Programming Language Semantics and Compiler Design / Sémantique des Langages de Programmation et Compilation

Introduction, Compilation & Semantics, Compiler Architecture

Yliès Falcone
ylies.falcone@univ-grenoble-alpes.fr — www.ylies.fr
Univ. Grenoble Alpes, and LIG-Inria team CORSE

Master of Sciences in Informatics at Grenoble (MoSIG)
Master 1 info

Univ. Grenoble Alpes - UFR IM^{2}AG
www.univ-grenoble-alpes.fr - im2ag.univ-grenoble-alpes.fr

Academic Year 2017 - 2018

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 (source language)
- to a program executable by a machine (target language).

source program (p_S) of source Language (L_S) ($p_S \in L_S$) $\longrightarrow$ $\boxed{\text{COMPILER}}$ $\longrightarrow$ target program (p_T) of target Language (L_T) ($p_T \in L_T$)

The algorithms and design principles used in compilers are general, generic, and are used in many domains of computer science and ICT.

Outline - Introduction, Compilation & Semantics, Compiler Architecture

Outline - Introduction, Compilation & Semantics, Compiler Architecture

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

Describing a programming language P

Lexicon L: words of P
$\rightarrow$ a regular language over alphabet P

Syntax S: sentences of P
$\rightarrow$ a context-free language over L

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

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

Meaning?

But How to define the meaning of programs?
$\rightarrow$ The semantics of programs

Semantics?

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

Overview of the semantics part of the course

Different styles/techniques for different purposes

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

Semantics also depends on the language "family":

- imperative
- functional

Outline - Introduction, Compilation & Semantics, Compiler Architecture

Compilers: what you surely already know...

A compiler transforms a program

- from a language we can ("easily") understand: the source language,
- to a language the machine can understand and execute: the target language.

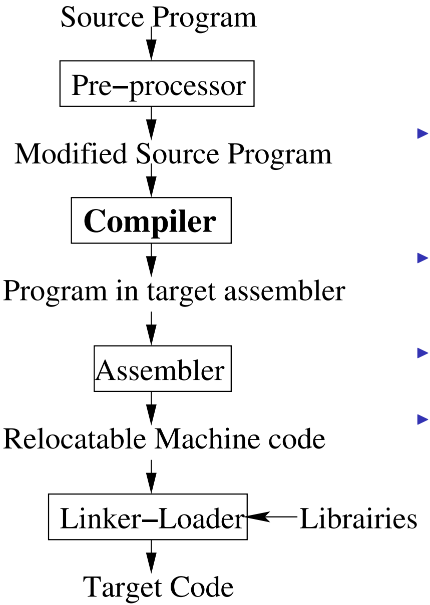
source program (p_S)
of source Language (L_S) $\longrightarrow$ $\boxed{\text{COMPILER}}$ $\longrightarrow$ of target Language (L_T)
($p_S \in L_S$)

target program (p_T)
($p_T \in L_T$)

A compiler is a *language processor*.

Remark The language in which the compiler is written is referred to as the *implementation language*. □

Compilers: what you surely already know. . . (ctd)



- 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.

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, easiness to produce, security (e.g., Java), better verification and error-reporting.

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, then passed to an interpreter (e.g., Java).

Compilers: what do we expect?

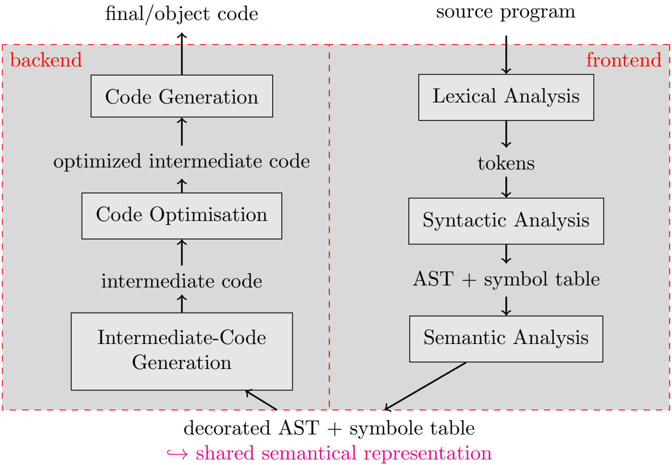
source program (p_S) of source Language (L_S) ($p_S \in L_S$)	$\longrightarrow$	$\boxed{\text{COMPILER}}$	$\longrightarrow$	target program (p_T) of target Language (L_T) ($p_T \in L_T$)

Expected Properties?

