# Lecture 14

COP3402 FALL 2015 – DR. MATTHEW GERBER – 10/21/2015

FROM EURIPIDES MONTAGNE, FALL 2014

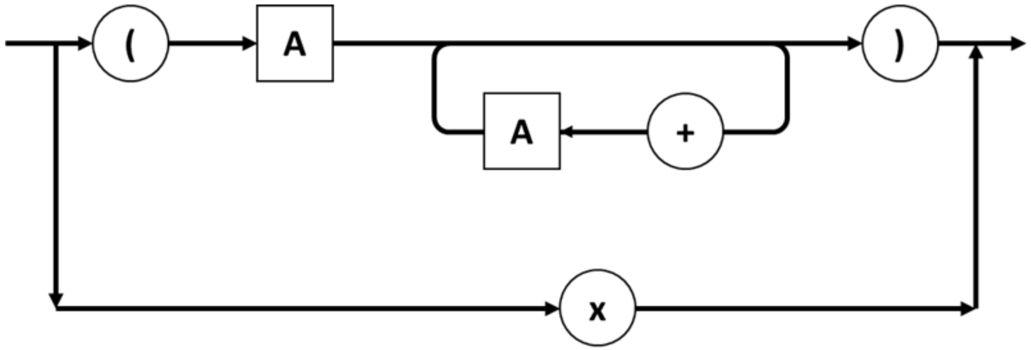
# Tonight

- From Syntax Graphs to Parsers
- Tiny-PL/0 Syntax
- Code Generation
- Generating Pseudocode

# From Syntax Graphs to Parsers

Building parsers from syntax graphs is a relatively systematic process via seven rules laid out by – again – Nikolas Wirth.  The first rule, we've already followed:

> **B1. Reduce the system of graphs to as few individual graphs as possible by appropriate substitution.**




We already did that when we made this graph out of its three component graphs at the end of Lecture 13.  As for the next rule…

> **B2. Translate each graph into a procedure declaration according to the subsequent rules B3 through B8.**

…it just says we need others.  So let's look at them.

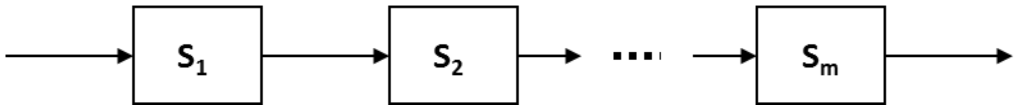
# Rule 3: Concatenation

**B3. A sequence of elements [as shown here] is translated into the compound statement**

```
{ T(S1); T(S2); …; T(Sn) }
```

**where T(S) denotes the translation of graph S.**

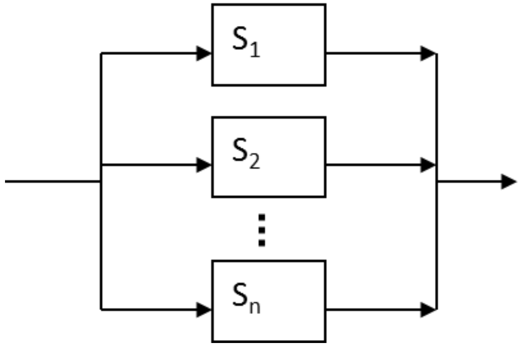
*In other words, where we have a sequence of elements and transitions in a line with no complications between them, we can safely string the code to handle them all together.*



# Rule 4: Choice

**B4.- A choice of elements [as shown here] is translated into a selective or conditional statement as follows:**

Note that **Li**, for every **i** 1 to **n**, is the set of symbols that imply a transition to **Si**. If **Li** has only one symbol then we can use simple equality.




```
switch (ch) {
  case ch in L1 : T(S1);
  case ch in L2 : T(S2);
  ...
  case ch in Ln : T(Sn);
  default: error
}
```

```
if ch in L1 { T(S1) } else
if ch in L2 { T(S2) } else
...
if ch in Ln { T(Sn) } else
error
```

