Announcements

- **Homework 1 (on web site) due Wednesday at the *start* of class**
- **Make sure your remus account still works (you'll need it for programming in scheme)**

Formal Languages - 3

- **Ambiguity in PLs**
  - **Problems with if-then-else constructs**
  - **Harder problems**
- **Tools for building compilers: lex and yacc**
- **Chomsky hierarchy for formal languages**
  - **Regular and context-free languages**
  - **Type 0 languages and Turing machines**
  - **Type 1: Context-sensitive languages**

Dangling Else

Here is a simplified grammar for Pascal:

<stmt> ::= <if-stmt> | <assgn> | ...

<if-stmt> ::= if <expr> then <stmt> |
if <expr> then <stmt> else <stmt>

<assgn> ::= <id> := <digit>

<expr> ::= <id> = 0

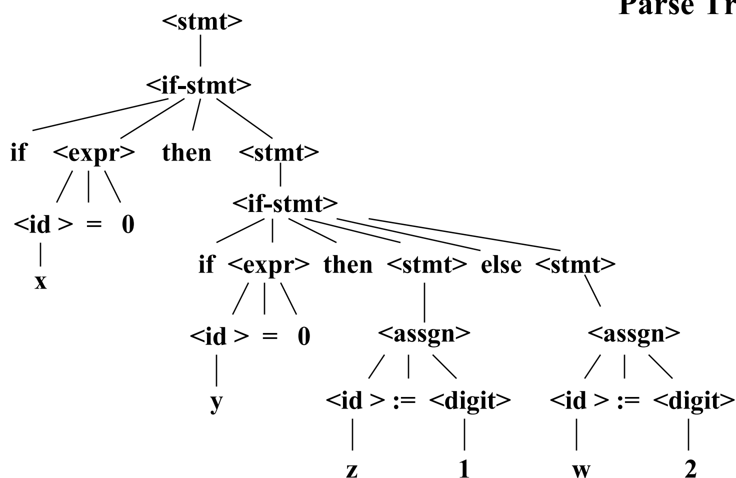
<id> ::= a | b | c | ... | x | y | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

How are compound "if" statements parsed using this grammar?

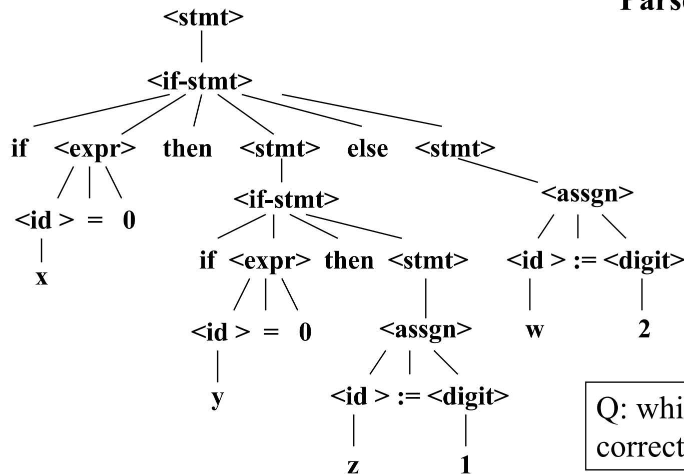
if x = 0 then if y = 0 then z := 1 else w := 2

Parse Tree 1



if x = 0 then if y = 0 then z := 1 else w := 2

Parse Tree 2



Q: which tree is correct?

5

How to Fix the Dangling Else?

- ***Algol60:*** **use block structure**

  if x = 0 then begin if y = 0 then z := 1 end else w := 2
- ***Algol68:*** **use statement begin/end markers**

  if x = 0 then if y = 0 then z := 1 fi else w := 2 fi
- ***Pascal:*** **change the grammar of "if" statement to disallow the second parse tree, i.e.,** ***always associate an "else" with the closest "if".***

6

How to Fix the Dangling Else?

Here is a revised grammar for Pascal:

<stmt> ::= <stmt1> | <stmt2>

<stmt1> ::= if <expr> then <stmt1> else <stmt1> |
<assgn> | ...

<stmt2> ::= if <expr> then <stmt> |
if <expr> then <stmt1> else <stmt2>

<assgn> ::= <id> := <digit>

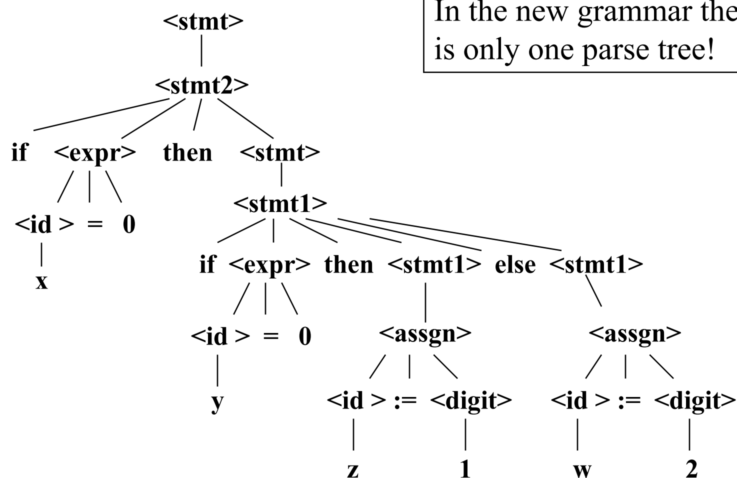
<expr> ::= <id> = 0

<id> ::= a | b | c | ... | x | y | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

if x = 0 then if y = 0 then z := 1 else w := 2

In the new grammar there is only one parse tree!

