Computer Engineering Department
Technical University of Lodz

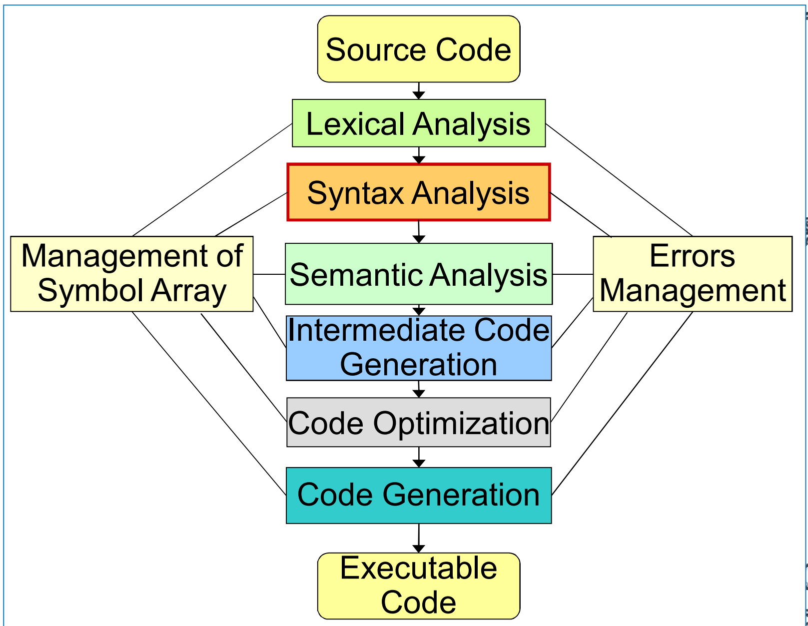
Mathematical Linguistics
Syntax Analysis

SYNTAX ANALYSER

Syntax Analyser (*syntactic analyser, parser*) is a translator component, which checks program or program module for syntax correctness.

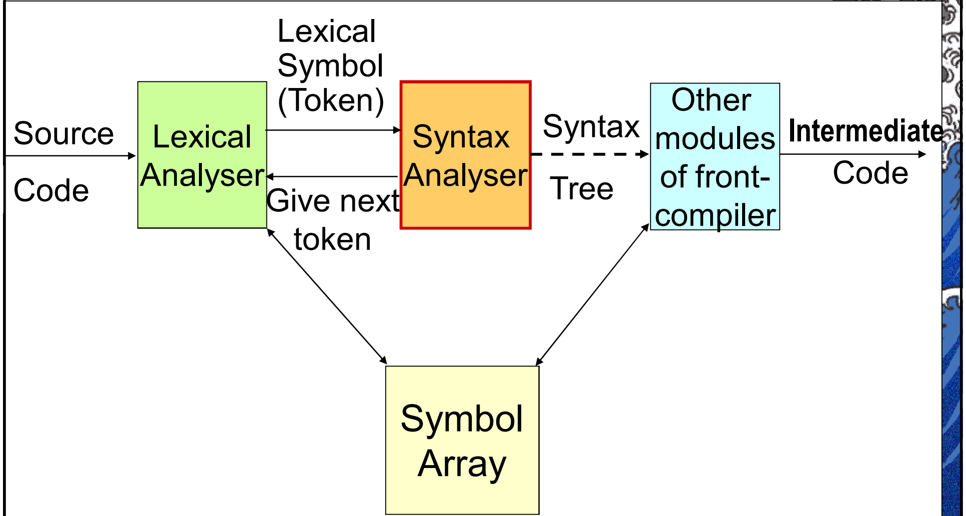
Important task in this translation process is finding and reporting errors in the source program.

Compiling Phases



Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

Position of Lexical Analyser in a Compiler



Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

CLASSIFICATION OF SYNTAX ANALYSERS

With respect to:

- **grammar:**
  - LR parser (Left-to-right, Rightmost derivation),
  - LL parser (Left-to-right, Leftmost derivation);
- **direction of derivation :**
  - Bottom-up parser,
  - To-down parser,
  - predictive parser;
- **way of syntax representation :**
  - parser for a given grammar,
  - parser controlled by the grammar.

Syntax Diagrams

Syntax diagrams are graphical representations of a grammar, equivalent to Backus-Naur form. This representation is made of a set of syntax diagrams, which is a scheme of ***syntax analyser*** program.

Diagrams features:

- Convenient form of language description
- Compact and clear scheme of language structure
- Help to understand syntax analysis
- A proper form for language design

Construction Rules of Syntax Diagrams

1. Each diagram defines a non-terminal symbol. Each non-terminal symbol A is defined by the production:

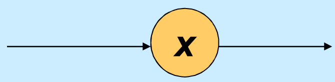
$$A ::= \xi_1 \mid \xi_2 \mid \ldots \mid \xi_n$$

is represented by a syntax diagram of A, which structure is defined by the right side of the production according to rules 2 - 6.

Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

Construction Rules of Syntax Diagrams

2. Terminals are represented by circles:

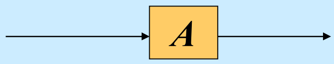


Each occurrence of a terminal symbol x in a sequence ξ_i corresponds to symbol recognition instruction and fetching the next lexical symbol (token) from the input string.

Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

Construction Rules of Syntax Diagrams

3. Non-terminals are represented by rectangles:



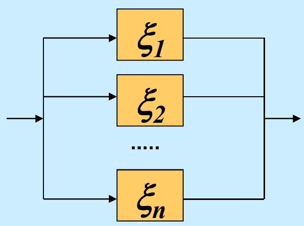
Each occurrence of non-terminal symbol A in a sequence ξ_i corresponds to calling a function which executes an algorithm defined by the diagram of symbol A.

Construction Rules of Syntax Diagrams

4. Each production of a form:

$$A ::= \xi_1 \mid \xi_2 \mid \ldots \mid \xi_n$$

is transformed into a diagram:



where each ξ_i is created according to rules 2 – 6 for ξ_i.

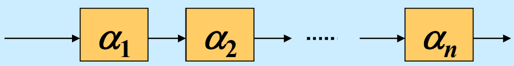
Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

Construction Rules of Syntax Diagrams

5. Each sequence of a form:

$$\xi = \alpha_1\ \alpha_2 \ldots\ \alpha_n$$

is transformed into a diagram:



where each α_i is created according to rules 2 – 6 for α_i.

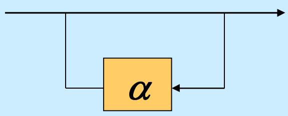
Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

Construction Rules of Syntax Diagrams

6. Each sequence of a form:

$$\xi = \{\ \alpha\ \}$$

is transformed into a diagram:



where α is created from α according to rules 2 – 6.

Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

EXAMPLE 1

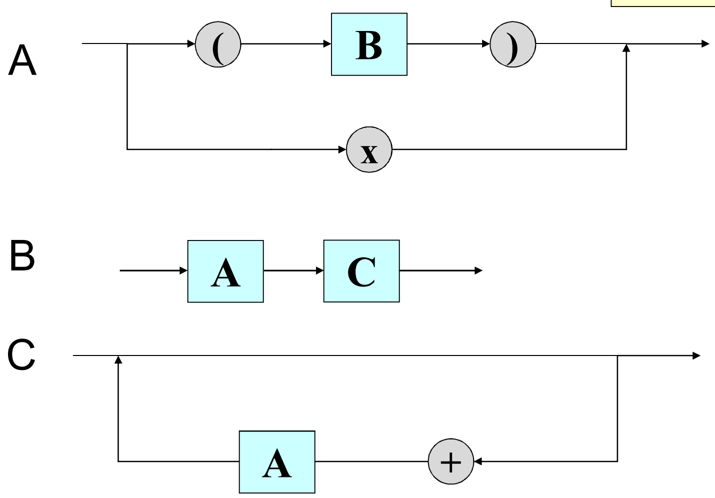
Draw syntax diagrams for the given grammar:

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

and reduce them to the optimal form.

EXAMPLE 1

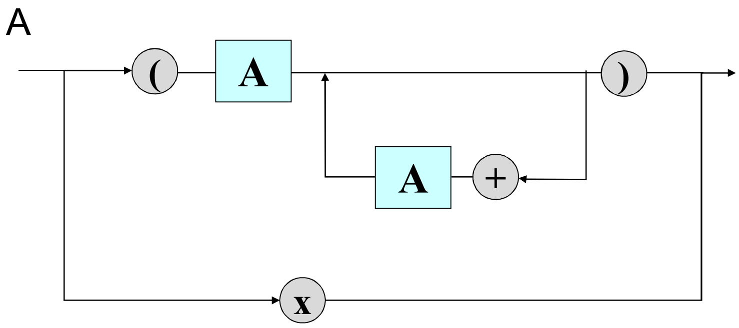
A::= x | (B)
B::= AC
C::= {+A}



EXAMPLE 1

A::= x | (A{+A})

Reduced syntax diagram



Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

EXAMPLE 2

Simplified grammar of arithmetic expressions in BNF

<expression> ::= <term> | <term> + <expression>
<term> ::= <factor> | <factor> * <term>
<factor> ::= <constant> | <variable> | (<expression>)
<variable> ::= x | y | z
<constant> ::= <digit> | <digit> <constant>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

EXAMPLE 2

The grammar after transformation into MBNF

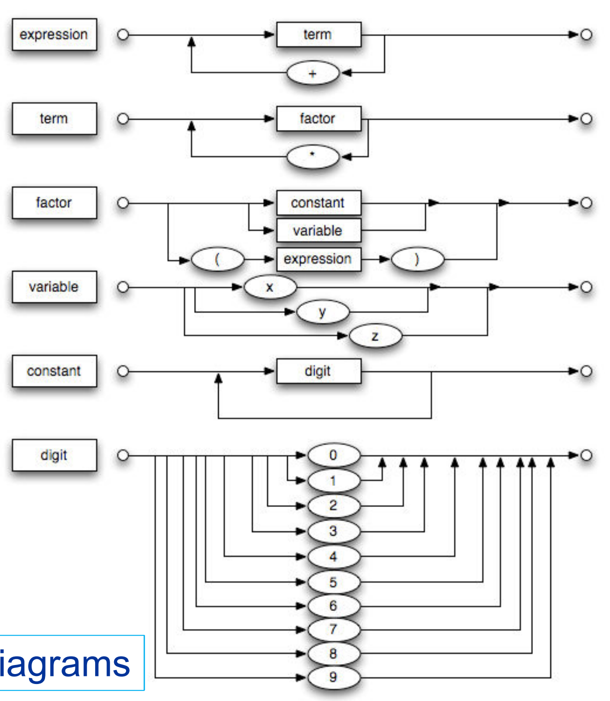
expression = term {"+" term}.
term = factor {"*" factor}.
factor = constant | variable | "(" expression ")".
variable = "x" | "y" | "z".
constant = digit {digit}.
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

