Fundamentals of Programming Languages

Definition of PLs

Lecture 03

sl. dr. ing. Ciprian-Bogdan Chirila

Lecture outline

- The definition of PLs
- Syntax
- Syntax grammars
- Syntax diagrams
- Semantics
- Operational semantics
- Attributed grammars
- Axiomatic semantics
- Denotational semantics

The PL

- Is a formal notation
- The form and meaning are described by a set of rules
- The rules establish
  - program correctness
  - what will happen during execution
- Syntactical rules –> PL syntax
- Semantic rules -> PL semantics

PL definition

- L=<Sm,St,f:St->Sm>
- Sm is the language semantics
- St is the language syntax
- f is the association function of syntax to certain semantics

Formal definition method of PLs

- defining an alphabet A out of base symbols
- defining A* set containing all possible symbol strings which may be constructed from the elements of A
- a set of rules to select the set of correct programs
- the semantic specification of each element

$$P \subseteq A^*$$

$$p \in P$$

The syntax

- Syntax rules generate an infinite set of sentences
- Only a subset of them are semantically correct
- Sentences are made out of symbols
- Symbols are made out of characters respecting the lexical rules
- Lexical rules belong to the PL syntax
- All symbols form the PL vocabulary
  - Identifiers, keywords
    - begin, end in Pascal
    - +, ++, <=, in C
  - Integer literals, float literals, string literals

Grammars

- All syntactical rules of a language form the grammar
- How to write a grammar?
- BNF Bachus Naur Form
  - Used for Algol 60
- Extended BNF –EBNF
  - Metalanguage
    - A language used to define another language

EBNF

::= defined as

| or

< and > used for non-terminals

[and] used for optional sequences

{ and } used for sequences repeated zero or more times

The syntax

- The syntax is a set of EBNF relations or rules
- A relation defines
  - A non-terminal specified at left hand side of ::=
  - Non-terminals and terminals at the right hand side of ::=
- Terminals are language symbols
- Each right hand side used non-terminal must be defined in a different relation
- A complete grammar must define all non-terminals
- One non-terminal is defined as the starting symbol of the grammar
- Usually is called <program>

The program

- A string of symbols or terminals
- Is syntactically correct if the symbol string can be derived based on the grammar rules beginning from the starting symbol

