[Note: These are the notes I use for the lectures. To use these as lecture notes, you need to ignore some irrelevant parts. They should work as a summary of the lecture topics, though.]

1 Intro, lecture 3

Note: Some stuff from these notes actually happened during lecture 4.

2 Repetition/Summary

Grammar, derivations.

Give intuitive argument for how precedence is encoded in grammars.

(Formal) Language L. Classes of languages:
(1) ones described by some FSM (DFA, NFA)
(2) ones described by context-free grammars
(3) other languages

For some fixed language L, many c-f grammars define L. The grammars may have different (interesting) properties.

When we speak of "a grammar", it refers to a specific description of some language. A language have various properties, and a grammar have various properties.

3 Today

Parsing algorithms and grammar classes

Parse trees: concrete and abstract

Parse trees for non-exps like stmt sequences

4 Concrete/Abstract syntax trees

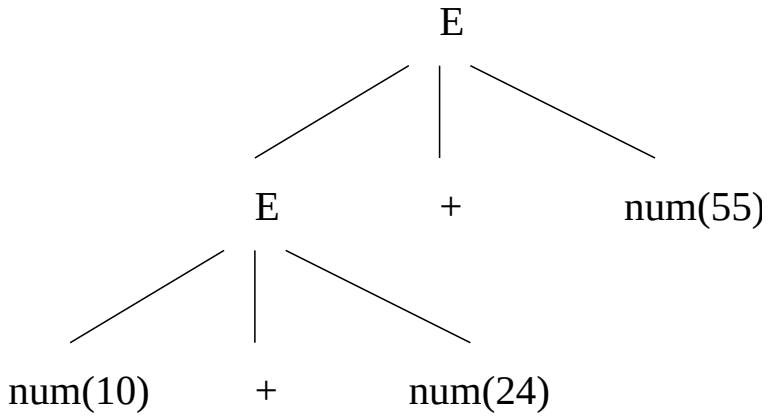
A *concrete parse tree* has a internal node for each production invoked, and one leaf for each token.

This parse tree will thus have ";" and strange extra non-terminals such as "T'".

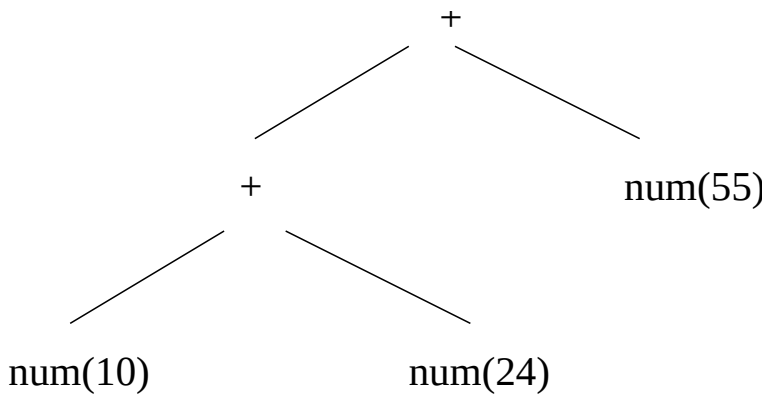
An abstract parse tree moves closer to the semantics of the language; semicolon The grammar structure is now no longer evident.

Example: $E \rightarrow E + E; E \rightarrow num$
With 10 + 24 + 55 (left assoc)

Concrete:



Abtract:



5 Parsing algorithms: Predictive Parsing

5.1 Example 1

[Write this grammar first with silly terminal symbols, as for meaning. Reason about implicit semantic interpretations.]

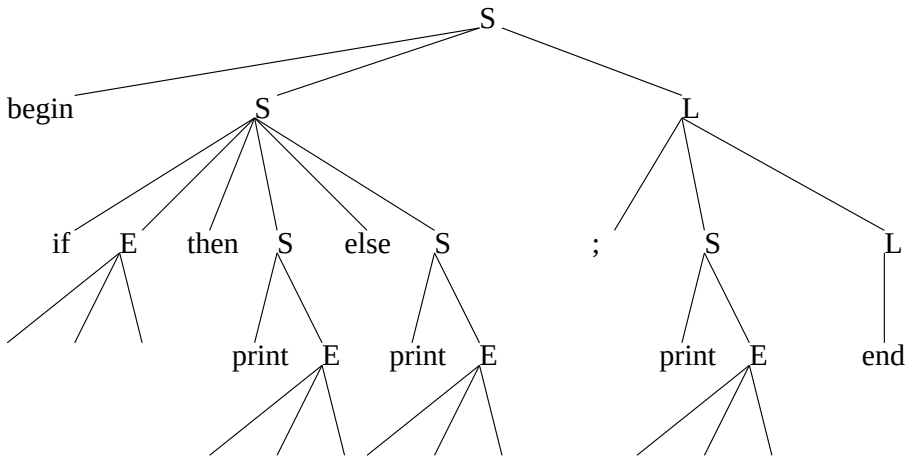
Consider the grammar:

$$
\begin{array}{lcl}
S & \rightarrow & \text{if } E \text{ then } S \text{ else } S \\
S & \rightarrow & \text{begin } S\ L \\
S & \rightarrow & \text{print } E \\
L & \rightarrow & \text{end} \\
L & \rightarrow & ;\ S\ L \\
E & \rightarrow & \text{num} = \text{num}
\end{array}
$$

Let the terminal symbol num = [0-9][0-9]*

What language does S define in grammar define, intuitively? (I.e., we use S as start symbol.)

