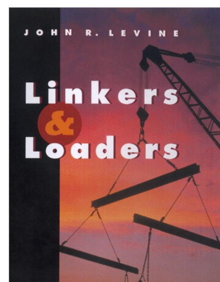
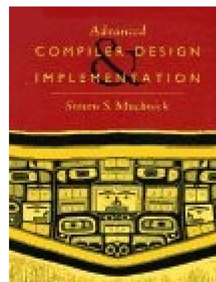
Compiler Construction

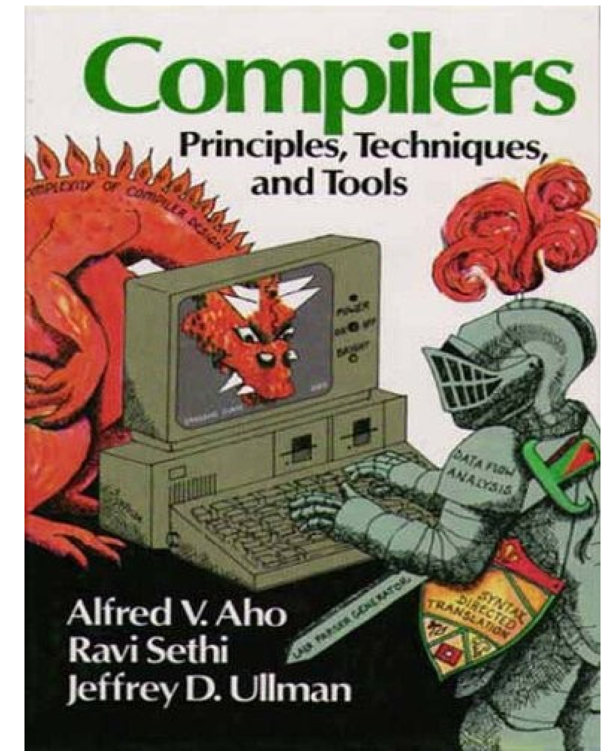
Grzegorz Jabłoński
Department of Microelectronics and Computer Science
tel. (631) 26-48
gwj@dmcs.p.lodz.pl
http://neo.dmcs.p.lodz.pl/cc

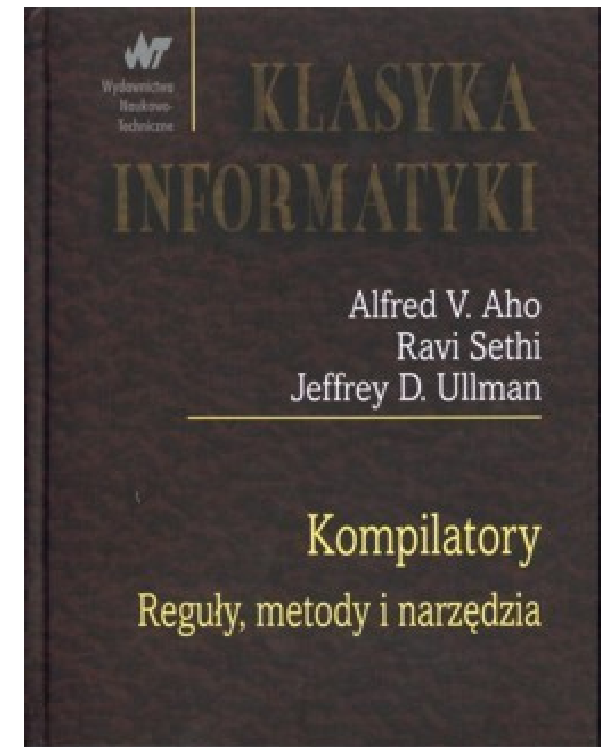
Recommended Reading

- Basic
  - A.V. Aho, R. Sethi, J. D. Ullman, "Compilers - Principles, Techniques, and Tools", Addison-Wesley 1986 (Polish edition WNT 2002)
- Auxiliary
  - John R. Levine, "Linkers and Loaders", Morgan Kaufmann Publishers 1999 (manuscript available online)
  - W. M. Waite, G. Goos, "Konstrukcja kompilatorow", WNT 1989
  - S. S. Muchnick, "Advanced Compiler Design and Implementation", Morgan Kaufmann Publishers 1997









Introduction

Course Scope

- Construction and operation of programming language compilers
  - Definitions of programming languages
  - Compilation stages
  - Design of a simple compiler for a procedural language

Why Compilers?

- Compiler
  - A program that translates from one language to another
  - It must preserve semantics of the source
  - It should create an efficient version of the target language
- In the beginning, there was machine language
  - Ugly – writing code, debugging
  - Then came textual assembly – still used on DSPs
  - High-level languages – Fortran, Pascal, C, C++
  - Machine structures became too complex and software management too difficult to continue with low-level languages

Language Families

- Imperative (or Procedural, or Assignment-Based)
- Functional (or Applicative)
- Logic (or Declarative)
- In this course, we concentrate on the first family

Imperative Languages

- Mostly influenced by the von Neumann computer architecture
- Variables model memory cells, can be assigned to, and act differently from mathematical variables
- Destructive assignment, which mimics the movement of data from memory to CPU and back
- Iteration as a means of repetition is faster than the more natural recursion, because instructions to be repeated are stored in adjacent memory cells

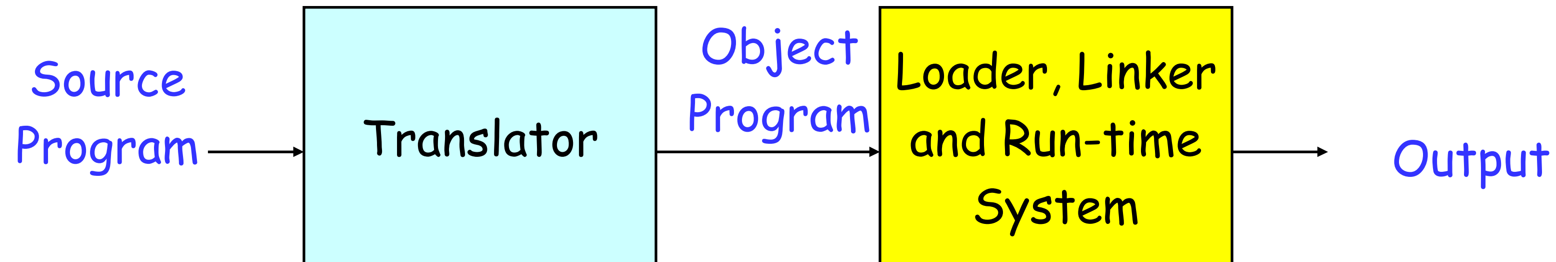
Functional Languages

- Model of computation is the lambda calculus (of function application)
- No variables or write-once variables
- No destructive assignment
- Program computes by applying a functional form to an argument
- Program are built by composing simple functions into progressively more complicated ones
- Recursion is the preferred means of repetition

Logic Languages

- Model of computation is the Post production system
- Write-once variables
- Rule-based programming
- Related to Horn logic, a subset of first-order logic
- AND and OR non-determinism can be exploited in parallel execution
- Almost unbelievably simple semantics
- Prolog is a compromise language: not a pure logic language

Compiler Structure

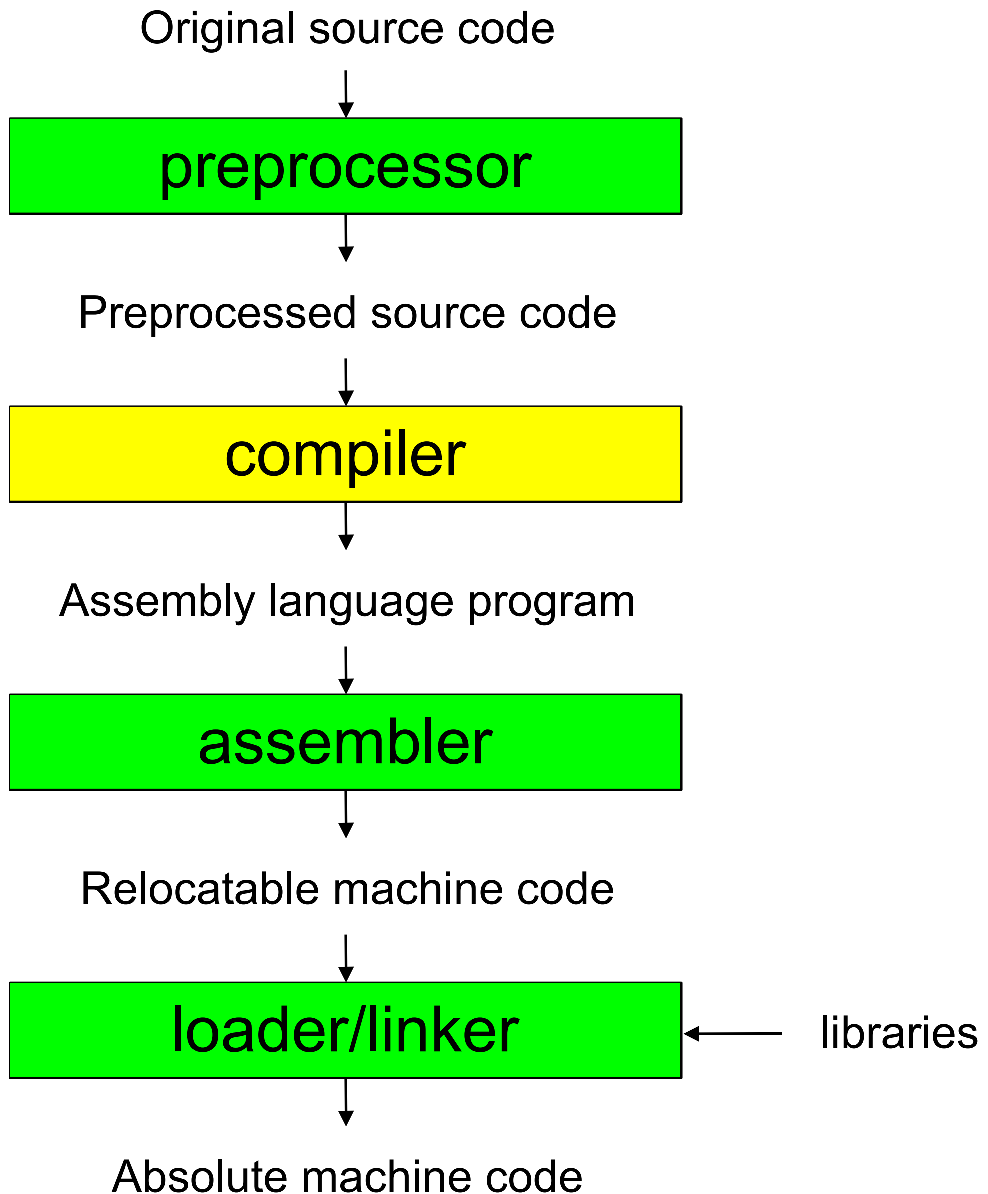


- Source language
  - Fortran, Pascal, C, C++
  - Verilog, VHDL, Tex, Html
- Target language
  - Machine code, assembly
  - High-level languages
  - IC mask layout

Compiler vs Interpreter

- Compiler
  - Translates the entire program into a form, which processor can understand
- Interpreter
  - Processes every instruction separately
- Intermediate solutions
  - Intermediate code, interpreted by a virtual processor
  - Special case: JIT (Just-in-Time) Compiler

Processing of The Program



libraries

gcc -E hello.c > hello.i

gcc -S hello.i

as hello.s -o hello.o

gcc hello.o -o hello

ld -dynamic-linker /lib/ld-linux.so.2 -o hello /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc/i586-mandrake-linux-gnu/3.4.1/crtbegin.o -L/usr/lib/gcc/i586-mandrake-linux-gnu/3.4.1 hello.o -lgcc -lc -lgcc /usr/lib/gcc/i586-mandrake-linux-gnu/3.4.1/crtend.o /usr/lib/crtn.o