Grammar example

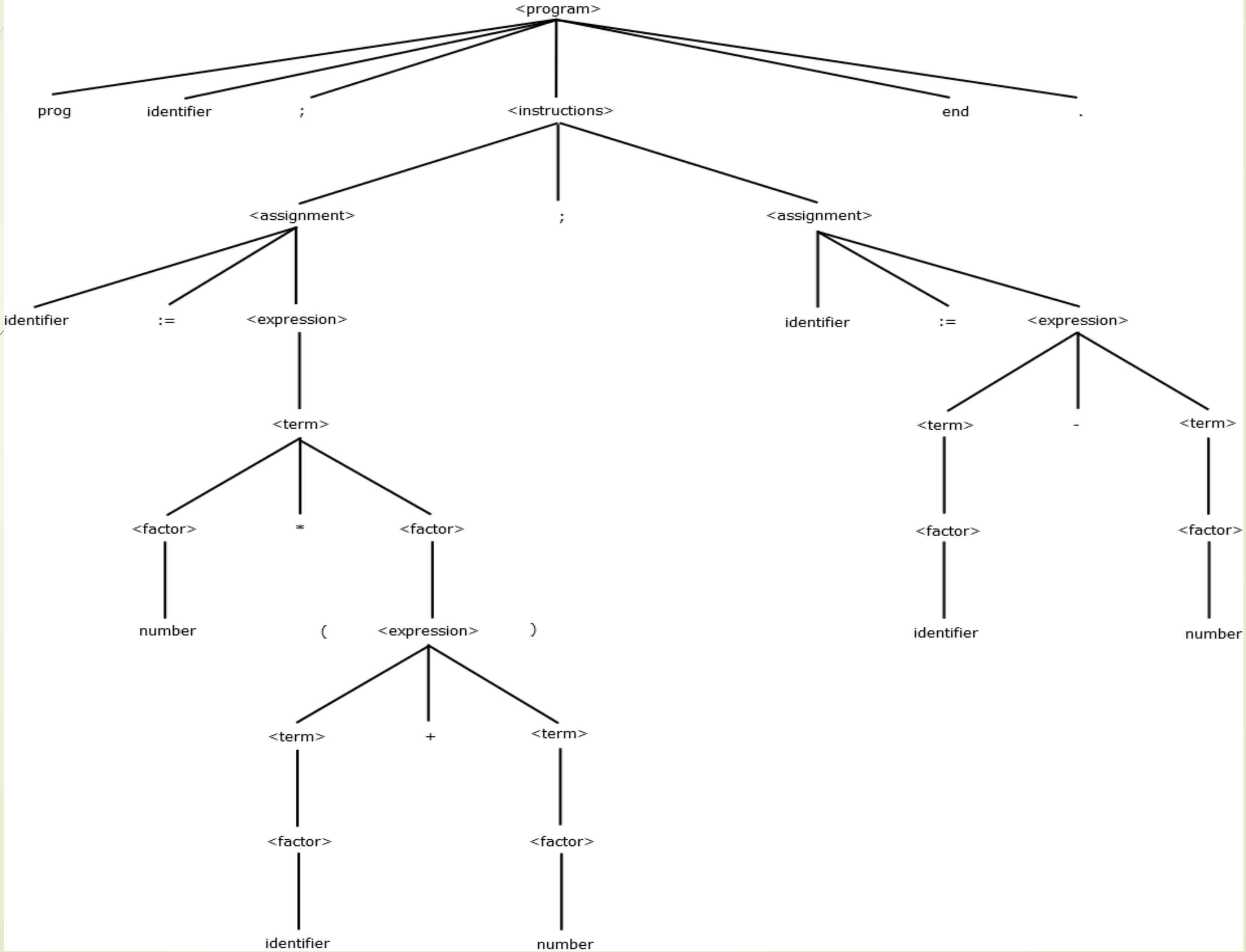
```
<expression> ::= <term> { +|- <term> }

<term> ::= <factor> { *|: <factor> }

<factor> ::= number | identifier  | (
   <expression> )

<assignment> ::= identifier := <expression>

<instructions> ::= <assignment> { ;
   <assignment> }

<program> ::= prog identifier; <instructions>
   end.
```

Program example

```
prog example;
   a:=2*(x+3);
   b:=a-1
end.
```

- Is syntactically correct if it can be derived based on the rules and starting from the <program> non-terminal
- The derivation process can be better illustrated drawing a tree where
  - The root is the starting symbol
  - The inner nodes are non-terminals
  - The leaves are terminals
- Syntax tree

