COP 3402 Systems Software

Euripides Montagne
University of Central Florida

COP 3402 Systems Software

Intermediate Code Generation

Outline

1. From syntax graph to parsers
2. Tiny-PL/0 syntax
3. Intermediate code generation
4. Parsing and generating Pcode.

Building a parser from a Syntax Graph

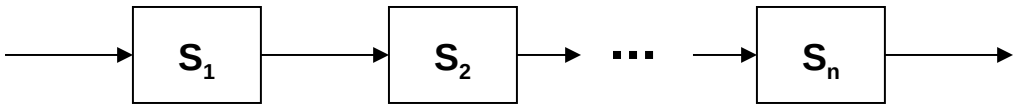
Transforming a grammar expressed in EBNF to syntax graph is advantageous to visualize the parsing process of a sentence because the syntax graph reflects the flow of control of the parser.

Rules to construct a parser from a syntax graph (N. Wirth):

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

B2.- Translate each graph into a procedure declaration according to the subsequent rules B3 through B7.

B3.- A sequence of elements



Is translated into the compound statement

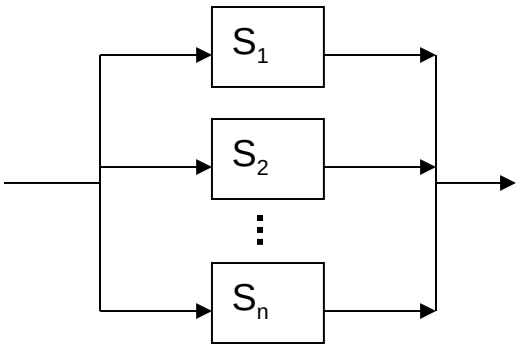
$$\{ T(S_1); T(S_2); \ldots; T(S_n) \}$$

T(S) denotes the translation of graph S

Building a parser from a Syntax Graph

Rules to construct a parser from a syntax graph:

B4.- A choice of elements



is translated into a selective or conditional statement

Selective

```
Switch (ch) {
case ch in L1 : T(S1);
case ch in L2 : T(S2);
. . .

case ch in Ln : T(Sn);
default: error
}
```

Conditional

if ch in L_1 {$T(S_1)$ else
if ch in L_2 { $T(S_2)$ else
. . .

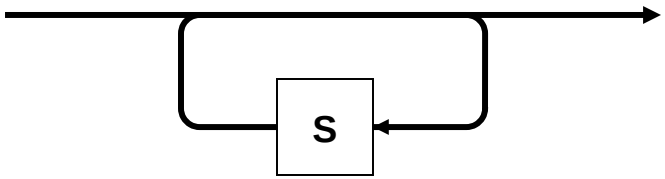
if ch in L_n { $T(S_n)$} else
error

If L_i is a single symbol, say a, then "ch in L_i" should be expressed as "ch == a"

Building a parser from a Syntax Graph

Rules to construct a parser from a syntax graph:

B5.- A loop of the form



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 B7,

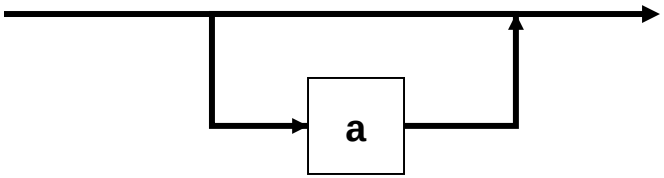
and L_i is a single symbol, say a, then "ch in L_i" should be expressed as "ch == a",

however L could be a set of symbols.

Building a parser from a Syntax Graph

Rules to construct a parser from a syntax graph:

B6.- A loop of the form



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,

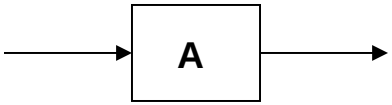
and L_i is a single symbol, say a, then "ch in L_i" should be expressed as "ch == a",

however L could be a set of symbols.

