Programming Languages and Compiler Design

Introduction: Compiler architecture, Compiling & Semantics

Yliès Falcone, Jean-Claude Fernandez

Master of Sciences in Informatics at Grenoble (MoSIG)
Univ. Grenoble Alpes
(Université Joseph Fourier, Grenoble INP)

Academic Year 2015 - 2016

Some practical information

Lecture sessions:

- Yliès Falcone (international parcours)
- Jean-Claude Fernandez (french parcours)

Exercise sessions:

- Fabienne Carrier (french parcours)
- Gwenaël Delaval (international parcours)
- Yliès Falcone (international parcours)
- Laurent Mounier (french parcours)

Emails: FirstName.LastName@imag.fr (with no accents)

Web pages:

- http://www.ylies.fr
- http://www-verimag.imag.fr/~fernand/

Phone numbers:

- YF: +33 4 38 78 29 51
- JCF: +33 4 56 52 03 79

Office location: CEA Minatec (YF), LIG (GD), Vérimag (FC, JCF, LM).

Meetings are possible (on appointment)

Global objectives of the course

- Programming languages, and their description:
  - syntax,
  - semantics.
- General compiler architecture.
- Some more detailed compiler techniques .

Basic objective

Study how to translate a program written in a programming language to a program executable by a machine.

References

Pedagogical Resources

All pedagogical resources are on the Moodle:

http://imag-moodle.e.ujf-grenoble.fr/

A. Aho, R. Sethi and J. Ullman

Compilers: Principles, techniques and tools

InterEditions, 1989

H. R. Nielson and F. Nielson.

Semantics with Applications: An Appetizer.

Springer, March 2007. ISBN 978-1-84628-691-9

W. Waite and G. Goos.

Compiler Construction

Springer Verlag, 1984

R. Wilhelm and D. Maurer.

Compilers - Theory, construction, generation

Masson 1994

Outline - Introduction: Compiler architecture, Compiling & Semantics

Compilers: some light reminders

Compiling and semantics: two connected themes

Some Maths Reminders

Summary

Outline - Introduction: Compiler architecture, Compiling & Semantics

Compilation vs interpretation

- Refers to the property of the *implementation* of a language (and not the language itself).
- Any language can be compiled or interpreted for a machine.

Interpretation

- The source code is read by another program, the interpreter, and not translated to native code.
- The interpreter is implemented for a particular machine/architecture.
- On-the-fly translation to machine code, and execution.
- Pros: portability, easier to produce.

Compilation

- The source code is translated to machine code by the compiler.
- Pros: programs run faster.

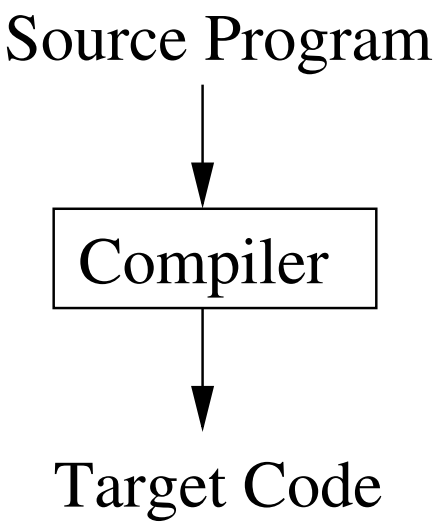
Remark

- Notions are not exclusive. A language can have an interpreter and a compiler (e.g., OCaml).
- Source code is often compiled before being passed to an interpreter.

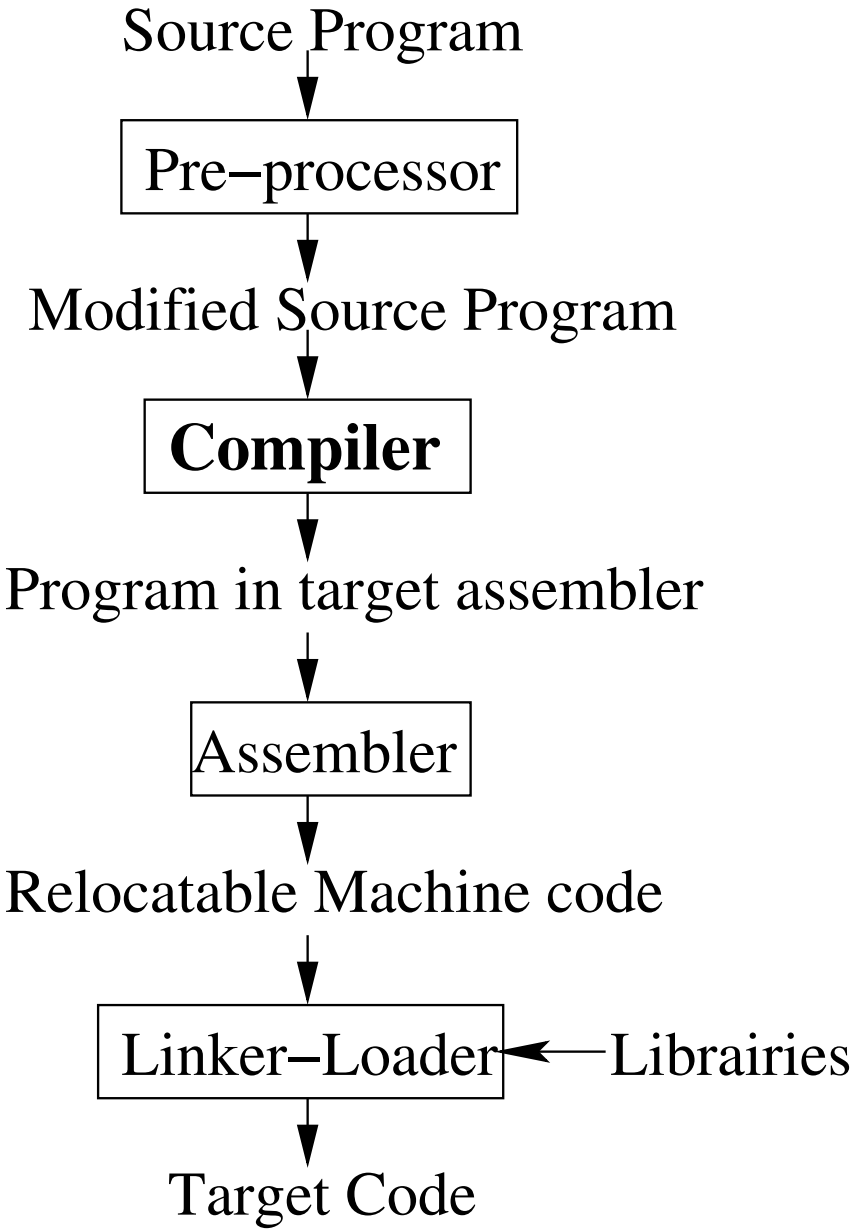
Compilers: what you surely already know...