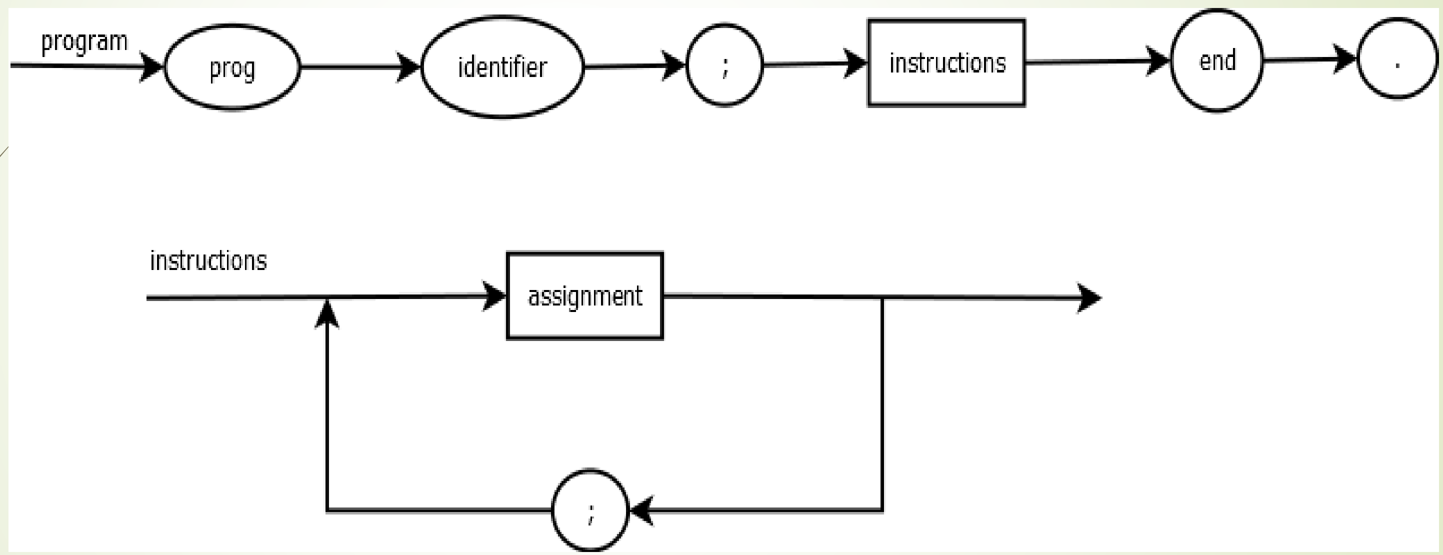
The derivation process



Syntactic analysis

- To check the syntactical correctness of a program
- Bottom-up
  - To start from the symbols
  - To replace right hand side sequences with rules
  - To repeat until the starting symbol is reached
- Top-down
  - To begin at the starting symbol
  - To replace non-terminals according to the grammar rules

Syntax diagrams



Syntax diagrams

- A symbols succession is correct if it can be generated by traversing the diagram from the beginning to the end
- Meeting a rectangle
  - the corresponding non-terminal must be verified
- Meeting a circle/ellipse
  - the corresponding terminal must be found

Regular expressions

- Lexical rules can be expressed using regular expressions
- Each regular expression *e* denotes a set of strings *S* formed with the letters of an alphabet *A* applying a set of operators

Regular expressions

- Let us suppose that S, S1 , S2 are sets of strings
- Reunion

$$S_1 \cup S_2 = \{s \mid s \in S_1 \; or \; s \in S_2\}$$

- Product or Catenation

$$S_1 S_2 = \{s_1 s_2 \mid s_1 \in S_1 \text{ and } \mathrm{s}_2 \in S_2\}$$

- Power

$$S^n = \begin{cases} \{\varepsilon\} \; n = 0, \varepsilon \; the \; empty \; string \\ S^{n-1} S \; \forall \, \mathrm{n} \in \mathrm{N}, \mathrm{n} \geq 1 \end{cases}$$

Regular expressions

- Kleene closure or Star $S^* = \bigcup_{i=0}^{\infty} S^i$
- Positive closure or Plus $S^+ = \bigcup_{i=1}^{\infty} S^i$
- e.g.
  - L={A,B,C,…Z,a,b,…,z}
  - C={0,1,…,9}

Defining new sets

- the set of all letters and digits $L \cup D$
- all two length strings where LD
  - the first character is letter
  - the second character is digit
- the set of strings of 4 letters L^4
- the set of strings of letters of any length including the empty length string L^*
- the set of digit strings containing at least a digit D^+

The construction of a regular expression

- Starting from an alphabet A
- ε is a regular expression
- a is a regular expression
- e1, e2, e regular expressions denoting the string sets S1, S2, S
- on these sets we can apply several operators
- the result will be a regular expression

Regular expression operators

- Reunion (e1)|(e2) denote the set S1 U S2
- Product or catenation (e1)(e2) denoting the set S1S2
- Star (e)* denoting the set (S)*
- all operators are left-associative
- the priority
  - from high to low
  - star, product, reunion

Regular expression operators

- One or more
  - Plus “+” operator
  - ee* is equivalent to e+
- Zero or one
  - The question mark “?” operator
- Character classes
  - The notation [c1c2c3c4] will designate the c1|c2|c3|c4 regular expression
  - a|b|…|z will become [a-z]