Building a parser from a Syntax Graph

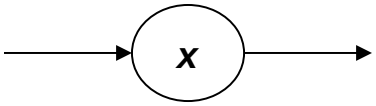
Rules to construct a parser from a syntax graph:

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

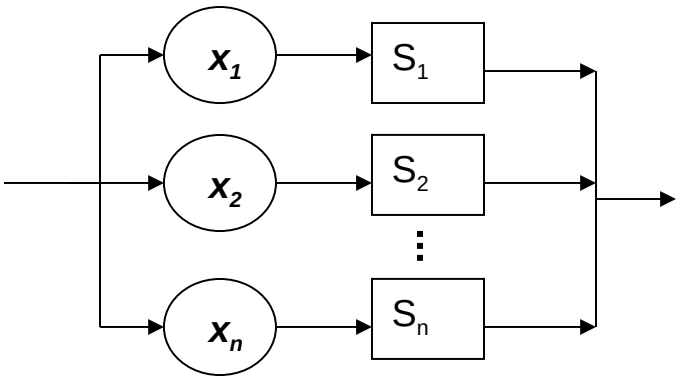
if (ch = x) { read(ch) } else {*error* }

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

Building a parser from a Syntax Graph

Useful variants of rules B4 and B5:

B4a.- A choice of elements



Conditional

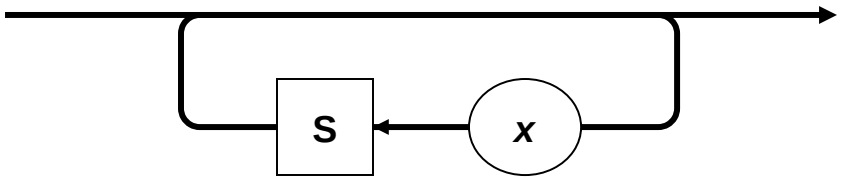
if ch == ‘x_1’ { read(ch) T(S_1) } else
if ch == ‘x_2‘ { read(ch) T(S_2) } else
. . .

if ch == ‘x_n‘ { read(ch) T(S_n)} else
error

Building a parser from a Syntax Graph

Useful variants of rules B4 and B5:

B5a.- A loop of the form



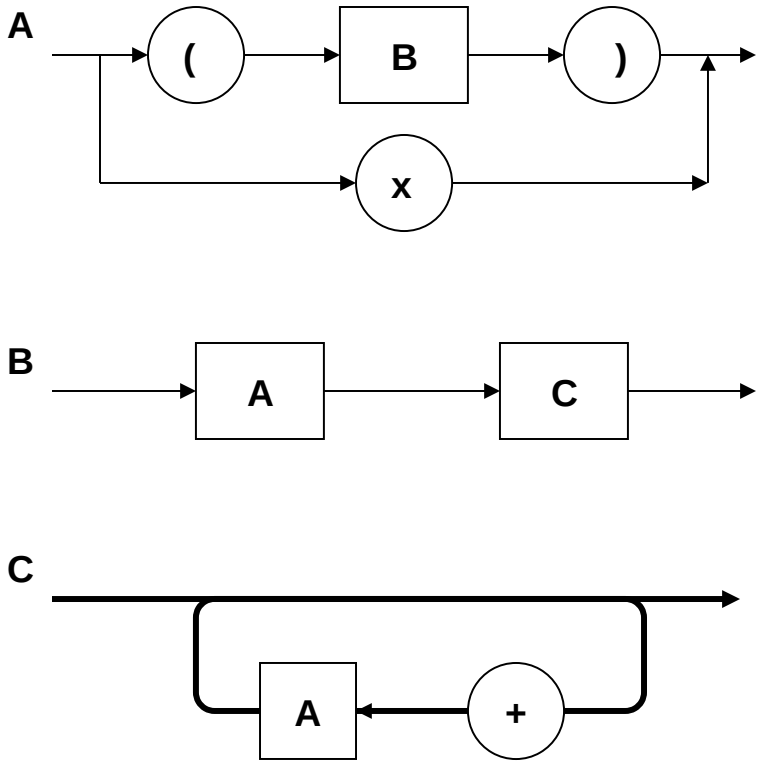
is translated into the statement

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

Example

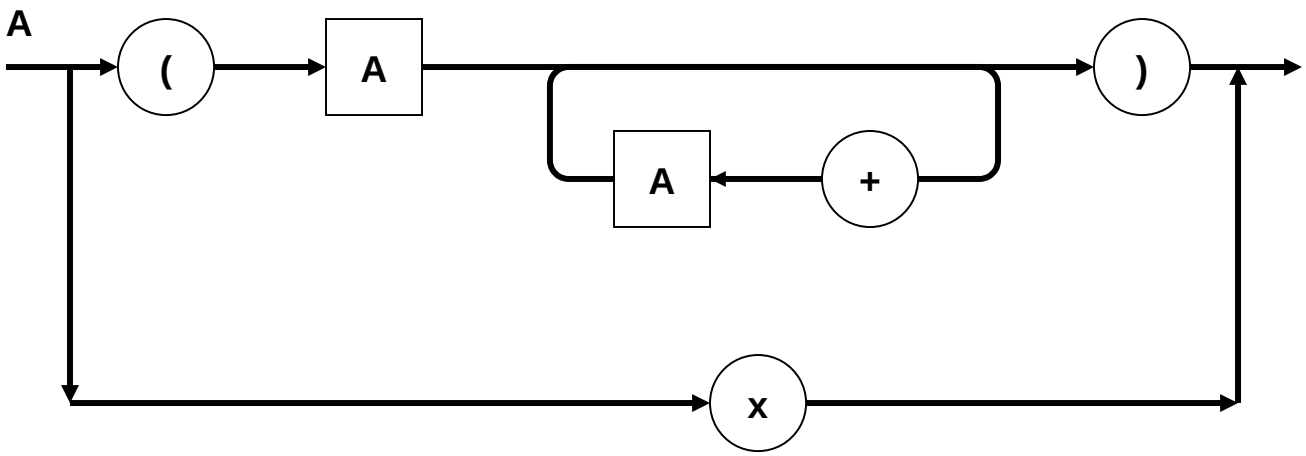
Applying the above mentioning rules to create one graph to this example:

A ::= “x” | “(“ B “)”
B ::= A C
C ::= { “+” A }



Syntax Graph

We will obtain this graph:



Using this graph and choosing from rules B1 to B8 a parser program can be generated.

Parser program for the graph A (in PL/0)

```
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.
```

EBNF grammar for Tiny PL/0 (1)