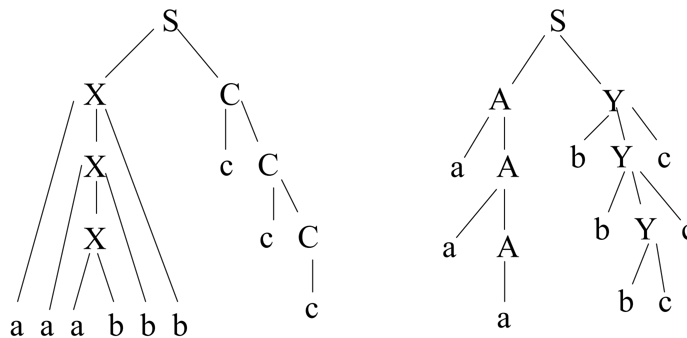


Inherent Ambiguity

- Sometimes we can remove an ambiguity from a grammar by restructuring the productions, but it is not always possible
  - An *inherently ambiguous* language does not possess an unambiguous grammar
  - Such a grammar is unsuitable as a PL!
  - E.g., L = { $a^i b^j c^k$ | i = j or j = k for i,j,k >=1} generated by

S ::= X C | A Y	X ::= a X b | ab	C ::= c | cC
	Y ::= b Y c | bc	A ::= a | aA

Parse Trees



two parse trees for $a^3 b^3 c^3$ in L

S ::= X C | A Y	X ::= a X b | ab	C ::= c | cC
	Y ::= b Y c | bc	A ::= a | aA

Tools for building parsers (in C): lex and yacc

- **lex**
  - implements a "tokenizer" given a specification of the lexemes (chunks of the input file) using regular expressions
  - code is given for how to respond to each kind of token
- **yacc**
  - implements a parser (from a variant of BNF)
  - code is given for each production of grammar

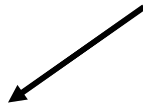
Example:
- arithmetic expressions using: + - * / ^ ( )
- from http://pltplp.net/lex-yacc/example.html.en

11

lex example

```
%{
#include "global.h"
#include "calc.h"
#include <stdlib.h>
%}
```

regular expressions for lexemes
- {} indicates a nonterminal
- ? indicates optional content



```
white       [ \t]+
digit       [0-9]
integer     {digit}+
exponent    [eE][+-]?{integer}
real        {integer}("."{integer})?{exponent}?

%%
```

12

lex example

(continued)

```
{white}        { }
{real}         {
                    yylval=atof(yytext);
                    return(NUMBER);
               }

"+"            return(PLUS);
"-"            return(MINUS);
"*"            return(TIMES);
"/"            return(DIVIDE);
"^"            return(POWER);
"("            return(LEFT_PAREN);
")"            return(RIGHT_PAREN);
"\n"           return(END);
```

code to respond to each lexeme as its encountered

yacc example: definitions

```
%{
#include "global.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
%}

%token NUMBER
%token PLUS   MINUS   TIMES   DIVIDE   POWER
%token LEFT_PAREN    RIGHT_PAREN
%token END

%left    PLUS   MINUS
%left    TIMES   DIVIDE
%left    NEG
%right   POWER

%start   Input
%%
```

token definitions (from lex)

associativity and precedence ↓ *higher*

start symbol for grammar

yacc example: grammar

(continued)

BNF	Code (per production)
Input:	
/* Empty */	
| Input Line ;	
Line:	
END |	display result
Expression END	{ printf("Result: %f\n",$1); } ;
Expression:	
NUMBER	{ $$=$1; }
| Expression PLUS Expression	{ $$=$1+$3; }
| Expression MINUS Expression	{ $$=$1-$3; } compute result
| Expression TIMES Expression	{ $$=$1*$3; }
| Expression DIVIDE Expression	{ $$=$1/$3; }
| MINUS Expression %prec NEG	{ $$=-$2; }
| Expression POWER Expression	{ $$=pow($1,$3); }
| LEFT_PAREN Expression RIGHT_PAREN	{ $$=$2; } ;

%%

yacc example: main program

(continued)

```
int yyerror(char *s)
{
    printf("%s\n",s);
}
int main(void)
{
    yyparse();
}
```

Example run

```
$ calc
1+2*3
Result : 7.000000
2.5*(3.2-4.1^2)
Result : -34.025000
```