Example

- letter(letter | digit)*
  - Regular expression for identifiers
  - L(LUC)* from previous examples
- digit->[0-9]
- letter->[A-Z, a-z]
- identifier -> letter(letter | digit)*
- digits->digit+
- exponent->((E | e)(+ | -)?digits)?
- fraction->(.digits)?
- number-> digits fraction exponent
- when names are used in the right hand side -> regular definition

The semantics

- Semantic rules
  - The meaning associated to correct syntactical constructions
- Describing syntax
  - BNF
  - EBNF
- Describing semantics
  - Coexist a series of methodologies
  - In research for one fully satisfiable

The semantics of a PL

- Is described in natural language
  - text, drawings, diagrams
- More or less exact or rigorous
- Good for learning
- Ambiguities are clarified by experiments

Why formal description of semantics?

- PLs
  - large spreading
  - tend to be complex and diverse
- because applications
  - are complex, large, diverse
  - demand high reliability
- the solution: formal, mathematical notation
  - With no ambiguities
  - Difficult access
  - Needs special preparing to decipher the formalism

Formalisms advantages

- Avoiding gaps in language definition
  - Gaps are probable in the informal definition
- Reference documentation for the programmer
  - The programmer may clarify problems when reading the informal definition
- Reference documentation for the implementation
  - For the PL implementation team
  - Used for implementation validation and homologation

Formalisms advantages

- Formal base for automated program checking
  - Formal checking algorithms need a rigorous PL definition
- Implementation independence
  - The formalisms guarantee that is independent of the implementation

Appreciations criteria for formal methods

- Completeness
  - The method capability of covering all syntax and semantic issues
- Simplicity
  - The ease of model creation no matter how complex the language is

Appreciations criteria for formal methods

- Clarity
  - understanding definitions easily
  - natural PL description
- Expressivity on errors
  - the method capability of detecting program errors

Appreciations criteria for formal methods

- Changeability
  - The method capability of defining places where restrictions or options are left free to implementers
- Modifiability
  - The method capability of easily making modifications in the previous PL description
  - Important in the PL definition phase

Formal methods for PL semantics

- 2 methods
  - Intuitive
  - Based on program translation concepts
- 2 methods
  - Mathematical
  - With a strong theoretical base
- methods comparison
  - using the presented criteria

Operational semantics

- Is defined by the effect its constructions make over a real or virtual processor
- The instruction semantics is denoted by
  - Knowing the computer state
  - Executing an instruction
  - Examining the new computer state