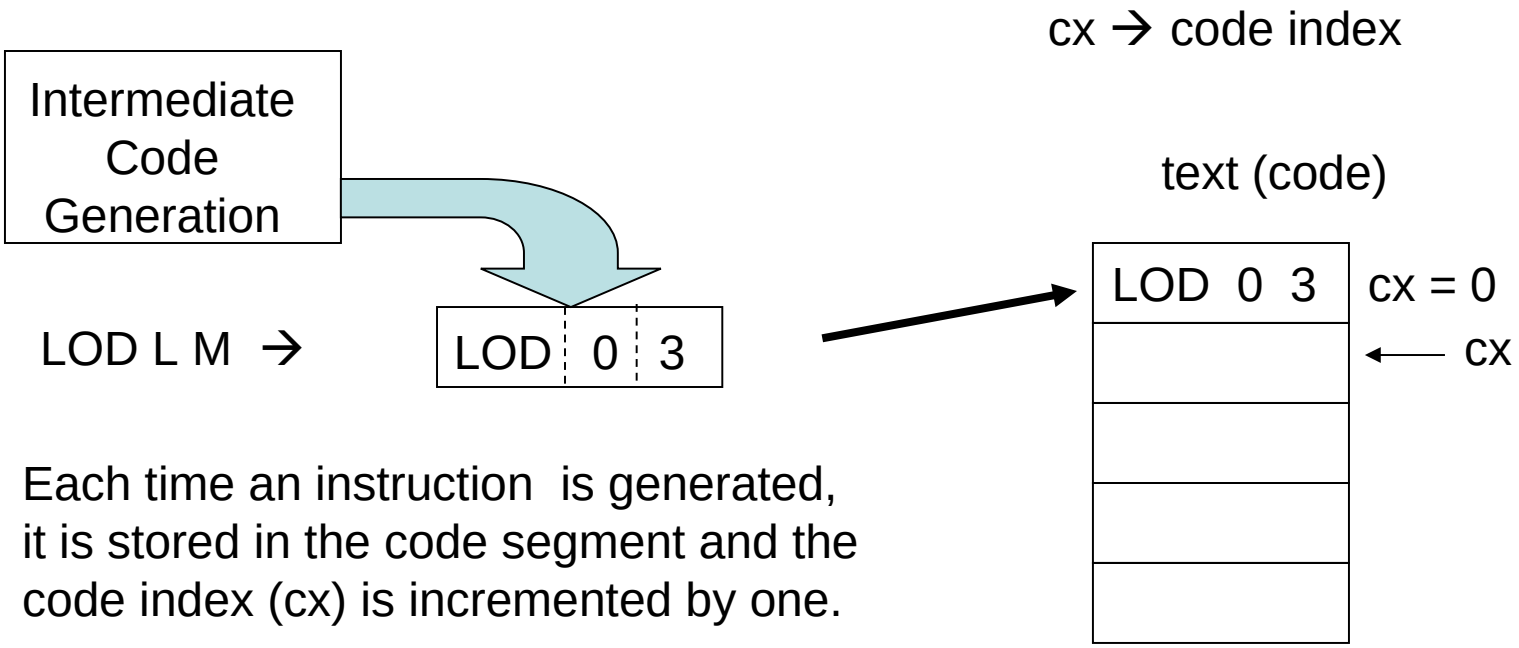
<program> ::= block "**.**" .
<block> ::= <const-declaration> <var-declaration> <statement>
<constdeclaration> ::= [“**const”** <ident> "**=**" <number> {"**,**" <ident> "**=**" <number>} "**;**"]
<var-declaration> ::= ["**var**" <ident> {"**,**" <ident>} "**;**"]
<statement > ::= [<ident> "**:=**" <expression>
| "**begin**" <statement> {"**;**" <statement> } "**end**"
| "**if**" <condition> "**then**" <statement>
| ”**while**" <condition> ”**do**" <statement>
| ε]

<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**"

```
procedure PROGRAM;
        begin
            GET(TOKEN);
            BLOCK;
            if TOKEN != "periodsym" then ERROR
        end;
```

Intermediate code generation



Each time an instruction is generated, it is stored in the code segment and the code index (cx) is incremented by one.

LOD L M → LOD 0 3

a := b + c;

Parsing and generating pcode

emit function

```
void emit(int op, int L, int M)
{
  if(cx > CODE_SIZE)
    error(25);
  else
  {
    text[cx].op = op;     //opcode
    text[cx].L = L;       // lexicographical level
    text[cx].M = M;       // modifier
    cx++;
  }
}
```

Parsing and generating pcode

<expression> → [+ | -] <term> {(+ | -) <term>}

```
void expression( )
{
  int addop;
  I f (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); stack machine ← (addition) → Register machine //emit(OPR, RF[ri], 0, M);
    else
      emit(OPR, 0, OPR_SUB); // subtraction
  }
}
```

← Function to parse an expression

Parsing and generating pcode

<term> → <factor> { (* | /) <factor> }

```
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
  }
}
```