EXAMPLE 2

Simplified BNF

E::= T {+T }
T::= F {*F }
F::= C | V |(E)
V::= x | y | z
C::= D {D}
D::= 0|1|2|3|4
D::= 5|6|7|8|9



Syntax diagrams

SYNTAX ANALYSER FOR THE GIVEN GRAMMAR

The aim of the ***syntax analysis*** is checking the correctness of the program grammar and sending messages about the errors.
All the syntax errors should be reported.

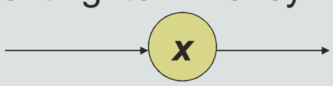
Syntax diagram is a block diagram of the program algorithm.

Specific rules transforming ***deterministic syntax diagram*** into a ***program*** are used for building the ***analyser for the given grammar.***

Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

Rules for transforming deterministic syntax diagram into a program

1. Reduce the number of diagrams as much as possible, applying the proper substitutions.
2. Replace each diagram with procedure declaration according to rules 3 - 7.
3. Replace a diagram's element representing terminal symbol *x*

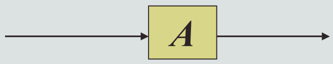


with an instruction:
if *ch* = *'x'* **then** *read(ch)* **else** *error*;

Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

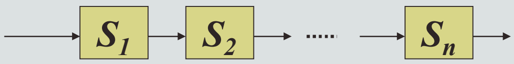
Rules for transforming deterministic syntax diagram into a program

4. Replace the diagram's element representing another diagram



with an instruction calling the procedure A.

5. Replace a sequence of elements



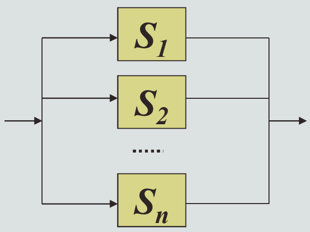
with a block of instructions:

begin $T(S_1); T(S_2); \ldots; T(S_n)$ **end**

where: $T(S_i)$ is derived from S_i according to rules 3 – 7.

Rules for transforming deterministic syntax diagram into a program

6. Replace an alternative diagram with a choice or conditional instruction:

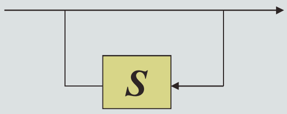


if *ch* **in** L_1 **then** $T(S_1)$ **else**
if *ch* **in** L_2 **then** $T(S_2)$ **else**
…
if *ch* **in** L_n **then** $T(S_n)$ **else** *error*;

case *ch* **of**
L_1: $T(S_1)$;
L_2: $T(S_2)$;
…
L_n: $T(S_n)$;
else *error*
end

Rules for transforming deterministic syntax diagram into a program

7. Replace a loop



with an instruction:

while *ch* **in** *L* **do** *T(S);*

where: *T(S)* is derived from *S* according to rules 3 - 7, and a set *L* = *first*(*S*)

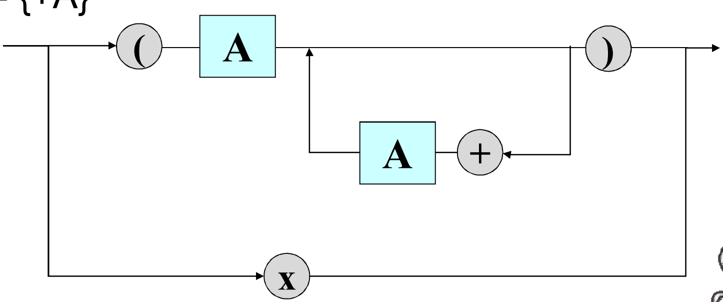
EXAMPLE 1

Write a program implementing syntax analyser for the given grammar:

A::= x | (B)

B::= AC

C::= {+A}



```
Program analyser;
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
                end     else error
end;
```



```
{main program}
begin
   read (ch); A
end.
```

CONSTRUCTING A TABLE-DRIVEN PARSING PROGRAM

A table-driven parser is a general parsing program. Its desigh is straightforward for LL(1) clas grammars. Then the simple top-down parsing method can be used.

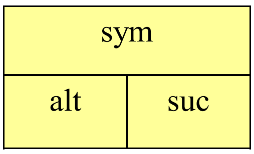
The given grammar, which we assume to be represented in the form of a deterministic set of syntax graphs, is translated into an appropriate data structure.

The program parsing is controlled by the dynamic data structure representing the given grammar.

Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

The nodes of the data structure

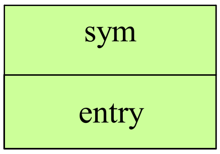
```
type pointer =  ^ node;
   node =
   record suc,alt: pointer;
        case terminal: boolean of
                true: (tsym: char);
                false: (nsym: hpointer)
   end
```



Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

The header node

```
type hpointer = ^ header;
   header =
        record entry: pointer;
                sym: char
        end
```



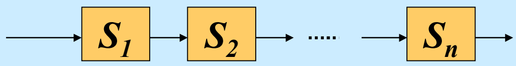
Rules of graph to data structure translation

1. Reduce the system of graphs (syntax diagrams) to as few individual graphs as possible by suitable substitution.
2. Translate each graph into a data structure according to the subsequent rules 3 - 6.
3. Replace each diagram's element representing terminal or non-terminal symbol with the appropriate node of the dynamic data structure.

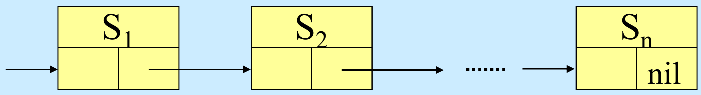
Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

Rules of graph to data structure translation

4. Translate a sequence of elements



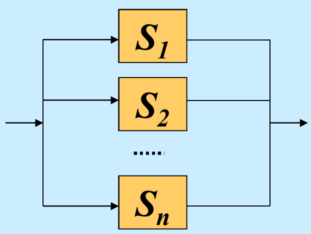
into the following list of data nodes:

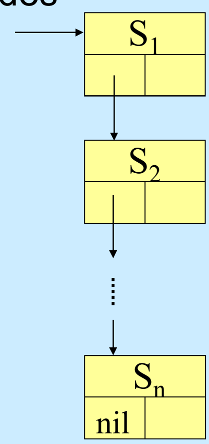


Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

Rules of graph to data structure translation

5. Translate an alternative diagram into the alternative list of data nodes

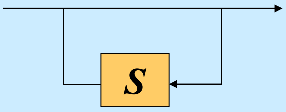




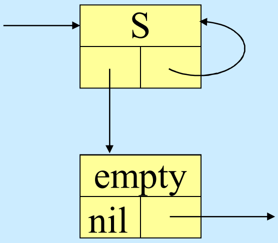
Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

Rules of graph to data structure translation

6. Translate a loop



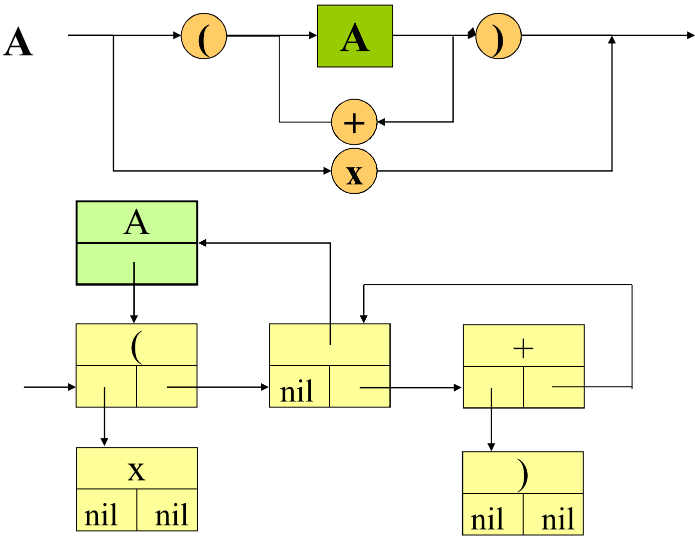
into the structure



Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

EXAMPLE

Translate the diagram into the data structure



Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

RBNF Notation

<table>
<tr><th>BNF</th><th>RBNF</th></tr>
<tr><td>::=</td><td>=</td></tr>
<tr><td>|</td><td>,</td></tr>
<tr><td>{</td><td>[</td></tr>
<tr><td>}</td><td>]</td></tr>
</table>

Each production is terminated by a dot.

The meta-language of syntax productions

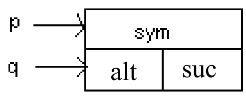
<production> ::= <symbol> = <expression>.
<expression> ::= <term> {,<term>}
<term> ::= <factor> {<factor>}
<factor> ::= <symbol> | [<term>]

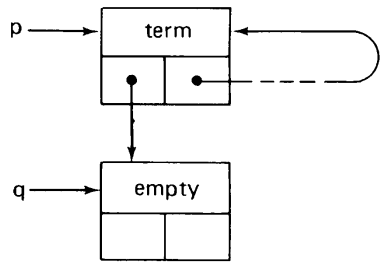
Parser accepts productions in RBNF notation.

