Intermediate Code Generation
Outline

1. From a syntax graph to a recursive decent parser
2. Syntax of tiny-PL/0 (subset of PL/0)
3. Generation of PM/0 code for programming constructs of tiny-PL/0
4. Symbol table
Building a Parser from a Syntax Graph

Transforming a grammar expressed in EBNF to syntax graph is helps 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

\[ S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_m \]

is translated into the compound statement

\[ \{ T(S_1); T(S_2); \ldots ; T(S_n) \} \]

T(S_1) denotes the translation of graph S_1
Rules to construct a parser from a syntax graph:

**B4.- A choice of elements**

\[
\begin{align*}
L_1 & \text{ denotes the set of tokens that can occur at the beginning of } S_1. \\
L_i & \text{ contains a single token, say } t, \text{ then } tok \text{ in } L_i \text{ should be expressed as } tok == t
\end{align*}
\]
Rules to construct a parser from a syntax graph:

B5.- A loop of the form

\[ \text{while tok in L do T(S)} \]

is translated into the statement

\[ \text{while tok in L do T(S)} \]

where \( T(S) \) is the translation of \( S \) according to rules B3 through B7.
Building a Parser from a Syntax Graph

Rules to construct a parser from a syntax graph:

B6. A loop of the form

\[
\text{if } ch \text{ in } L \{ T(S) \}
\]

is translated into the statement

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

$$\text{if (tok == t) \{ read(tok) \} else \{ error \}}$$

where error is a routine called when an ill-formed construct is encountered.
Useful variants of rules B4 and B5:

B4a. A choice of elements

Conditional

if tok == t₁ { read(ch); T(S₁); } else
if tok == t₂ { read(ch); T(S₂); } else
... 
if tok == tₙ { read(ch); T(Sₙ); } else
error
Useful variants of rules B4 and B5:

B5a. - A loop of the form

\[ \text{while (tok == t) \{ read(tok); T(S); \}} \]

is translated into the statement

\[ \text{while (tok == t) \{ read(tok); T(S); \}} \]
We now apply these rules to write the parser based on the syntax graph

\[ A ::= \text{“x”} \mid \text{“(“ B “)”} \]
\[ B ::= A C \]
\[ C ::= \{ \text{“+”} A \} \]
We will obtain this graph:

We can generate a parser program by transforming this graph following the appropriate rules from B1 to B8.

The parser is written in PL/0-based pseudocode.
Parser Program for the Graph A

```pascal
var ch : char;

procedure A;
begin
  if ch = 'x' then get(tok)
  else if ch = '(' then
    begin
      get(tok);
      A;
      while tok = '+' do
        begin
          get(tok);
          A;
        end;
      if tok = ')' then get(tok) else error(err_number)
    end;
  else error(err_number)
end;

begin
  get(tok);
  A;
end.
```

EBNF grammar for Tiny-PL/0

<program> ::= block ".
<block> ::= <const-declaration> <var-declaration> <statement>

<const-declaration> ::= [ "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>
| "read" <ident>
| "write" <ident>
]

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

<rel-op> ::= "=" | "<>" | "<" | "<=" | ">" | ">="
<expression> ::= [ "+" | "-" ] <term> { ( "+" | "-" ) <term> }
<term> ::= <factor> { ( "*" | "/" ) <factor> }
<factor> ::= <ident> | <number> | "(" <expression> ")"

<number> and <ident> are tokens with semantic values
The symbol table or name table records information about each symbol name in the program.

Each piece of information associated with a name is called an attribute.

For instance, the type of a variable, the parameters of a procedure, the number of dimensions for an array etc.

The symbol table can be organized as a linear list, a tree, or as a hash table.
Symbol Table

Symbol table operations

Enter (insert)

Lookup (retrieval)

Enter
When a declaration is processed the name is inserted into the symbol table

If the programming language does not require declarations (such as JavaScript or Python), then the name is inserted when the first occurrence of the name is found.

In tiny PL/0, we only enter constants and variables while parsing <const-declaration> and <var-declaration>.

Lookup
Each subsequent use of the name causes a lookup operation.

In tiny PL/0, we only lookup constants and variables while parsing <ident>.

For tiny PL/0 the symbol table is flat because we do not have procedures (nothing is nested).
Symbol Table

We put symbols into the symbol table while parsing the <ident>’s inside <const-declaration> and <var-declaration>

<const-declaration> ::= [ “const” <ident> “=” <number> { “,” <ident> “=” <number> } “;” ]

<var-declaration> ::= [ “var” <ident> { “,” <ident>} “;” ]

We get symbols from the symbol table while parsing the <ident> inside <factor> and inside <statement>

<factor> ::= <ident> | ...

<statement> ::= [ ... | “read” <ident> | “write” <ident> ]
We assume that we have the following definitions available.

```c
// token type
typedef enum {
    nulssym = 1, identsym, numbersym, plussym, minussym,
    multsym, slashsym, oddsym, eqsym, neqsym, lessym, leqsym,
    gtrsym, geqsym, lparentsym, rparentsym, commasym, semicolonsym,
    periodsym, becomesym, beqnsym, endsym, ifsym, thensym,
    whilesym, dosym, callsym, constsym, varsym, procsym, writesym,
    readsym, elsesym
} token_type;

// semantic value
//   either string (char *) for the name of a constant/an identifier
//   or     int for the value of the number
typedef union {
    string id;
    int num;
} LVAL;
```
Lexer variables and functions

We assume that we have the following definitions available.