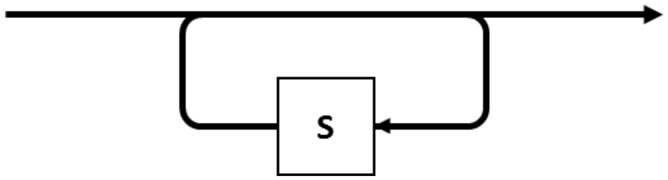
# Rule 5: Loops

**B5. A loop of the form [shown here] is translated into the statement:**

```
while ch in L do T(S)
```

**where T(S) is the translation of S according to rules B3 through B8.**

L is the set of symbols that implies continuing the loop. If L has only one symbol we can use simple equality.



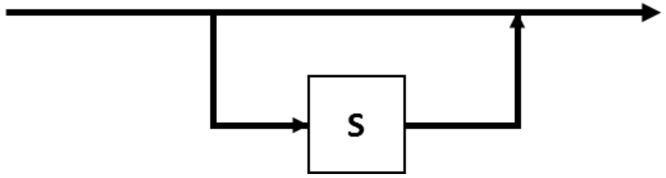
# Rule 6: Conditions

**B6. A loop of the form [shown here] is translated into the statement:**

```
if ch in L { T(S)}
```

**where T(S) is the translation of S according to rules B3 through B8.**

L is the set of symbols that implies a transition into S. If L has only one symbol we can use simple equality.



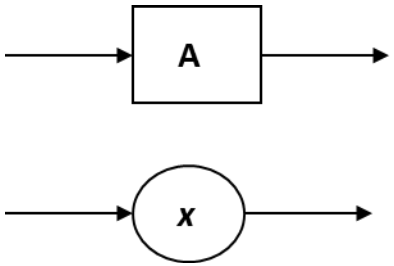
# Rules 7 and 8: Branching and Terminals

**B7. An element of the graph denoting another graph A is translated into the procedure call statement** A**.**

**B8. An element of the graph denoting a terminal symbol x is translated into the statement:**

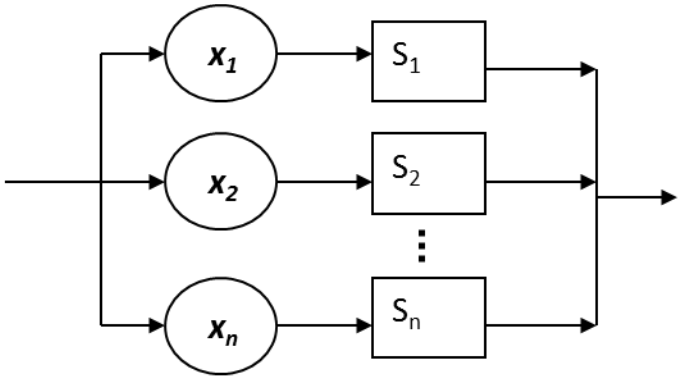
```
if (ch = x) { read(ch) }
else        { error }
```

**Where** `error` **is a routine called when an ill-formed construct is encountered.**

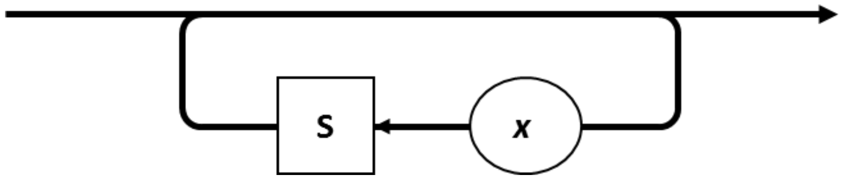


# Useful Variants

```
if ch == ‘x1’ { read(ch); T(S1) } else
if ch == ‘x2‘ { read(ch); T(S2) } else
. . .
if ch == ‘xn‘ { read(ch); T(Sn) } else
error
```




```
while (ch == ‘x’ ) {
  read(ch); T(S);
}
```



