code as well as macro definitions for regular expressions.

Rules: Regular expressions of tokens to match and their corresponding actions.

User Subroutines: For our purposes, just contains an optional main function.

LEX

Let’s start by creating a Lex file which simply matches our integers. We defined integers as being any whole number and its negation (i.e. …-3, -2, -1, 0, 1, 2, 3 …). So what’s an appropriate regular expression to match this regular set?

LEX

Let’s start by creating a Lex file which simply matches our integers. We defined integers as being any whole number and its negation (i.e. …-3, -2, -1, 0, 1, 2, 3 …). So what’s an appropriate regular expression to match this regular set?

```
(-?[1-9][0-9]*)|0
```

‘-’ is optional

LEX

Let’s start by creating a Lex file which simply matches our integers. We defined integers as being any whole number and its negation (i.e. …-3, -2, -1, 0, 1, 2, 3 …). So what’s an appropriate regular expression to match this regular set?

```
(-?[1-9][0-9]*)|0
```

We can match zero or more numbers from the range 0-9, but it must be preceded by a number from the range 1-9.

LEX

Let's start by creating a Lex file which simply matches our integers. We defined integers as being any whole number and its negation (i.e. …-3, -2, -1, 0, 1, 2, 3 …). So what's an appropriate regular expression to match this regular set?

```
(-?[1-9][0-9]*)|0
```

Alternatively, just match zero.

LEX

simple.l:

```
%{
     int numints = 0;
%}
inttoken (-?[1-9][0-9]*)|0

%%
{inttoken}      {printf("Matched integer: %s\n", yytext); numints++;}
%%

int main(){
    yylex();
    printf("Number of integer tokens: %d \n", numints);
    return 0;
}
```

LEX

simple.l:

```
%{
	int numints = 0;
%}
inttoken (-?[1-9][0-9]*)|0

%%
{inttoken}      {printf("Matched integer: %s\n", yytext); numints++;}
%%

int main(){
    yylex();
    printf("Number of integer tokens: %d \n", numints);
    return 0;
}
```

Macro created for integers

Lex creates a pointer to the matched string, *yytext*. We'll use it to print what we found.

LEX

Let's give this a little test before we move on.

What's with the output?

```
carnahan@diablo>./a.out < test_nums.txt
Matched integer: 12
 Matched integer: -356776434678
 Matched integer: 1
 Matched integer: 4487654

.Matched integer: 456

Matched integer: -4
.Matched integer: 567

Matched integer: 35677654
.Matched integer: 3

Matched integer: -45

Matched integer: 4862

Number of integer tokens: 11
```

test_nums.txt

```
12 -356776434678 1 4487654
.456
-4.567
35677654.3
-45
4862
```

LEX

Now, let's take care of the reals (e.g. 6.7, -3.0, -.54). How about this regular expression?

```
(-|(-?[1-9][0-9]*)|0))?“.”[0-9]+
```

We already defined inttoken to be `(-?[1-9][0-9]*)|0`, so we can also do this:

```
(-|{inttoken})?“.”[0-9]+
```

LEX

Now, let’s take care of the reals (e.g. 6.7, -3.0, -.54). How about this regular expression?

```
(-|(-?[1-9][0-9]*)|0))?“.”[0-9]+
```

We already defined inttoken to be `(-?[1-9][0-9]*)|0`, so we can also do this:

```
(-|{inttoken})?“.”[0-9]+
```

- We either allow inttokens before the decimal, a single negative sign, or nothing.
- Followed by the decimal itself.
- Followed by at least one digit in the range 0-9.
- What are our limitations? What do we not allow?

LEX

<u>simple.l:</u>

```
%{
     int numints = 0, numdoubles = 0;
%}
inttoken (-?[1-9][0-9]*)|0
%%
{inttoken}                 {printf("Matched integer: %s\n", yytext); numints++;}
(-|{inttoken})?“.”[0-9]+  {printf("Matched real: %s\n", yytext); numdoubles++;}
%%
int main(){
   yylex();
   printf("Number of integer tokens: %d \n", numints);
   printf("Number of real tokens: %d \n", numdoubles);
   return 0;
}
```

LEX

```
carnahan@diablo>./a.out < test_nums.txt
Matched integer: 12
 Matched integer: -356776434678
 Matched integer: 1
 Matched integer: 4487654

Matched real: .456

Matched real: -4.567

Matched real: 35677654.3

Matched integer: -45

Matched integer: 4862

Number of integer tokens: 6
Number of real tokens: 3
```

test_nums.txt

```
12 -356776434678 1 4487654
.456
-4.567
35677654.3
-45
4862
```

