

Programming Language Concepts and Semantics Part I

Bernd Meyer
bernd.meyer@infotech.monash.edu.au

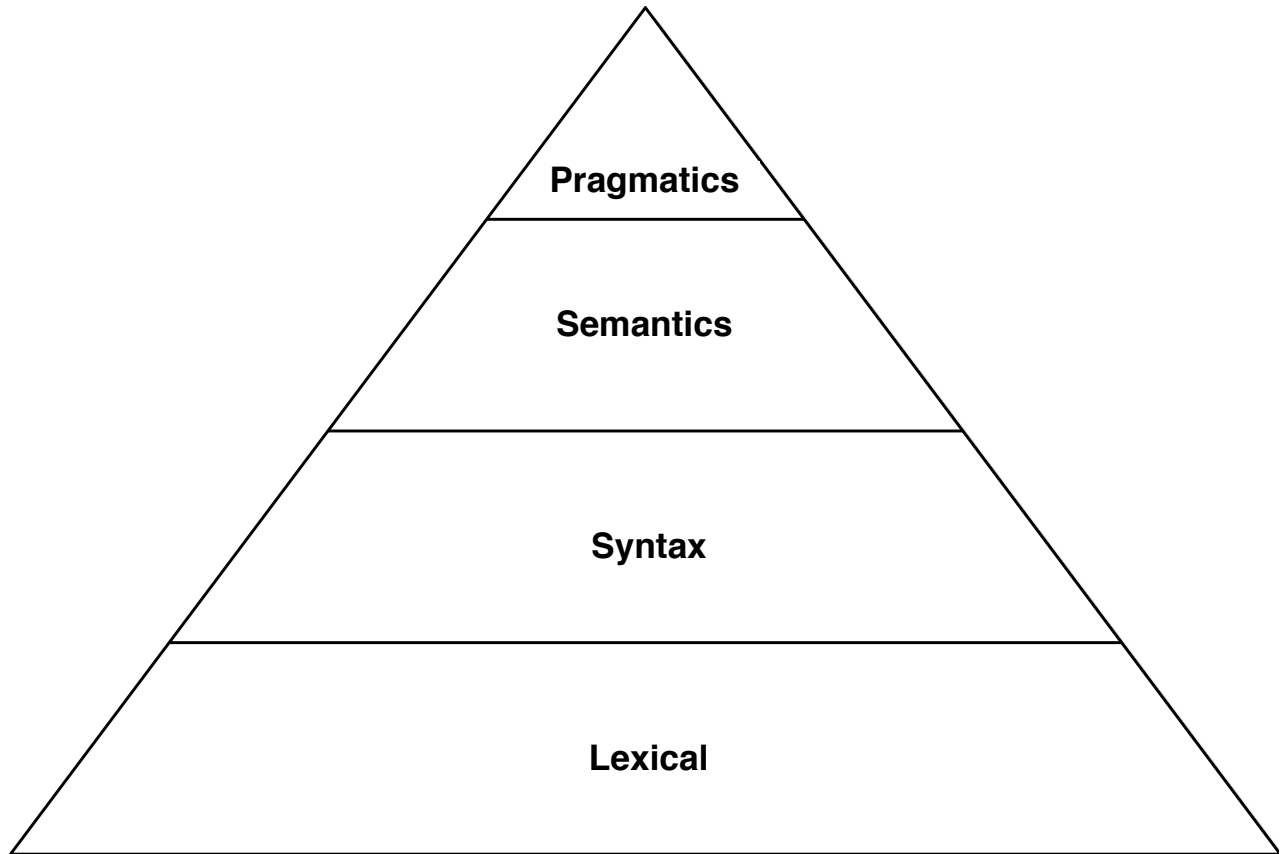
Consultation
11am-1pm, wednesdays, Rm 148, Bldg 75

Part I treats the Basics of Formal Semantic Specification and provides an introduction to the different types of formal semantic specification.

- **Wk 1: Introduction, Syntax vs Semantics, Grammars.**
- **Wk 2: Parsing and Translational Semantics.**
- **Wk 3: Operational Semantics of IMP.**
- **Wk 4: Denotational Semantics of IMP.**
- **Wk 5: Axiomatic Semantics of IMP.**
- **Wk 6-12: Semantics of ML using Modular Structured Operational Semantics**

The material in weeks 3-5 is based on Glynn Winskel's book, the material in weeks 6-12 on Peter Mosses script.