A compiler is a *language processor*: it transforms a program

- from a language we can understand: the programming language,
- to a language the machine can understand: the target language.

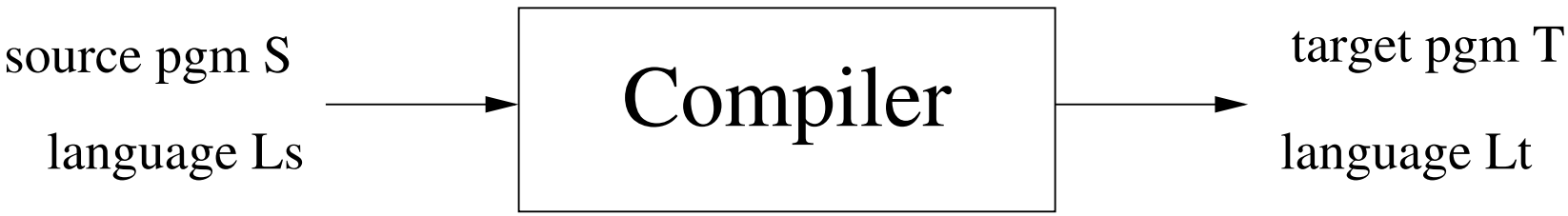


Compilers: what you surely already know...



- Pre-processing (lexical): macro substitution, textual inclusion of other files, and conditional compilation or inclusion.
- Pre-processing (syntactic): language extension/customization with new primitives.
- Linking: combining object modules and librairies.
- (OS) Loader: moving object modules to memory and resolving addresses.

Compilers: what do we expect ?



Expected Properties?

- correctness: execution of T should preserve the **semantics** of S
- efficiency: T should be optimized w.r.t. some execution resources (e.g., time, memory, energy, etc.)
- "user-friendliness": errors in S should be accurately reported
- completeness: any correct *Ls*-program should be accepted

Many programming language paradigms . . .

Imperative languages

- *ex: FORTRAN, Algol-xx, Pascal, C, Ada, Java*
- notions: control structure, (explicit) memory assignment, expressions, types, . . .

Functional languages

- ex: *ML, CAML, LISP, Scheme, etc*
- notions: term reduction, function evaluation, recursion, . . .

Object-oriented languages

- ex: *Java, Ada, Eiffel, C++*
- notions: objects, classes, types, inheritance, polymorphism, . . .

Logical languages

- ex: *Prolog*
- notions: resolution, unification, predicate calculus, . . .

Web languages

- ex: *JavaScript*, *PHP*, *HTML*
- notions: scripts, markers, . . .

etc.

. . . and many architectures to target!

- Complex instruction set computer (CISC)
- Reduced instruction set computer (RISC)
- VLIW, multi-processor architectures
- dedicated processors (DSP, . . .)
- embedded systems (mobile phones, . . .).
- etc.

We will mainly focus on:

Imperative languages

- data structures
  - basic types (integers, characters, pointers, etc)
  - user-defined types (enumeration, unions, arrays, ...)
- control structures
  - assignments
  - iterations, conditionals, sequence
  - nested blocks, sub-programs

"Standard" general-purpose machine architecture: (e.g. ARM, iX86)

- heap, stack and registers
- arithmetic and logical binary operations
- conditional branches

Describing a programming language P

Lexicon L: words of P
→ a regular language over P alphabet

Syntax S: sentences of P
→ a context-free language over L

Static semantic (e.g., typing): "meaningful" sentences of P
→ subset of S, defined by inference rules or attribute grammars

Dynamic semantic: the meaning of P programs
→ describes how program execute, and their execution sequences

Meaning?