LEX

Now, we'll do the next three all at once (check out simple.l on the website):

- Identifiers: any sequence of letters, digits, and underscores which start with a letter.
  `[a-zA-Z][a-zA-Z_0-9]*`
- Operators: `[+\-/*=]` (Note: we have to escape '-'…it has special meaning in the brackets.)
- Whitespace: `[ \n\t]`

LEX

Okay, so let's try this with a new test file.

```
carnahan@diablo>./a.out < test_all.txt
Matched identifier: my_int1
Matched operator: =
Matched integer: 1
Matched identifier: my_int2
Matched operator: =
Matched integer: 3
Matched operator: +
Matched identifier: my_int1
Matched identifier: Myreal1
Matched operator: =
Matched real: -3.4
Matched operator: -
Matched real: 2.0
Matched identifier: Myreal2
Matched operator: =
Matched identifier: Myreal1
Matched operator: /
Matched real: -2.5
Number of integer tokens: 2
Number of real tokens: 3
Number of identifiers: 6
Number of operators: 7
Number of whitespace characters: 17
```

<u>test_all.txt</u>

```
my_int1 = 1
my_int2 = 3 + my_int1

Myreal1 = -3.4 - 2.0
Myreal2 = Myreal1/-2.5
```

LEX

<table>
<tr><th>Metacharacter</th><th>Matches</th></tr>
<tr><td>.</td><td>any character except newline</td></tr>
<tr><td>\n</td><td>newline</td></tr>
<tr><td>*</td><td>zero or more copies of the preceding expression</td></tr>
<tr><td>+</td><td>one or more copies of the preceding expression</td></tr>
<tr><td>?</td><td>zero or one copy of the preceding expression</td></tr>
<tr><td>^</td><td>beginning of line</td></tr>
<tr><td>$</td><td>end of line</td></tr>
<tr><td>a|b</td><td>a or b</td></tr>
<tr><td>(ab)+</td><td>one or more copies of ab (grouping)</td></tr>
<tr><td>"a+b"</td><td>literal "a+b" (C escapes still work)</td></tr>
<tr><td>[]</td><td>character class</td></tr>
</table>

<table>
<tr><th>Expression</th><th>Matches</th></tr>
<tr><td>abc</td><td>abc</td></tr>
<tr><td>abc*</td><td>ab abc abcc abccc ...</td></tr>
<tr><td>abc+</td><td>abc, abcc, abccc, abcccc, ...</td></tr>
<tr><td>a(bc)+</td><td>abc, abcbc, abcbcbc, ...</td></tr>
<tr><td>a(bc)?</td><td>a, abc</td></tr>
<tr><td>[abc]</td><td>one of: a, b, c</td></tr>
<tr><td>[a-z]</td><td>any letter, a through z</td></tr>
<tr><td>[a\-z]</td><td>one of: a, -, z</td></tr>
<tr><td>[-az]</td><td>one of: - a z</td></tr>
<tr><td>[A-Za-z0-9]+</td><td>one or more alphanumeric characters</td></tr>
<tr><td>[\t\n]+</td><td>whitespace</td></tr>
<tr><td>[^ab]</td><td>anything except: a, b</td></tr>
<tr><td>[a^b]</td><td>a, ^, b</td></tr>
<tr><td>[a|b]</td><td>a, |, b</td></tr>
<tr><td>a|b</td><td>a, b</td></tr>
</table>

Source: http://epaperpress.com/lexandyacc/prl.html

LEX

Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN condition`	switch start condition
`ECHO`	write matched string

Source: http://epaperpress.com/lexandyacc/prl.html

LEX

There are some excellent Lex references out there! Go read about it. We will do a little project on Lex ☺

Lex is available on all linprog machines, so you can start playing with it! Just create a simple .l file and try to make it more and more detailed.

Remember to compile with the –lfl flag on linprog (e.g. `gcc lex.yy.c -lfl`).

PARSING

So now that we know the ins-and-outs of how compilers determine the valid tokens of a program, we can talk about how they determine valid *patterns* of tokens.

A *parser* is the part of the compiler which is responsible for serving as the *recognizer* of the programming language, in the same way that the scanner is the recognizer for the tokens.

PARSING

Even though we typically picture parsing as the stage that comes after scanning, this isn't really the case.

In a real scenario, the parser will generally call the scanner as needed to obtain input tokens. It creates a parse tree out of the tokens and passes the tree to the later stages of the compiler.