The Levels of Language



Pragmatics: Intends to make the listener pass the salt

Semantics: Query whether the listener has the ability to pass the salt

Syntax: question(inflexion(can), nounphrase(pronoun(you)),
verbphrase(verb(pass)),
nounphrase(determiner(the), noun(salt)))

Lexical Level: can, you, pass, the, salt, ...

The Lexical Level

Main Entry: lex·i·cal 🔊

Pronunciation: 'lek-si-k&l

Function: *adjective*

1 : of or relating to words or the vocabulary of a language as distinguished from its grammar and construction

2 : of or relating to a [lexicon](#) or to [lexicography](#)

- **lex·i·cal·i·ty** 🔊 /'lek-s&-'ka-l&-tE/ *noun*

- **lex·i·cal·ly** 🔊 /'lek-si-k(&-)lE/ *adverb*

Source: Merriam Webster online edition at <http://www.m-w.com>

Syntax

Main Entry: **syn·tax** 🗣️

Pronunciation: 'sɪn-'taks

Function: *noun*

Etymology: French or Late Latin; French *syntaxe*, from Late Latin *syntaxis*, from Greek, from *syntassein* to arrange together, from *syn-* + *tassein* to arrange

1 a : the way in which linguistic elements (as words) are put together to form constituents (as phrases or clauses) **b** : the part of grammar dealing with this

2 : a connected or orderly system : harmonious arrangement of parts or elements

3 : syntactics especially as dealing with the formal properties of languages or calculi

Source: Merriam Webster online edition at <http://www.m-w.com>

Note how only here the word *formal* occurs. This also reflects the treatment that programming languages are often given in practice: only the syntactic level and the lexical level below it are defined formally.

Semantics

Main Entry: **se·man·tics** 🗣️

Pronunciation: *si-'man-tiks*

Function: *noun plural but singular or plural in construction*

1 : the study of meanings: **a** : the historical and psychological study and the classification of changes in the signification of words or forms viewed as factors in linguistic development **b**

(1) : **SEMIOTIC** (2) : a branch of semiotic dealing with the relations between signs and what they refer to and including theories of denotation, extension, naming, and truth

2 : **GENERAL SEMANTICS**

3 a : the meaning or relationship of meanings of a sign or set of signs; *especially* : connotative meaning **b** : the language used (as in advertising or political propaganda) to achieve a desired effect on an audience especially through the use of words with novel or dual meanings

Source: Merriam Webster online edition at <http://www.m-w.com>

Pragmatics

Main Entry: prag·mat·ics 

Pronunciation: prag-'ma-tiks

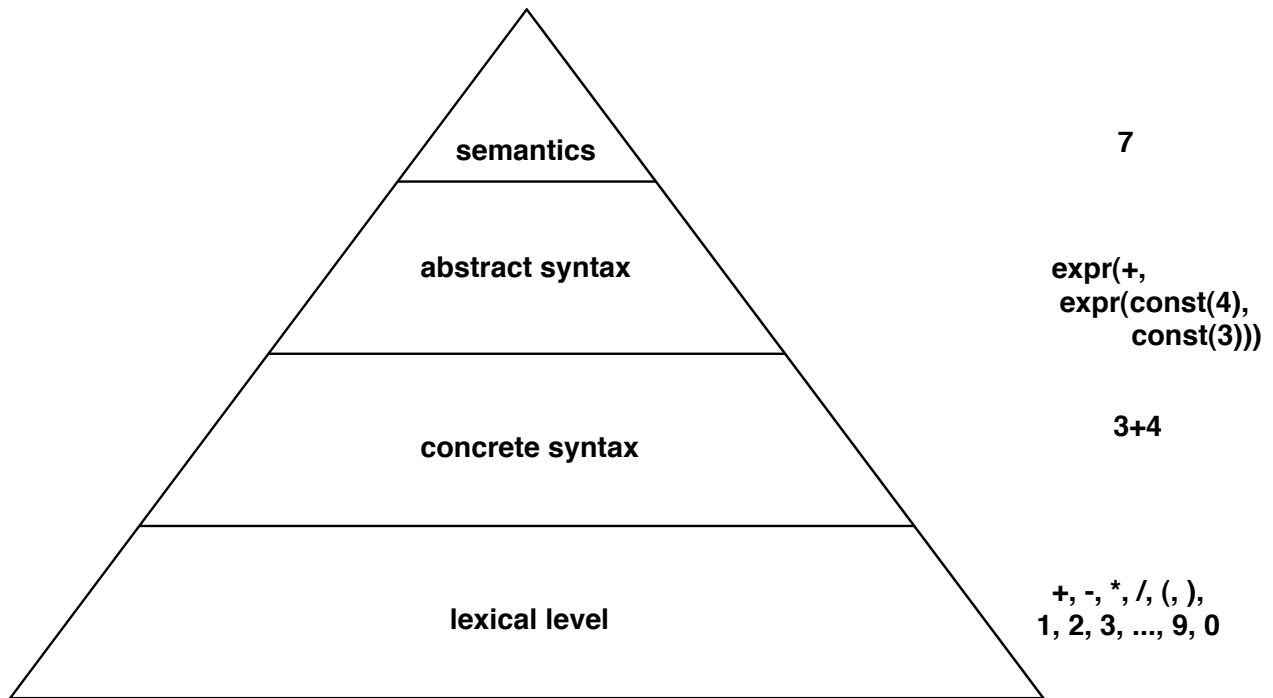
Function: *noun plural but singular or plural in construction*