Compiler Project

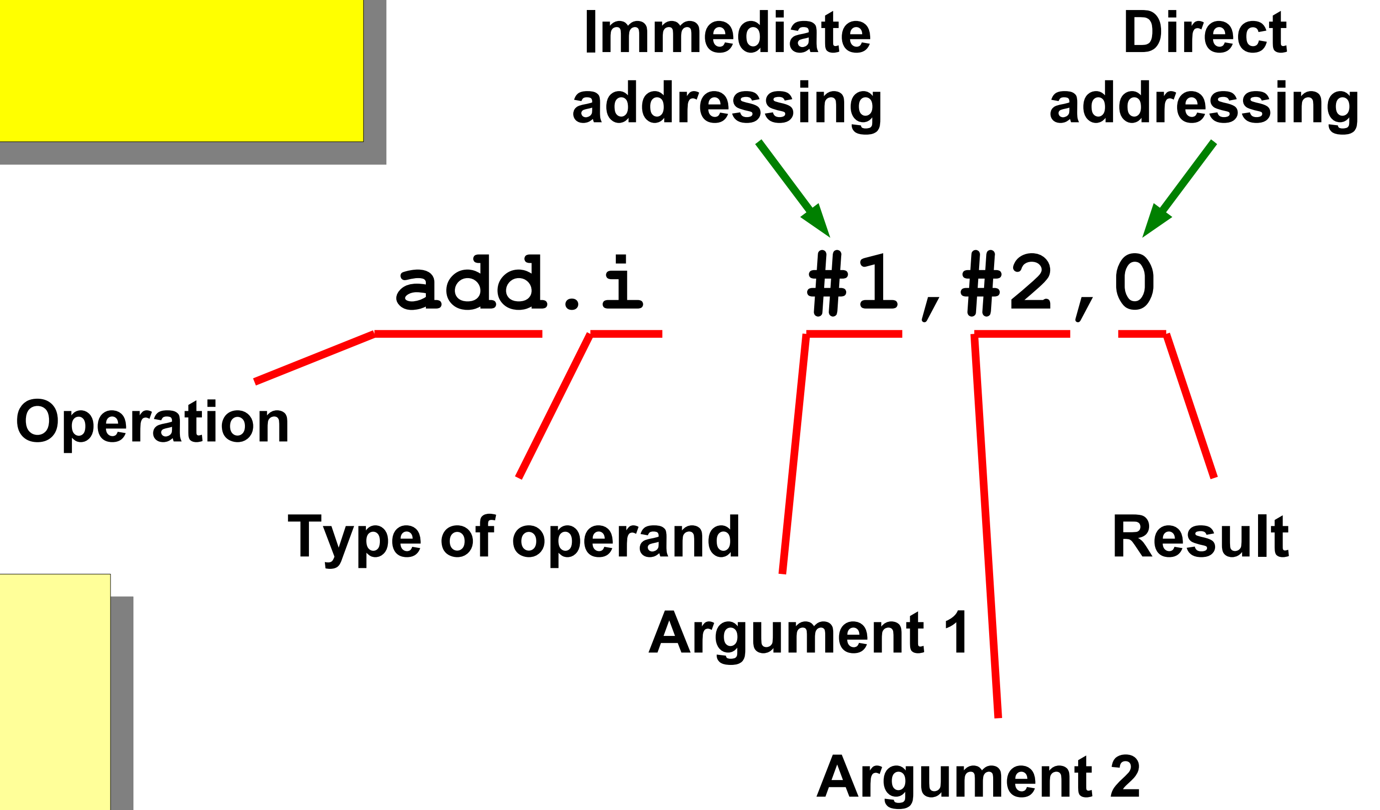
Compiler Project

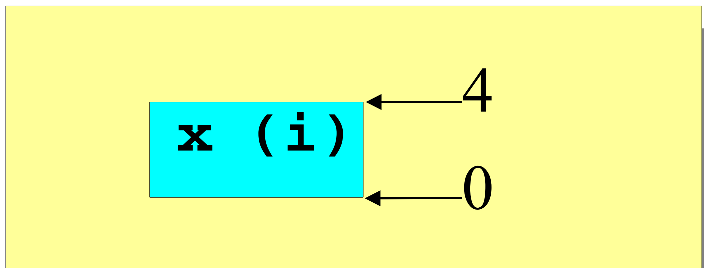
- Compiler of a language with a syntax similar to Pascal
- Target processor:
  - Harvard architecture
  - Memory-memory architecture
  - Compilation to symbolic assembly language
    - With absolute variable addresses
    - With symbolic instruction addresses
  - Assembly interpreter (virtual machine) partially implemented
- Example implementation of compiler available – executable only, no source code

Simple Program

```
program example1(input, output);
var x: integer;

begin
  x:=1+2;
  write(x)
end.
```

```
add.i    #1,#2,0
write.i  0
exit
```

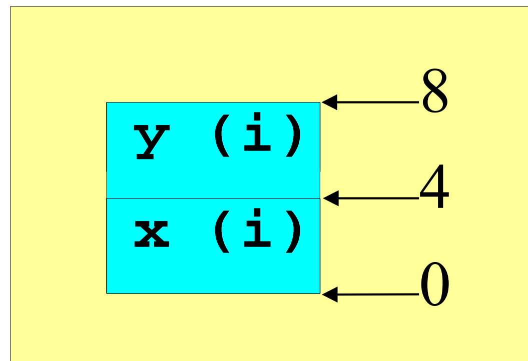




Two Variables

```
program example2(input, output);
var x,y: integer;

begin
  x:=1+2;
  y:=x+1;
  write(y)
end.
```

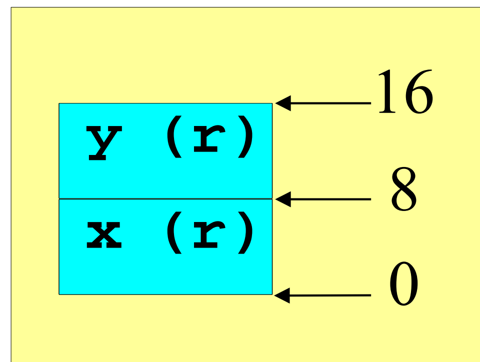
```
add.i    #1,#2,0
add.i    0,#1,4
write.i 4
exit
```



Real Variables

```
program example3(input, output);
var x,y: real;

begin
  x:=1.0;
  y:=x+2.0;
  write(y)
end.
```

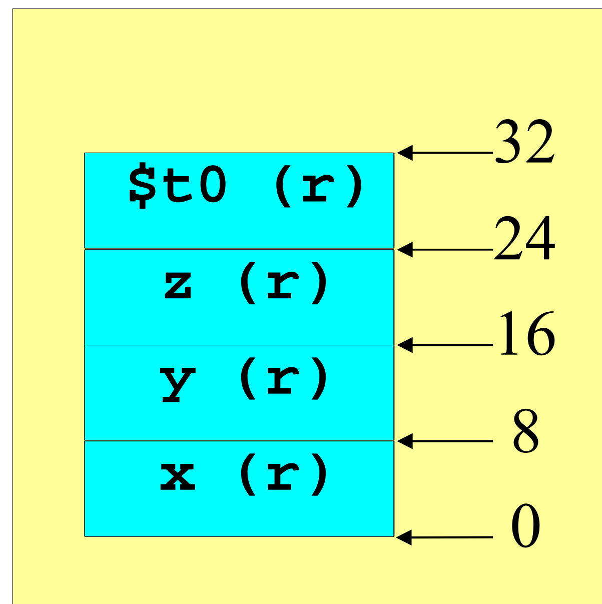
```
mov.r   #1,0
add.r   0,#2,8
write.r 8
exit
```



Complex Expression

```
program example4(input, output);
var x,y,z: real;

begin
  x:=1.0;
  y:=2.5;
  z:=x+2.0*y;
  write(z)
end.
```

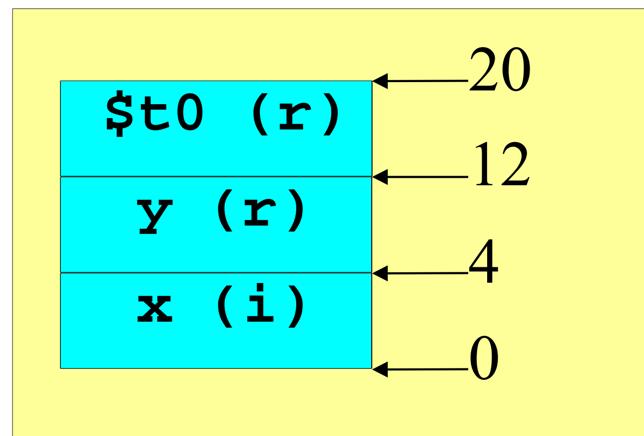
```
mov.r    #1,0
mov.r    #2.5,8
mul.r    #2,8,24
add.r    0,24,16
write.r 16
exit
```



Mixed-Mode Expressions

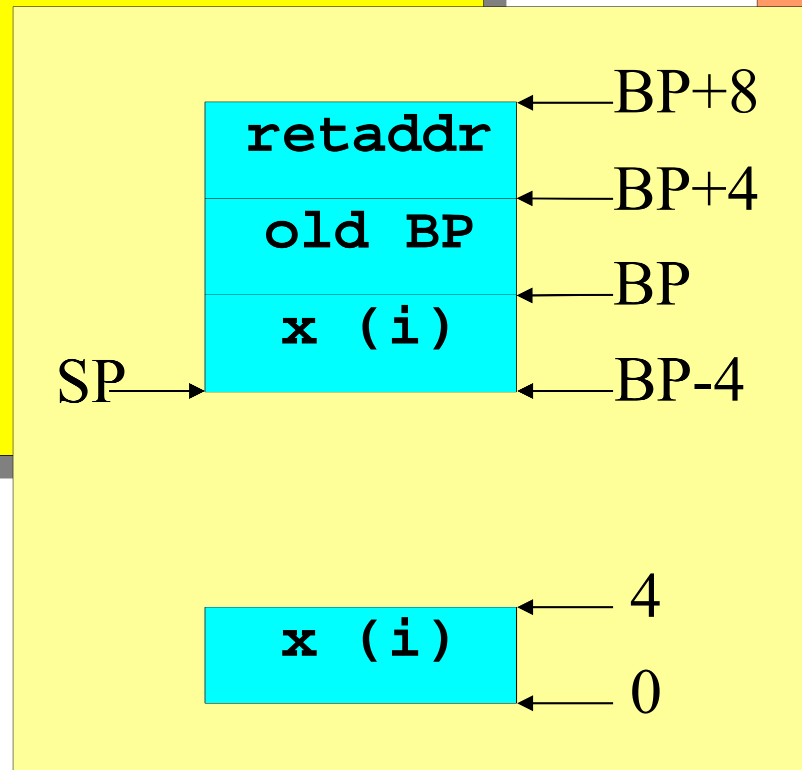
```
program example5(input, output);
var x: integer;
var y: real;

begin
  x:=1;
  y:=x+2.0;
  write(y)
end.
```

```
mov.i         #1,0
inttoreal     0,12
add.r         12,#2,4
write.r       4
exit
```



Procedures and Local Variables

```
program example6(input, output);
var x: integer;

procedure p;
var x: integer;
begin
 x:=0;
 write(x)
end;

begin
  x:=1;
  p;
  write(x)
end.
```



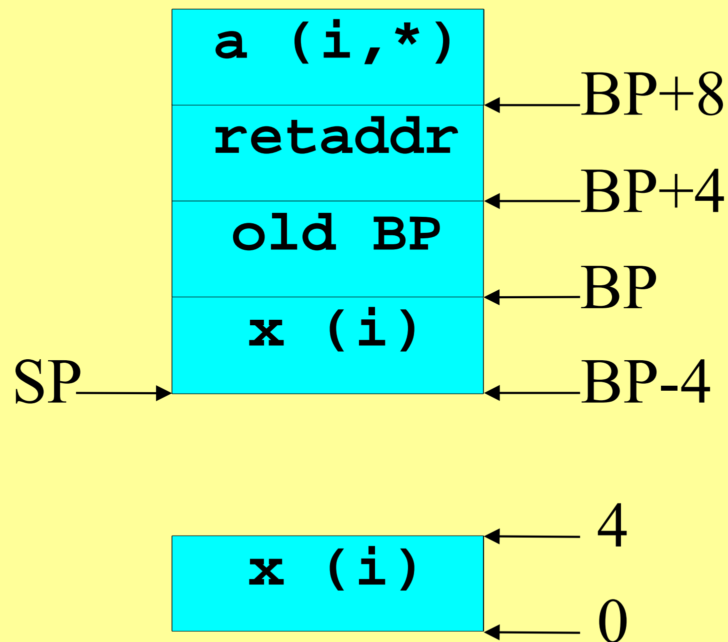

```
        jump.i   #lab0
p:
        enter.i #4
        mov.i    #0,BP-4
        write.i BP-4
        leave
        return
lab0:
        mov.i    #1,0
        call.i   #p
        write.i 0
        exit
```

```
enter.i n
    push.i #BP
    mov.i  #SP,#BP
    sub.i  #SP,#n,#SP

leave
    mov.i  #BP,#SP
    pop.i  #BP
```