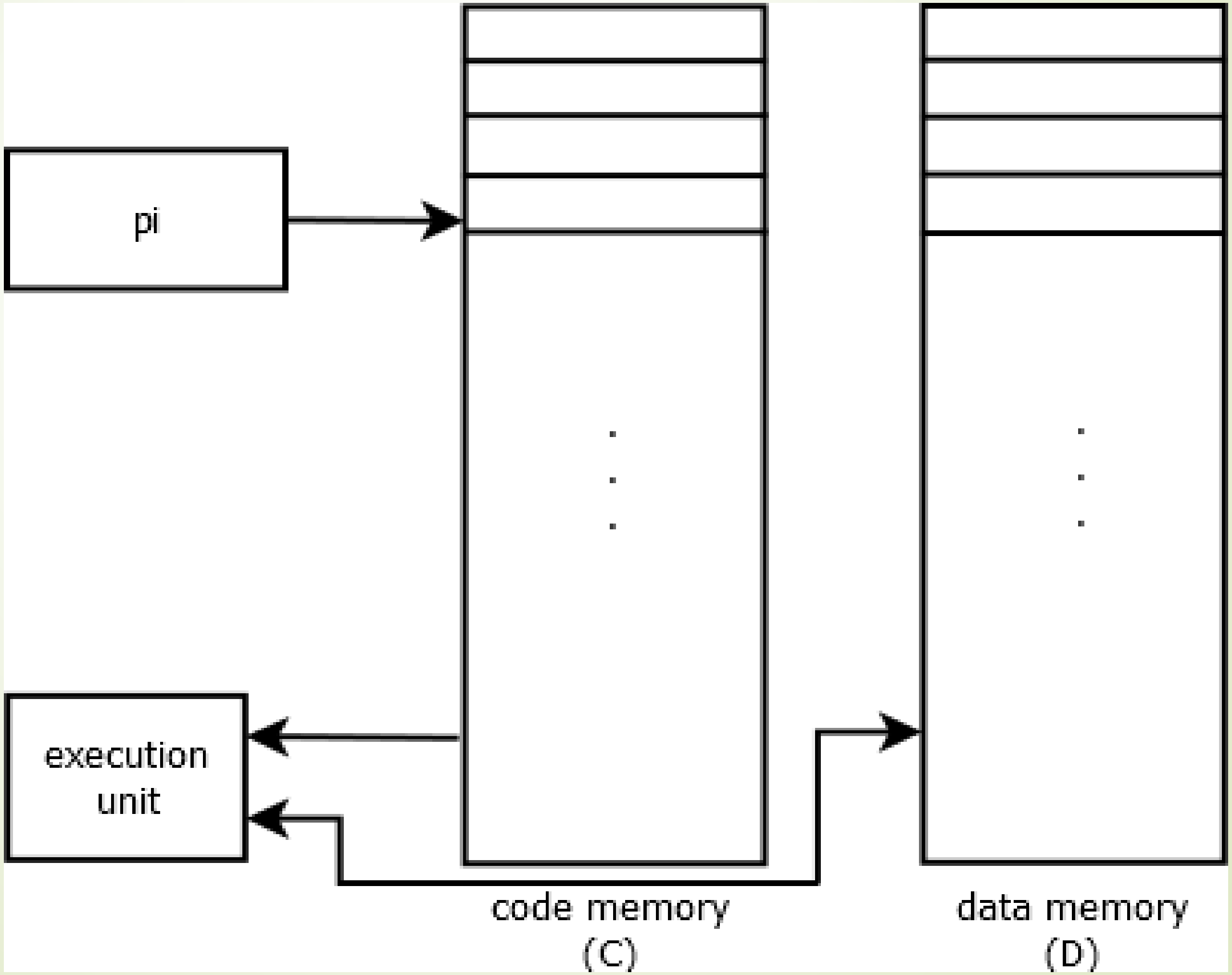
Operational semantics

- Real computer architectures are very complex
- Virtual machines are used instead
- Software interpreter executing virtual instructions
- Virtual machines can be designed such as PL semantics can be easily expressed with virtual instructions
- The set of virtual instructions must be simple enough to be implemented on any hardware machine

Applying the operational method

- Defining and implementing a virtual machine VM
- A translator converting the instructions of the L language into the instructions of the virtual machine VM
- State changes produced on the virtual machine by executing the virtual code resulted by translating L language instructions defines the instruction semantics

The structure of a virtual machine VM



Virtual machine run

- Virtual processor
  - Instruction pointer IP
- Code memory C
- Data memory D
- VM cycle
  - Execute the instruction pointed by IP
  - If the instruction does not change IP, the IP register will be incremented and will denote the next instruction in C

Program example

```
for i:=first to last do
begin
 ...
end;
```

```
i:=first;
loop: if i>last goto out
    ...
    i:=i+1
    ...
    goto loop
out: ...
```

Operational description

- Was used for the first time in the IBM Vienna subsidiary
- Was used to define PL/I 1969
- VDL - Vienna Definition Language
- Good for both programmers and implementers
- Is not based on a complicated mathematical formalism
- Is based on translation algorithms
- The PL semantic is defined in the terms of a different known low level language

Attributed grammars

- Used when the translation process is coordinated by a grammar
- The semantics can be specified by attaching semantic attributes to the grammar symbols
  - Terminals
  - Non-terminals
- Proposed by Donald Knuth 1968

Attributed grammars

- The attribute values are computed through expressions or functions called semantic rules associated to the grammar rules
- The evaluation of semantic rules means semantic analysis
- This process is also called syntax directed translation
- There are several associations possible between semantic rules and grammar relations
  - syntax directed definitions SDD