- Correctness: execution of p_T should preserve the **semantics** of p_S.
- Efficiency: p_T should be optimized w.r.t. some execution resources (e.g., time, memory, energy, etc.).
- "User-friendliness": errors in p_S violating the "rules" of L_S should be accurately reported.
- Completeness: any correct L_S-program should be accepted.
- Reasonable compilation time: only expect that it is linear in the size of p_S.

Structure and architecture of a compiler

The logical steps/modules and architectural considerations



- each module is as "filter"

+ runtime system for admin., e.g., allocation, task management

Architectural considerations

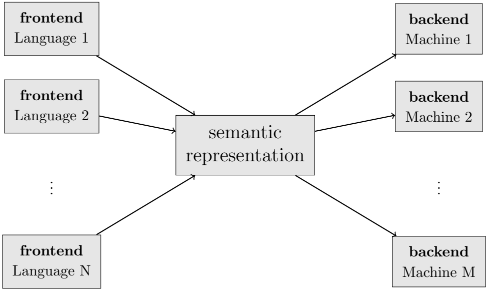
- width of a compiler (size of processed "chunks"): **wide** vs narrow
- passes (iterations on the code)
- steps can be grouped into passes

Structure of a compiler: a quick note

On the importance of the semantic representation

Ideally:

- frontend is independent from the target language
- backend is independent from the source language



Running example: an assignment

Consider the assignment $\texttt{position} = \texttt{initial} + \texttt{speed} * 60$

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

```
position = initial + speed * 60
```

$\downarrow$

COMPILER

Lexical Analysis

Syntactic Analysis

Semantic Analysis

Intermediate-Code Generation

Optimisation

Code Generation

$\downarrow$

?

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

`position = initial + speed * 60`

↓

Lexical Analysis

↓

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

Symbol Table

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

About the symbol table

Some features

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

Store the various attributes of identifiers:

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

Implementation with linear lists, hash tables, trees.

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 :

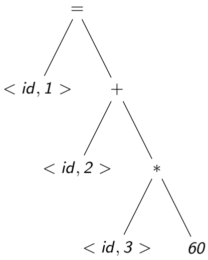
```
%type <u_node> program
%type <u_node> e
```

- Example of rule description :

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

Syntactic analysis on the running example

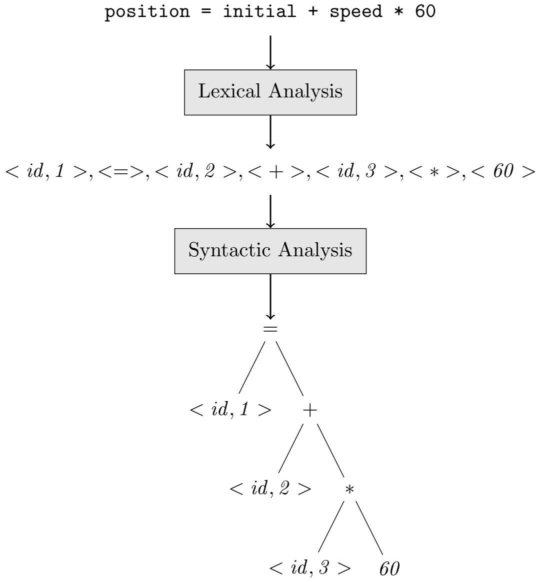
Example (Syntactic analysis of $< id, 1 >, <=>, < id, 2 >, < + >, < id, 3 >, < * >, < 60 >$)



- $*$ 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

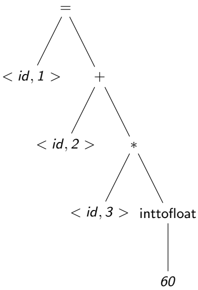
1. name identification:
   $\rightarrow$ bind **use-def** occurrences: variable not declared multiple times, variables declared before assigned, variables assigned before referenced.
2. type verification and/or type inference
   $\rightarrow$ type system
   (e.g., $*$ uses integers, indexes of arrays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