1 : a branch of semiotic that deals with the relation between signs or linguistic expressions and their users

2 : linguistics concerned with the relationship of sentences to the environment in which they occur

Source: Merriam Webster online edition at <http://www.m-w.com>

Levels of Programming Languages



understanding normal language requires mastering all layers, the same holds for programming languages.

- important: distinguish *concrete* and *abstract* syntax.
- semantics and pragmatics merge in programming languages.

Syntax of Programming Languages

for i = 0 to 10	for i := 0 to 10 step 1
begin	{
j := j + 1	j := j + 1;
end	}

There are some subtle differences in the syntax of these two program segments. Some are in the concrete syntax, some in the abstract syntax.

If you want to write a program you have three ways to figure out the correct syntax:

- use trial and error and rely on the compiler to give you syntax warnings.
- look at examples in books and manuals and infer the syntax intuitively.
- look up a formal syntax definition (grammar or syntax diagram)
*loop ::= for var := **int** to **int** *block**

A program that is compiled without syntax errors is definitely syntactically correct. But it may not necessarily do what you want!

Why Semantics Definitions?

The semantics of a program tells you, roughly speaking, what a syntactically correct program will do or what it will compute (but not necessarily how it will do this).

It also gives you additional conditions for correctness of a program that go beyond simple syntax (for example, the correctness of typing or that methods are called with appropriate parameters).

Example 1 (Java)

<code>float y = 2.0f;</code>	<code>int y = 2;</code>	<code>int y = 2;</code>
<code>float x :=</code>	<code>float x :=</code>	<code>float x :=</code>
<code> (1 / y);</code>	<code> (float) (1 / y);</code>	<code> 1 / (float) y;</code>
<code>println(x);</code>	<code>println(x);</code>	<code>println(x);</code>

Advanced applications of formal semantics are, for example, to prove the correctness of a program segment or to prove equivalence of two program segments (such that they are interchangeable).

Why Semantics Definitions? (cont.)

Example 3 (Lisp)

(let ((x 1))	(let ((x ... <i>some function</i>))
(let ((x 2)	(let ((x 2)
(y (+ x 1)))	(y (+ x 1)))
(print y)))	(print y)))

Without a proper definition of semantics you cannot be sure what the result of a computation is. Note that this is different from syntax which can statically be checked at compile time, so that once a program is compiled without error you can be sure it is syntactically correct.

- Trial and error is unreliable. You may not even be aware that there could be a problem with different input values.
- Using an informal semantics definition from a textbook or manual, is the most common approach to understanding semantics, but in principle has the the same problems as trial and error. You have to *hope* that the authors have thought of and illustrated every pitfall.
- If available you can look up a formal semantics definition. Unfortunately, these are usually very complex and many languages have never been fully defined formally. One laudable counterexample is the functional language ML which we will use in this course.

The *only* way to be absolutely sure what a program will do for any input is to consult a formal definition of the program language semantics.