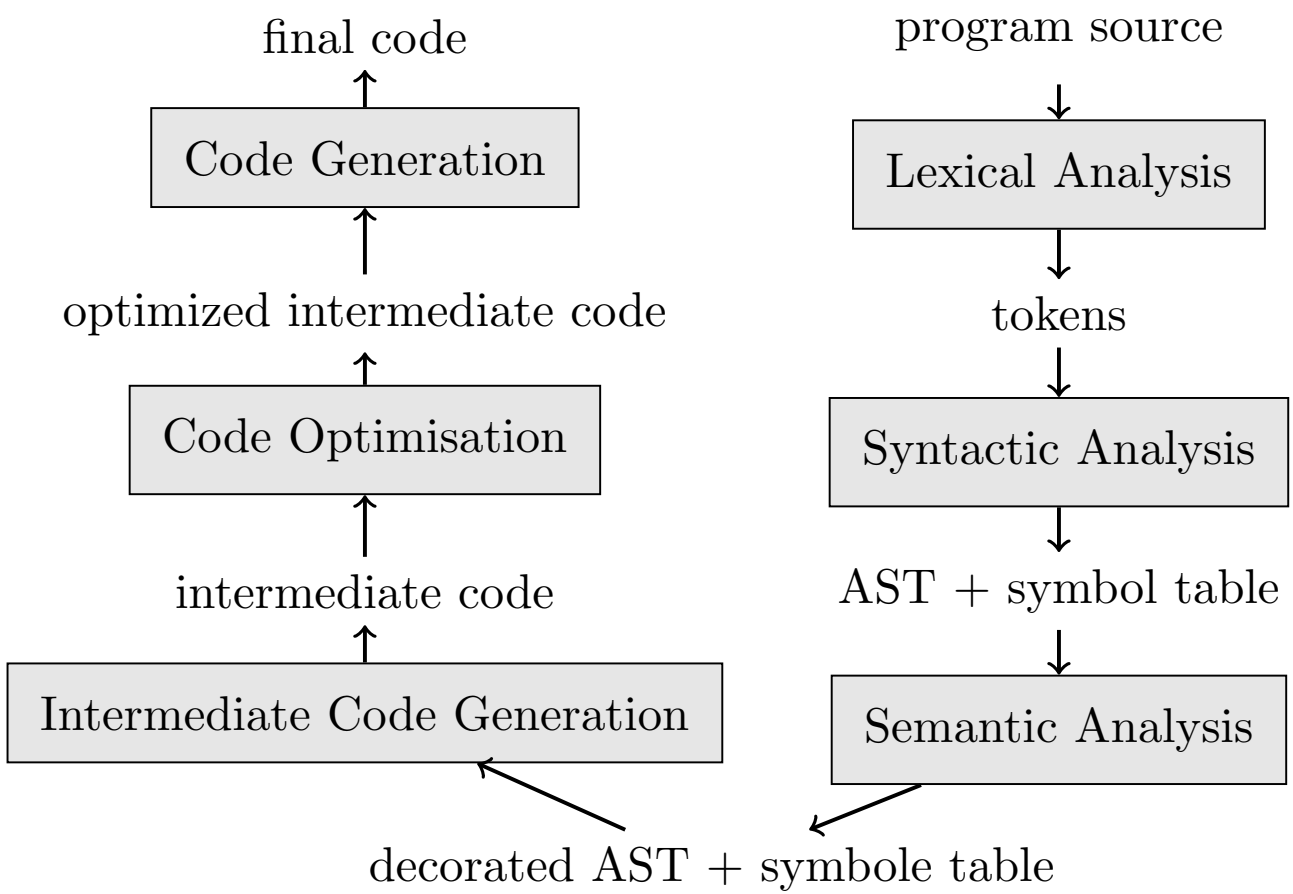
But How to define the meaning of program?

→ The semantics of programs

Semantics?

- Several notions/visions of semantics
  → transition relation, predicate transformers, partial functions
- Depends on "what we want to do/know on programs"

Compiler architecture: the logical steps



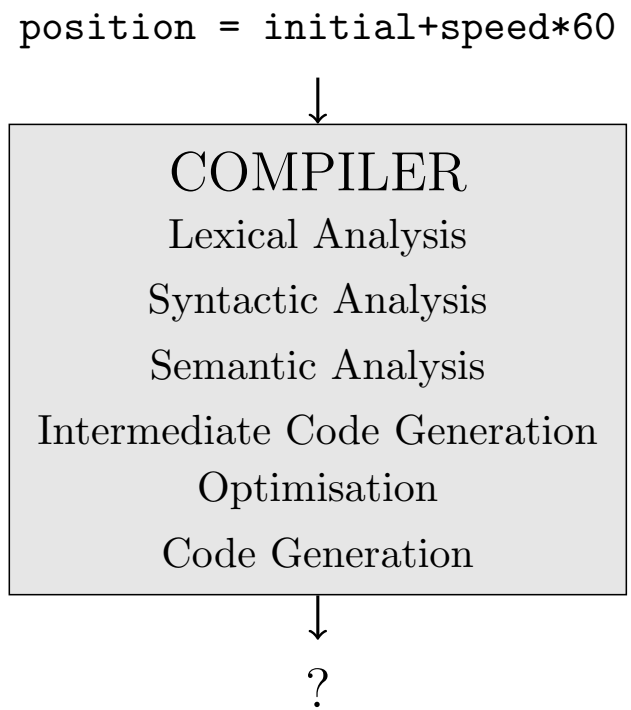
In practice:

- steps regrouped into passes
- one passe vs multi-pass

Running example: an assignment

Consider the assignment `position` = `initial` + `speed` ∗ 60

Example (Processing `position` = `initial` + `speed` ∗ 60)



Lexical analysis by a scanner

Input: sequence of characters

Output: sequence of lexical unit classes

1. compute the longest sequence $\in$ a given lexical class
   $\hookrightarrow$ *lexems* of the program
2. insert a reference in the *symbol table* for identifiers
3. returns to the syntactical analyzer:
   - lexical class (*token*): constants, identifiers, keywords, operators, separators, ...
   - the element associated to this class: the *lexem*
4. skip the comments
5. special token: error

Based on formal tools: regular languages

- described by regular expressions
- recognized by (deterministic) finite automata

Example of scanner: LeX (scanner generator)

Lexical Analyzer / Scanner Generator

A scanner generator yields an implementation of a Finite-State Automaton from a regular expression.

Example (LeX)

- Description:

```
declarations
%%
rules
%%
procedures
```

- Examples of declaration:

```
digit [0-9]
integer {digit}+
```

- Example of rule description:

```
{integer} {val=atoi(yytext);return(Integer);}
```

Lexical analysis on the running example

Example ($\texttt{position} = \texttt{initial} + \texttt{speed} * 60$)

lexem	**token**
`position`	$< id, 1 >$
$=$	$<=>$
`initial`	$< id, 2 >$
$+$	$< + >$
`speed`	$< id, 3 >$
$*$	$< * >$
60	$< 60 >$

Symbol Table

1	`position`	...
2	`initial`	...
3	`speed`	...