Procedures and Parameters

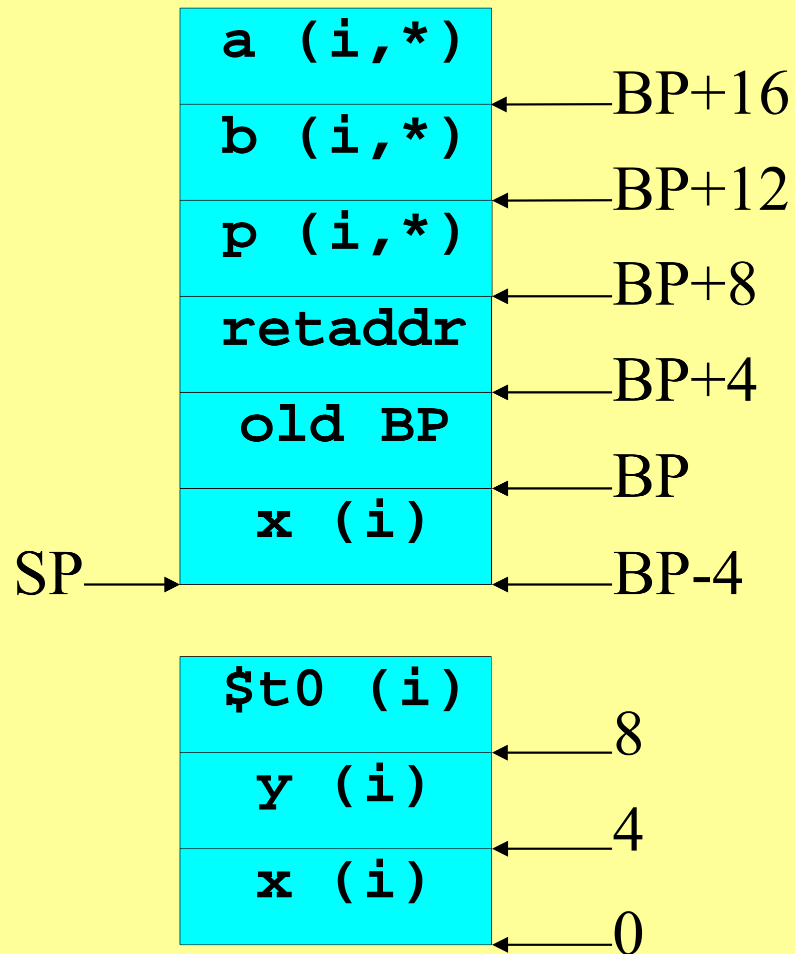
```
program example7(input, output);
var x: integer;

procedure p(a:integer);
var x: integer;
begin
 x:=a+1;
 write(x)
end;

begin
 x:=1;
 p(x);
 write(x)
end.
```

```
        jump.i   #lab0
p:
        enter.i  #4
        add.i    *BP+8,#1,BP-4
        write.i  BP-4
        leave
        return
lab0:
        mov.i    #1,0
        push.i   #0
        call.i   #p
        incsp.i  #4
        write.i  0
        exit
```



Functions

```
program example8(input, output);
var x,y: integer;
function p(a,b:integer): integer;
var x: integer;
begin
 x:=a+1;
 p:=x+b
end;

begin
 x:=1;
 y:=2;
 x:=p(x,y);
 write(x)
end.
```

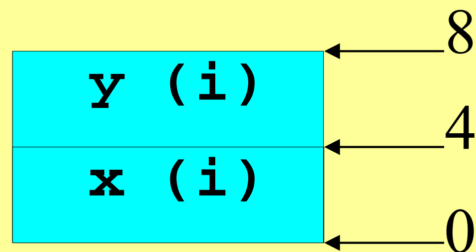



```
    jump.i   #lab0
p:
    enter.i #4
    add.i    *BP+16,#1,BP-4
    add.i    BP-4,*BP+12,*BP+8
    leave
    return
lab0:
    mov.i    #1,0
    mov.i    #2,4
    push.i   #0
    push.i   #4
    push.i   #8
    call.i   #p
    incsp.i #12
    mov.i    8,0
    write.i 0
    exit
```

Conditional Instructions

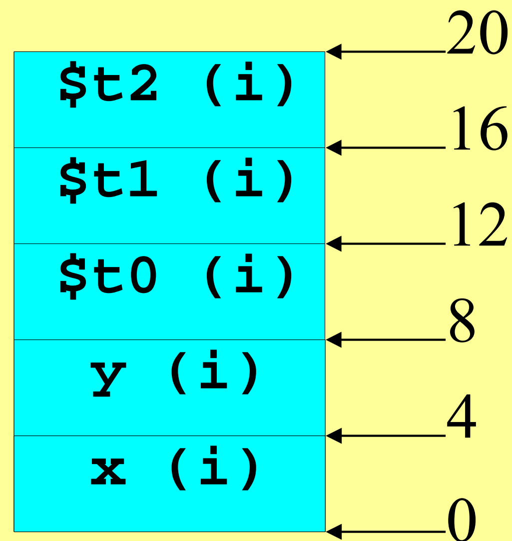
```
program example9(input, output);
var x,y: integer;

begin
 read (x,y);
 if x>y then
   write (x)
 else
   write (y)
end.
```

```
        read.i   0
        read.i   4
        jg.i     0,4,#then
        write.i 4
        jump.i   #endif
then:
        write.i 0
endif:
        exit
```



Complex Conditional Instructions

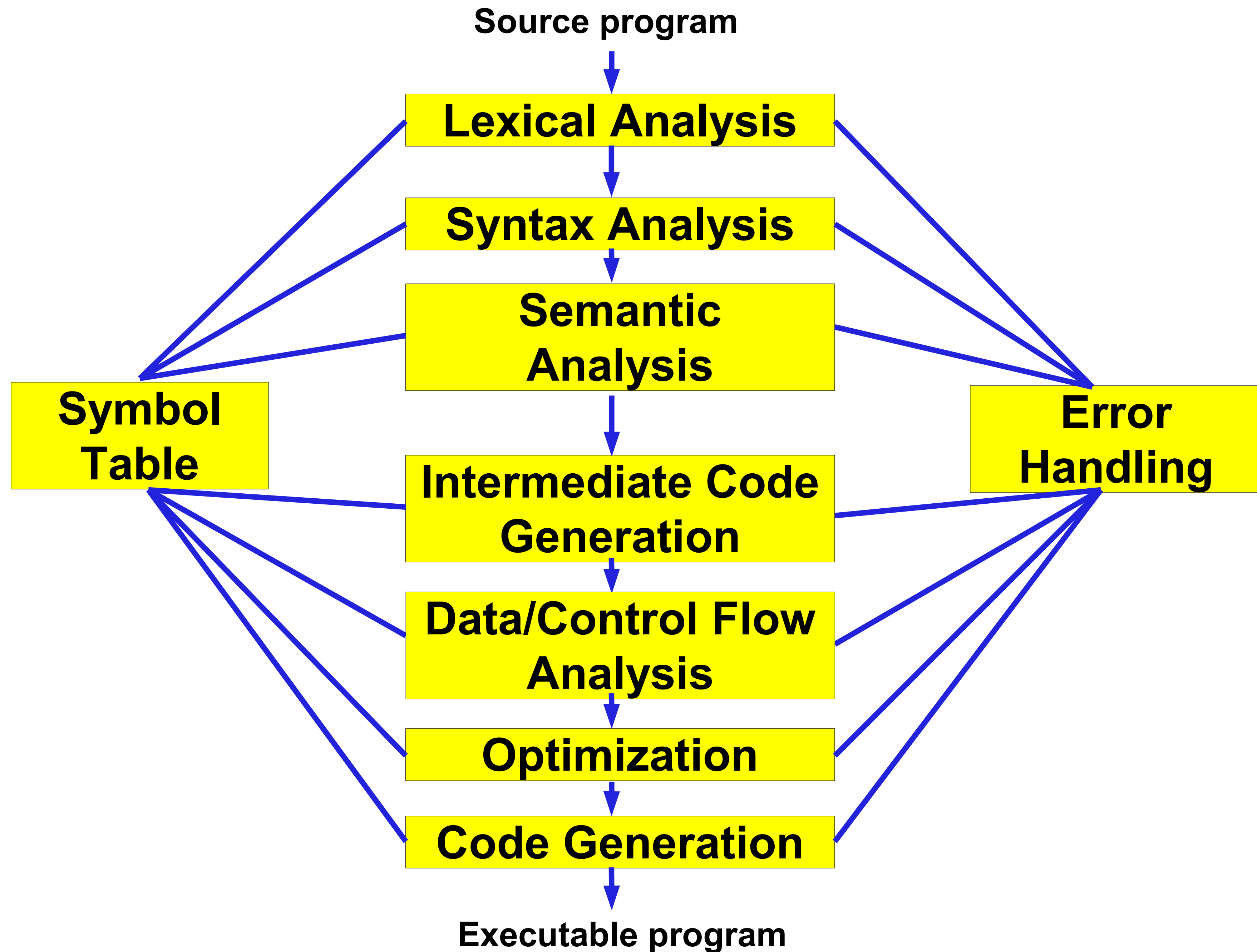
```
program example10(input, output);
var x,y: integer;

begin
 read (x,y);
 if (x>y) and (x>0) then
   write (x)
 else
   write (y)
end.
```




```
        read.i   0
        read.i   4
        jg.i     0,4,#lab1
        mov.i    #0,8
        jump.i   #lab2
lab1:   mov.i    #1,8
lab2:   jg.i     0,#0,#lab3
        mov.i    #0,12
        jump.i   #lab4
lab3:   mov.i    #1,12
lab4:   and.i    8,12,16
        je.i     16,#0,#lab5
        write.i 0
        jump.i   #lab6
lab5:   write.i 4
lab6:   exit
```

Compiler Overview

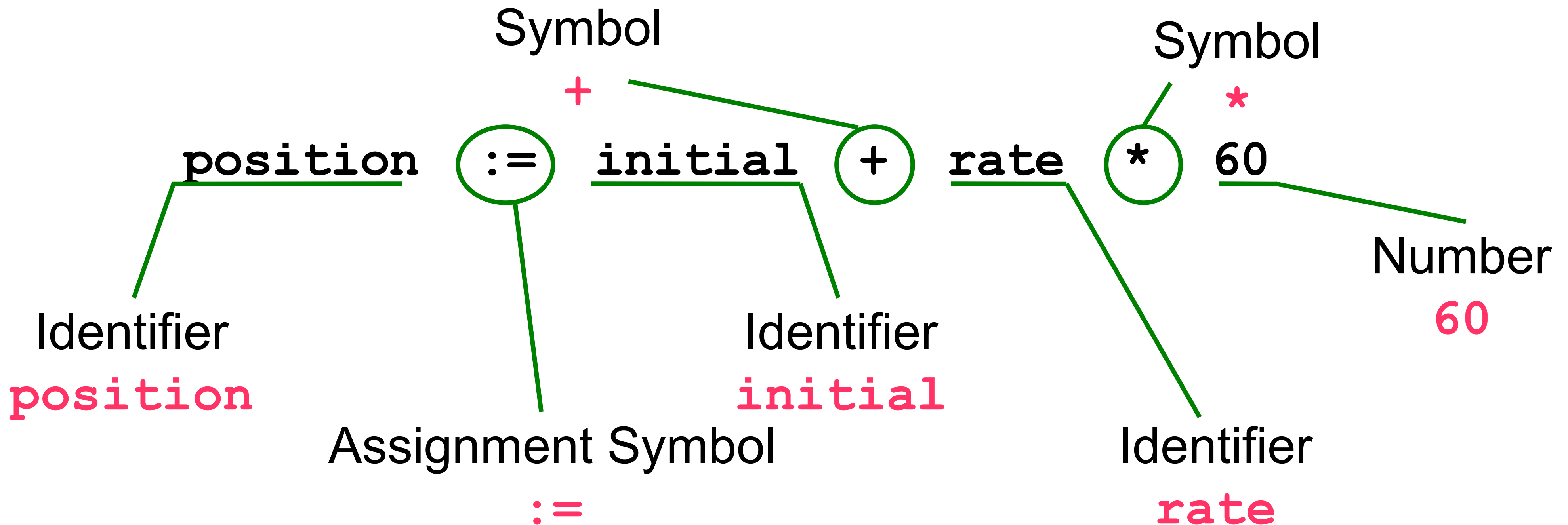
Compilation Stages



Lexical Analyser (Scanner)

- Extracts and identifies lowest level lexical elements from a source stream
  - Reserved words: for, if, switch
  - Identifiers: “i”, “j”, “table”
  - Constants: 3.14159, 17, “%d\n”
  - Punctuation symbols: “(“, “)”, “,”, “+”
- Removes non-grammatical elements from the stream – i.e. spaces, comments
- Implemented with a Finite State Automata (FSA)
  - Set of states – partial inputs
  - Transition functions to move between states

