COP 3402 Systems Software

Intermediate Code Generation

Thanks to Euripides Montagne

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

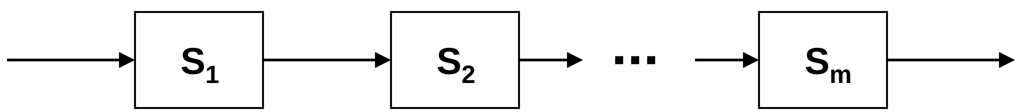
Transforming a grammar expressed in EBNF to syntax graph is advantageous to visualize the parsing process of a sentence because the syntax graph reflects the flow of control of the parser.

<u>Rules to construct a parser from a syntax graph (N. Wirth):</u>

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

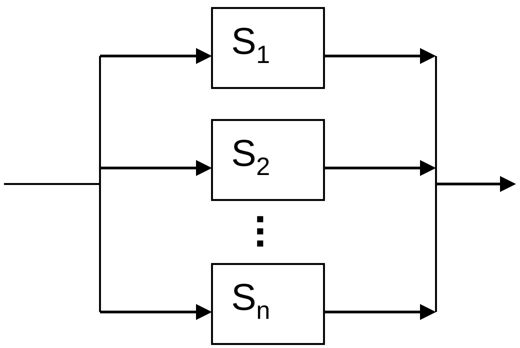
$$\{\ T(S_1);\ T(S_2);\ \ldots;\ T(S_n)\ \}$$

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
. . .

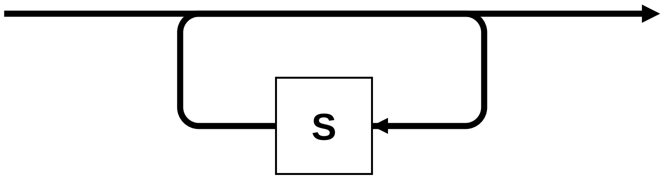
if ch in L_n { $T(S_n)$} else
error

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,

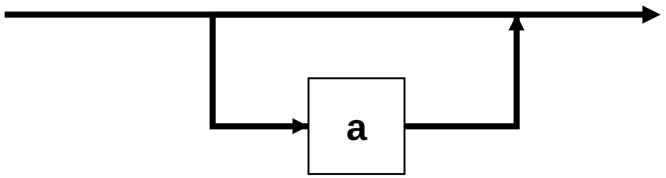
and L_i is a single symbol, say a, then "ch in L_i" should be expressed as "ch == a",

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,

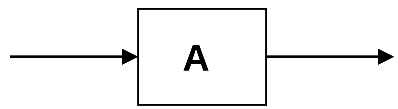
and L_i is a single symbol, say a, then "ch in L_i" should be expressed as "ch == a",

however L could be a set of symbols.

Building a parser from a Syntax Graph

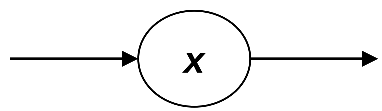
Rules to construct a parser from a syntax graph:

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

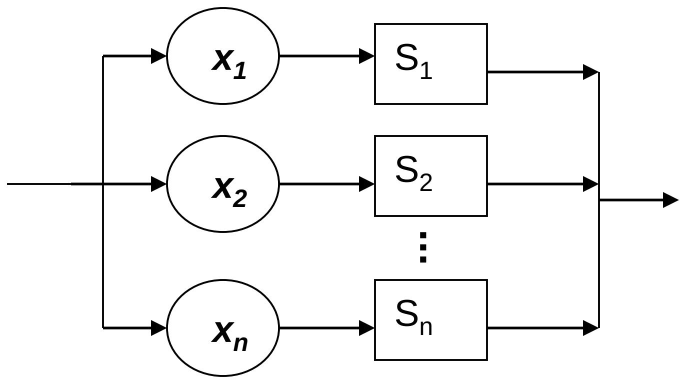
if (ch = x) { read(ch) } else {*error* }

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

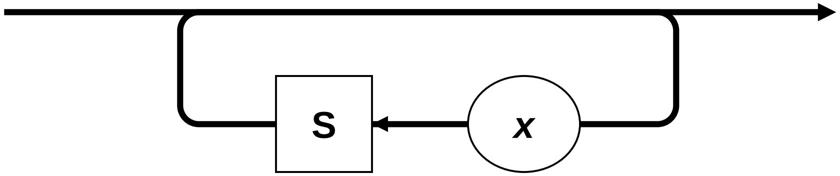
if ch == 'x_1' { read(ch) T(S_1) } else
if ch == 'x_2' { read(ch) T(S_2) } else
. . .