Some remarks:

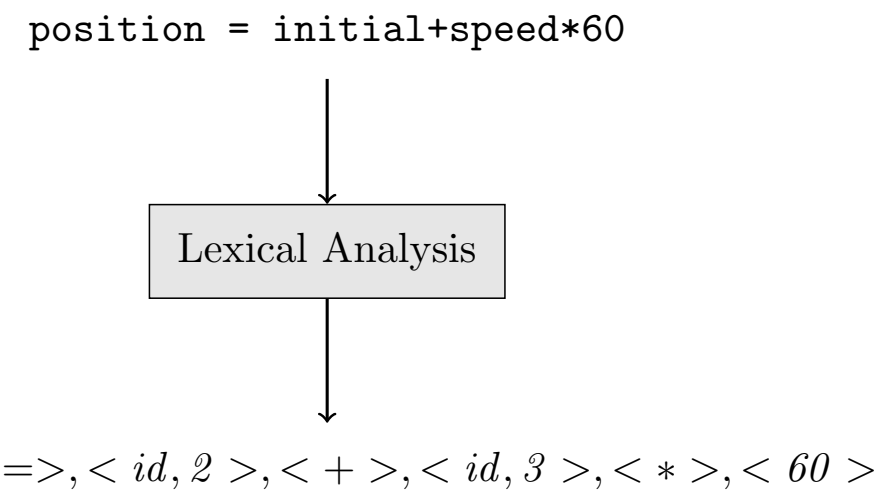
- *id* is an abstract symbol meaning *identifier*
- In $< id, i >$, i is a reference to the entry in the **symbol table** (The entry in the symbol table associated to an identifier contains information on the identifier such as name and type)
- normally $< 60 >$ is represented $< number, 4 >$

Running example: an assignment

Example (Lexical analysis of
`position` $=$ `initial` $+$ `speed` $*$ 60)

Symbol Table

1	`position`	...
2	`initial`	...
3	`speed`	...



$< id, 1 >, <=>, < id, 2 >, < + >, < id, 3 >, < * >, < 60 >$

About the symbol table

Some features

- Data structure containing an entry for each identifier (variable name, procedure name. . .)
- Rapid Read/Write accesses.

Store the various attributes of identifiers:

- allocated memory,
- type,
- scope (locations of the program where the variable can be used),
- for procedure names: number and types of the parameters,
- . . .

Syntactic analysis by a parser

Input: sequence of tokens

Output: abstract syntax tree (AST) + (completed) symbol table

1. syntactic analysis of the input sequence
2. AST construction (from a derivation tree)
   $\hookrightarrow$ depicts the grammatical structure of the program
   - node: an operation
   - children: arguments of operations
3. complete the symbol table

Based on Formal tools: **context-free languages** (CFG)

- described by (LR) context-free grammars
- recognized by (deterministic) push-down automata

Example of parser: Yacc (parser generator)

Parser Generator

Example (Yacc/Bison description)

- description:

```
declarations
%%
rules
%%
procedures
```

- Example of declaration :

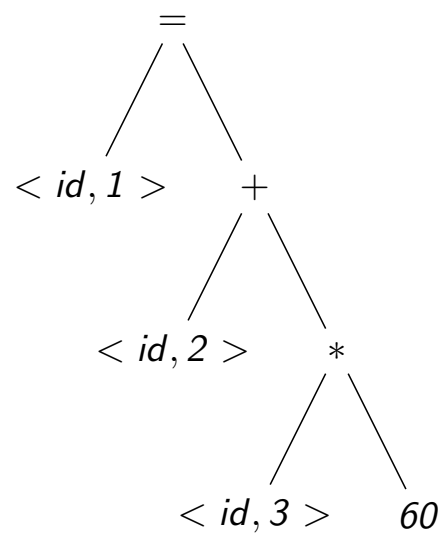
```
%type <u_node> program
%type <u_node> e
```

- Example of rule description :

```
e :  e '+' t
 {$$=m_node(PLUS,$1,$3);}
| t
 {$$=$1;}
;
```