Lexical Analysis



Symbol Table

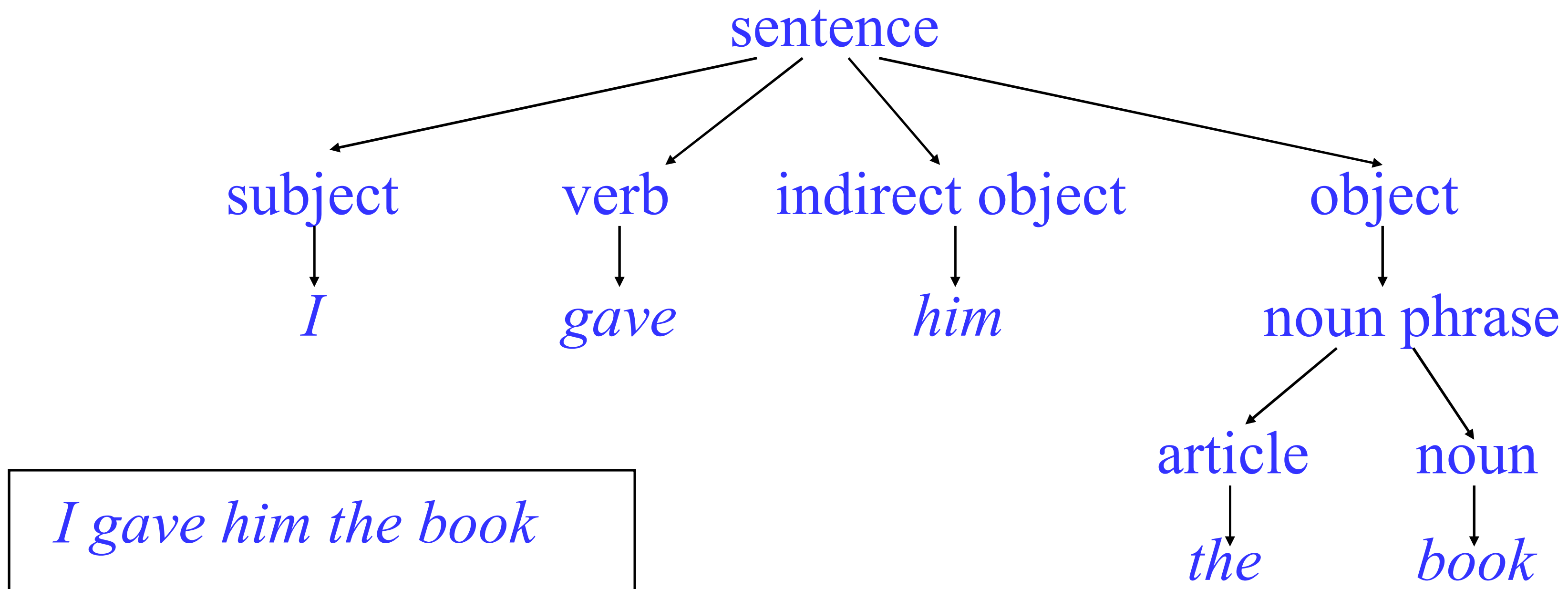
Identifier	Type	Address
position	real	0
initial	real	8
rate	real	16

Lex/Flex

- Automatic generation of scanners
  - Hand-coded ones are faster
  - But tedious to write, and error prone!
- Lex/Flex
  - Given a specification of regular expressions
  - Generate a table driven FSA
  - Output is a C program that you compile to produce your scanner

Parsing Analogy

- Syntax analysis for natural languages
  - Recognize whether a sentence is grammatically correct
  - Identify the function of each word

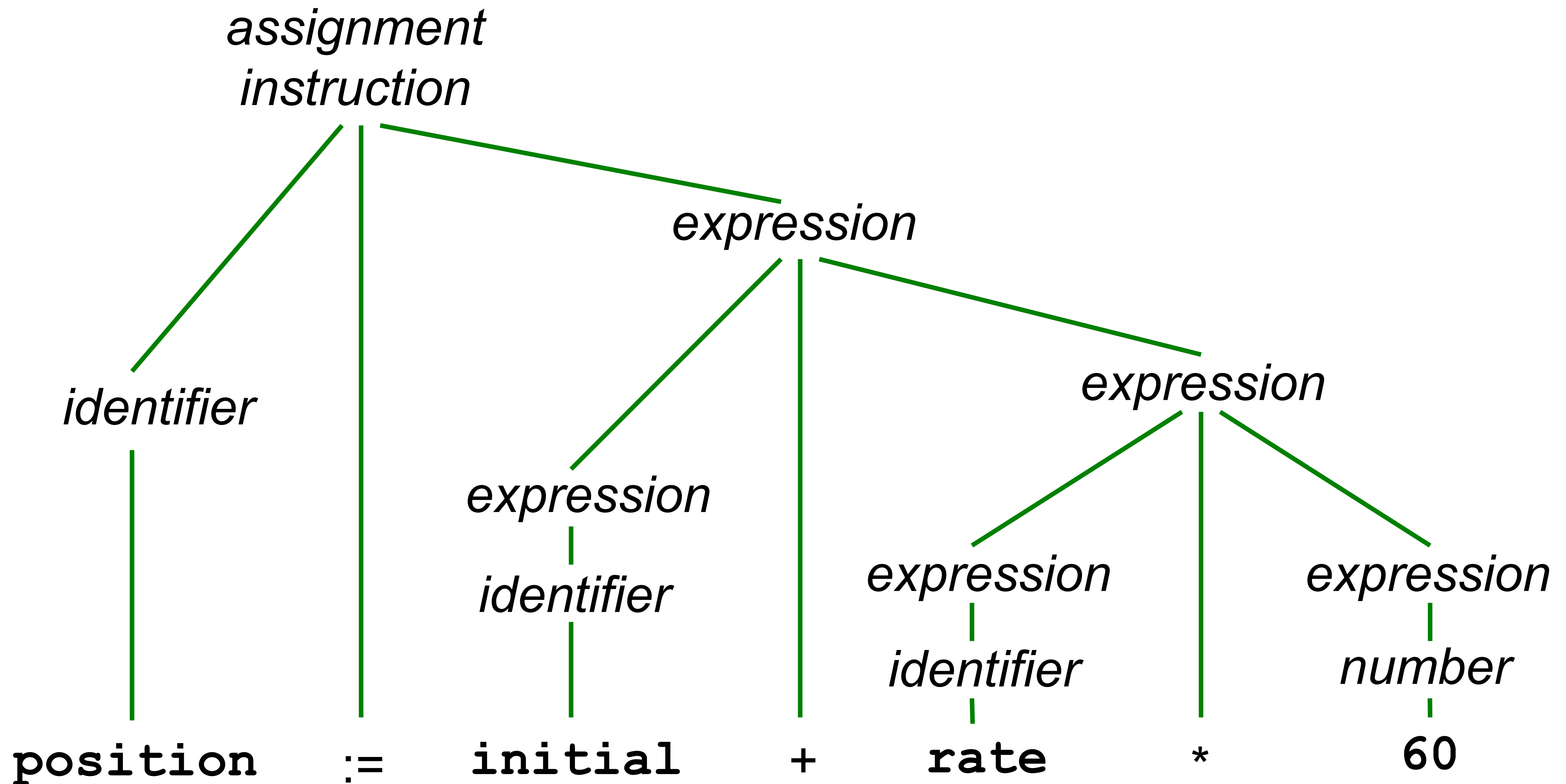


Syntax Analyzer (Parser)

- Check input stream for syntactic correctness
  - Framework for subsequent semantic processing
  - Implemented as a push down automaton (PDA)
- Lots of variations
  - Hand coded, recursive descent?
  - Table driven (top-down or bottom-up)
  - For any non-trivial language, writing a correct parser is a challenge
- Yacc (yet another compiler compiler)/bison
  - Given a context free grammar
  - Generate a parser for that language (again a C program)

Syntax Analysis

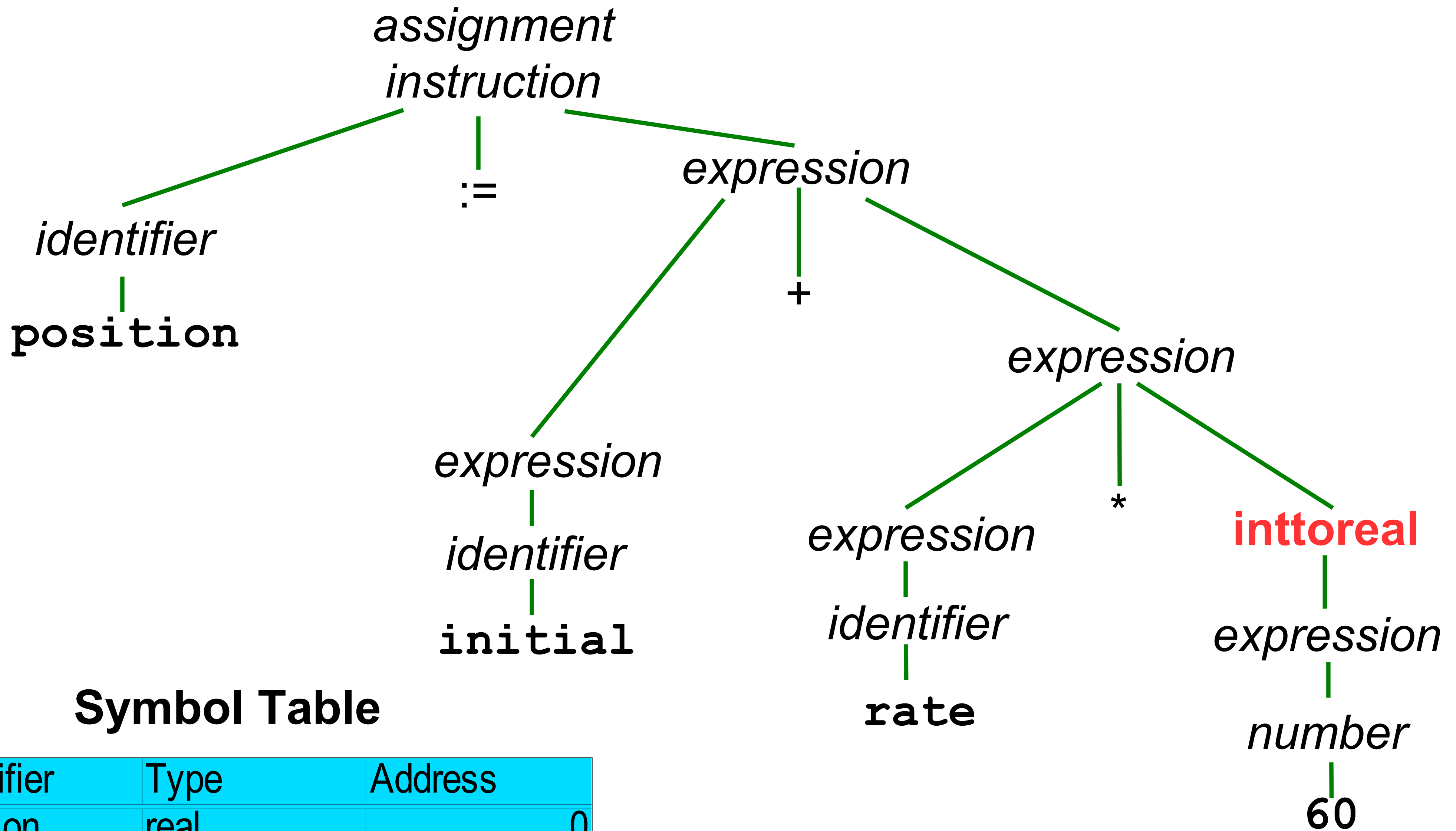
```
assignment_instruction -> identifier ':=' expression
            expression -> number
            expression -> identifier
            expression -> expression '+' expression
            expression -> expression '*' expression
```



Semantic Analysis

- Several distinct actions to perform
  - Check definition of identifiers, ascertain that the usage is correct
  - Disambiguate overloaded operators
  - Insert type conversions when necessary

Semantic Analysis



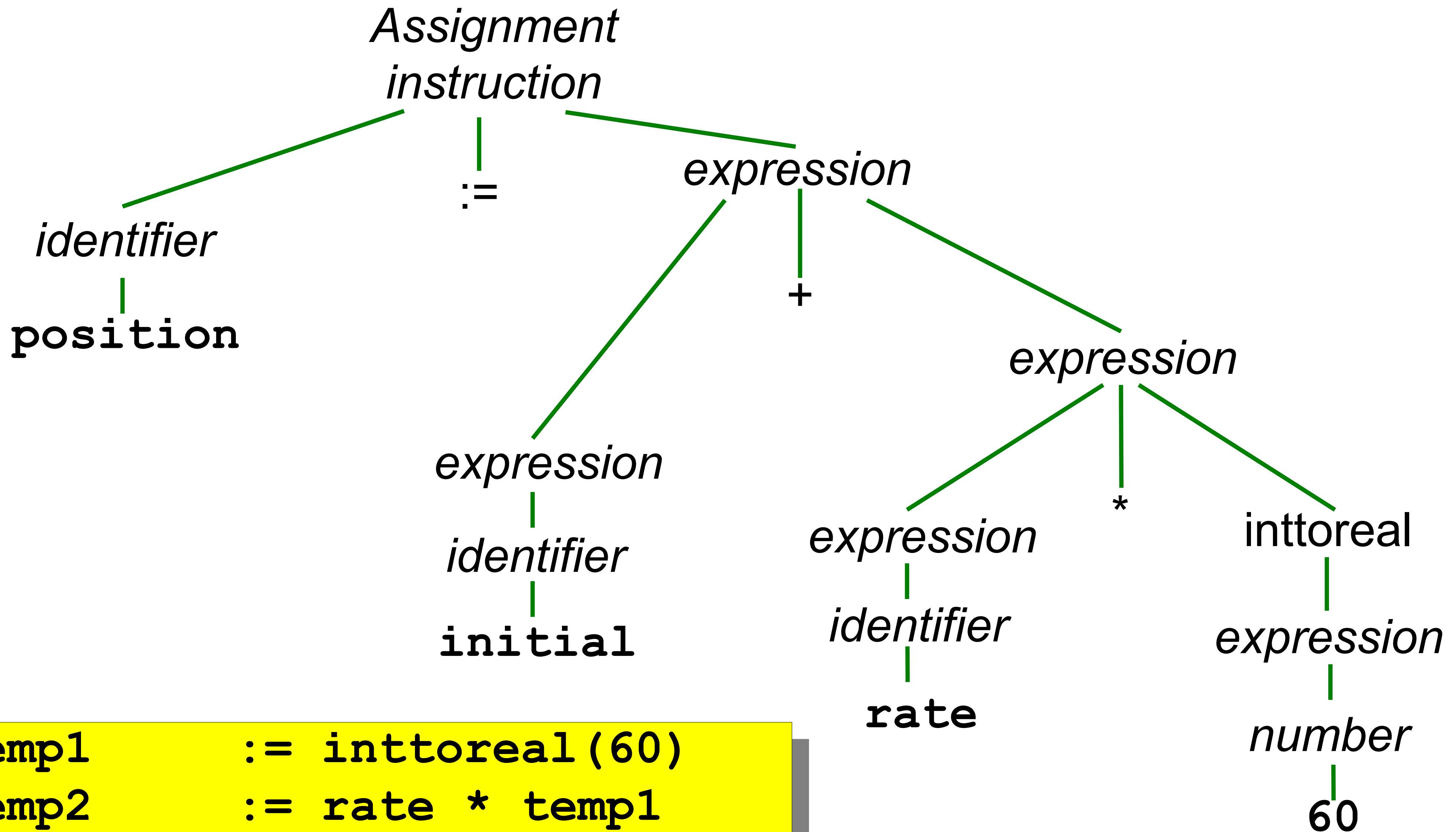
Symbol Table

Identifier	Type	Address
position	real	0
initial	real	8
rate	real	16

Intermediate Code Generation

- Converting a tree to linear form
  - Sequence of statements
- Usually IR is a three address code
  - a = b OP c
  - Originally, because instruction had at most 3 addresses or operands
    - This is not enforced today, ie MAC: a = b * c + d
- May have fewer operands
- Also called quadruples: (a,b,c,OP)