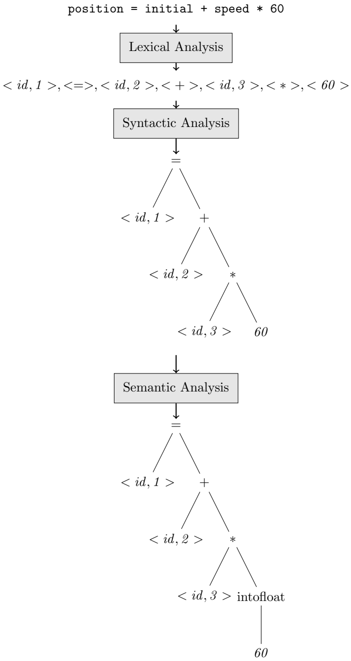


`speed` ($< id, 2 >$) is declared as a float

- 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.

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

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

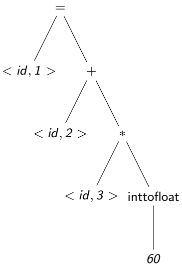
Three concerns for the intermediate representation:

- 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
```

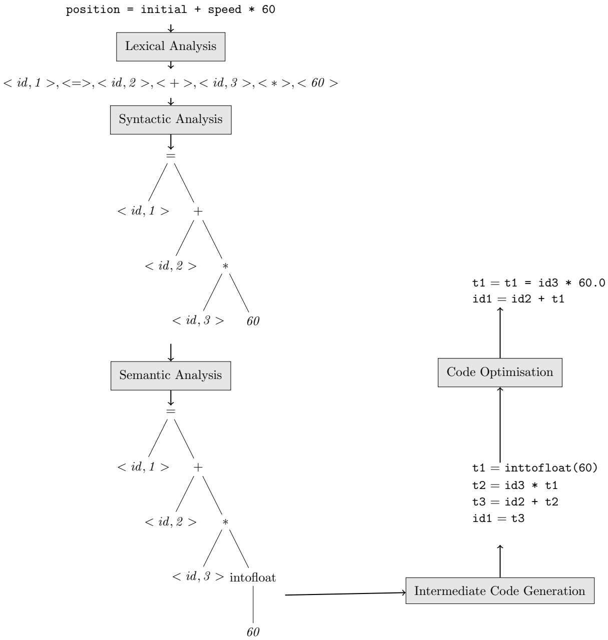
we can notice that:

- 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:

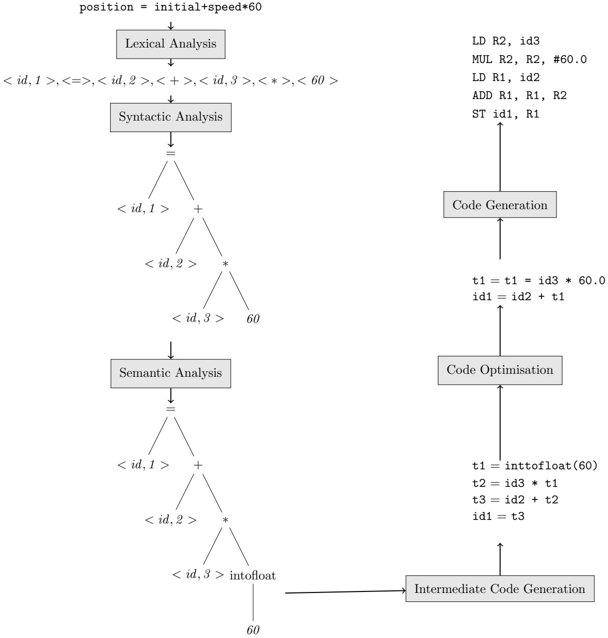
```
t1   = id3 * 60.0
id1  = id2 + t1
```

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, Compilation & Semantics, Compiler Architecture

Summary of Introduction, Compilation & Semantics, Compiler Architecture

Key points

- Architecture/structure of a compiler and its logical steps:
  - *front end*: lexical analysis, syntactic analysis, semantic analysis,
  - *back end*: intermediate-code generation, code optimization, final-code generation.
- The semantics of a programming language:
  - is a key element in its definition;
  - allows to write the back-end of a compiler (describes the meaning of the AST representation of programs) and the semantic analysis part of the frontend (to reject incorrect programs);
  - should be formally defined to avoid ambiguity and permit automation.