Syntactic analysis on the running example

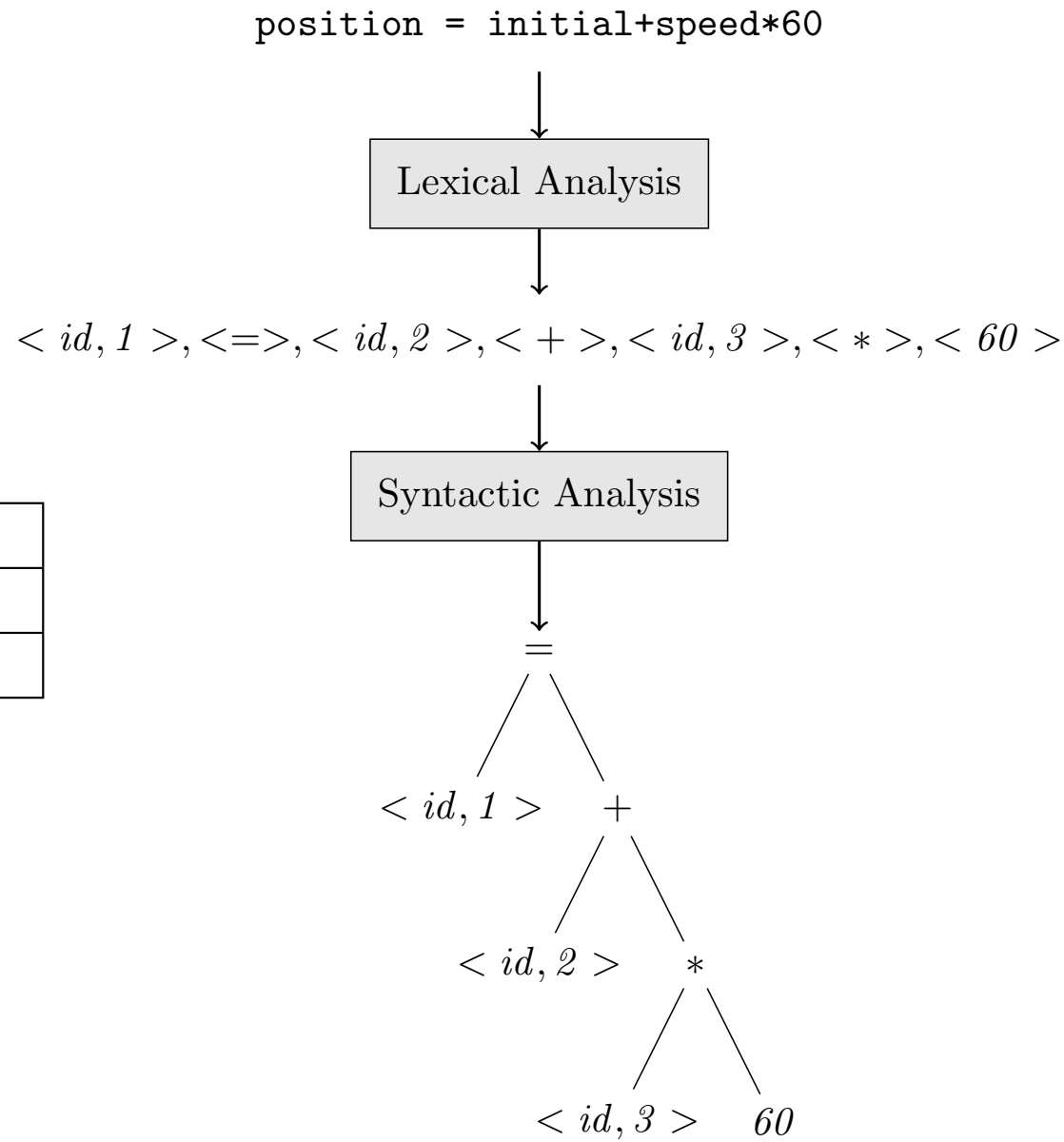
Example (Syntactic analysis of
$< id, 1 >, <=>, < id, 2 >, < + >, < id, 3 >, < * >, < 60 >$)



- $*$ has $< id, 3 >$ as a left-child and 60 as a right child
- The AST indicates the order to compute the assignment (compatible with arithmetical conventions)

The next steps (of analysis and generation) will use the syntactic structure of the tree

Running example: an assignment



Symbol Table

1	position	...
2	initial	...
3	speed	...

Semantic analysis

Input: : Abstract syntax tree (AST) + Symbol Table

Output: : enriched AST / error wrt. semantics

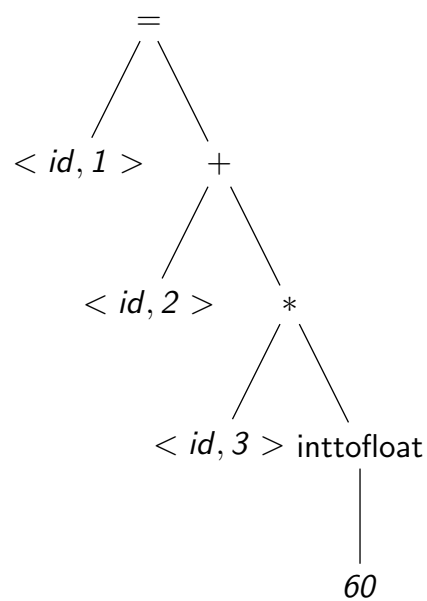
1. name identification:
   $\rightarrow$ bind **use-def** occurrences: variable not declared multiple times, variables declared before assigned, variables assigned before referenced.
2. type verification and/or type inference
   $\rightarrow$ type system
   (e.g., $*$ uses integers, indexes of arrays are integers,...)
3. languages may allow type coercion

$\Rightarrow$ traversals and modifications of the AST

Based on the language semantics

Semantic Analysis of the running example

Example (from the previous AST)

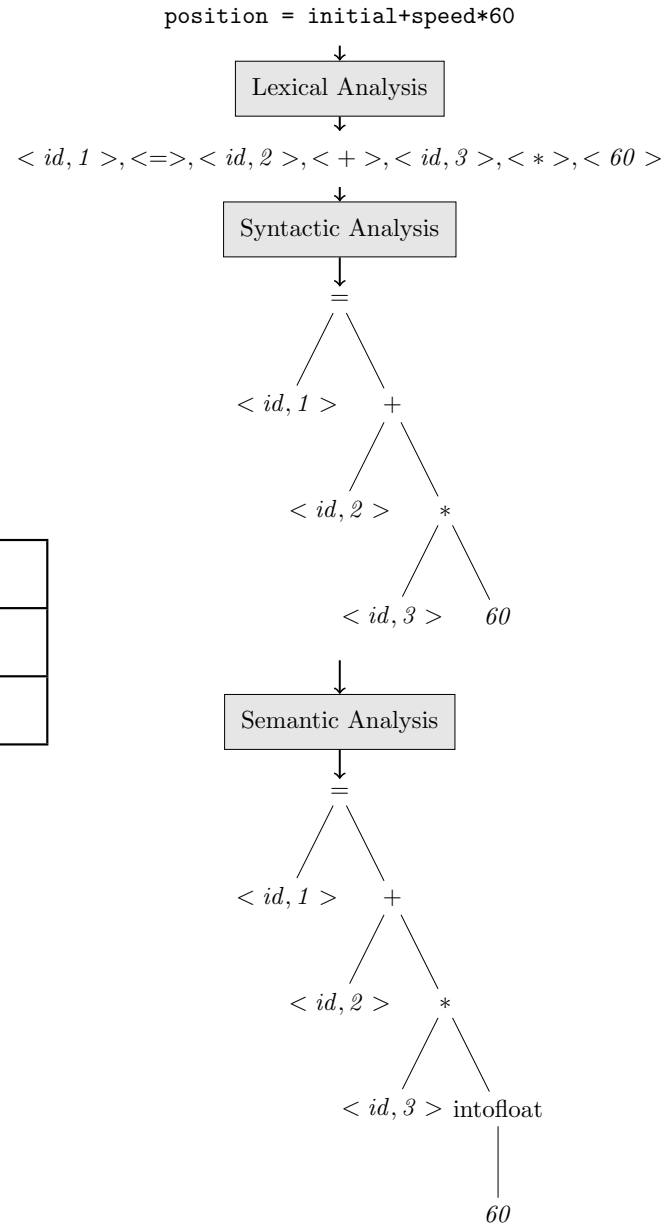


`speed` ($< id, 2 >$) is declared as a float

- Type inference: `position` ($< id, 1 >$) is a float
- Type coercion:
  - 60 denotes an integer