Syntax directed definition

- Is a generalization of a grammar
- To each symbol we attach a set of attributes
- Results an attributed grammar
- Attribute representation
  - Numbers
  - Strings
  - Types
  - Memory locations
- Attributes are computed during the development of the syntactic tree
- The attribute value is computed using a semantic rule associated with the production

Example SDD for an office calculator

<line>::=<expression>nl

<expression>::=<expression>+<term> | <term>

<term>::=<term>*<factor> | <factor>

<factor>::=(<expression>) | number

- the definition associates to each nonterminal (<expression>, <term>, <factor>) an attribute having integer value named val
- For each production we compute the val attribute associated with the left hand-side non-terminal based on the values of the val attribute from right-hand side non-terminals

SDD for the office calculator

Grammar production	**Semantic rules**
<line>::=<expression>nl	print(<expression>.val)
<expression>::=<expression1>+<term>	<expression.val>:=<expression1>.val+<term>.val
<expression>::=<term>	<expression>.val:=<term>.val
<term>::=<term1>*<factor>	<term>.val:=<term1>.val*<factor>.val
<term>::=<factor>	<term>.val:=<factor>.val
<factor>::=(<expression>)	<factor>.val:=<expression>.val
<factor>::=number	<factor>.val:=number.lexval

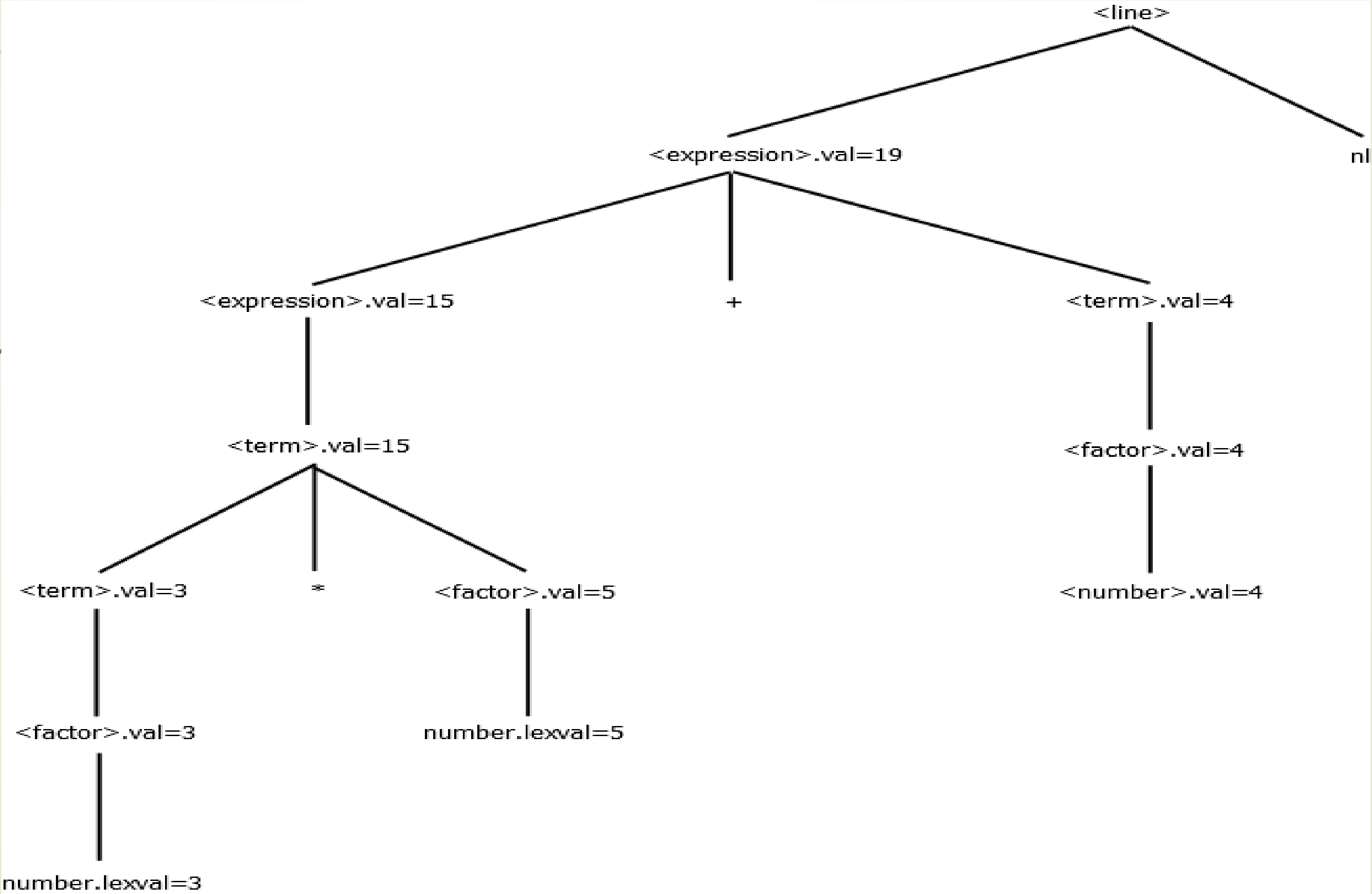
SDD for the office calculator

- number atom has an attribute named lexval
- the starting rule prints out the value of the <expression>

Annotated syntax tree

- is a syntax tree which shows the nodes attributes
- The process is called syntax tree annotation
- A definition using only synthesized attributes is called an S-attributed definition

Example of annotated syntax tree



Axiomatic semantic

- To translate a correct construction into a mathematical meta-language
- A notation having well defined mathematical rules
- To determine a set of translation rules between
  - language construction domain
  - mathematical formulae – meta-language

Axiomatic semantic meta-language

- C.A.R. Hoare 1969
- Has its roots in mathematical logic
- Based on predicates computation
- Predicates
  - Are logic expressions applied on program variables
  - Used to express states in the computing process