if ch == 'x_n' { read(ch) T(S_n)} else
error

Building a parser from a Syntax Graph

<u>Useful variants of rules B4 and B5:</u>

B5a.- A loop of the form



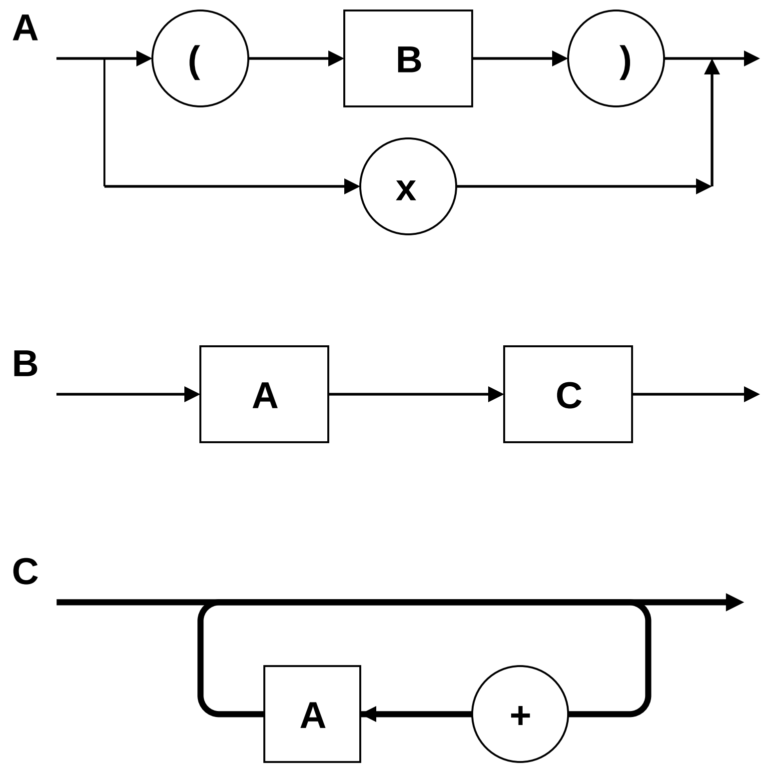
is translated into the statement

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

Example

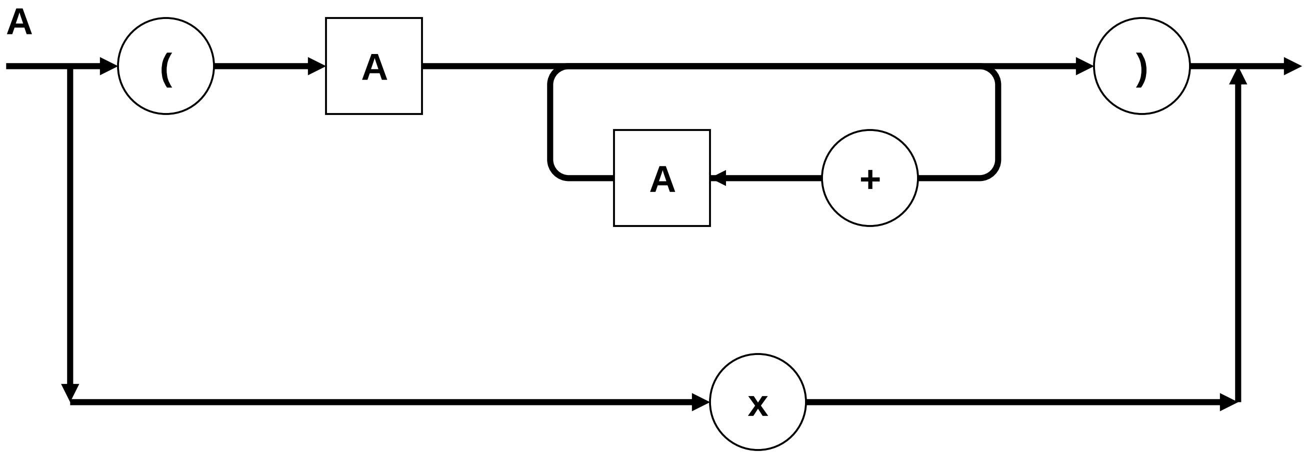
Applying the above mentioning rules to create one graph to this example:

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)