Pitfalls of Informal Semantics Definitions

Example 3 (Algol 60, call-by-name)

```
begin
  procedure zero(Arr, i, j, u1, u2);
    integer Arr;
    integer i,j,u1,u2;
    begin
      for i := 1 step 1 until u1 do
        for j := 1 step 1 until u2 do
          Arr := 0;
        end;
      end;

      integer array Work[1:100,1:200];
      integer p, q, x, y, z;

      x := 100;
      y := 200;

      zero(Work[p,q], p, q, x, y);
    end
end
```

What does this program do? (Hint: it uses call by name).

Call-by-name was only explained as a concept, but not in formal detail. This later led to unanticipated problems what call-by-name was *really* meant to do in more complex usages.

Formal Semantics Definitions

Example 4 (Operational Semantics, while loops)

$$\frac{\langle b, \sigma \rangle \rightarrow \mathbf{false}}{\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma} \text{ (w1)}$$

$$\frac{\langle b, \sigma \rangle \rightarrow \mathbf{true} \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle \mathbf{while } b \mathbf{ do } c, \sigma'' \rangle \rightarrow \sigma'}{\langle \mathbf{while } b \mathbf{ do } c, \sigma \rangle \rightarrow \sigma'} \text{ (w2)}$$

These rules state the formal semantics of a while loop in *operational* form.

Here, b and c are statements (or code blocks) and $\sigma, \sigma', \sigma''$ are execution states.

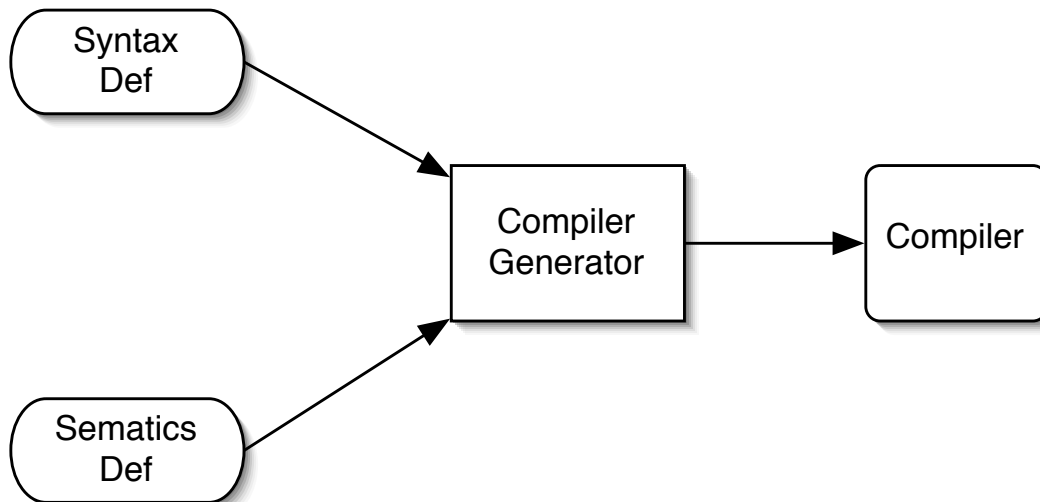
Roughly speaking, the rules can be read as follows:

- w1: If in a state σ the execution of b yields the value false, then the execution of the statement “**while** b **do** c ” in the same state σ will leave the execution state unchanged.
- w2: If in a state σ the execution of b yields the value true and the execution of c in the same state σ yields the state σ'' and the execution of the statement “**while** b **do** c ” in this state σ'' leads to the state σ' then the execution of “**while** b **do** c ” in the initial state σ will yield the state σ' .

You can see, that an operational semantics specification is in principle a very fine-grained specification of the execution of a program.

Benefits of Formal Semantics

- No guesswork, we know precisely what a program is doing and hence how to write a program that does what we want it to do.
- Formal proofs of program properties such as type equivalence, contextual equivalence or partial correctness become possible.
- The implementation of a programming language (compiler, interpreter) can be verified against the spec so that we can guarantee that the “meaning” of a program is independent from any given implementation.
- Formal specifications can serve as a basis for high-level meta tools that automatically generate compilers, interpreters, syntax-directed editors, and other parts of a programming environment.



Simple Proof Example, Axiomatic Semantics

Axiomatic Semantics uses pre-conditions and post-conditions as a specification mechanism. If the pre condition of a code fragment is guaranteed to hold, then so is the post condition. Different fragments can then be combined by chaining pre and post conditions using standard rules of formal reasoning.