Preconditions and postconditions

- Instruction S
- Predicate P
  - that must be true after executing S
  - is called postcondition for S
- Predicate Q
  - is true
  - S is executed normally
  - Postcondition P is true
  - Is called precondition for S and P

Example

- Notation
  - Q {S} P
- Example
  - S: x:=y+1 (integers)
  - P: x>0 y=3 {x:=y+1} x>0
  - Q: y=3
  - ------------
  - Q: y>-1 y>-1 {x:=y+1} x>0

Example

- For
  - instruction S
  - postcondition P
- there are multiple (an infinite) preconditions available
- One of them is called the weakest precondition
- All preconditions Q imply the weakest precondition W
- For any true Q, W is also true

Axiomatic semantic

- ∀ precondition Q $\rightarrow$ W (implication relationship)
- p $\rightarrow$ q (means anytime p is true also q is true)
- y=3 -> y>-1 is TRUE
- y>0 -> y>-1 is TRUE
- y>-5 -> y>-1 is FALSE
- Only the weakest precondition is important
- To express the construction effect by predicate transformation
- To define the axsem function
  - axsem(S,P)=W
  - S – language construction
  - P – postcondition
  - W – weakest precondition
- To define a language means defining axsem for all constructions

axsem for assignment instruction

- axsem{x:=E,P}-> $P_{x\text{->}E}$
- $P_{x\text{->}E}$ is the predicate P where all appearances of x were replaced by E
- In order that predicate P to be true after x got the value of E, before the assignment the predicate obtained replacing x by E must be true
- $P_{x\text{->}E}$ {x:=E} P
- y>-1 {x:=y+1} x>0
  - in x>0 we replace x with y+1
  - y+1>0 or y>-1
  - the semantic of the assignment is that if y>-1 then x>0

Example

- We will use the axsem function to find out in which conditions x:=x+3 will produce a result x>8
- axsem(x:=x+3,x>8)=x>5
  - in x>8 we replace x by x+3
  - x+3>8 and x>5
  - if x>5 then after the assignment x>8
  - the semantic of the assignment is that if x>5 then x>8

axsem for an instruction sequence

- considering
- axsem(S1,P)=Q
- asxem(S2,Q)=R
- for the sequence S2;S1
- axsem(S2;S1,P)=R
- the postcondition created by S2 becomes precondition for S1
- R S2 Q
- Q S1 P
- after sequencing we got R S2 Q S1 P or R S2;S1 P

axsem for if instruction

- if B then L1 else L2 endif
- B condition
- L1,L2 instruction sequences
- axsem(instr-if,P)=