Dynamic data structure

FACTORS

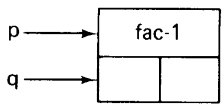
1. <symbol> ***2. [<term>]***

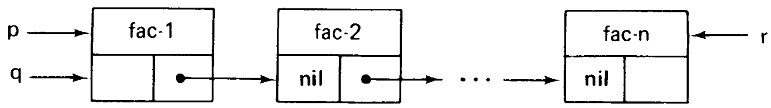




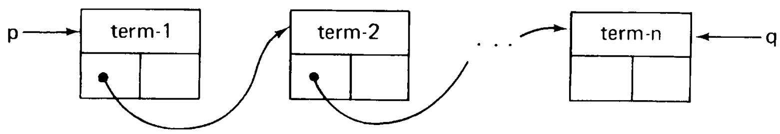
Dynamic data structure

TERMS **<factor-1> ... <factor-n>**





EXPRESSIONS **<term-1>, … , <term-n>**



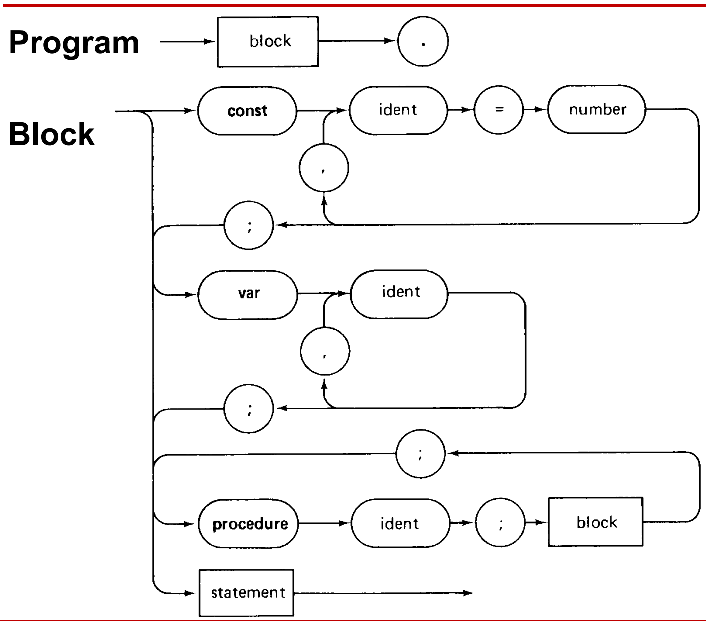
CONSTRUCTING A COMPILER FOR PL/0 LANGUAGE

Development of a compiler for PL/0 language is divided into the three main steps:

- constructing a parser for PL/0,
- recovering from syntactic errors,
- code generation.

PL/0 is a mini-language designed specially for didactic purposes. PL/0 is one possible compromise between sufficient simplicity and complexity. The designed compiler is reasonably small, but its project expose the most fundamental concepts of compiling high-level languages.

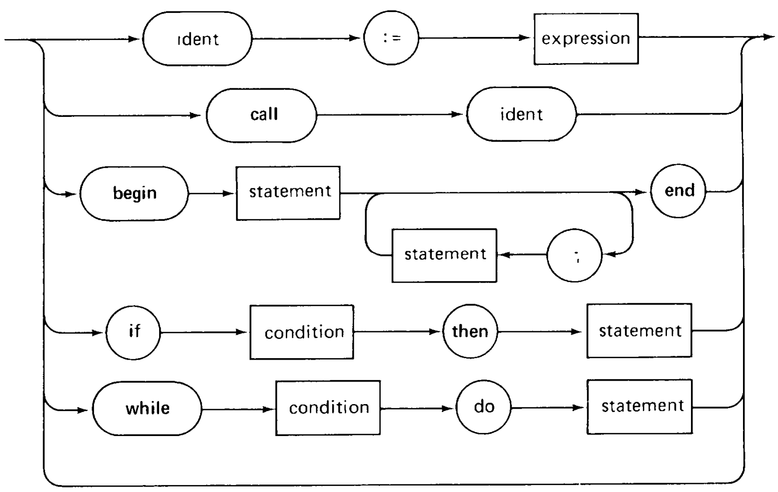
Syntax of PL/0



Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

Syntax of PL/0

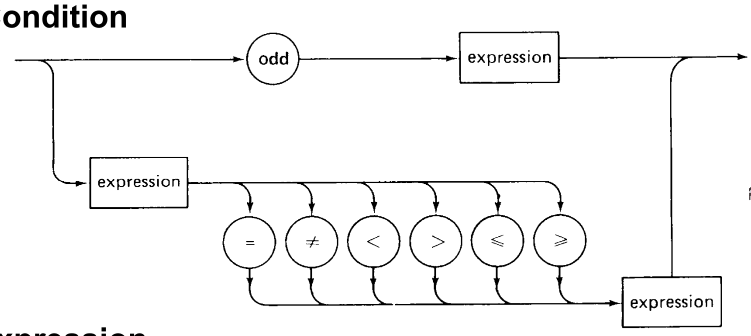
Statement



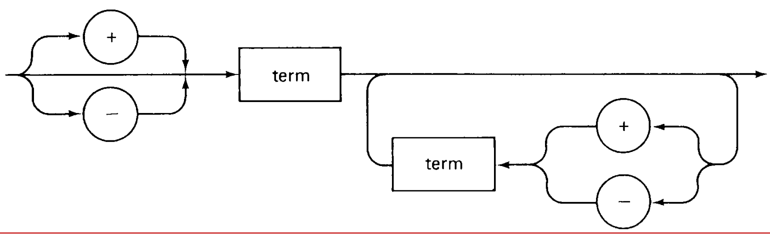
Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