Intermediate Code Generation




```
temp1      := inttoreal(60)
temp2      := rate * temp1
temp3      := initial + temp2
position   := temp3
```

Dataflow and Control Flow Analysis

- Provide the necessary information about variable usage and execution behavior to determine when a transformation is legal/illegal
- Dataflow analysis
  - Identify when variables contain “interesting” values
  - Which instructions created values or consume values
  - DEF, USE, GEN, KILL
- Control flow analysis
  - Execution behavior caused by control statements
  - If’s, for/while loops, goto’s
  - Control flow graph

Optimization

- How to make the code go faster
- Classical optimizations
  - Dead code elimination – remove useless code
  - Common subexpression elimination – recomputing the same thing multiple times
- Machine independent (classical)
  - Focus of this class
  - Useful for almost all architectures
- Machine dependent
  - Depends on processor architecture
  - Memory system, branches, dependences

Optimization and Code Generation

```
temp1      := inttoreal(60)
temp2      := rate * temp1
temp3      := initial + temp2
position   := temp3
```

```
temp1      := rate * 60.0
position   := initial + temp1
```

```
MOVF   16, R2
MULF   #60.0, R2
MOVF   8, R1
ADDF   R2,R1
MOVF   R1,0
```

Symbol Table

Identifier	Type	Address
position	real	0
initial	real	8
rate	real	16
temp1	real	24

A Look At Simple Compiler

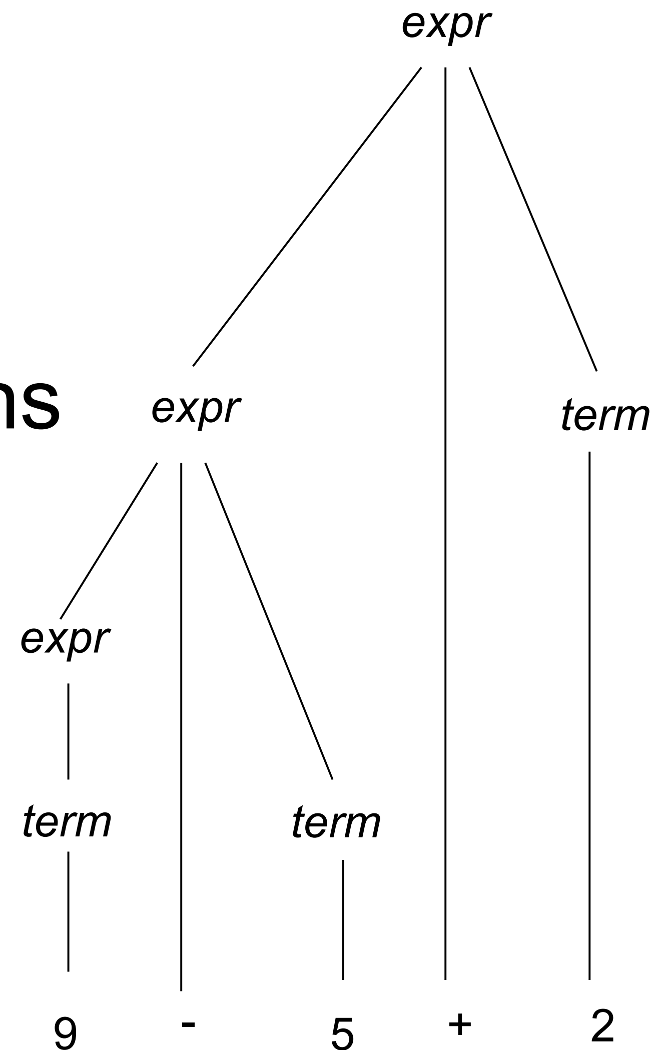
Context-Free Grammars

- Consist of 4 components:
  - Terminal symbols = token or $\in$
  - Non-terminal symbols = syntactic variables
  - Start symbol S = special non-terminal
  - Productions of the form LHS$\rightarrow$RHS
    - LHS = single non-terminal
    - RHS = string of terminals and non-terminals
    - Specify how non-terminals may be expanded

Parse Trees

- Nonterminals are nodes
- Terminals are leaves
- LHS - parent, RHS - children
- Example: arithmetic expressions with addition and subtraction

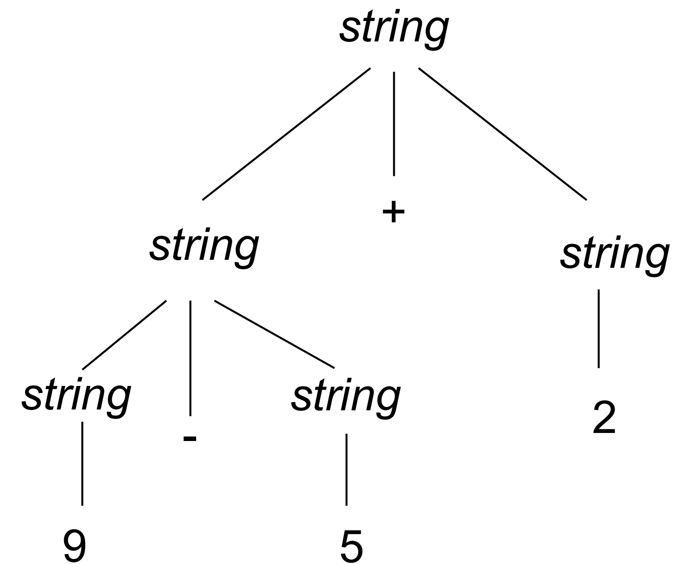
```
expr -> expr + term
expr -> expr - term
expr -> term
term -> 0
term -> 1
...
term -> 9
```

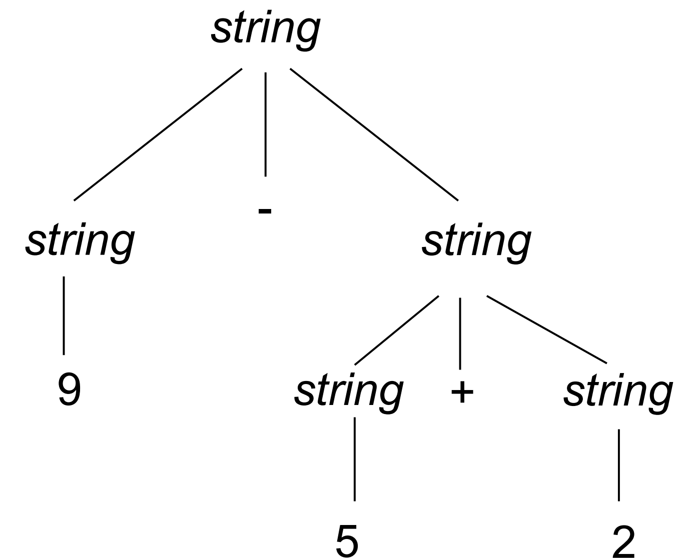


Ambiguity

- Not every grammar is unambiguous
- One string can have several parse trees

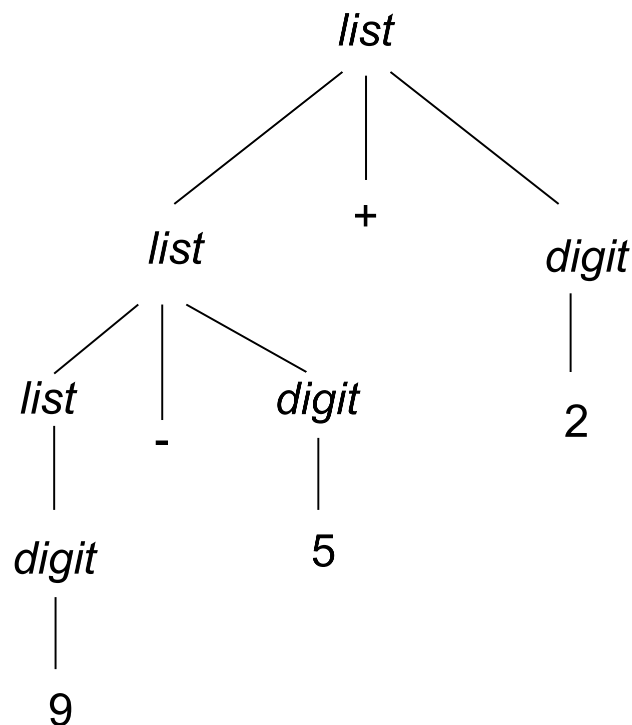
```
string -> string + string
string -> string - string
string -> 0
string -> 1
...
string -> 9
```

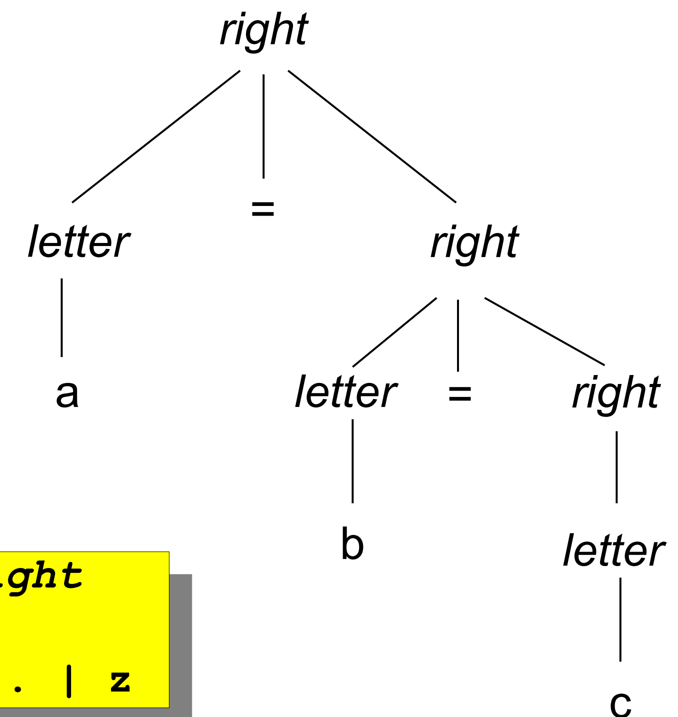




Associativity of Operators

- An operator is called left-associative, if in a situation, when operand has equal priority operators at both sides, left operator is evaluated first

