159.341 Programming Languages, Algorithms & Concurrency

Programming Languages (Part 3)

Daniel Playne
d.p.playne@massey.ac.nz

Reminder

Programming languages have four layers.

- ***Lexical*** layer - the keywords, symbols allowed in the language.
- ***Syntax*** layer - how keywords/symbols are structured in a valid program.
- ***Context*** layer - rules about allowable programs (e.g. type checking).
- ***Semantic*** layer - addresses the meaning of the program

Context Layer

Contextual Layer

Programming languages generally have an extra layer called the ***contextual layer*** that sits in-between syntax and semantics.

Think of it as a way of breaking down the concept of semantics into a form more easily explained in a program. This level is useful as pieces of the program may mean different things in different contexts.

This level is concerned with the feasibility of instructions and indicates what actions can be performed on a given object.

Context Analysis

A programming language may forbid certain things (although they may be syntactically correct).

- 'In an expression both values must be integers' (`1 + 'a'` is illegal).
- 'If a value is an identifier it must have a value'
- 'Identifiers must be declared before use'

We would like the compiler to check these conditions (not always done and not always possible).

Context Analysis

Type-checking is a common task in context analysis that enforces constraints on what values can be used in different circumstances depending on their type.

Different programming languages have different rules about how types are treated, converted etc.

Semantic Layer

Semantic Layer

Finally the semantics represents the ***meaning*** of the program, this can be hard to define and check formally.

The ***semantic layer*** describes what the permissible actions mean.

Logic and Functional languages have less distinction between context and semantic layers. These languages are less ***instruction-oriented***.

Semantics Analysis

What do language notations ***mean***?

For example - 'what will happen when the value of an expression is calculated?'

This is the semantics of the expressions. Semantics of each notion combine to define the semantics of the whole language.

Semantics can be described formally - but much harder than context-free grammars used to describe syntax. Beyond the scope of this paper.

Language Structure

Level	Sample
lexical	`begin`
syntactic	`if .. then .. else .. end if`
contextual	variable with type, name, declaration, uses
semantic	flow of control during procedure call and return

Error Checking

One of the important tasks of a compiler (or interpreter) is to detect errors in a program (and hopefully give the user a helpful error message).

This can often be difficult as it is not easy for a compiler to know where the start of the error actually is, compilers can often continue past an error until they eventually become stuck.

For a compiler to perfectly identify all errors, it would need to know what you were trying to do.

Lexical Checking

Lexical errors are fairly uncommon as most input will match some lexical pattern.

A misspelled keyword will often by detected as an identifier token, which will then cause problems for the parser.

However, some lexical errors can occur if the input contains characters that do not exist in the language.

Syntax Checking

Syntax errors are generally easy to find (provided the language is based on a context-free grammar). This type of error is mirrored by sentences in natural languages that are incorrect.

"Him of quickly five"

Which is a completely invalid sentence.

Syntax Checking

Syntax errors can be found by our compiler (parser) by determining that a particular sentence is malformed.

For example:

```
for(i<0;) ;i++) } {
```

There are even compiler/parser tools that can generate a parser for us (from a grammar definition) - YACC, Bison, lex, flex etc.

Context Checking

Contextual errors are generally much harder to identify:

"What colour are an elephant's antlers?"

Recognising that this sentence doesn't make sense requires a large general set of knowledge about colours, elephants, antlers etc.

Context Checking

In programming there are many contextual correctness test that are hard to do properly (and some that are unsolvable). No algorithms exist (or *can* exist) that can test them properly.

Luckily some contextual checks can be performed by the compiler.

- Undeclared identifiers
- Number of parameters vs function definition
- Types of variables and identifiers
- Heuristic based checks

Type Checking

Variables have values within a domain:

- logical values (true/false)
- 32-bit integers (`-2,147, 483,648 - 2,147,483,295`)

These ***domains*** are called ***data types***.

A compiler can check that the usage of data types is correct - "only use **int16** when **int16** is expected"

Type Checking

If a type-checking system picks up ***all inconsistencies*** we can say that is applies ***strong type checking***.

Otherwise it may apply ***weak type checking*** or sometimes even none at all.

C and C++ based languages apply weak type checking, it is easy for a programmer to coerce or cast data types and circumvent type checking.

```
int i;
double d = 10;
i = (int *) & d;
```

Type Checking

This can be a powerful tool and can be very useful for optimising performance, but generally it is very unsafe and can easily lead to type-related errors.

The C and C++ compilers are unable to protect you from these kinds of errors when you use this kind of functionality.

Heuristic Context Checking

Checking based on common experience rather than strict, formal rules.

Can often be implemented with compiler switches or flags that allow you to turn them off or on.