Syntax of PL/0

Condition



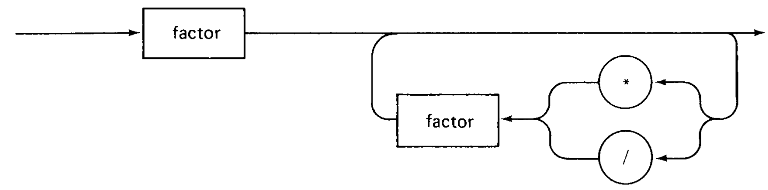
Expression



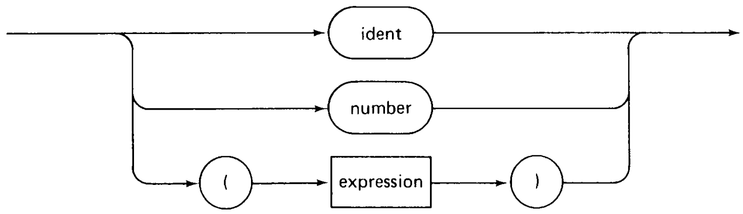
Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

Syntax of PL/0

Term



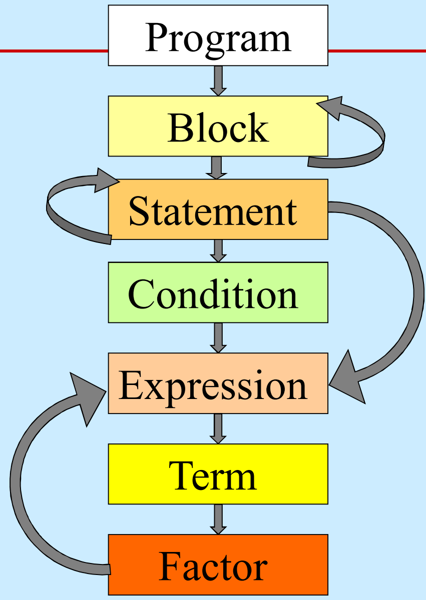
Factor



First and Follow Symbols in PL/0

Non-terminal Symbol	First (X)	Follow (X)
Block	const var procedure ident call begin if while	. ;
Statement	ident call begin if while	. ; end
Condition	odd + - (ident liczba	then do
Expression	+ - (ident liczba	. ; end then do R)
Term	(ident liczba	. ; end then do R) + -
Factor	(ident liczba	. ; end then do R) + - * /

Dependence Diagram for PL/0



Error Messages of PL/0 Compiler

1. Use = instead of : =
2. = must be followed by a number
3. Identifier must be followed by =
4. **const**, **var**, **procedure** must be followed by an identifier
5. Semicolon or comma missing
6. Incorrect symbol after procedure declaration
7. Statement expected
8. Incorrect symbol after statement part in block
9. Period expected
10. Semicolon between statements is missing
11. Undeclared identifier
12. Assignment to constant or procedure is not allowed

Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

Error Messages of PL/0 Compiler

13. Assignment operator : = expected
14. **call** must be followed by an identifier
15. Call of a constant or a variable is meaningless
16. **then** expected
17. Semicolon or **end** expected
18. **do** expected
19. Incorrect symbol following statement
20. Relational operator expected
21. Expression must not contain a procedure identifier
22. Right parenthesis missing
23. The preceding factor cannot be followed by this symbol
24. An expression cannot begin with this symbol
30. This number is too large

A PL/0 Machine

The PL/0 machine consists of two stores, an instruction register and three address registers.

The program store, called code, is loaded by the compiler and remains unchanged during interpretation of the code. It can then be considered as a read-only store.

The data store *S* is organized as a stack, and all arithmetic operators operate on the two elements on top of the stack, replacing their operands by a result. The top element is addressed (indexed) by **the top stack register *T***. **The instruction register *I*** contains the instruction that is currently being interpreted. **The program address register *P*** designates the next instruction to be fetched for interpretation.

Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

A PL/0 Machine

Every procedure in PL/0 may contain local variables. Since procedures may be activated recursively, storage for these local variables may not be allocated before the actual procedure call. Hence, the data segments for individual procedures are stacked up consecutively in the stack store ***S***. Since procedure activations strictly obey the **first-in-last-out** scheme, the **stack** is the appropriate storage allocation strategy. Every procedure owns some internal information of its own, namely, the program address of its call (the so-called **return address *RA***), and the address of the data segment of its caller (the so-called **dynamic link *DL***). These two addresses are needed for proper resumption of program execution after termination of the procedure.

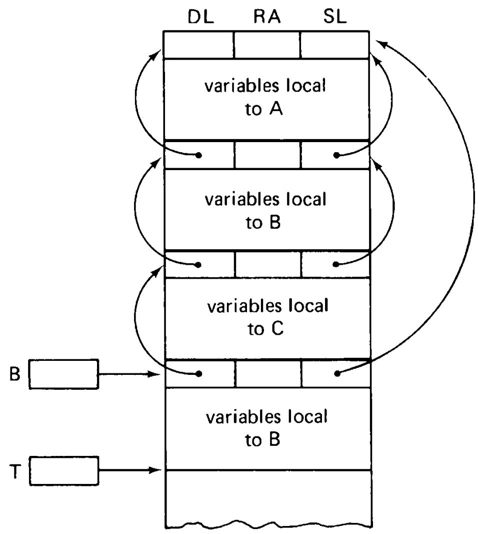
A PL/0 Machine

The address of the most recently allocated data segment, is retained in **the base address register *B***.