Example 4 (Axiomatic Semantics)

$$\{Y=a\} Y := Y + 1 \{Y=a + 1\} \quad \{Y \neq 0, Y=b\} X := 1 / Y \{X=1/b\}$$
$$\{Y=a, a > 0\}$$
$$Y:=Y+1$$
$$\{Y > 0, Y=a + 1\}$$
$$X := 1 / Y$$
$$\{X=\frac{1}{a+1}\}$$

Types of Semantic Specifications

The following are the main types of semantics specifications:

- Operational Semantics,
particularly *structural operational semantics*
 - uses state transformation rules
 - is based on term rewriting
- Axiomatic Semantics
 - uses pre and post-conditions
 - is based on Hoare Logic, which in turn is based on first order predicate logic
- Denotational Semantics
 - uses “meaning functions” (denotations)
 - is based on Domain Theory
(continuous functions and complete partial orders)
- Another relatively popular form is Abstract Machine Semantics, which in principle is a form of operational semantics.
- Finally, any formal mapping of a program language to any other language can be understood as a form of *Translational Semantics*. If this mapping and the semantics of the target language are formally well-defined, this can be understood as a formal semantics.

Overview of Provisional Schedule

- Wk 1: Overview; Syntax vs Semantics (Bernd)
- Wk 2: Grammars, Parsing and Translational Semantics (Bernd)
- Wk 3: Operational Semantics (Bernd)
- Wk 4: Denotational Semantics (Bernd)
- Wk 5: Axiomatic Semantics (Bernd)
- Wk 6: Semantics of ML: Introduction to ML0 (Maria)
- Wk 7: Modular Structural Operational Semantics (Maria)
- Wk 8: Expressions and Declarations in ML0 (TBA)
- Wk 9: Types and References in ML (Bernd)
- Wk 10: Type Inference and MSOS Rules for Types (Bernd)
- Wk 11: Imperatives in ML0 (Maria)
- Wk 12: Abstractions and Recursion in ML0 (Maria)
- Wk 13: Revision (Maria & Bernd)

2 Hours of lectures per week, tutorials will only be held in some weeks.

Please refer to <http://www.csse.monash.edu.au/courseware/cse5340/> for details and courseware.

Assessment

There will be two written assignments each worth 50%. The first assignment covers weeks 1-6 and is handed out at the end of week 6, the second one covers weeks 7-12 and is handed out at the end of week 12.

Assignment 1 (50%)

due September 20, COB (Week 10);

Assignment 2 (50%)

due November 1, COB (Exam Period, Week 2)

Prerequisite Knowledge

Proficiency in at least one programming language, knowledge of alternative programming paradigms, such as functional programming (as for example given in CSE3322), logic programming (as for example given in CSE2393) or concurrent programming paradigms (as for example given in CSE4333). While any knowledge of alternative programming paradigms is advantageous, basic knowledge of functional programming (as given in CSE3322) is particularly important. A solid understanding of recursion. At least 12 points of Computer Science-oriented mathematics (for example, MAT1841+MAT1830 or MAT1811+1812). Fundamentals of first-order predicate logic (for example from CSE2303). Understanding of the notion of a proof, induction.

Formal Prerequisites are CSE3322 or entry to the Master of Computer Science degree. Unless you meet these, your prerequisites must be waived by one of the unit leaders for you to participate in this unit.

Reading

- The first half of the course is in parts based on Chapters 2 and 5–7 of
 - Glynn Winskel. *The Formal Semantics of Programming Languages*, The MIT Press, Cambridge/MA, 1993.
- The second half of the course is based on the following script. It is not publicly available, but will be made available to students of this subject in the lectures (courtesy of Peter D. Mosses).
 - Peter D. Mosses. *Fundamental Concepts and Formal Semantics of Programming Languages*, Lecture Notes. University of Aarhus.
- The following books are not required for the course, but constitute useful and good reading
 - David A. Watt. *Programming Language Syntax and Semantics*, Prentice Hall, Englewood Cliffs, 1991.
 - Bertrand Meyer. *Introduction to the Theory of Programming Languages*, Prentice Hall, Englewood Cliffs, 1990.
 - Peter D. Mosses. *Action Semantics*, Cambridge University Press, Cambridge, UK, 1992.