// global variable holding the token type of the current token
token_type token;

// global variable holding the semantic value of the current token
LLVAL lval;

// function advance that move to the next token and
// updates the above two global variables
void advance();
We assume that we have the following definitions available.

typedef struct symbol {
  int kind;       // constant = 1, variable = 2
  string name;    // name of constant or variable
  int num;        // number number is only set for constant
  int level;      // L level level and modifier are only set for variable,
  int modifier;   // M modifier but level is always 0 for tiny PL/0
} symbol_type;

// function get_symbol that looks up a symbol in symbol table by name and
// returns pointer symbol if found and NULL if not found

symbol_type *get_symbol(string name)

// function put_symbol that puts a symbol into symbol table provided that
// a symbol with this name does not exist (calls error function if name already exists)

void put_symbol(int kind, string name, int num, int level, int modifier);
void const_declaration() {
    if (token != constsym) return;
    string id;

    do {
        advance();

        if (token != identsym) error("expected identifier in constant declaration");
        id = lval.id;
        advance();

        if (token != eqsym) error("expected '=' after identifier in constant declaration");
        advance();

        if (token != numbersym) error("expected number after '=' in constant declaration");
        put_symbol(1, id, lval.num, 0, 0); // constant => kind = 1
        advance();
    } while (token == commasym);

    if (token != semicolonsym) error("expected ';' at the end of constant declaration");
    advance();
}
void var_declaration() {
    int num_vars = 0;

    if (token == varsym) {
        do {
            advance();

            if (token != identsym) error("expected identifier in variable declaration");
            num_vars++;
            put_symbol(2, lval.id, 0, 0, 3 + num_vars); // variable => kind = 2
            advance();
        } while (token == commasym);

        if (token != semicolonsym) error("expected ';' at the end of variable declaration");
        advance();
    }

    emit(INC, 0, 4 + num_vars); // emit is defined on the following slides
}
Each time an instruction is generated, it is stored in the code store and the code index ($cx$) is incremented by one.
void emit(int op, int l, int m)
{
    if(cx > CODE_SIZE)
        error("code too long");
    else
    {
        code[cx].op = op; // opcode
        code[cx].l = l;   // lexicographical level
        code[cx].m = m;  // modifier
        cx++;
    }
}
Parsing and Generating PM/0-code

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

```c
void expression()
{
    int addop;
    if (token == plussym || token == minussym)
    {
        addop = token;
        advance();
        term();
        if(addop == minussym)
            emit(OPR, 0, OPR_NEG); // negate
    } else
        term ();

    while (token == plussym || token == minussym)
    {
        addop = token;
        advance( );
        term();
        if (addop == plussym)
            emit(OPR, 0, OPR_ADD); // addition
        else
            emit(OPR, 0, OPR_SUB); // subtraction
    }
}
```

Function to parse an expression
<term> ::= <factor> { ( "*" | "/" ) <factor> }

void term()
{
    int mulop;
    factor();
    while(token == multsym || token == slashsym)
    {
        mulop = token;
        advance();
        factor();
        if(mulop == multsym)
            emit(OPR, 0, OPR_MUL); // multiplication
        else
            emit(OPR, 0, OPR_DIV); // division
    }
}
Parsing and Generating PM/0-code

“if” <condition> “then” <statement>

if (token == ifsym)
{
    advance();
    condition();
    if (token != thensym)
        error("then expected");
    else
        advance();

    ctemp = cx;
    emit(JPC, 0, 0);
    statement();
    code[ctemp].m = cx;
}
Parsing and Generating PM/0-code

“if” <condition> “then” <statement>
Parsing and Generating PM/0-code

“while” <condition> “do” <statement>

if (token == whilesym)
{
  cx1 = cx;
  advance();
  condition();
  cx2 = cx;
  emit(JPC, 0, 0)
  if (token != dosym)
    error("do expected");
  else
    advance();
  statement();
  emit(JMP, 0, cx1);
  code[cx2].m = cx;
}

\[
\begin{array}{|c|}
\hline
\text{condition} & \text{cx1} \\
\text{JPC 0 cx} & \text{cx2} \\
\text{statement} & \text{statement} \\
\text{JMP 0 cx1} & \text{cx} \\
\hline
\end{array}
\]
"read" <ident>, "write" <ident>, <ident> inside <factor>

Figure on your own how to generate code for <ident>.

• invoke the get_symbol function to determine if <ident> is a variable or constant and to obtain either
  
  o the modifier (indicating where the variable is stored in the activation record) or

  o the constant number

• emit ST0 and LIT depending on whether we have a variable or a constant.

Figure our how to generate code for <number> inside <factor> (much simpler than above because not necessary to invoke get_symbol).