Since the actual allocation of storage takes place during execution (interpretation) time, the compiler cannot equip the generated code with absolute addresses. Since it can only determine the location of variables within a data segment, it is capable of providing relative addresses only. The interpreter has to add to this so-called ***displacement*** to the base address of the appropriate data segment. Therefore a second link chain of data segments is provided (the so-called **static link *SL***).

Addresses are therefore generated as pairs of numbers indicating the static level difference and the relative displacement within a data segment.

Stack of PL/0 Machine



Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

The Instruction Set of the PL/0 Machine

1. An instruction to load numbers (literals) onto the stack (LIT)
2. An instruction to fetch variables onto the top of the stack (LOD)
3. A store instruction corresponding to assignment statements (STO)
4. An introduction to activate a subroutine corresponding to a procedure call (CAL)
5. An instruction to allocate storage on the stack by incrementing the stack pointer T (INT)
6. Instructions for unconditional and conditional transfer of control, used in if- and while statements (JMP, JPC)
7. A set of arithmetic and relational operators (OPR)

Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

Code Generation

The format of instructions is determined by the need for three components, namely, an operation code ***f*** and a parameter consisting of one or two parts. In the case of operators the parameter ***a*** determines the identity of the operator; in the other cases it is either a number (LIT, INT), a program address (JMP, JPC, CAL), or a data address (LOD, STO).

f	ℓ	a

Instruction format

Code Generation

Examples of expressions in infix and postfix notations (RPN – Reverse Polish Notation)

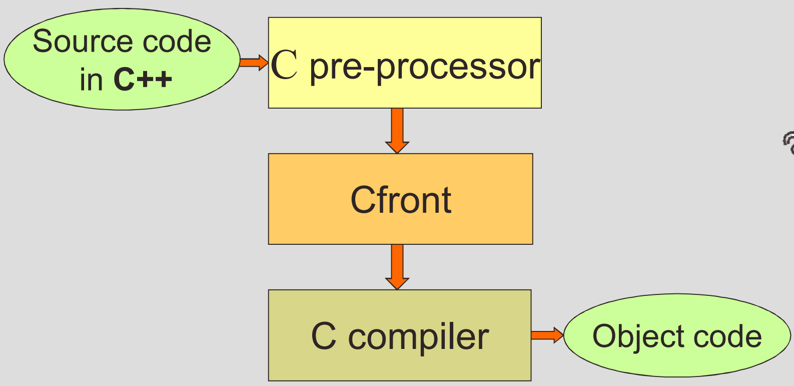
Conventional infix notation	Postfix notation (RPN)
x+y	*xy+*
(*x–y*)+*z*	*xy–z+*
x– (*y+z*)	*xyz+–*
*x**(*y+z*)**w*	*xyz+*w**

Code Generation

The patterns of code generated for the if and while statements

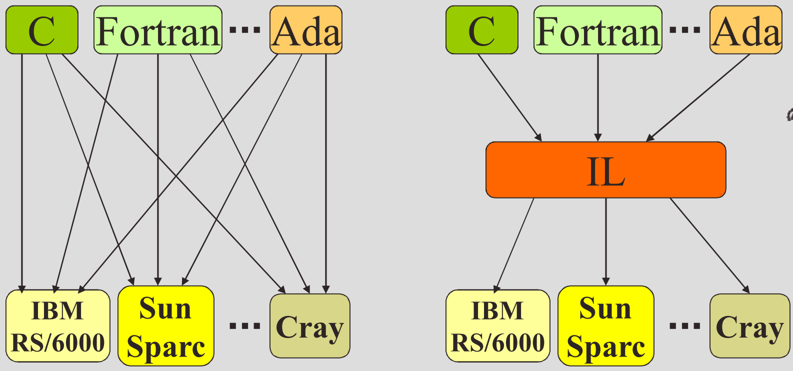
if C then S	while C do S
code for condition C JPC L1 Code for statement S LI :	LI: code for C JPC L2 code for S JMP LI L2:

Cfront Compiler



Using Cfront to translate C++ to C

Intermediate Language (IL) in Compiling Process



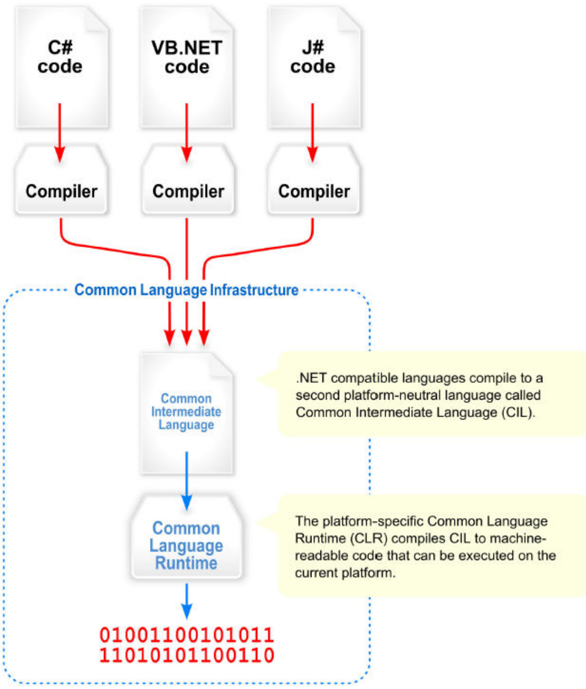
An IL can reduce the effort needed to re-source or re-target a compiler

Selected Intermediate Languages

Source Language	Intermediate Language
Pascal	Pcode
Java	Java VM
Ada	Diana

.NET platform

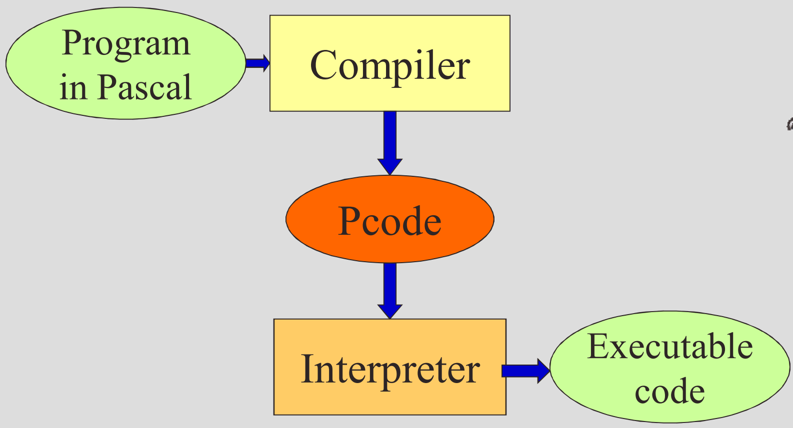
Common Intermediate Language in .NET platform



Languages supported by .NET platform

Supported programming languages		
APL	Fortran	Pascal
C++	Haskell	Perl
C#	Java Language	Python
COBOL	Microsoft JScript®	RPG
Component Pascal	Mercury	Scheme
Curriculum	Mondrian	SmallTalk
Eiffel	Oberon	Standard ML
Forth	Oz	Microsoft Visual Basic®

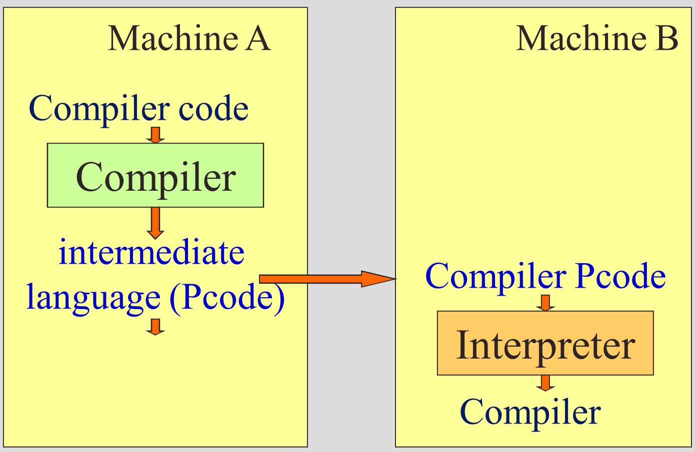
Intermediate Language (IL) in Compiling Process



Pascal compiler using Pcode IL

Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

Porting a compiler using IL



Porting Pascal compiler to another platform using Pcode intermediate language

Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz

GRAMMARS WITH TRANSLATION

- A grammar with translation is a context-free grammar, in which a set of terminal symbols is extended by additional symbols called symbols of translation.
- Symbols of translation generate an extra output statement in addition to the statement generated from the grammar.

Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

Example 1

Grammar of arithmetic expressions

$E ::= T\ E_1$

$E_1 ::= +T\ E_1 \mid -T\ E_1 \mid \varepsilon$

$T ::= F\ T_1$

$T_1 ::= *\ F\ T_1 \mid /\ F\ T_1 \mid \varepsilon$

$F ::= -\ F \mid (E) \mid id$

$id ::= a \mid b \mid c$

Lidia Jackowska-Strumillo, Computer Engineering Department Technical University of Lodz

Example 2

Grammar of arithmetic expressions extended with translation into RPN (Reverse Polish Notation)

$E ::= T\ E_1$

$E_1 ::= +T\ \{+\}\ E_1 \mid -T\ \{-\}\ E_1 \mid \varepsilon$

$T ::= F\ T_1$

$T_1 ::= *\ F\ \{*\}\ T_1 \mid /\ F\ \{/\}\ T_1 \mid \varepsilon$

$F ::= -\ F\ \{-\} \mid (E) \mid id\ \{id\}$

$id ::= a \mid b \mid c$

Lidia Jackowska-Strumiłło, Computer Engineering Department Technical University of Lodz