  $\rightarrow$ the integer 60 is converted to a float

Running example: an assignment



Symbol Table

1	position	...
2	initial	...
3	speed	...

Intermediate Code generation

Input: AST

Output: intermediate code (in some intermediate representation)

- based on a systematic translation function f s.t.

$$\mathrm{Sem}_{\mathrm{source}}(P) = \mathrm{Sem}_{\mathrm{target}}(f(P))$$

- in practice: several intermediate code levels
  (to ease the optimization steps)

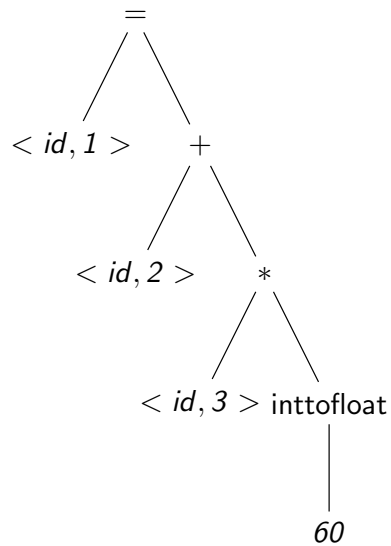
Three concerns for the intermediate representation:

- easy to produce,
- easy to analyze,
- easy to translate to the target machine.

Based on the semantics of the source and target languages

Intermediate code generation on the running example

Example (Generated code from the decorated AST)



$\rightarrow$

```
 t1   = inttofloat(60)
 t2   = id3 * t1
 t3   = id2 + t2
id1   = t3
```

Remarks:

- Every operation has at most one right-hand operand.
- Use the order described by the AST.
- Compiler may create temporary names that receive values created by one operation: `t1, t2, t3`.
- Some operations have less than 3 operands.

Intermediate Code Optimization

Input/Output: Intermediate code

- several criteria: execution time, size of the code, energy
- several optimization levels (source level vs machine level)
- several techniques:
  - data-flow analysis
  - abstract interpretation
  - typing systems
  - etc.

Intermediate Code Optimization on the running example

Example (Optimization of the generated code)

Examining the generated code

```
t1   = inttofloat(60)
t2   = id3 * t1
t3   = id2 + t2
id1  = t3
```

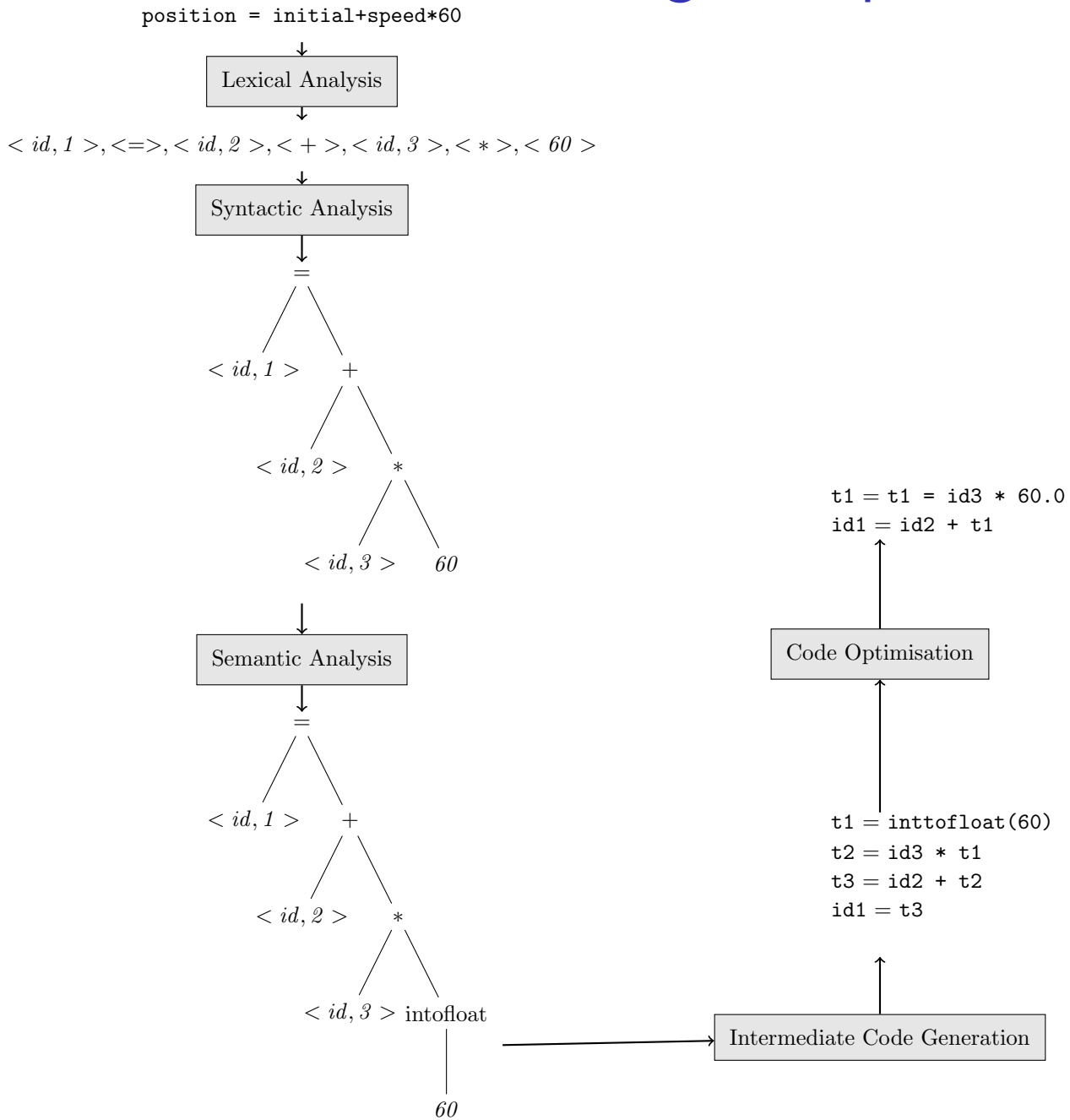
we can notice that:

- conversion of 60 to a float can be done once for all by replacing the inttofloat operation by number 60.0;
- `t3` is only used to transmit the value to `id1`.

→ **The code can be "shortened":**

```
t1   = id3 * 60.0
id1  = id2 + t1
```

Optimization for the running example



(Final) Code Generation

Input: Intermediate code

Output: Machine code

Principles:

- Each intermediate statement is translated into a sequence of machine statements that "does the same job"
- Each variable corresponds to a register or a memory address

Challenge: wisely use the registers

We will study 3-address code (Assembly code)

```
OPER Ri, Rj, Rk     or     OPER Ri, @
```

- at most 3 operands per statement
- 1 operand $\approx$ 1 register
- first operand is the destination

Final Code Generation for the running example

Example (Final code generation from the optimized code)

Input:

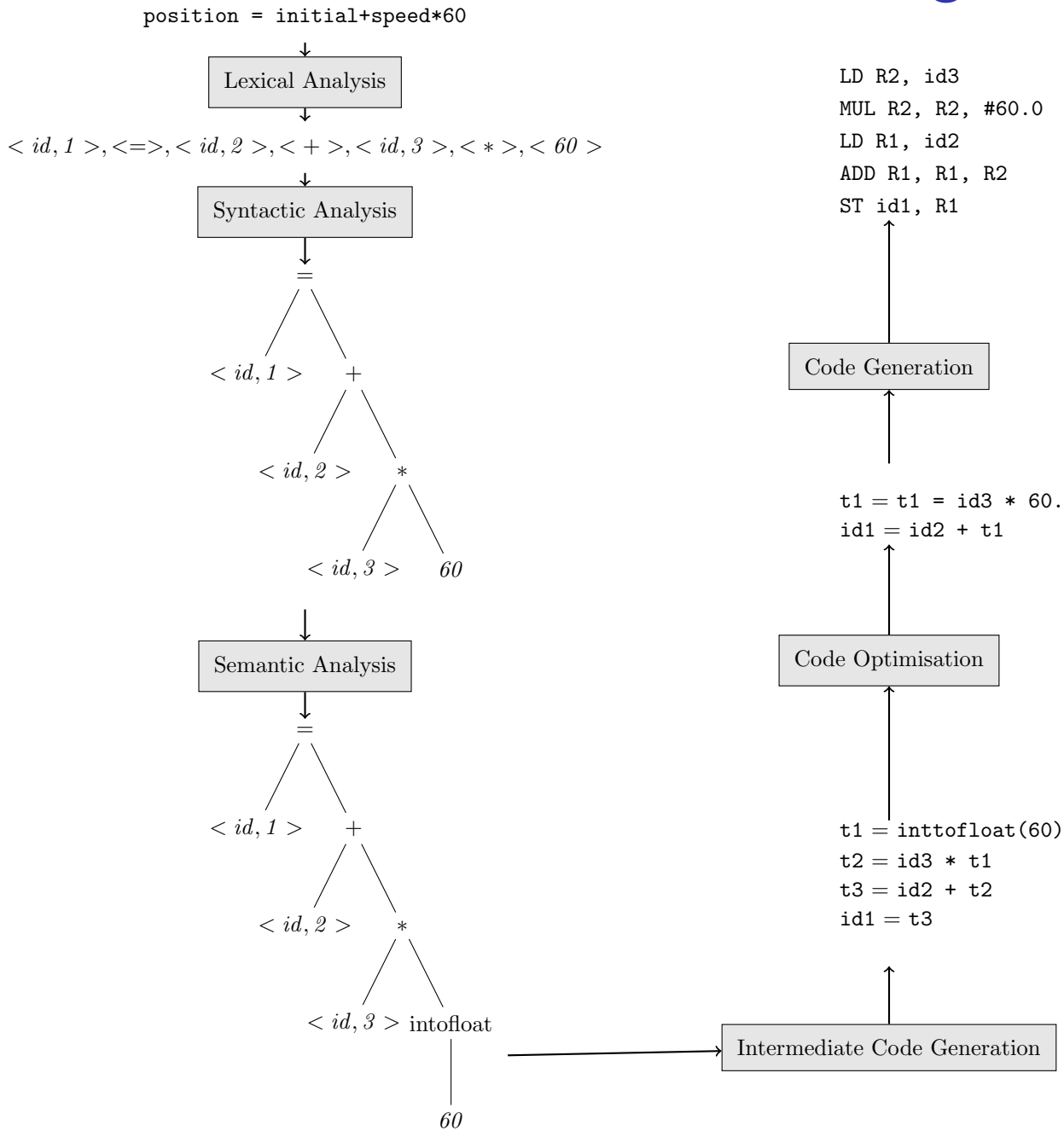
```
t1   = id3 * 60.0
id1  = id2 + t1
```

Output:

```
LD    R2, id3          (loads the content at memory @ id3 into R2)
MUL   R2, R2, #60.0    (multiplies 60.0 by the content of R2)
LD    R1, id2          (loads the content at memory @ id2 into R1)
ADD   R1, R1, R2       (adds the content of registers R1 and R2
                          and stores the result into R1)
ST    id1, R1          (stores the content of register R1 at memory @ id3)
```

- # in #60.0 indicates that it is an immediate constant
- The topic of memory allocation will be addressed in the lecture on code generation.