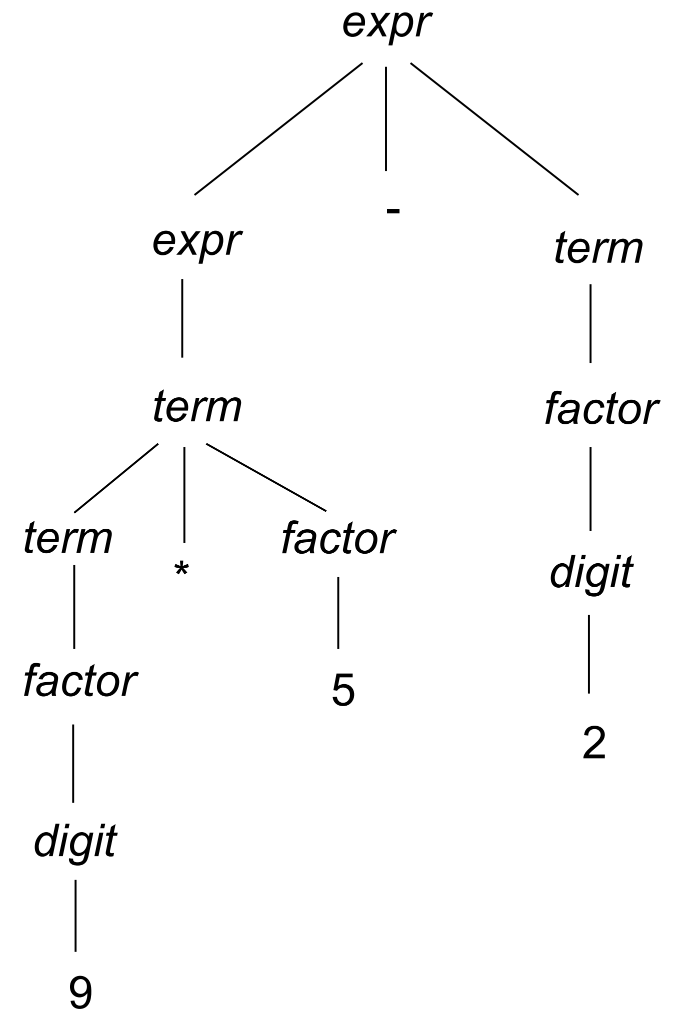





```
right -> letter = right
right -> letter
letter -> a | b | ... | z
```

Operator Priority

```
expr -> expr + term | expr - term | term
term -> term * factor | term / factor | factor
factor -> digit | ( expr )
digit -> 0 | 1 | ... | 9
```

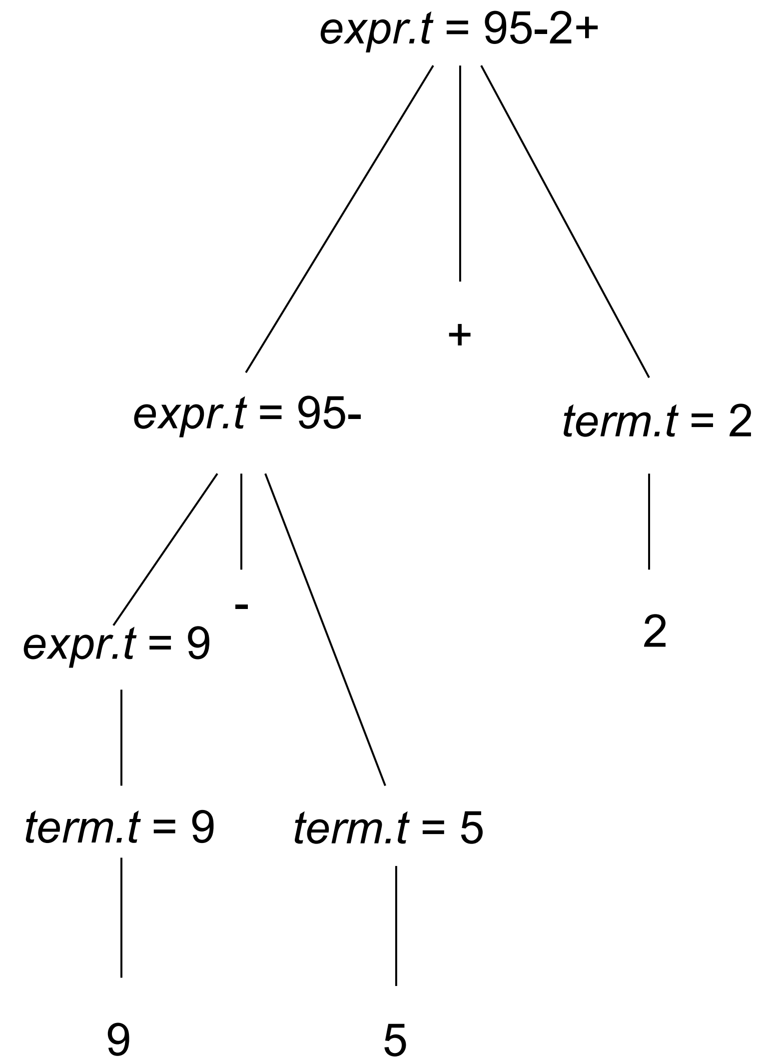


Postfix Notation

- The postfix notation for an expression *E* can be defined inductively as follows:
  - If *E* is a variable or constant, then the postfix notation for *E* is *E* itself
  - If *E* is an expression of the form E_1 op E_2, where *op* is any binary operator, then the postfix notation for *E* is $E_1' \, E_2' \, op$, where E_1' and E_2' are the postfix notations for E_1 and E_2 ,respectively
  - If *E* is an expression of the form (E_1), then the postfix notation for E_1 is also the postfix notation for *E*

Syntax-Directed Definitions

Production	Semantic Rule
expr → *expr*$_1$ + *term*	*expr.t* := *expr1.t* || *term.t* || '+'
expr → *expr*$_1$ – *term*	*expr.t* := *expr1.t* || *term.t* || '-'
expr → *term*	*expr.t* := *term.t*
term → 0	*term.t* := '0'
term → 1	*term.t* := '1'
…	…
term → 9	*term.t* := '9'

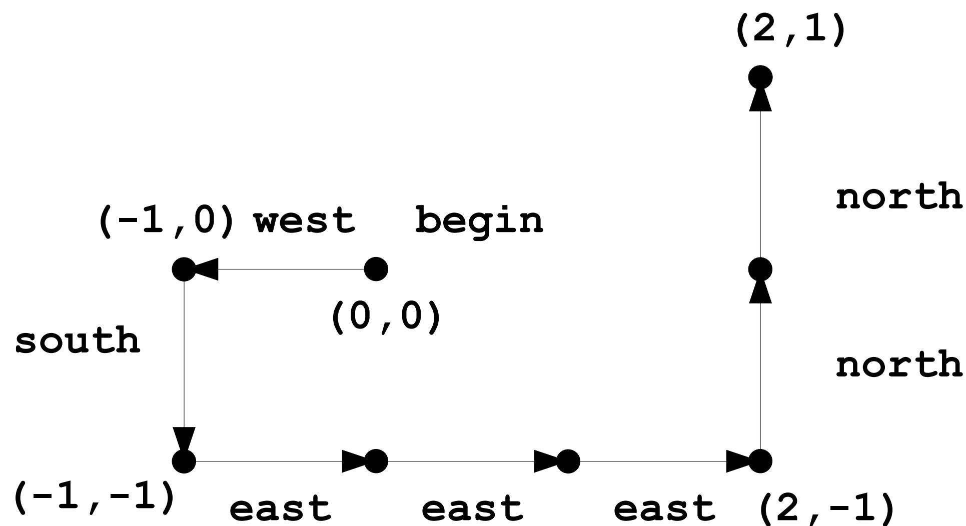


- Simple compiler – transforms infix notation into postfix notation (reverse Polish notation)
- 9-5+2 -> 95-2+

Mobile Robot Control Language

```
seq -> seq instr | begin
instr -> east | north | west | south
```

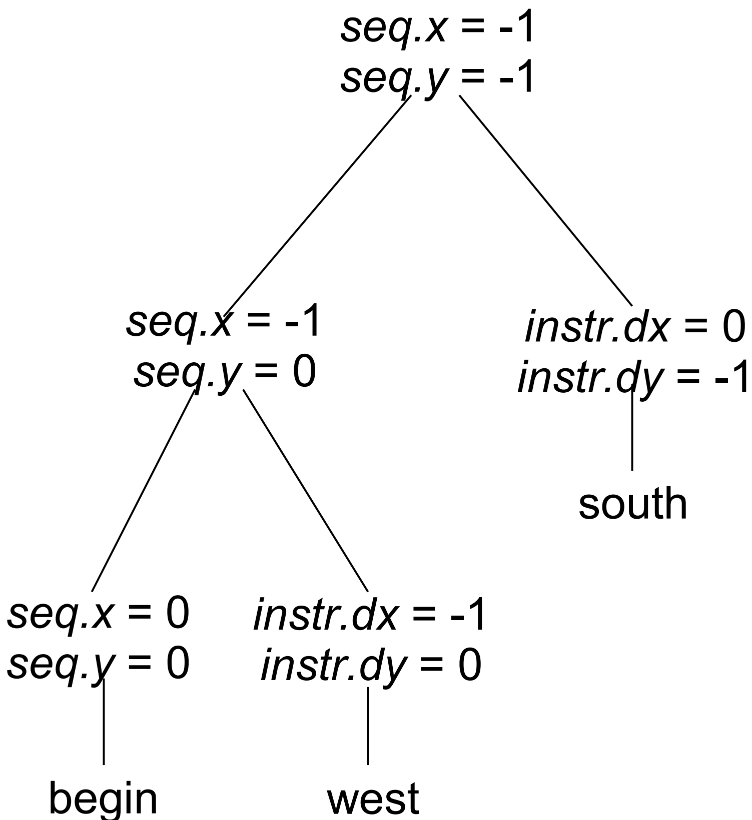
`begin west south east east east north north`



Production	Semantic Rule
seq → **begin**	*seq.x* := 0 *seq.y* := 0
seq → seq_1 *instr*	*seq.x* := seq_1*.x* + *instr.dx* *seq.y* := seq_1*.y* + *instr.dy*
instr → **east**	*instr.dx* := 1 *instr.dy* := 0
instr → **north**	*instr.dx* := 0 *instr.dy* := 1
instr → **west**	*instr.dx* := -1 *instr.dy* := 0
instr → **south**	*instr.dx* := 0 *instr.dy* := -1

Mobile Robot Control Language

```
begin west south
```



Production	Semantic Rule
$seq \rightarrow$ **begin**	$seq.x := 0$ $seq.y := 0$
$seq \rightarrow seq_1\ instr$	$seq.x := seq_1.x + instr.dx$ $seq.y := seq_1.y + instr.dy$
$instr \rightarrow$ **east**	$instr.dx := 1$ $instr.dy := 0$
$instr \rightarrow$ **north**	$instr.dx := 0$ $instr.dy := 1$
$instr \rightarrow$ **west**	$instr.dx := -1$ $instr.dy := 0$
$instr \rightarrow$ **south**	$instr.dx := 0$ $instr.dy := -1$

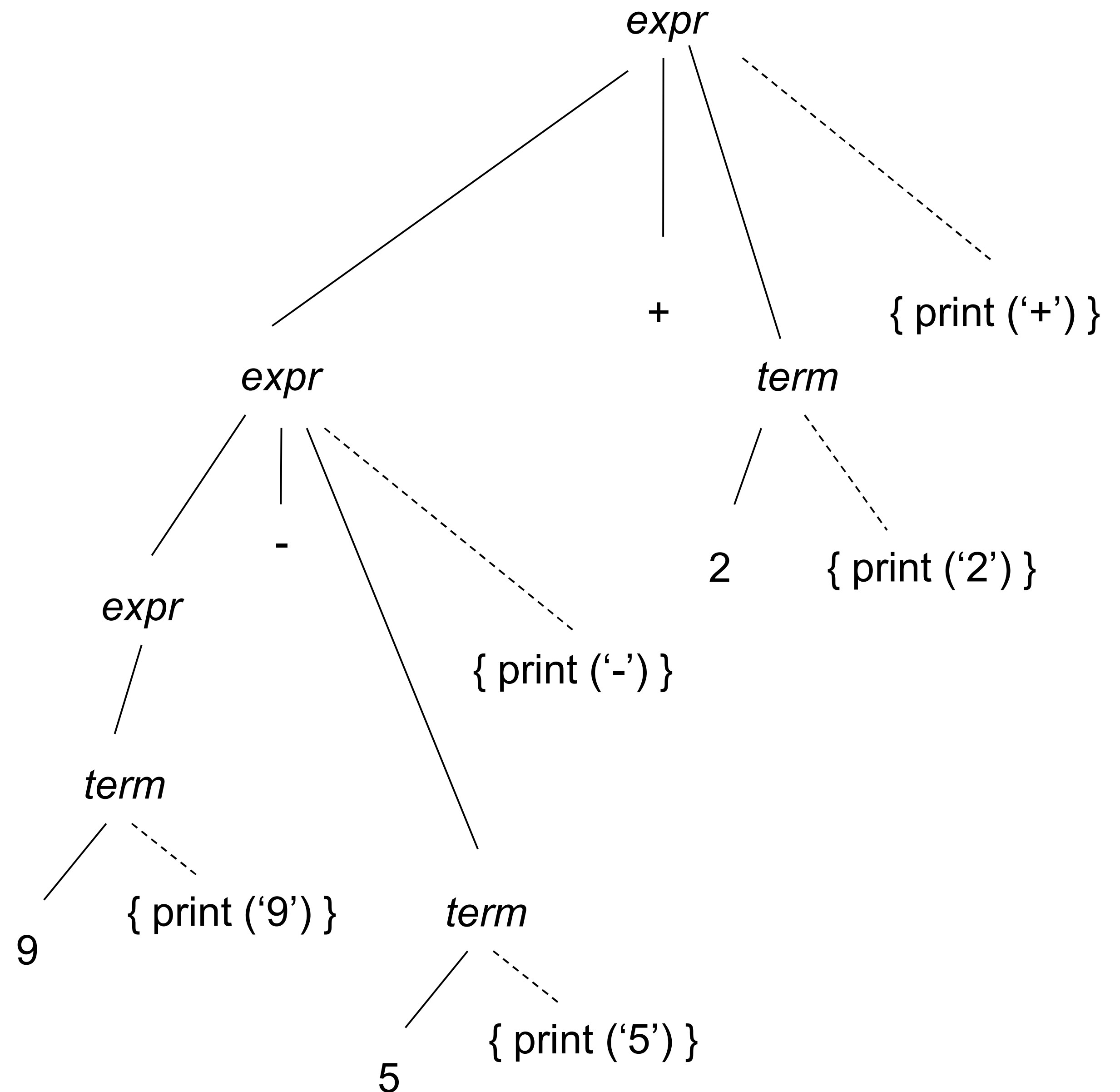
Depth-First Traversals

- A syntax-directed definition does not impose any specific order of evaluation on a parse tree
  - any evaluation order taking into account dependencies among attributes is acceptable
- In examples presented so far the attributes could be evaluated in a depth-first order