This style of compilation is known as *syntax-directed translation.*

PARSING

Let's review context-free grammars. Each context-free grammar has four components:

- A finite set of tokens (terminal symbols), denoted T.
- A finite set of non-terminals, denoted N.
- A finite set of productions $N \rightarrow (T \mid N)^*$
- A special nonterminal called the start symbol.

The idea is similar to regular expressions, except that we can create recursive definitions. Therefore, context-free grammars are more expressive.

PARSING

Given a context-free grammar, parsing is the process of determining whether the start symbol can derive the program.

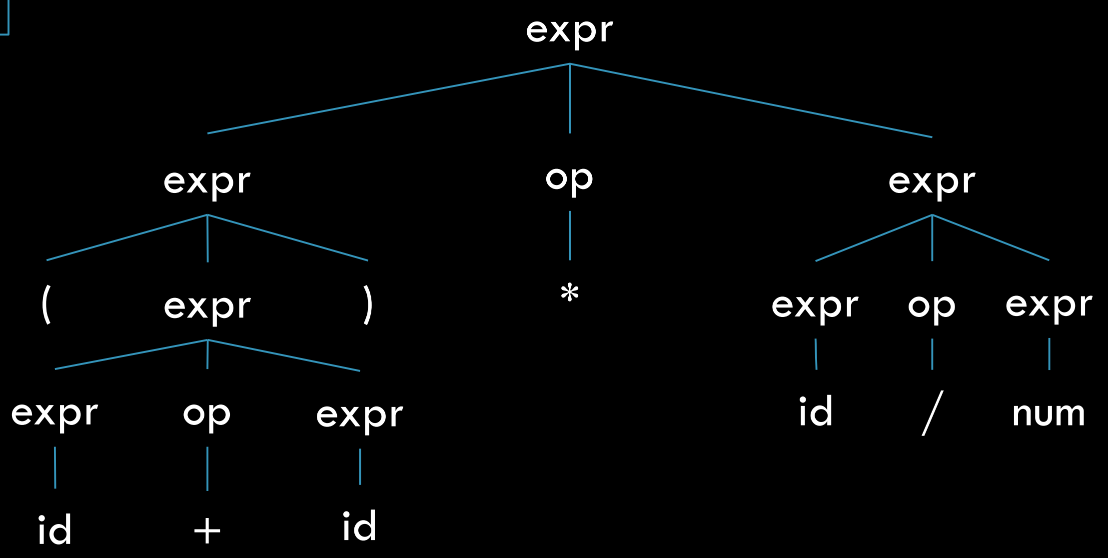
- If successful, the program is a valid program.
- If failed, the program is invalid.

PARSING

We can derive parse trees from context-free grammars given some input string.

expr → id | number | - expr | (expr) | expr op expr
op → + | - | * | /

expr ⟶ expr op expr
⟶ expr op expr op expr
⟶ expr op expr op number
⟶ expr op expr / number
⟶ expr op id / number
⟶ expr * id / number
⟶ (expr) * id / number
⟶ (expr op expr) * id / number
⟶ (expr op id) * id / number
⟶ (expr + id) * id / number
⟶ (id + id) * id / number



PARSING

There are two classes of grammars for which linear-time parsers can be constructed:

- LL – “Left-to-right, leftmost derivation”
  - Input is read from left to right.
  - Derivation is left-most, meaning the left-most nonterminal is replaced at every step.
  - Can be hand-written or generated by a parser generator.
  - Use “top-down” or “predictive” parsers.
- LR – “Left-to-right, rightmost derivation”
  - Input is read from left to right.
  - Derivation is right-most, meaning the right-most nonterminal is replaced at every step.
  - More common, larger class of grammars.
  - Almost always automatically generated.
  - Use “bottom-up” parsers.

PARSING

- LL parsers are *Top-Down ("Predictive") parsers.*
  - Construct the parse tree from the root down, predicting the production used based on some look-ahead.
  - LL parsers are easier to understand, but the grammars are less intuitive.
- LR parsers are *Bottom-Up parsers.*
  - Construct the parse tree from the leaves up, joining nodes together under single parents.
  - LR parsers can parse more intuitive grammars, but are harder to create.

When you see a () suffix with a number (e.g. LL(1)), that indicates how many tokens of look-ahead the parser requires.

We will be focusing on LL parsers in this class.

RECURSIVE DESCENT PARSING

Recursive descent parsers are an LL parser in which every non-terminal in the grammar corresponds to a subroutine of the parser.

- Typically hand-written but can be automatically generated.
- Used when a language is relatively simple.

RECURSIVE DESCENT PARSER

Let's look at an example. Take the following context-free grammar. It has certain properties (notably, the absence of left recursion) that make it a good candidate to be parsed by a recursive descent parser.

