# Introduction

Vertalerbouw **HC1**

**VB HC1** http://fmt.cs.utwente.nl/courses/vertalerbouw/

Michael Weber
kamer: INF 5037
telefoon: 3716
email: michaelw@cs.utwente.nl

Theo Ruys
University of Twente
Department of Computer Science
Formal Methods & Tools



## Vertalerbouw 2010/2011

- Course material taken over from 2008/2009 (Theo Ruys)
- VB 2010/2011: ditto (due to BOZ regulations)
- After 2011
  - Redesign of VB likely
  - Voiding earlier passed partial course goals (OS + P)!
- Options for "herhalers":
  - Pass the course this year, or
  - Redo from scratch with new VB material later
- Recommendation:
  - VB >= 2 years ago? Come to HCs and redo P!





## Vertalerbouw - kernpunten

- ASTs staan centraal: geen one-pass compilatie
- nadruk op LL(k) compilatie (recursive descent)
- modern en populair practicumgereedschap (ANTLR)
- Java als implementatietaal
- OO-achtige aanpak van het bouwen van een vertaler
- aandacht voor moderne taalaspecten
- minder aandacht voor theorie achter scanning en parsing
- eindopdracht: bouwen van eigen vertaler





## Overview of Lecture 1

- Ch 1 – Introduction
  - 1.1 Levels of programming language
  - 1.2 Programming Language Processors
  - 1.3 Specification of Programming Languages
  - 1.4 The Triangle Programming Language
- Ch 2 – Language Processors
  - 2.1 Translators and Compilers
  - 2.2 Interpreters
  - 2.3 Real and abstract machines
  - 2.4 Interpretive compilers
  - 2.5 Portable compilers
  - 2.6 Bootstrapping
  - 2.7 Triangle language processors
- Organisatie van Vertalerbouw (in Dutch)

# Ch 1 - Introduction







# Why Compilers?

- Programming languages (Pascal, C, Java, Python,…)
  - "easy" to use and write by humans
  - "difficult" to be interpreted by computers
- Machine language (microcode/assembler)
  - "difficult" to use and write by humans
  - "easy" to execute by hardware/microprocessor
- Not just for programming languages
  - analysis/translation of natural language
  - analysis of generated data

Compilers: close interplay between theory and practice.





# Levels of Programming Languages (1)

- Programming language = notation for algorithms (which might be run on computers).
- Levels of programming language:
  - High-level (abstract)
    Java, Pascal, Miranda

```
let
   var n: Integer
in
   n := n-1
```

  - Low-level (detailed)
    assembler program

```
LOAD  r1, n
LOAD  r2, 1
SUB   r1, r2
STORE r1, n
```

  - Machine code

```
0001000100100 1101
0100001001010 0011
11100011111...
```





# Levels of Programming Languages (2)

- Aspects typically found in high-level languages, but not in low-level languages:
  - expressions
  - control structures:
    - **if-then-else**, **while**, procedures
  - data types:
    - different types of data: boolean, integer, char, float
    - composite data types: arrays
    - user defined data types: records, classes
  - declarations:
    - type checking
  - **abstraction**
  - **encapsulation**

## Language Specification

Language of the end project has to be defined like Triangle!

- syntax (or *grammar*)
  - defines the structure of correct sentences
- contextual constraints (or *static semantics*)
  - well-formedness depends on context of an expression (e.g. scope rules, type rules)
- semantics
  - defines the meaning of correct sentences (denotational or operational semantics)
- [Watt & Brown 2000]
  - formal syntax using (E)BNF
  - informal contextual constraints
  - informal semantics

See Appendix B for a "complete" specification of Triangle.





## Syntax (1)

- Syntax is specified using "Context Free Grammars" (CFG, known from "Basismodellen").
  A CFG G is defined by a 4-tuple (N, T, P, S)
  - N: a finite set of non-terminal symbols
  - T: a finite set of terminal symbols
  - P: a finite set of production rules
  - S: a start symbol
- CFGs are usually written using Backus-Naur Form (BNF) notation.
  - Two types of production rules
    - N ::= $\alpha$
    - N ::= $\alpha$ | $\beta$ | …





## Syntax (2)

- A CFG defines a set of strings, which is called the language of the CFG.
- Example:

start symbol

```
id      ::=  letter
        |    id letter
        |    id digit
letter  ::=  a | b | c | ... | z
digit   ::=  0 | 1 | 2 | ... | 9
```

non terminal

terminals

- This grammar generates "identifiers": finite strings that start with a letter and are followed by zero or more letters or digits.
  - "`a`", "`foo`", "`x123141243124124`", "`a1b2c3d4e5f6`"