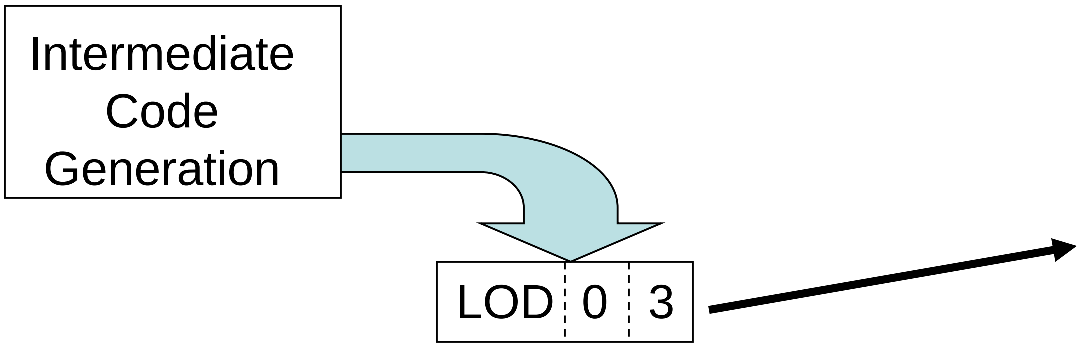
<program> ::= block "**.**" .
<block> ::= <const-declaration> <var-declaration> <statement>
<constdeclaration> ::= ["**const**" <ident> "**=**" <number> {"**,**" <ident> "**=**" <number>} "**;**"]
<var-declaration> ::= ["**var**" <ident> {"**,**" <ident>} "**;**"]
<statement > ::= [<ident> "**:=**" 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

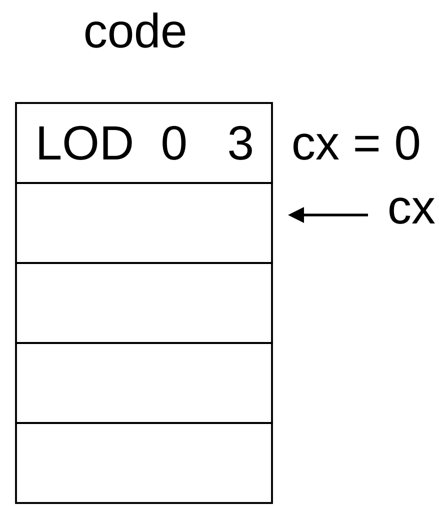
<condition> ::= "**odd**" <expression>
| <expression> <rel-op> <expression>

<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**"

Intermediate code generation

cx → code index





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

Parsing and generating pcode

emit funtcion

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

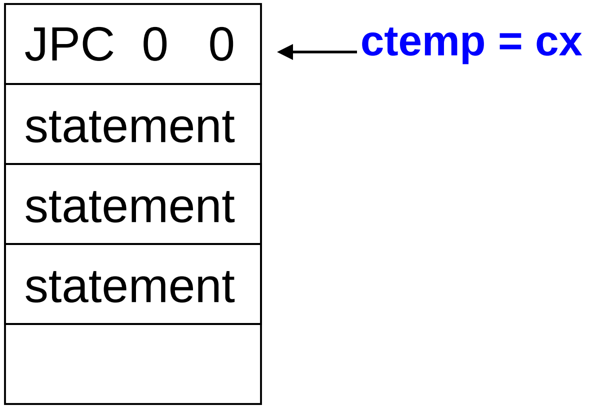
Parsing and generating pcode

If <condition> then <statement>

Parsing the construct IF-THEN

```
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;
  }
```

code

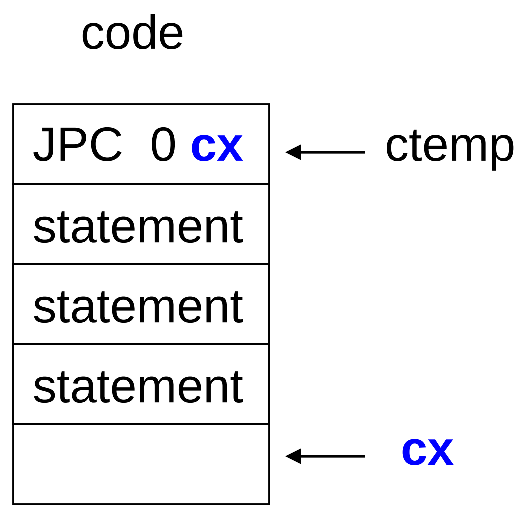


Parsing and generating pcode

If <condition> then <statement>

Parsing the construct IF-THEN

```
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;
  }
```



changes JPC 0 0 to JPC 0 cx

Parsing and generating pcode

while <condition> do <statement>

```
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;
  }
```

Parsing the construct WHILE-DO