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

Note: a grammar is *left-recursive* if nonterminal A can derive to a sentential form A → Aw where w is a string of terminals and nonterminals. In other words, the nonterminal appears on the left-most side of the replacement.

LL grammars cannot be left-recursive!

RECURSIVE DESCENT PARSER

Some strings we can derive from this grammar include:

- (1 * 2)
- (1 * 2) + 3 + 4
- 1 + 2 + 3 * 4
- etc!

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

RECURSIVE DESCENT PARSER

In order to create a parser for this grammar, all we have to do is create appropriate subroutines for each nonterminal. Let's start with program.

Because program is our starting nonterminal, it's always the first function called. Now, inside of program, let's think about what we want to do!

We'll probably want to call the expr() function because it is the only production for program. But in which cases do we make this call?

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

RECURSIVE DESCENT PARSER

In order to create a parser for this grammar, all we have to do is create appropriate subroutines for each nonterminal. Let's start with program.

```
procedure program
        case input of:
                '(', int:       expr()  match('$')
                else error
```

program → *expr*

expr → *term expr_tail*

expr_tail → + *term expr_tail* | ϵ

term → *factor term_tail*

term_tail → * *factor term_tail* | ϵ

factor → (*expr*) | int

Note: '$' is a symbol meaning end-of-input.
Typically, this would be the EOF character.
It is the last thing we should "consume" to know we're done parsing.

We use match() calls to consume terminal tokens.

RECURSIVE DESCENT PARSER

Now let's look at expr.

```
procedure expr
    case input of:
        '(', int:  term() expr_tail()
        else       error
```

program → *expr*

expr → *term expr_tail*

expr_tail → + *term expr_tail* | ϵ

term → *factor term_tail*

term_tail → * *factor term_tail* | ϵ

factor → (*expr*) | int

RECURSIVE DESCENT PARSER

Now let's look at term.

```
procedure term
    case input of:
        '(', int:    factor() term_tail()
        else         error
```

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

RECURSIVE DESCENT PARSER

Now let's look at factor.

```
procedure factor
    case input of:
        '(':   match('(')  expr()  match(')')
        int:   match(int)
        else  error
```

program → *expr*

expr → *term expr_tail*

expr_tail → + *term expr_tail* | ϵ

term → *factor term_tail*

term_tail → * *factor term_tail* | ϵ

factor → (*expr*) | int

RECURSIVE DESCENT PARSER

Now let's look at expr_tail

```
procedure expr_tail
        case input of:
                '+':   match('+')  term() expr_tail()
                '$', ')':      skip
                else  error
```

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

RECURSIVE DESCENT PARSER

Now let's look at expr_tail

```
procedure expr_tail
        case input of:
                '+':   match('+')  term() expr_tail()
                '$', ')':      skip
                else  error
```

program → *expr*

expr → *term expr_tail*

expr_tail → + *term expr_tail* | ϵ

term → *factor term_tail*

term_tail → * *factor term_tail* | ϵ

factor → (*expr*) | int

This is where it gets a little tricky – notice, we check for an input of '$' or ')'. This is how we handle the case where expr_tail is the empty string. The only thing that could follow expr_tail in that case is '$' or ')'.

RECURSIVE DESCENT PARSER

Now let’s look at term_tail

```
procedure term_tail
        case input of:
                '*':    match('*')  factor()  term_tail()
                '+', '$‘, ‘)’:        skip
                else  error
```

program → *expr*
expr → *term expr_tail*
expr_tail → + *term expr_tail* | ϵ
term → *factor term_tail*
term_tail → * *factor term_tail* | ϵ
factor → (*expr*) | int

Again – notice that we check for ‘+’, ‘)’ and ‘$’. These are the only possible valid tokens that could follow term_tail if it were the empty string.

RECURSIVE DESCENT PARSER

Putting all of these subroutines together would give us a nice little recursive descent parser for our grammar. But this code only verifies that the program is syntactically correct. We know that parsers must create a parse tree for the next step in the compilation process.

Basically, we can build in the construction of a parse tree by creating and linking new nodes as we encounter the terminal or non-terminal symbols. But nodes created for non-terminal symbols must be expanded further.

RECURSIVE DESCENT PARSER

Some recursive descent parsers require backtracking.

The grammar we used was an LL(1) grammar – it requires a look-ahead of only one character. This allowed us to create a *predictive* parser, which does not require backtracking. Any LL(k) grammar can be recognized by a predictive parser.

NEXT LECTURE

More LL parsing.