# An Example

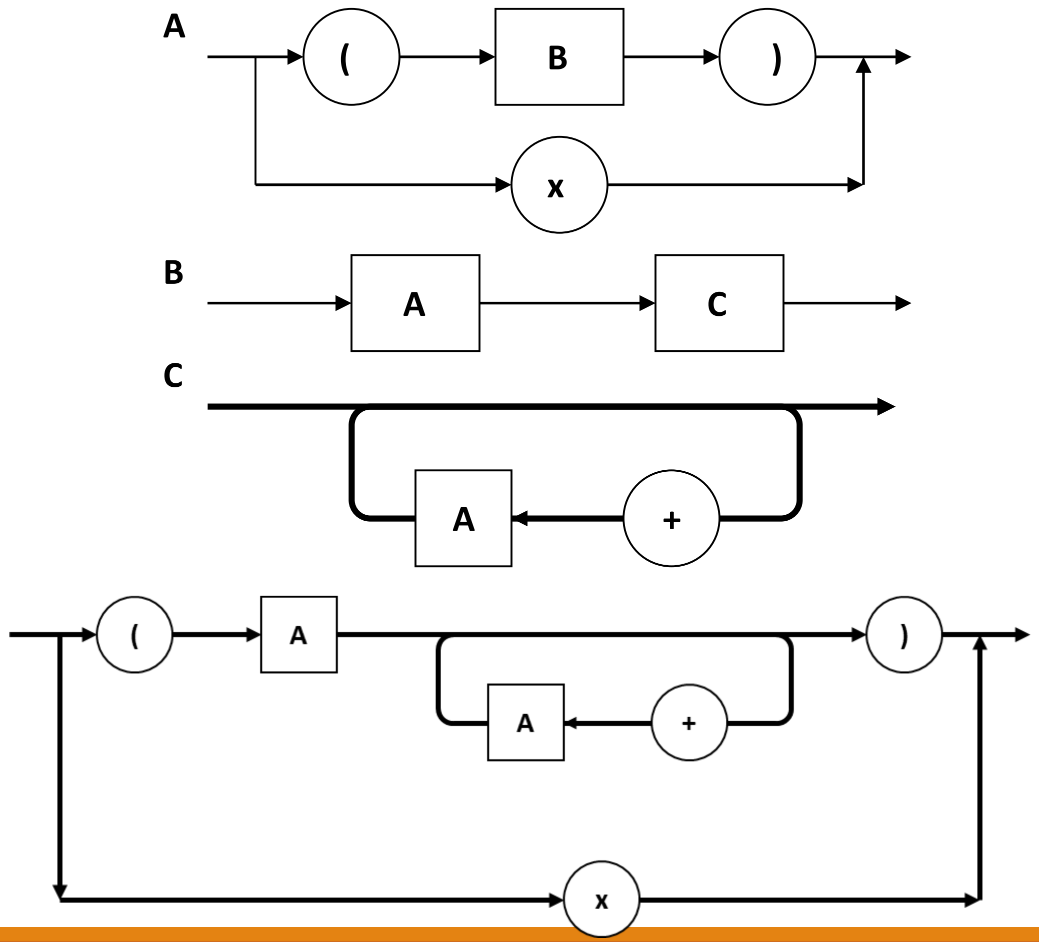
Again recall combining three other graphs to get a single syntax graph for the language:

A ::= “x” | “(“ B “)”