Cheating

The fineprint: It is important that your solutions to the assignment questions be your own work. It is perfectly acceptable to seek help and advice when completing the assignments, but this must not be taken to the point where what is submitted is in part someone else's work.

Please note that, since the assignments are used in assessing your final grade in this subject, the following Faculty policy applies. "Students should note that cheating is regarded as a very serious offence which is likely to lead not only to failure in the subject concerned but also to additional penalties including exclusion. Students should carefully note that the taking of any unauthorised material into examinations such as notes and unauthorised dictionaries will be regarded as cheating. Students should also note that essays, assignments and other work are generally understood to be the student's own work and where such work is identical with, or similar to, another student's work, an assumption of cheating may arise. Where students wish to undertake work in conjunction with other students, it is suggested that the matter be discussed with the lecturer concerned." Faculty of Computing and Information Technology Handbook In addition, the following School policy applies. "The assignments set in this subject are designed primarily as learning exercises, but they also contribute to your final grade.

Copying of other student's assignment solutions is unacceptable.

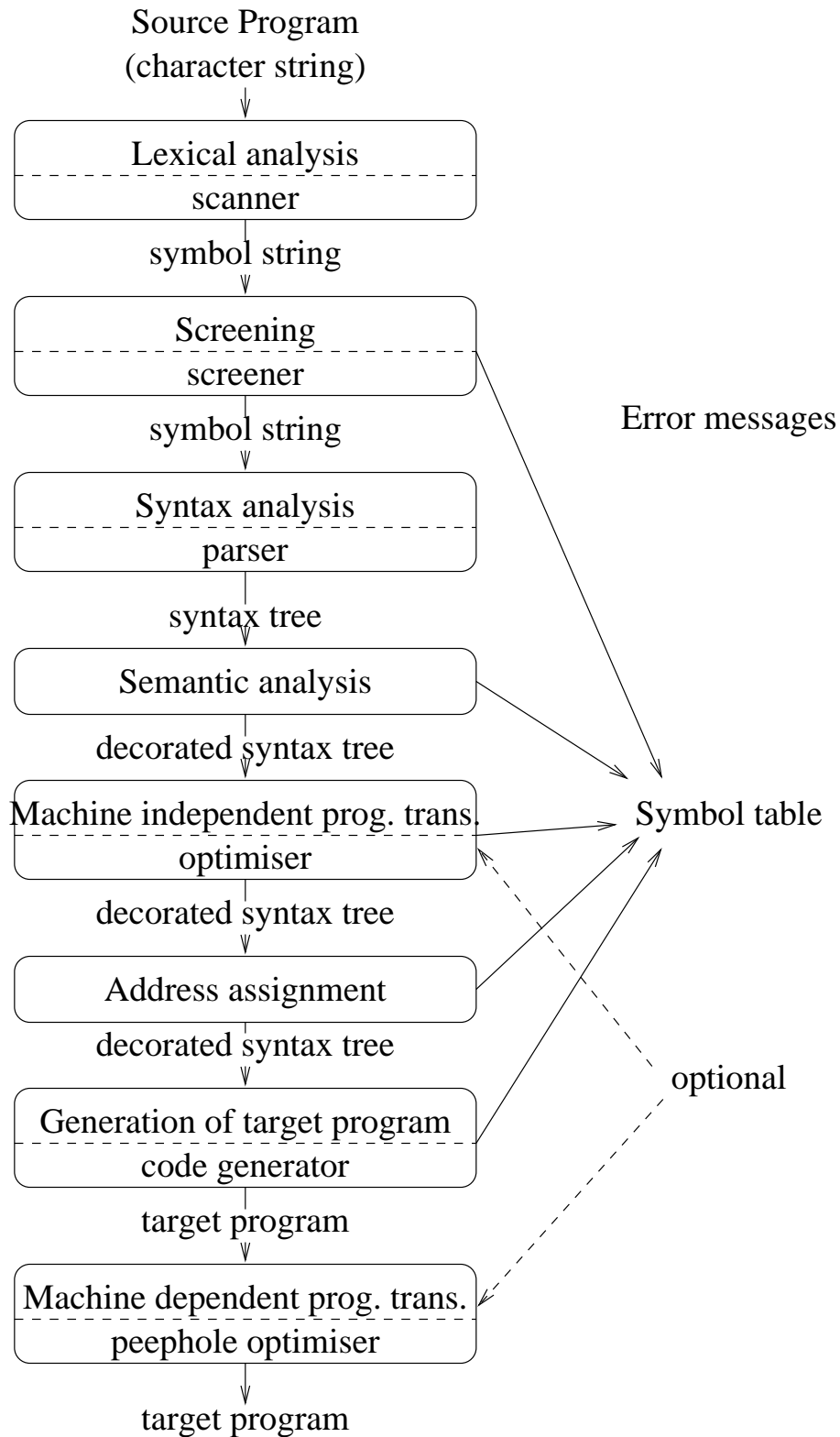
All students have a responsibility to ensure that their assignment solutions are their own work. You must ensure that others do not obtain access to your solutions for the purpose of copying a part of them. Where such plagiarism is detected, both of the assignments involved will receive no marks.