May represent cases that are normally a mistake (but in some cases may be valid).

Heuristic Context Checking

For example: a compiler could check whether a recursive routine has a terminating condition, if not there is an error.

However, the compiler may be able to find a terminating condition that is still unreachable by the program.

```
int f(int i) {
    if(i > 25) {
        return i;
    } else {
        return f(i+1);
    }
}
```

Heuristic Context Checking

What about the following case?

```
int f(int a, int b, int c, int n) {
    if(pow(a,n) + pow(b,n) == pow(c,n)) {
        return a;
    } else if(pow(a,n) + pow(b,n) < pow(c,n)) {
        return f(a+1, b, c);
    } else {
        return f(a, b, c+1);
    }
}
```

Heuristic Context Checking

This is a recursive function that attempts to solve a simplified version of ***Fermat's Last Theorem*** which states that no three positive integers a, b and c can satisfy the equation $a^n + b^n = c^n$ for $n > 2$.

Expecting a compiler to be able to determine this recursion will never end is rather unrealistic given that it took mathematicians over 350 years to prove it.

Semantics Checking

Semantics checking is hard to define as it concerns the *meaning* of the program. For a compiler to actually check that your program is correct, it would need some way of knowing what your program is supposed to do.

However, it may be used internally within compilers for optimisation and code modification to ensure that a particular optimisation does not change the meaning of your program.

Program Processing

EBNF can be used to define the ***lexical*** and ***syntax*** layers of a language and we could agree on some rules about what contextual checking to perform.

The last step is to generate machine instructions that will convert programs written in our language into machine instructions.

The two main ways of doing this are - ***compilation*** and ***interpretation***.

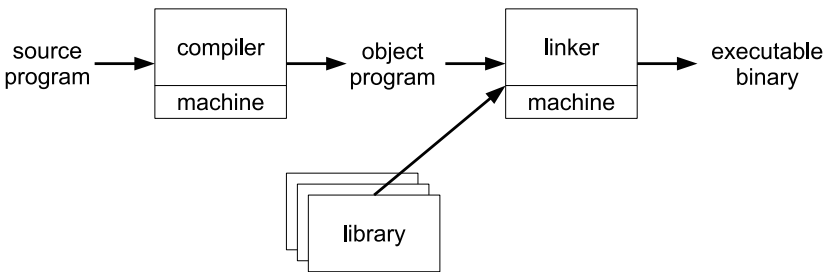
Program Processing

Compilers read the source code of the program and generate machine instructions which can be executed on your machine. This executable program will need input (and to link to libraries) at run-time.

Interpreters will read the source code of the program and execute machine instructions as soon as it has determined some operations that need to be executed. Interpreters read the source code and may link to libraries and read input from user all at the same time.

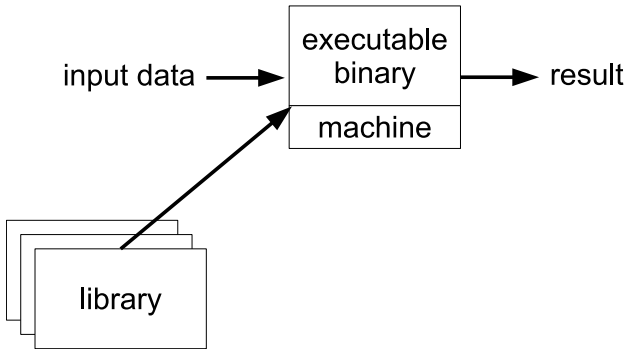
Compiling

Compilers are usually broken up into stages of compiling into object code and linking it into a final executable. Static libraries may be linked at this stage.



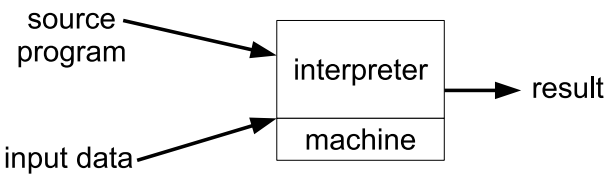
Executables

Running an executable then requires the program input and may require dynamically linked libraries.



Interpreters

Interpreters require both the source code and the input to be provided at the same time. The interpreter must also be linked to any necessary libraries.



Program Processing

In general ***compiled*** executables still run faster than interpreted code. The compiler can perform all processing of the source code and will often apply optimisations to produce the most efficient executable code possible.

Interpreted code is generally a lot slower but has the advantage of being interactive and can be more portable as the source code doesn't need to be recompiled for each different architecture.

Summary

- Programming Language Layers
  - Context Layer
  - Semantic Layer
- Error Checking
- Program Processing
  - Compiler
  - Interpreter