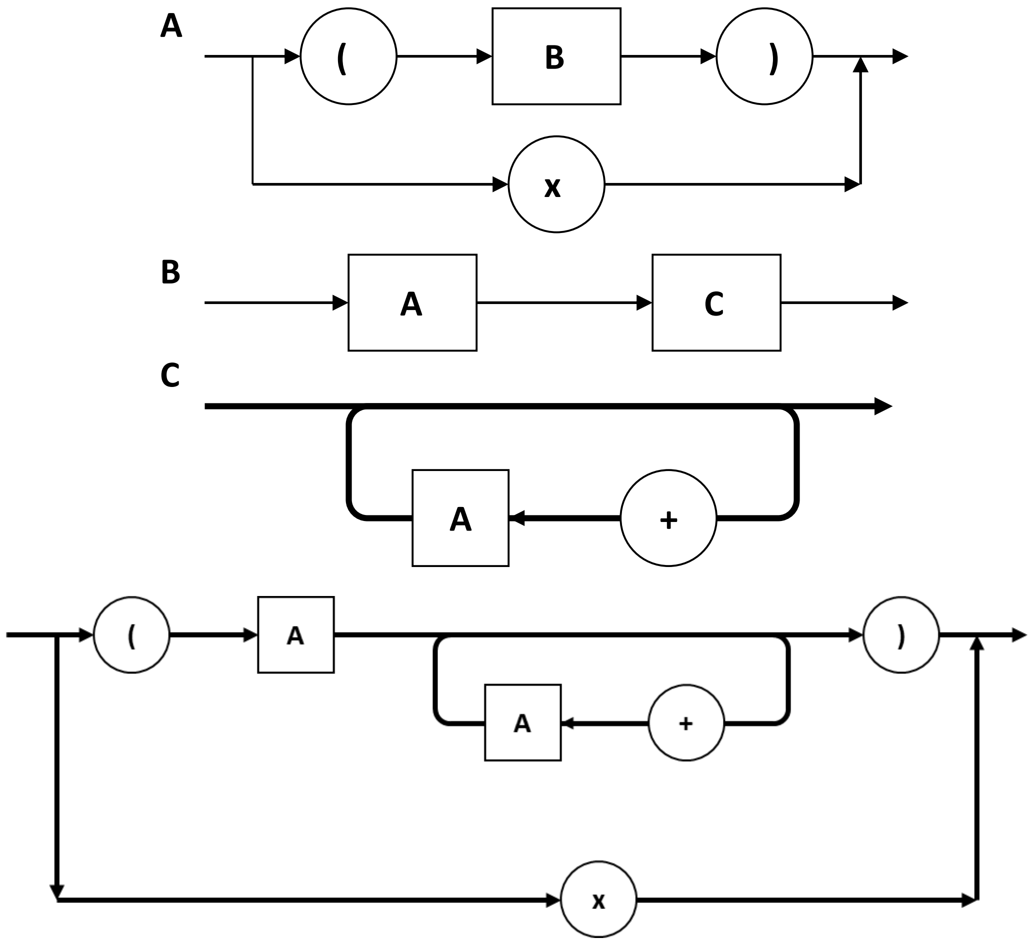
B ::= A C

C ::= { “+” A }

Now that we have Wirth’s rules, we can use them to turn that syntax graph into parser pseudocode.



# Pseudocode Example




```
var ch: char;
procedure A;
    begin
        if ch = 'x' then
            read(ch)
        else if ch = '(' then begin
            read(ch);
            A;
            while ch = '+' do begin
                read(ch);
                A
            end;
            if ch = ')' then read(ch)
            else error(err_number)
        end else error(err_number)
   end;
begin
    read(ch);
    A
end.
```

# Parsing PL/0

```
program             ::= block "."
block               ::= const-declaration var-declaration proc-declaration statement
const-declaration   ::= [ "const" ident "=" number {"," ident "=" number} ";"]
var-declaration     ::= [ "var" ident {"," ident} “;"]
proc-declaration    ::= {"procedure" ident ";" block ";" }
statement           ::= [ ident ":=" expression
                        | "call" ident
                        | "begin" statement { ";" statement } "end"
                        | "if" condition "then" statement [“else" statement]
                        | "while" condition "do" statement
                        | "read" ident
                        | "write" ident
                        | e ]
condition           ::= "odd" expression
                        | expression rel-op expression
rel-op              ::= "=" | “<>" | "<" | "<=" | ">" | ">="
expression          ::= [ "+"|"-"] term { ("+"|"-") term}
term                ::= factor {("*"|"/") factor}
factor              ::= ident | number | "(" expression ")"
number              ::= digit {digit}
ident               ::= letter {letter | digit}
digit               ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
letter              ::= "a" | "b" | … | "y" | "z" | "A" | "B" | ... | "Y" | "Z"
```

This is the EBNF for PL/0.