In significant cases of plagiarism, action may be taken against the offenders under the University's disciplinary regulations.

Please be aware of the faculty's plagiarism policies at:

<http://www.csse.monash.edu/~ajh/adt/policies/>

Phases of a Compiler



Lexical Analysis

The **scanner** (lexical analyser) converts the source program (string of characters) into a sequence of tokens (i.e. lexical units).

Tokens are, for example

- identifiers (“this”, “x”, ...)
- numbers and other constants (“3.14”, “99”, “A String t”, ...)
- operators (“+”, “-”, “(”, ...)

At the same time meaningless whitespace (blanks, tabs, line breaks, etc.) are stripped.

In some cases the scanner already strips comments, too.

Lexical Analysis

Example

```
strcpy(s, t)
  char *s, *t;
  { while(*s++ = *t++); }
```

```
[ strcpy, (, s, ,, t, ), char, *, s, ,, *, t, ;, {,
while, (, *, s, +, +, =, *, t, +, +, ),;, } ]
```

Methods

For the specification of the token-level we use regular expressions or regular (type 3) grammars

Recognition of these can be implemented with finite state automata (FSA).

Screening

The Screener extends the lexer:

- recognition of reserved words (“while”, “return” ...)
- stripping comments
- recognizing compiler directives (so-called pragmas)
- creating and initializing the *symbol table*

Screening and Lexing can run in a single pass before the syntax analysis or interleaved with syntax analysis (on-demand).

Example

```
[ strcpy, (, s, ,, t, ), char,  
*, s, ,, *, t, ;, {,  
while, (, *, s, +, +, =,  
*, t, +, +, ),;, } ]
```

```
[ res(strcpy), (, id(1), ,, id(2), ),  
res(char), op(*), id(1), ,, op(*), id(2), ;,  
{, res(while), (, op(*), ... ]
```

Methods

Regular grammars or regular expressions implemented with finite state automata (FSA). Additionally table look-up and calls to arbitrary (programmed) functions are used to achieve the additional capabilities, such as recognition of reserved words.

Syntax Analysis

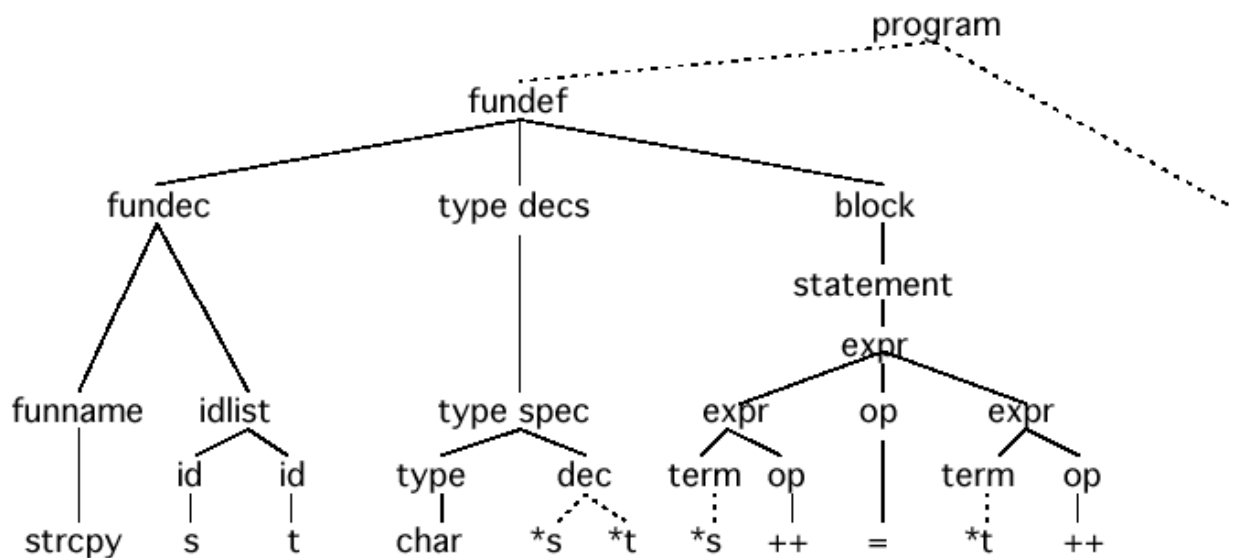
The task of the syntax analyser (called a parser) is to recover the hierarchical syntactical structure of the program which reflects its logical structure.