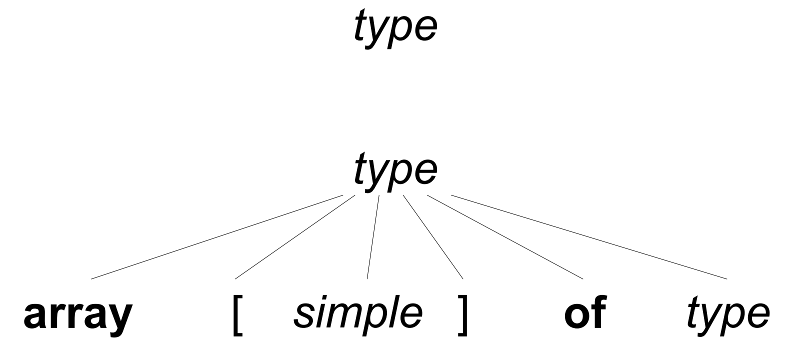
```
procedure visit(n: node);
begin
  for each child m of n, from left to right do
    visit(m);
  evaluate semantic rules at node n
end
```

Translation Schemes

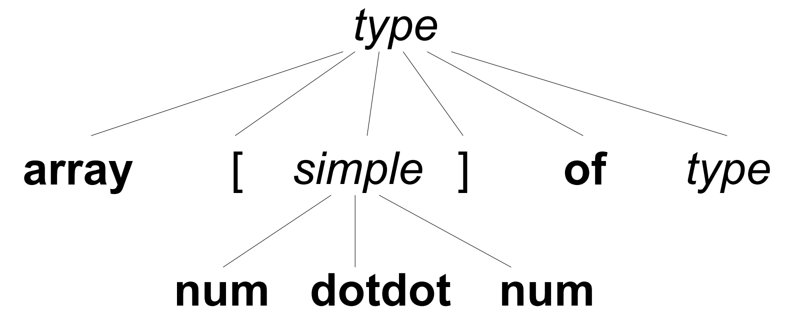
$expr \rightarrow expr_1$ + $term$ { print('+') }
$expr \rightarrow expr_1$ – $term$ { print('-') }
$expr \rightarrow term$
$term \rightarrow$ 0 { print('0') }
$term \rightarrow$ 1 { print('1') }
…
$term \rightarrow$ 9 { print('9') }



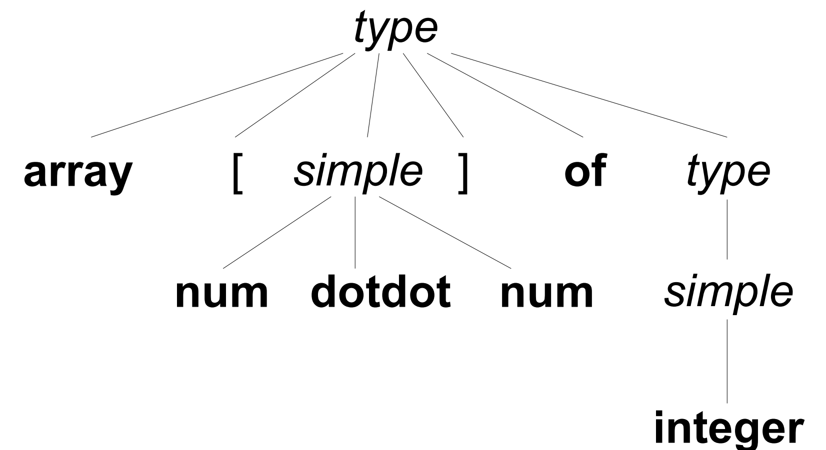
Top-Down Parsing



array [num dotdot num] of integer




```
type    ->  simple
          | ^id
          | array [ simple ] of type
simple ->  integer
          | char
          | num dotdot num
```



Predictive Parsing

```
procedure match(t: token)
begin
     if lookahead = t then
          lookahead := nexttoken
     else error
end;

procedure type;
begin
     if lookahead is in { integer, char, num } then
           simple
     else if lookahead = '^' then begin
           match('^'); match(id)
     end
     else if lookahead = array then begin
           match(array); match('['); simple; match(']'); match(of); type
     end
     else error
end;

procedure simple;
begin
     if lookahead = integer then
            match(integer)
     else if lookahead = char then
            match(char)
     else if lookahead = num then begin
            match(num); match(dotdot); match(num)
     end
     else error
end;
```

```
type   ->  simple
         | ^id
         | array [ simple ] of type
simple ->  integer
         | char
         | num dotdot num
```

FIRST(*simple*) = { **integer**, **char**, **num** }
FIRST(^**id**) = {^}
FIRST(**array** [*simple*] **of type**) = { **array** }

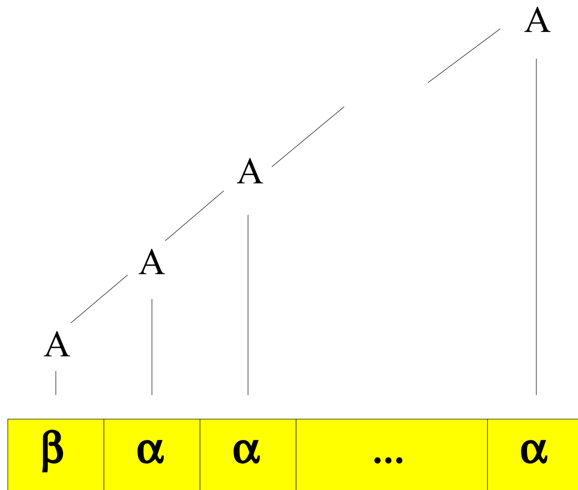
Empty Productions

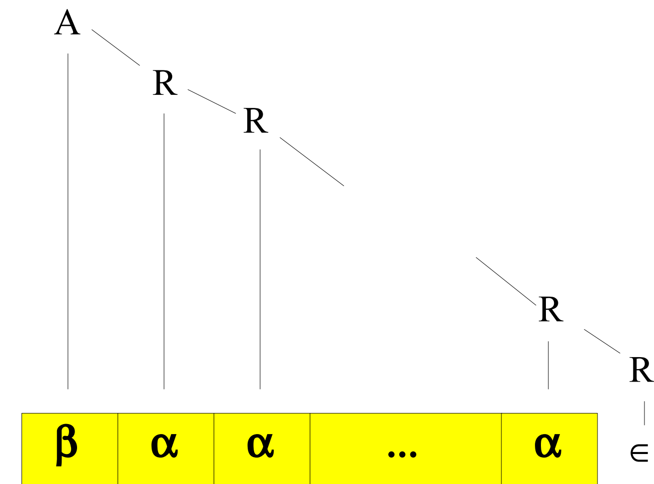
stmt → **begin** *opt_stmts* **end**
opt_stmts → *stmt_list* | ∈

- Empty production is a default when no other production can be used

Left Recursion

```
expr -> expr + term
```





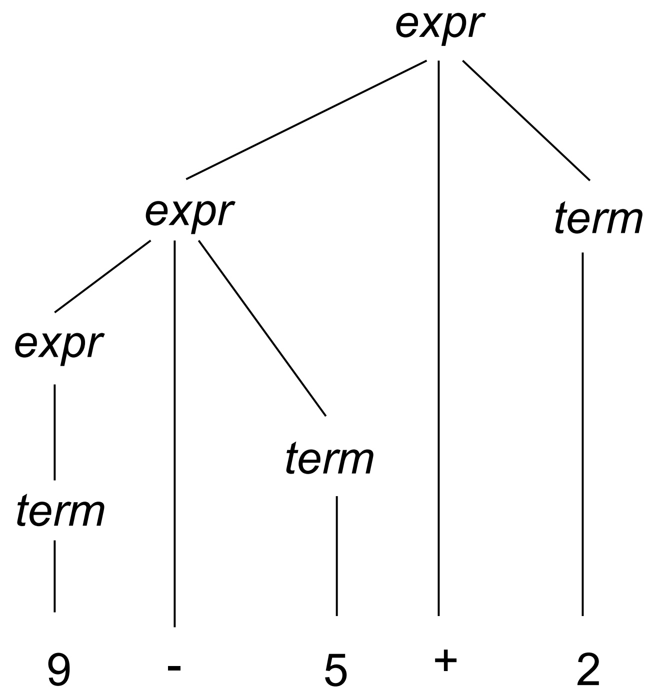
$A \to A\alpha \mid \beta$

$expr \to expr + term \mid term$
$A = expr;\ \alpha = +\ term;\ \beta = term$

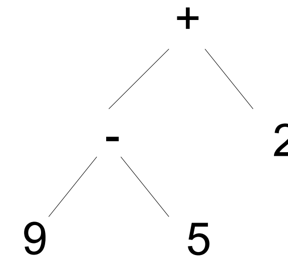
$A \to \beta R$
$R \to \alpha R \mid \in$

$A \to A\alpha \mid A\beta \mid \gamma$
$R \to \alpha R \mid \beta\ R \mid \in$

Concrete and Abstract Trees



A concrete tree



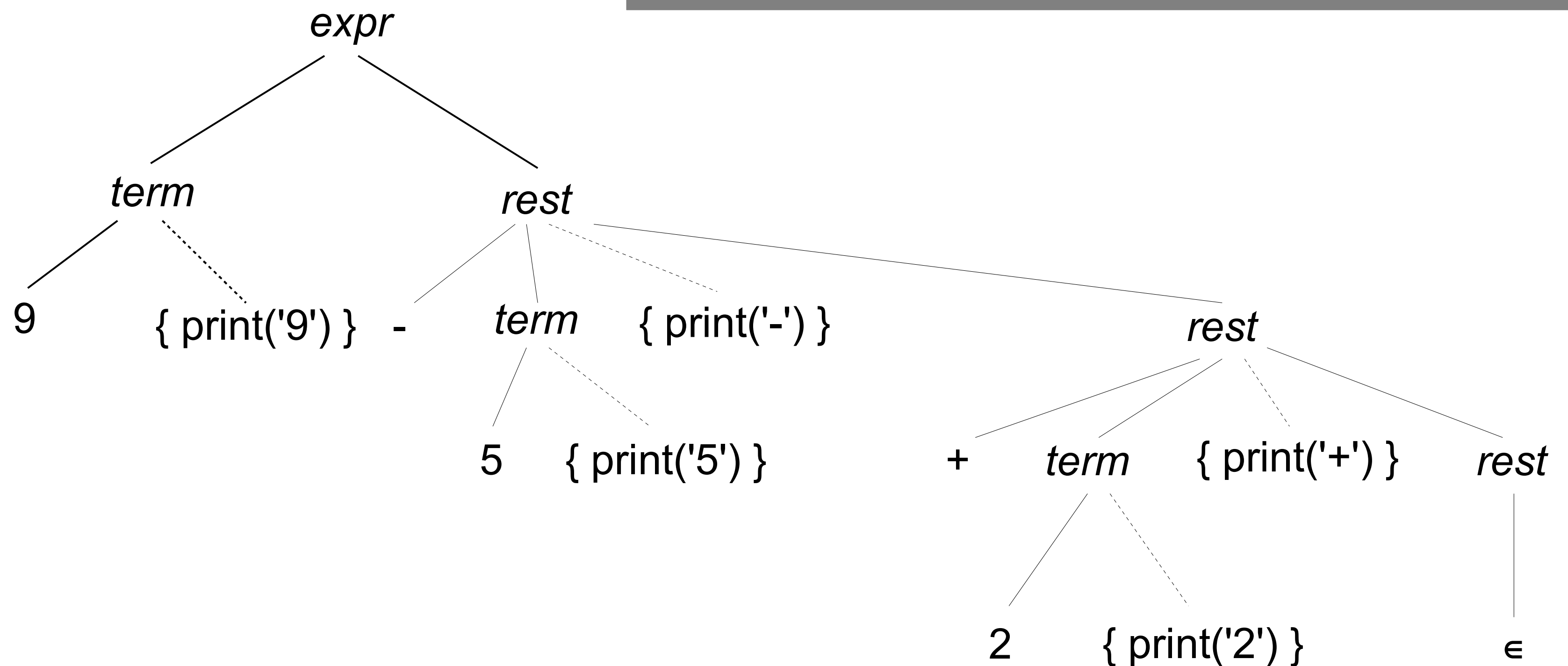
An abstract tree

The Tree of Utah, Karl Momen, 1982-1986, 87 feet

Adaptation of A Translation Scheme for A Predictive Parser

```
expr -> expr + term { print('+') }
expr -> expr - term { print('-') }
expr -> term
term -> 0 { print ('0') }
term -> 1 { print ('1') }
        …
term -> 9 { print ('9') }
```

```
expr ->  term rest
rest ->  + term { print('+') } rest
       | - term { print('-') } rest
       | ∈
term -> 0 { print ('0') }
term -> 1 { print ('1') }
        …
term -> 9 { print ('9') }
```