Parsing <term>

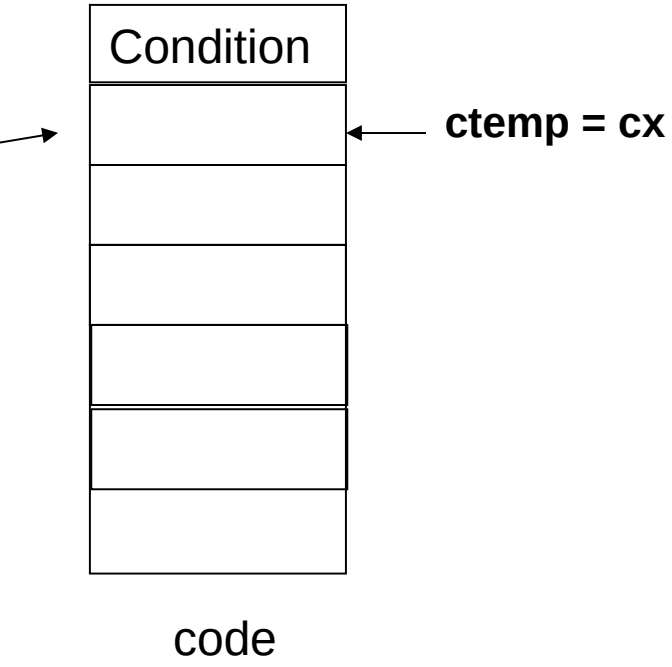
Parsing and generating pcode

if <condition> **then** <statement>

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

Parsing the construct IF-THEN

if <condition> then a := b + 5;



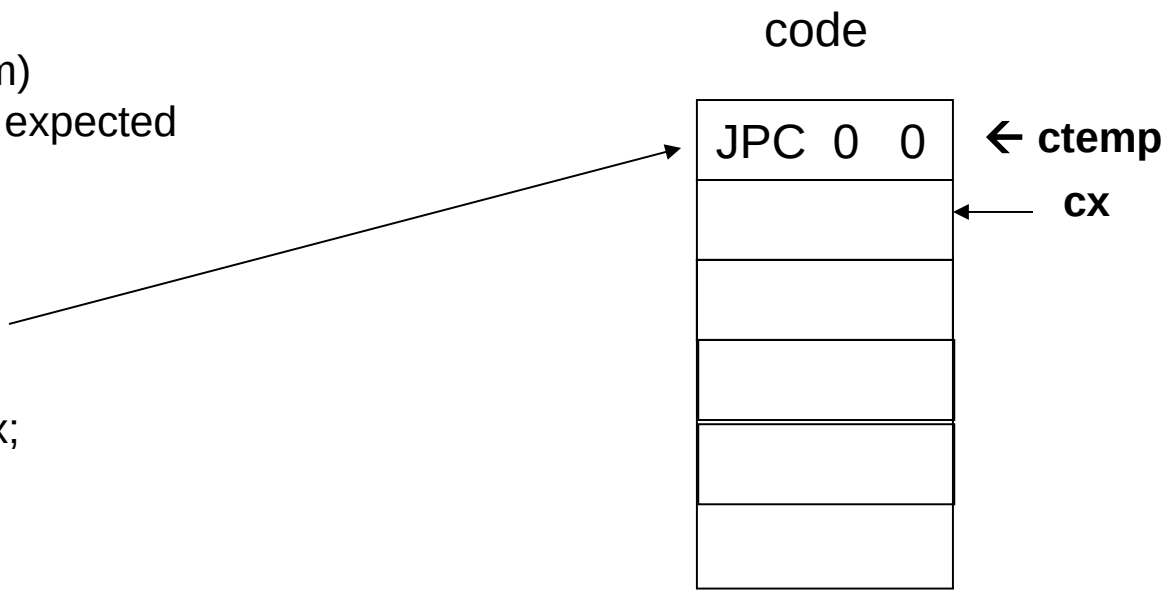
Parsing and generating pcode

if <condition> **then** <statement>

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

Parsing the construct IF-THEN

if <condition> then a := b + 5;