Chomsky Hierarchy

- Describes categories of languages which correspond to more and more powerful automata
- 4 level hierarchy
  - We've studied the bottom two levels
    - regular and context-free languages

Chomsky Hierarchy

- Type 0: Arbitrary Languages
  - Recognized by: Turing Machines
- Type 1: Context Sensitive Languages
  - Recognized by: Linearly-bounded automata
- Type 2: Context Free Languages
  - Generated by: Context Free Grammars
  - Recognized by: Push-Down Automata
- Type 3: Regular Languages
  - Generated by: Regular Grammars
  - Described by: Regular Expressions
  - Recognized by: Finite State Automata

Type 3 (regular) languages

- *Recognizer*: finite state automaton
- Can be written with all right-linear or all left-linear rules
  - A → a | Aa (this is left-linear)
- Only can do simple recursive constructs
- No memory: can't count or match parentheses
- Not regular: $\{a^n b^n, n >= 1\}$

Type 2 (context-free) languages

- *Recognizer*: push down automaton
- Productions with 1 nonterminal on the left-hand side (i.e. A → ab | aAb)
- Limited memory: can see the input <u>twice</u> (the second time is only in reverse order)
  - can match parenthesis, etc…
- Can't check *context*
  - Cannot check that no variable is declared twice
  - Cannot do type-checking
- Not context-free: $\{a^n b^n c^n, n >= 1\}$

Context-free Languages

- **Cannot check match-up of arguments with parameters**
  - **consider Pascal's nested function definitions:**

    procedure p (x :integer, y:real)

    procedure q (w: integer)

    … P(50,1.2)…

    end q

    …Q(1)…

    end P
- **pattern seen is (parms p) (parms q) (args p) (args q)**
  - **corresponding language is $\{a^n\ b^m\ c^n\ d^m,\ m,n >= 1\}$ which is not context-free**

Turing Machines (lightly)

- **Abstract model of computation**
- **<finite set of states, alphabet, "blank" symbol, start state, final state, transition function>**
- **an infinite tape contains the input (and contains the output upon completion)**
- **the tape has a "read-head" (a current location)**
- **transition function:**

  <state, input tape symbol> → <state, tape output symbol, {L,R,N}>
  - **L,R,N moves the current tape location 1 square to the Left or Right, or No movement**

The halting problem

- Turing Machine Halting problem
  - Given a TM in an arbitrary configuration with nonblank symbols on its tape, will the TM eventually halt? → *unsolvable!*
  - There cannot exist an algorithm to solve this problem for an arbitrary choice of Turing machine on arbitrary input
    - although for a specific TM with specific input, there may be a solution
- Extends to real programs

Decidability

What do we mean by "unsolvable" ?

- a statement is decidable when there is an algorithm <u>both</u> for
  - reporting the answer when the statement is true
  - reporting the answer when the statement is false
- a statement is semi-decidable when there is an procedure for
  - reporting the answer when the statement is true
  - but that procedure <u>may not terminate</u> when the statement is false

This can be formalized using Turing machines

- Turing-Church thesis: every effective computation can be carried out on a Turing Machine (semi-decidable = computable)

Ambiguity

- **There is no algorithm which can examine two arbitrary context-free grammars and tell if they generate the same language**
  - **This is *undecidable***
  - **A procedure does exist that always terminates when the languages are different (the problem is *semi-decidable*)**
- **There is no algorithm which can examine an arbitrary context-free grammar and tell if it is ambiguous or not**
  - **This is *undecidable***
  - **A procedure does exist that always terminates when the language is ambiguous (the problem is *semi-decidable*)**

Type 1 (context-sensitive) language

- ***Recognizer*: linear bounded automaton (a Turing Machine with a finite tape proportional in size to its input length)**
- **Grammar rules can have more than 1 symbol on the left-hand side as long as |rhs| >= |lhs|**
- **Can do parameter-argument matching (in number) and simple type-checking**
  - **But it is very messy! So CFGs are used, and are augmented with computations beyond the grammar**
- **Examples:**

  $\{a^n b^m c^n d^m, m,n >= 1\}$

  $\{a^n b^n c^n, n >= 1\}$

Context-sensitive Example

1. T ::= S
2. S ::= a S B C (2a) | a B C (2b)
3. CB ::= BC --reverse B's and C's
4. aB ::= ab
5. bB ::= bb --expand B
6. bC ::= bc --expand C
7. cC ::= cc

Derive aabbcc:

T $\xrightarrow{1}$ S $\xrightarrow{2a}$ a S B C $\xrightarrow{2b}$ aaB C B C $\xrightarrow{3}$ aa B B C C $\xrightarrow{4}$ aabB C C $\xrightarrow{5}$ aabbCC $\xrightarrow{6}$ aabbcC $\xrightarrow{7}$ aabbcc

Type 0 (recursively enumerable) languages

- ***Recognizer*: Turing machine**
- **All languages that can be recognized by a procedure**
  - **i.e. languages describing computable problems**

Subclass of Type 0:

- **Recursive languages: languages recognized by an algorithm that always halts**
  - **i.e. languages describing decidable problems**