  B => axsem(L1,P)

  and

  not B => axsem(L2,P)

Example

- if x>=y then max:=x else max:=y endif
- if the sequence computes max correctly
- then (x>=y and max=x) or (y>=x and max=y) must be true
- P is this postcondition, what is the precondition ?
- (x>=y)=>((x>=y and x=x) or (y>x and x=y)) and

not(x>=y)=>((x>=y and y=x) or (y>x and y=y)) = true

Denotational semantic

- Scott 1970
- S=<mem,i,o>
  - mem is a function representing the memory
  - Mem:Id->Z U {undef}
    - Id is the set of all identifiers
    - Z is the set of all integers
    - undef is the value of an undefined indentifier
  - i,o input and output sequences
    - their values can be integers sequences or void sequence

Denotational semantic

- Using this representation each language construction is expressed as a function
- Functions show the modifications of the language construction produced on the system state
- All functions + all composition rules represents the semantic definition of the language
- Mathematical metalanguage for denotational semantic is the functional calculus

Arithmetical expression

- dsemEx:EX x S -> Z U {error}
  - S the set of states
  - EX the set of expressions
- dsemEx(E,s)=error
  - if s=<mem,i,o> and mem(v)=undef for a variable v from E; else
- dsemEx(E,s)=e
  - if s=<mem,i,o> and e is the result of E expression evaluation after replacing each v identifier from E with mem(v)
- we assumed that expression have
  - no collateral effects
  - no overflows
  - no type errors

Assignment instruction

- dsemAS:AS x S -> S U {error}
  - AS: the set of assignment instructions
  - dsemAs(x:=E,s)=error
    - if dsemEx(E,s)=error; else
  - dsemAs(x:=E,s)=s'
    - where s=<mem,i,o>, s'=<mem',i',o'>
    - i'=i, o'=o
    - mem'(y)=mem(y) for any y≠x
    - mem'(x)=dsemEx(E,s)

Read instruction

- x<-read
- dsemRd:RDxS->S U {error}
  - RD the set of read instructions
  - dsemRd(x<-read,s)=error
    - if s=<mem,i,o> and i is void; else
  - dsemRd(x<-read,s)=s'
    - Where s=<mem,i,o> s'=<memi,i',o'>
    - o= o' i=Ii'
    - mem'(y)=mem(y) for all y≠x
    - mem'(x)=I

Instruction sequence

- dsemIS:ISxS->S U {error}
  - SI is the set of al instruction sequences
  - In case of a void list ε
    - dsemIs(ε,s)=s
  - In case of a list T;L
    - dsemIs(T;L,s)=error
      - if dsem(T,s)=error; else
    - desemIs(T;L,s)=dsemIs(L,dsem(T,s))
    - Dsem describes the semantic of T

If instruction

- if B then L1 else L2 end if
- B is an expression

  = 0 false

  ≠ 0 true
- L1,L2 instruction sequences

If instruction

- dsemIf:IFxS->S U {error}
- IF is the set of all if instructions
- dsemIf(if B then L1 else L2 end if, s)=error
  - if dsemEx(B,s)=error; else
- dsemIf(if B then L1 else L2 end if, s)

  =dsemSi(L1,s) if dsemEx(B,s) ≠0; else

  =dsemSi(L2,1)

While instruction

- while B do L end while
- B is an expression

  = 0 false

  ≠ 0 true
- L instruction sequence
- dsemWhile:WHILE x S->S U {eroare}
  - WHILE the set of all while instructions

While instruction

dsemWhile(while B do L end while, s)=error

if dsemEx(B,s)=error; else

dsemWhile(while B do L end while, s)=s

if dsemEx(B,s)=0; else

dsemWhile(while B do L end while, s)=error

if dsemIS(L,s)=error; else

dsemWhile(while B do L end while, dsemIs(L,s))