Code of A Predictive Parser

```
expr ->  term rest
rest ->  + term { print('+') } rest
       | - term { print('-') } rest
       | ∈
term -> 0 { print ('0') }
term -> 1 { print ('1') }
         …
term -> 9 { print ('9') }
```

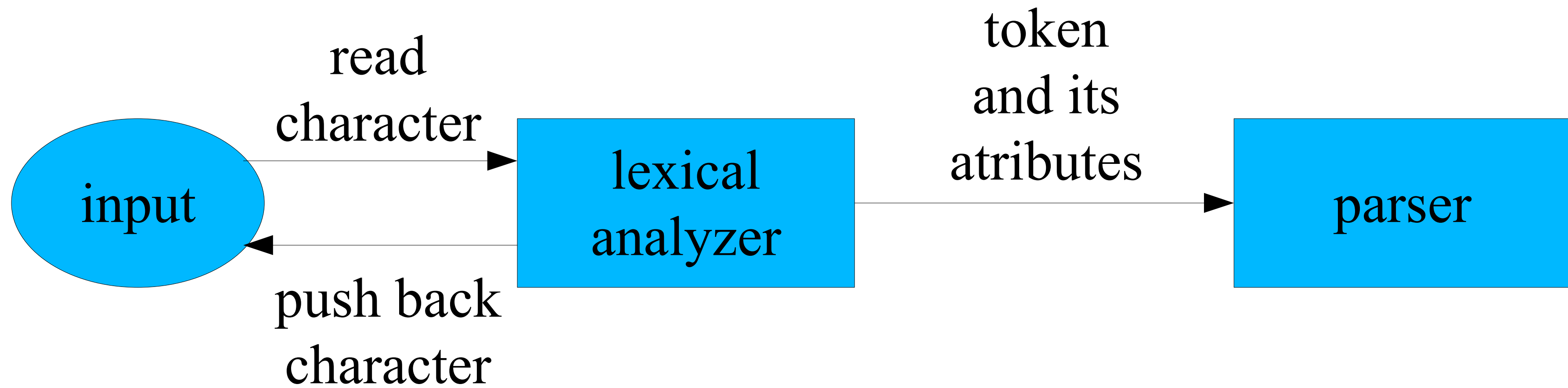
```
expr()
{
	term(); rest();
}

rest()
{
	if(lookahead == '+') {
		match('+'); term(); putchar('+'); rest();
	}
	else if (lookahead == '-') {
		match('-'); term(); putchar('-'); rest();
	}
	else;
}

term()
{
	if(isdigit(lookahead)) {
		putchar(lookahead); match(lookahead);
	{
	else error();
}

match(int t)
{
	if (lookahead == t)
		lookahead = getchar();
	else error();
}
```

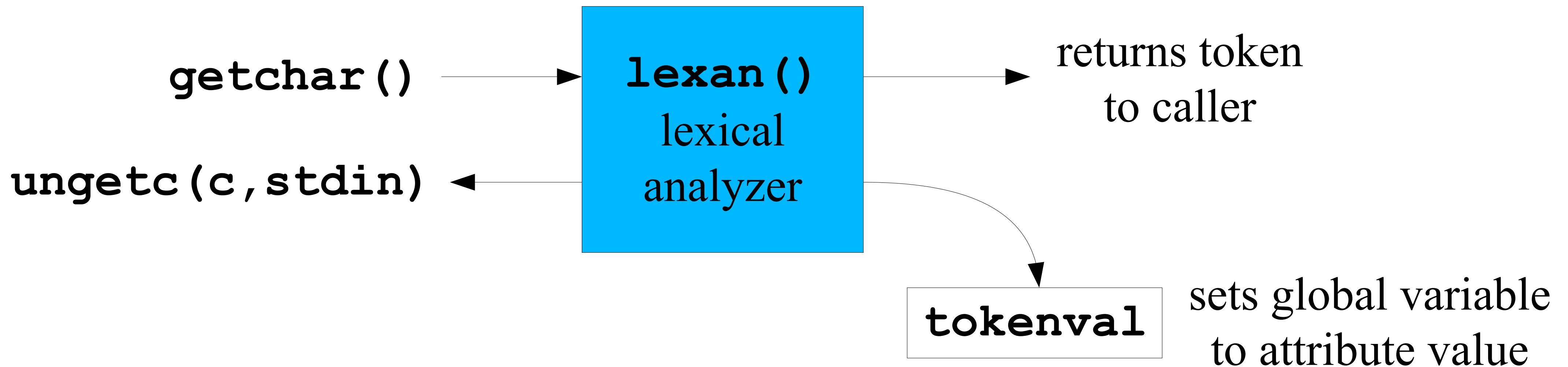
Elimination of A Tail Recursion

```
rest()
{
L:    if(lookahead == '+') {
          match('+'); term(); putchar('+'); goto L;
      }
      else if (lookahead == '-') {
          match('-'); term(); putchar('-'); goto L;
      }
      else ;
}
```

```
expr()
{
      term();
while(1)
      if (lookahead == '+') {
          match('+'); term(); putchar('+');
      }
      else if (lookahead == '-') {
          match('-'); term(); putchar('-');
      }
      else break;
}
```

Lexical Analyzer Interface





Code of A Lexical Analyzer

```
int lineno = 1;
int tokenval = NONE;

int lexan ()
{
   int t;
   while (1) {
        t = getchar ();
        if (t == ' ' || t == '\t')
             ; /* strip out blanks and tabs */
        else if (t == '\n')
             lineno++;
        else if (isdigit (t)) {
             tokenval = t - '0';
             t = getchar();
             while (isdigit(t)) {
                  tokenval = tokenval*10 + t - '0';
                  t = getchar();
             }
             ungetc (t, stdin);
             return NUM;
        }
        else {
             tokenval = NONE;
             return t;
        }
   }
}
```

The Symbol Table

```
int insert(const char* s, int t); /* returns index of new entry for string s, token t */
int lookup(const char* s); /* returns index of the entry for string s, or 0 if s is not found */
```

- The symbol table is used to store variables and keywords
- It is initialized with insertion of keywords:

```
insert("div", DIV);
insert("mod", MOD);
```

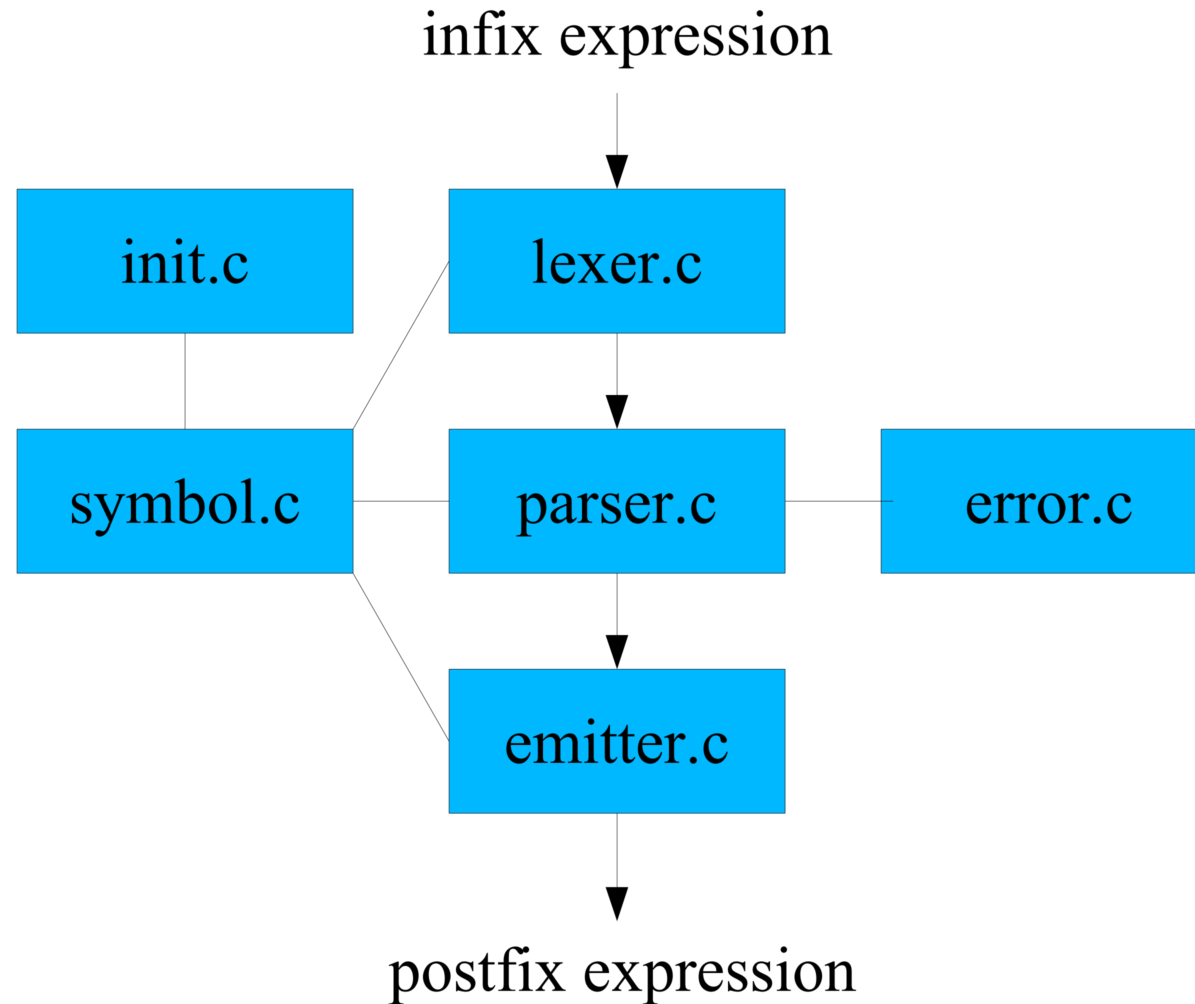
Code of A Lexical Analyzer with A Symbol Table

```
int lineno = 1;
int tokenval = NONE;
int lexan () {
 int t;
 while (1) {
      t = getchar ();
      if (t == ' ' || t == '\t');
      else if (t == '\n')
           lineno++;
      else if (isdigit (t))  {
           ungetc (t, stdin);
           scanf ("%d", &tokenval);
           return NUM;
      } else if (isalpha (t)) {
           int p, b = 0;
           while (isalnum (t)) {
                lexbuf[b] = t;
                t = getchar ();
                b++;
                if (b >= BSIZE) error ("compiler error");
           }
           lexbuf[b] = EOS;
           if (t != EOF) ungetc (t, stdin);
           p = lookup (lexbuf);
           if (p == 0) p = insert (lexbuf, ID);
           tokenval = p;
           return symtable[p].token;
      } else if (t == EOF)   return DONE;
      else {
           tokenval = NONE;
           return t;
      }
 }
}
```

Example Translator Grammar

```
start  ->  list eof
list   ->  expr ; list
        | ∈
expr   -> expr + term      { print('+') }
        | expr - term      { print('-') }
        | term
term   -> term * factor    { print('*') }
        | term / factor    { print('/') }
        | term div factor { print('DIV') }
        | term mod factor { print('MOD') }
        | factor
factor -> ( expr )
        | id               { print(id.lexeme) }
        | num              { print(num.value) }
```

Translator Structure



Lexical Analyzer

TOKENS: '+' '-' '*' '/' DIV MOD '(' ')' ID NUM DONE

Lexeme	Token	Attribute value
white space		
sequence of digits	NUM	numeric value of sequence
`div`	DIV	
`mod`	MOD	
other sequences of a letter then letters and digits	ID	index into **`symtable`**
end-of-file character	DONE	
any other character	that character	

Modified Translator Grammar

```
start       -> list eof
list        -> expr ; list
             | ∈
expr        -> term moreterms
moreterms   -> + term { print('+') } moreterms
             | - term { print('-') } moreterms
             | ∈
term        -> factor morefactors
morefactors -> * factor { print('*') } morefactors
             | / factor { print('/') } morefactors
             | div factor { print('DIV') } morefactors
             | mod factor { print('MOD') } morefactors
             | ∈
factor      -> ( expr )
             | id               { print(id.lexeme) }
             | num              { print(num.value) }
```