…so no problem writing parser pseudocode for this, right?

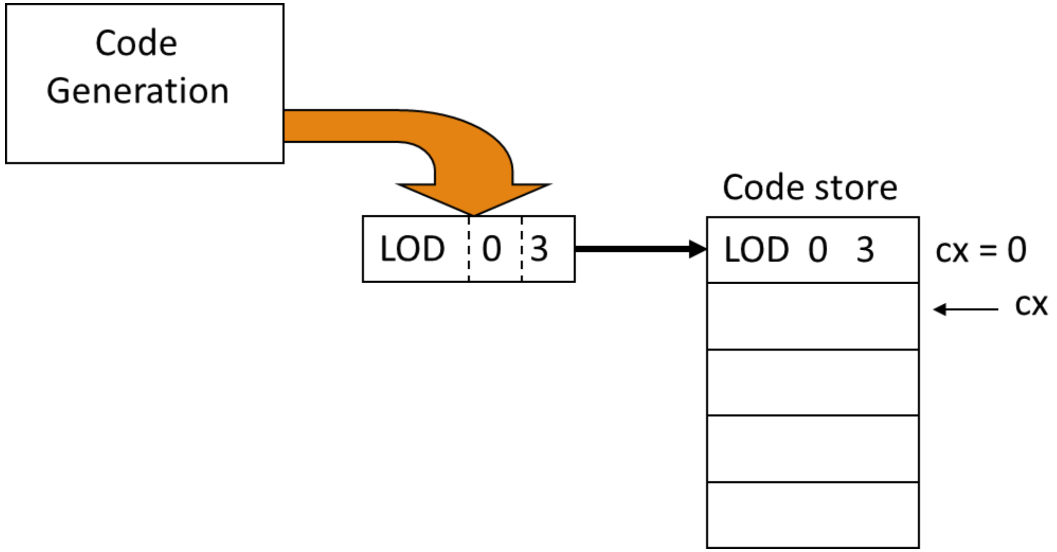
Just kidding.  We've helped you out quite a bit.

- Look in the reference document: There's pseudocode for all of PL/0 in there
- It's *intentionally* not in a format you can just copy and paste into a C program…
- …but it will get you started

…and once you can parse source code, you can get right to generating code for it.

# PL/0 Code Generation: Instruction by Instruction

Keep a code index (cx) that indicates where you are generating. Each time an instruction is generated, it is stored in the code segment and cx is incremented by one.




```
void emit(int op, int l, int m)
{
  if(cx > CODE_SIZE)
    error(25);
  else
  {
    code[cx].op = op; // opcode
    code[cx].l = l;   // lex level
    code[cx].m = m;   // modifier
    cx++;
  }
}
```

# Major Example 1: Expressions

```
void expression()
{
  int addop;
  if (token == plussym || token == minussym)
  {
    addop = token;
    getNextToken();
    term();
    if(addop == minussym)
      emit(OPR, 0, OPR_NEG); // negate
  }
  else
    term();
  while (token == plussym || token == minussym)
  {
    addop = token;
    getNextToken();
    term();
    if (addop == plussym)
      emit(OPR, 0, OPR_ADD); // addition
    else
      emit(OPR, 0, OPR_SUB); // subtraction
  }
}
```