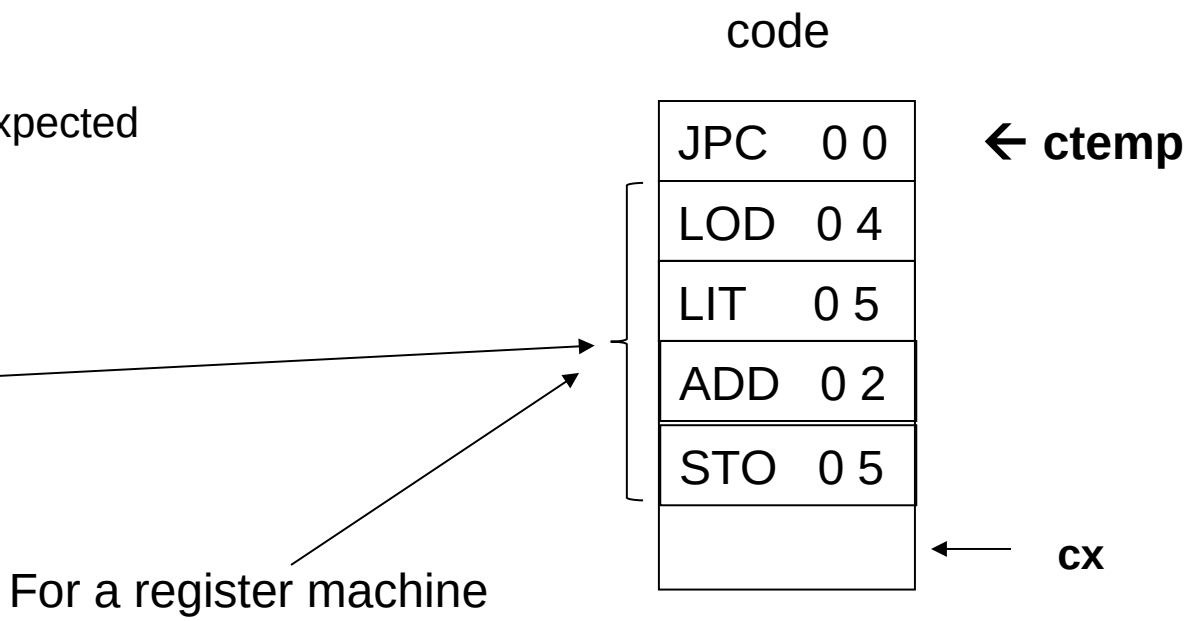
Parsing and generating pcode

if <condition> **then** <statement>

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

Parsing the construct IF-THEN

if <condition> then a := b + 5;

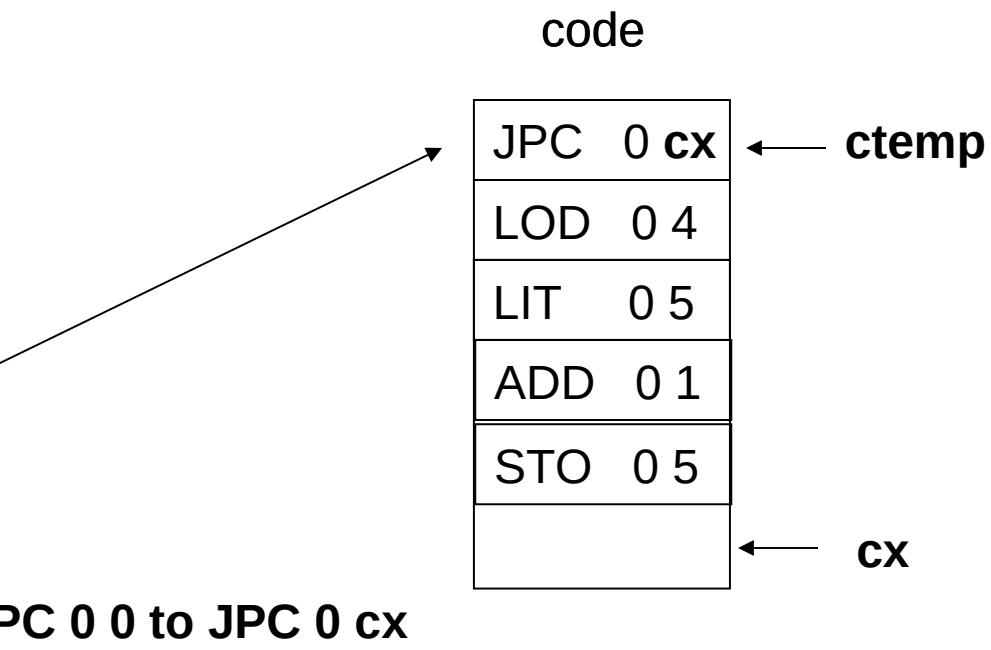


Parsing and generating pcode

if <condition> **then** <statement>

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

Parsing the construct IF-THEN
if <condition> then a := b + 5;