Consider this string: begin if 10=11 then print 20=21 else print 30=31; print 0=0 end



Which of the following strings are in the language of S?

ϵ (empty string)	no
␣ (space)	no

[I have not defined the lexicals very well, so this might be hard to answer...]

print 12 = 34	ok
print 12 = -34	no

if 0 = 0 then
begin print 1=1; print 1=2;
end else print 2=3
	no (extra semicolon)

end	no (part of language def by L, though)
12 = 34	no (part of language def by E, though)

(Is the grammar unambiguous, i.e. "can more than two parse trees be constructed from some string belonging to the language"?

Consider

```
if 0 = 1 then
  if 2 = 3 then
  else S
else S
```

To where does the first `else` belong, the inner `if` or the outer `if`?

It is unambiguous, but only because we require `else` to each `if`.)

Show slide 1.

5.2 Example 2

Consider now the grammar for arith exprs from last lecture:

$E \to T$	$T \to F$	$F \to \textbf{integer}$
$E \to E + T$	$T \to T * F$	$F \to (E)$
$E \to E - T$	$T \to T/F$	

A program in the same style will perhaps look like this:

Show slide 2.

There are a few problems here...
(1) The case labels
(2) The recursion does not make progress, therefore infinite

Key conclusion: We need to know the set of next *terminal symbol(s)* for each production, and two productions may not have any common elements of their sets.

Define FIRST(P) for production P.

We must follow the "grammar recursion" to compute FIRST. Special handling is needed to nullable symbols.

5.3 Eliminating left recursion

Consider a potion of the previous grammar:

$E \to T$
$E \to E + T$

Any token in FIRST(T) will also be in FIRST($E + T$).

The problem is that E is a first rhs symbol in an E production. We say that we have *left recursion* in the grammar.

No grammar with left recursion can use a predictive parser.

Rewrite exp grammar to right recursive form:

$E \to TE'$

$E' \to +TE'$
$E' \to$

For full exp grammar w/o left recursion, see pg 52, top.

5.4 Left factoring

Consider the grammar:

$S \rightarrow$ if E then S else S
$S \rightarrow$ if E then S

Which production should our parser choose?
(1) We have just one token of look-ahead, which should always be **if** here.
(2) E might consist of any number of tokens, and so may the recursive S.

Solution: a technique known as *left factoring*.

$S \rightarrow$ if E then S X
$X \rightarrow$
$X \rightarrow$ else S

Reason about the ambiguousness of this grammar. Solution: Allow it, prefer 2nd X production.

5.5 LL parsing

LL stands for Left-to-right parse, leftmost derivation. LL(k) uses a fixed k number of look-ahead tokens.

It looks at the leftmost k tokens of the rhs of a production, before making a decision.

A predictive parser can handle LL(k) grammars.

The name "predictive" comes from that the parser needs to "predict" the production after a fixed number of tokens. The term can be confusing; no guessing takes place.

6 LR parsing

LR stands for Left-to-right parse, rightmost derivation.

LR(k) production decisions are made after having seen the *entire rhs of the production* plus k tokens.

Hand-wave description of the LR parser engine:

We have a (1) *stack* onto which we can push terminal and non-terminal symbols and (2) the *token input* (to which we can lookahead at the first k tokens).

Based on the contents of the stack and the lookahead, it performs the actions:

Shift:
Move the input token to the top of the stack.

Reduce:
(1) Choose a production $X \rightarrow ABC$
(2) pop C, pop B, pop A
(2) push X

6.1 Parser generators

Writing a recursive descent parser by hand is not terribly hard.

Most grammars are left recursive, will need some rewriting.

IMHO: Except for trivial grammars, one should use a parser generator.

Here is an example using yacc/bison. It is a calculator. [Omit exponentiation when presenting, add later.]

```
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation        */
```

```
%%
input:    /* empty string */
        | input line
;

line:     '\n'
        | exp '\n'  { printf (``\t%.10g\n'', $1); }
;

exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
;
```

- This is run through `bison` to generate a C file
- Bison allows grammars to be ambiguous, with declarators as a meta-language
- The order of declarators determine precedence (line number = precedence)

6.2 Error in the input string

Do not report first error, but recover from the error. Different strategies exist, including additional productions, which extends the grammar to include common mistakes.

7 Final words

We have talked about grammar properties, and about rewriting grammars. Depending on which parser generator you choose, you will need to do little or much grammar rewriting.