Final Code Generation for the running example



Outline - Introduction: Compiler architecture, Compiling & Semantics

Motivation

Why do we need to study the semantics of programming languages?

Semantics is paramount to:

- write compilers (and program transformers)
- understand programming languages
- classify programming languages
- validate programs
- write program specifications

Why do we need to formalize this semantics?

Example: static vs. dynamic binding

```
Program     Static_Dynamic
begin       var  x := 0;
            proc p is x := x * 2;
            proc q is call p;
            begin
                var  x := 5;
                proc p is x := x + 1;
                call q; y := x;
            end;
end
```

What is the final value of y?

- dynamic scope for variables and procedures: $y = 6$
- dynamic scope for variables and static scope for procedures: $y = 10$
- static scope for variables and procedures: $y = 5$

Example: parameters

Program value_reference

var a;

proc $p(x)$;

  begin

    $x := x + 1$; $write(a)$; $write(x)$

  end;

begin

  $a := 2$; $p(a)$; $write(a)$

end;

What values are printed?

	$p(a)$		$write(a)$
call-by-value	2	3	2
call-by-reference	3	3	3

Overview of the semantics part of the course

Various Semantic styles:

Operational semantics: "How a computation is performed?" - meaning in terms of "computation it induces"

- Natural: "from a bird-eye view"
- Operational: "step by step"

Axiomatic semantics (Hoare logic): "What are the properties of the computation?"

- Specific properties using assertions, pre/post-conditions
- Some aspects of the computation are ignored

Denotational semantics: "What is performed by the computation?"

- Meaning in terms of mathematical objects
- Only the effect

Different styles/techniques for different purposes: not rival!

Language families:

- imperative
- functional

Outline - Introduction: Compiler architecture, Compiling & Semantics

Inductive/Compositional definitions

Let us consider:

- E a set ,
- $f : E \times E \times \ldots \times E \to E$ a partial function,
- $A \subseteq E$ a subset of E.

Definition (closure)

A is closed by f iff $f(A \times \ldots \times A) \subseteq A$.

Definition (Construction rule)

A construction rule for a set states either:

- that a *basis element* belongs to the set, or
- how to produce a new element from existing elements (*production rule* given by a partial function).

Definition (Inductive definition)

An inductive definition on E is a family of rules defining the smallest subset of E that is *closed* by these rules.

Inductive definitions: examples

Example (Natural numbers)

How can define them?

- basis element 0
- 1 rule: $x \mapsto succ(x)$

2 is the natural number defined as $succ(succ(0))$

Example (Even numbers)

- basis element: 0;
- 1 rule: $x \mapsto x + 2$.

Example (Palindromes on $\{a, b\}$)

- basis elements: ϵ, a, b;
- 2 rules: $w \mapsto a \cdot w \cdot a$, $w \mapsto b \cdot w \cdot b$.

A notation: derivation tree

Notation for $t = f(x_1, \ldots, x_n)$

$$\frac{x_1 \quad \ldots \quad x_n}{t}$$

"t is built/obtained from $x_1, \ldots, x_n$".

Example (Derivation trees)

- $2 = succ(succ(succ(0)))$ is a natural number

$$\frac{\frac{\frac{\overline{0}}{1}}{2}}$$

- *aba* is a palindrome:

$$\frac{\overline{b}}{aba}$$

- *ababa* is a palindrome:

$$\frac{\frac{\overline{a}}{bab}}{ababa}$$

Abstract syntax trees and derivation trees

Consider an abstract syntax tree, produced by syntactic analysis.

Derivation tree

For each node, computing information from the information of its sons.

Example (Derivation trees in type analysis)

We obtain for each node a type (or error) based on types of its sons.

Definition of derivation trees by a formal system

Generally, we have information, stored in some environment Γ. The formal system states how to *deduce* knowledge from existing knowledge, where knowledge is of the form $\Gamma \vdash \mathcal{P}$, which means $\mathcal{P}$ holds on Γ.

- a set of axiom schemes
- a set of inference rules : a rule of the form

$$\frac{\Gamma_1 \vdash \mathcal{P}_1 \quad \cdots \quad \Gamma_n \vdash \mathcal{P}_n}{\Gamma \vdash \mathcal{C}}$$

"if the hypothesis (premisses) $\mathcal{P}_i$ hold, then the conclusion $\mathcal{C}$ holds.

(Some simple) Proof techniques

Proof by contradiction, reducto-ad-absurdum, contraposition,...

Structural Induction

- Proof for the basic elements, atoms, of the set.
- Proof for composite elements (created by applying) rules:
  - assume it holds for the immediate components (induction hypothesis)
  - prove the property holds for the composite element

Induction on the shape of a derivation tree

- Proof for 'one-rule' derivation trees, i.e., axioms.
- Proof for composite trees:
  - For each rule R, consider a composite tree where R is the last rule applied
  - Proof for the composite tree
    - Assume it holds for subtrees, or premises of the rule (induction hypothesis)
    - Proof for the composite tree

Outline - Introduction: Compiler architecture, Compiling & Semantics

Summary of Introduction: Compiler architecture, Compiling & Semantics

Summary

- Architecture of a compiler and its logical steps: lexical analysis, syntactic analysis, semantic analysis, intermediate code generation, code optimization, final-code generation.
- The semantics of a programming language is a key element in its definition and should be formally defined.
- Math reminders: inductive definitions, proof techniques.