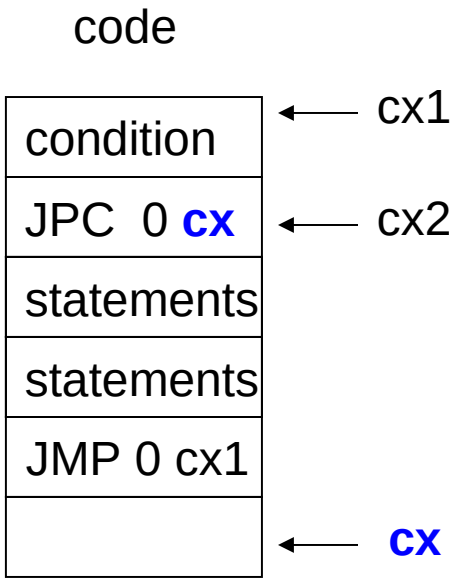


Parsing and generating pcode

while <condition> do <statement>

```
If  token = whilesym then
  begin
    cx1 := cx;
    getNextToken( );
    condition( );
    cx2 := cx;
    emit(JPC  0, 0)
    if token != dosym then
       error(18);  // do expected
    else
       getNextToken( );
    statement( );
    emit(JMP  0, cx1);
    code[cx2].m := cx;
  end
```

Parsing the construct WHILE-DO



I need help from my students to implement <condition>

COP 3402 Systems Software

<condition> ::= "**odd**" <expression> | <expression> rel-op <expression>**.**

```
Condition( )
{
  int relop;
  if(token==oddsym) {
    getNextToken(); expression( ); emit(2 0 6);}
    else {
      expression( );
      if ((token!=eql)&&(token!=neq)&&(token!=lss)&&
         (token!=leq)&&(token!=gtr)&&(token!=geq)) error(20);
      else {
        relop=token; getNextToken(); expression( );
        switch (relop) {
          case   9: emit(02, 0, 8); break;         /* equal */
          case  10: emit(02, 0, 9); break;       /* not equal */
          case 11: emit(02, 0, 10); break;      /* < */
          case 12: emit(02, 0, 11); break; ;    /* <= */
          case 13: emit(02, 0,12); break;      /* > */
          case 14: emit(02, 0,13);               /* >= */

        }
      }
    }
}
```

Parsing and generating opcode

<identifier> := <expression> // Assignment statement

Example:
var a,b;
a := a + b.

procedure Statement; **02 a 20 02 a 04 02 b 19**

```
  begin
     if token <> "identifier-symbol" then ERROR ( ).
     else
       begin
         get (next token)                    // gets name x from the token string.
         id := token;                        // id stores x
           i :=  find-in-symbol-table (id);  //  i != 0 variable found in ST
         if i := 0 then ERROR( );            // variable not declared
         if ST[ i ].kind != 2 then  ERROR( );
         get (next token);
          if token != ":=" then ERROR ( );
          get(next token);
          call  E;
          if i != 0 then emit (sto,  0, ST[ i ].adrr)  /
          if token != "." ERROR;  // for this example only
        end;
      if token <> "if-symbol" then ERROR ( ).  // In case we have a "if"
      else ……
  end;
```

```
Lod  0 6
Lod  0 8
Add  0 2
Sto  0 6
```

Discussion in class to test your compiler

```
var x ,y;
x := y + 777.
```

Address of x → 4
Address of y → 5

```
INC   0 6
LOD  0 5
LIT    0 777  → → VM
ADD  0 2
STO   0 4
SYS   0 3
```

COP 3402 Systems Software

The end