expression ::= [ "**+**"|"**-**"] term { ("**+**"|"**-**") term}

# Major Example 2: Terms

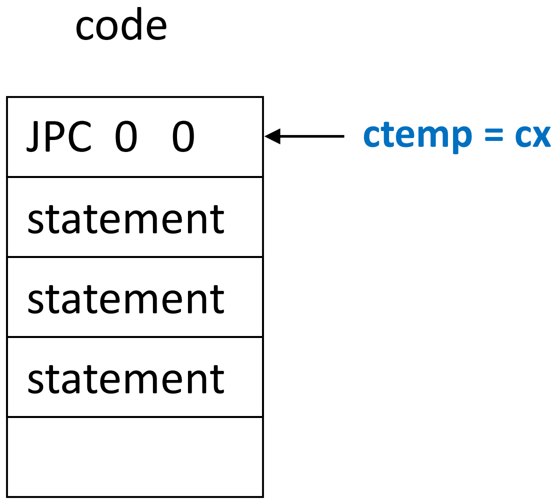
```
void term()
{
  int mulop;
  factor();
  while(token == multsym || token == slashsym)
  {
    mulop = token;
    getNextToken();
    factor();
    if(mulop == multsym)
      emit(OPR, 0, OPR_MUL); // multiplication
    else
      emit(OPR, 0, OPR_DIV); // division
  }
}
```

term ::= factor {("*"|"/") factor}

# Major Example 3: If

```
if(token == ifsym) {
  getNextToken();
  condition();
  if(token != thensym)
    error(16);  // then expected
  else
    getNextToken();
  ctemp = cx;
  emit(JPC, 0, 0);
  statement();
  code[ctemp].m = cx;
}
```

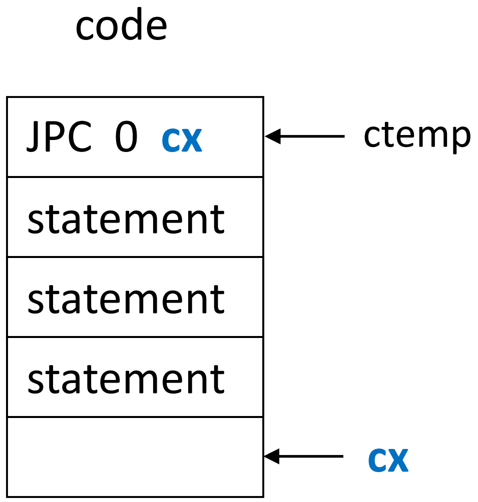
"**if**" condition "**then**" statement



# Major Example 3: If

```
if(token == ifsym) {
  getNextToken();
  condition();
  if(token != thensym)
    error(16);  // then expected
  else
    getNextToken();
  ctemp = cx;
  emit(JPC, 0, 0);
  statement();
  code[ctemp].m = cx;
}
```

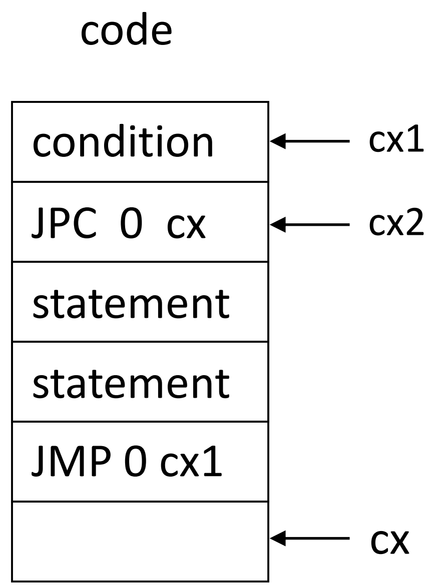
"**if**" condition "**then**" statement



# Major Example 3: While

```
if (token == whilesym) {
     cx1 =cx;
     getNextToken();
     condition();
     cx2 = cx;
     gen(JPC, 0, 0)
     if(token != dosym)
        error(18);  // then expected
     else
        getNextToken();
     statement();
     gen(JMP, 0, cx1);
     code[cx2].m = cx;
   }
```

"**while**" condition "**do**" statement



# Next Time:
# LL(1) Parsing