It generates a syntax tree which is (together with the symbol table) the central data structure for all following phases.

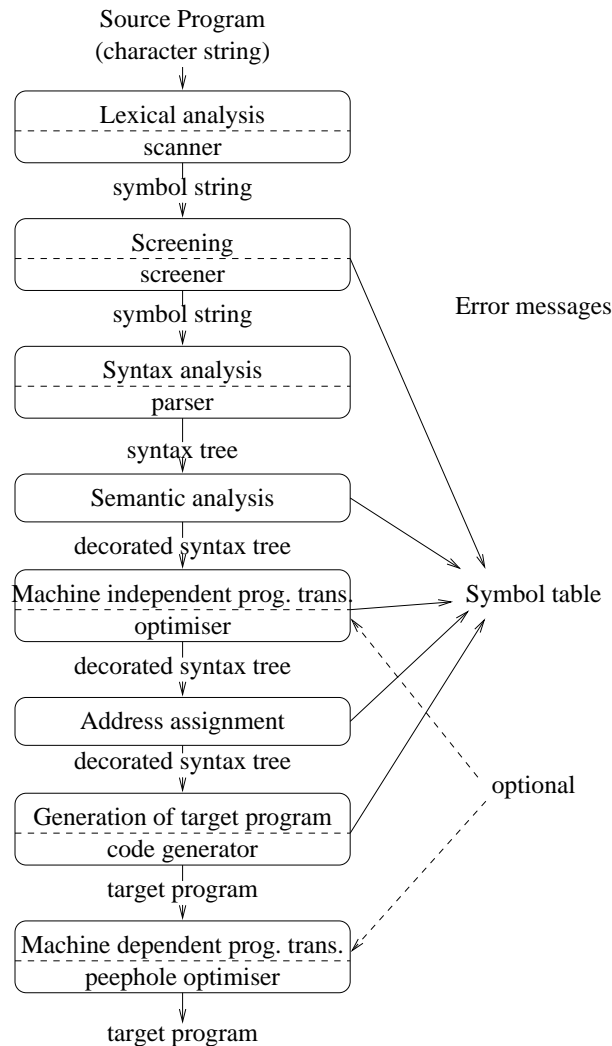
Parsing is theoretically very well understood. The theory is based on context-free grammars (type 2 grammars) and thus pushdown automata, but it makes use of several refined classes of grammars.

Parse Tree

The parse tree (syntax tree) reflects the hierarchical structure of the program.



Semantic Analysis in a Compiler



Determines some non-syntactic properties that can be determined from the program text.

- typically determines the **kind** of each identifier.
- performs **type checking** and **type inference**.
- augments the symbol table with this information.

However, it only determines some semantic properties, but not the entire semantics (meaning) of the program. This is, in a sense, done by the code generation in the form of a translation semantics.

Summary

We have looked at the reasons for formal semantic specifications and the basic ideas behind them

- programming languages require specifications of four levels: lexical, concrete syntax, abstract syntax and semantics.
- These levels of specification correspond to different phases in a compiler: scanning, parsing, semantic checking and code generation.
- The main ways to specify semantics formally are
 - operational (state rewriting rules)
 - denotational (functional)
 - axiomatic (logical)

Homework

- Revise regular expressions and finite state automata (FSA)
- Revise context-free grammars and BNF