## (Mini) Triangle

- Triangle is a small, but realistic, Pascal-like language with `let-in` constructs for local declarations.
- Example:

local declarations

```
let
   const MAX ~ 10;
   var n: Integer
in begin
   getint(var n);
   if (n>0) /\ (n<=MAX) then
      while n > 0 do begin
         putint(n); puteol();
         n := n-1
      end
   else
end
```

constant def ("~" instead "=")

variable may be changed by `getint`

sequence of commands can be grouped with `begin`/`end`

`else` is mandatory (but might be empty)

# (Mini) Triangle – Syntax (1)

| | | | |
|---|---|---|---|
| Program | ::= | single-Command | |
| Command | ::= | single-Command | |
| | \| | Command ; single-Command | |
| single-Command | ::= | | *skip* |
| | \| | V-name := Expression | |
| | \| | Identifier ( Expression ) | |
| | \| | **if** Expression **then** single-Command **else** single-Command | |
| | \| | **while** Expression **do** single-Command | |
| | \| | **let** Declaration **in** single-Command | |
| | \| | **begin** Command **end** | |
| Expression | ::= | primary-Expression | |
| | \| | Expression Operator primary-Expression | |
| primary-Expression | ::= | Integer-Literal | |
| | \| | V-name | |
| | \| | Operator primary-Expression | |
| | \| | ( Expression ) | |





# (Mini) Triangle – Syntax (2)

| | | |
|---|---|---|
| V-name | ::= | Identifier |
| Declaration | ::= | single-Declaration |
| | \| | Declaration ; single-Declaration |
| single-Declaration | ::= | **const** Identifier ~ Expression |
| | \| | **var** Identifier : Type-denoter |
| Type-denoter | ::= | Identifier |
| Operator | ::= | + \| - \| * \| / \| < \| > \| = \| \ |
| Identifier | ::= | Letter |
| | \| | Identifier Letter |
| | \| | Identifier Digit |
| Integer-Literal | ::= | Digit |
| | \| | Integer-Literal Digit |
| Comment | ::= | ! Graphic* **eol** |

EBNF for: zero-or-more





# Syntax Trees (1)

- A Context-Free Grammar (CFG) is a specification of a rewrite system.
  - A CFG generates a language, which is the set of all strings of terminal symbols derivable from the start symbol.
  - Such a language is defined in terms of syntax trees and phrases (sentences).
- A syntax tree of a grammar G is an ordered labeled tree:
  - the leaves are labeled by terminal symbols
  - the interior nodes are labeled by nonterminal symbols
  - each nonterminal node labeled by N has children labeled by $X_1, \ldots, X_n$, such that $N ::= X_1, \ldots, X_n$ is a production rule.
- A phrase of G is a string of terminal symbols (taken from left to right) of a syntax tree of G.

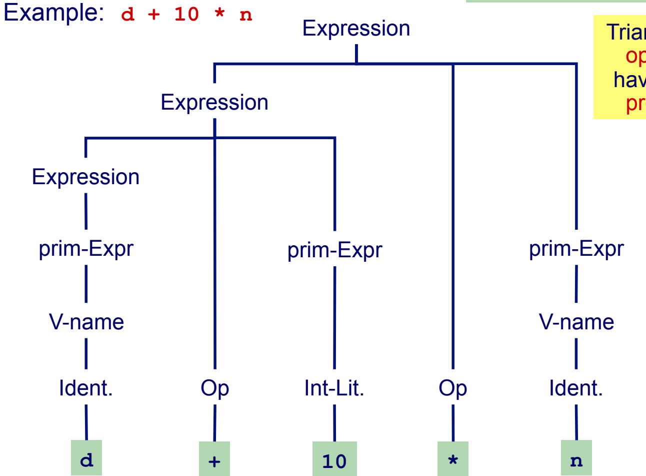




# Syntax Trees (2)

| | | |
|---|---|---|
| Expr | ::= | prim-Expr |
| | \| | Expr Op prim-Expr |
| prim-Expr | ::= | V-name |
| | \| | Int-Literal |
| | \| | ... |
| V-name | ::= | Identifier |

- Example: `d + 10 * n`




Triangle's binary operators all have the same precedence.

# Concrete vs. Abstract Syntax

- The defining grammar of a programming language specifies the concrete syntax of a language.
  - The concrete syntax is important for the programmer who needs to know exactly how to write syntactically well-formed programs.
  - But, the concrete syntax has no influence on the semantics of the programs.
- The abstract syntax omits irrelevant syntactic details and only specifies the essential structure of programs.
  - E.g. different concrete syntax for an assignment:

```
v := e
v <- e
assign e to v
set v=e
```





# (Mini) Triangle – Abstract Syntax

| | | | |
|---|---|---|---|
| Program | ::= | Command | Program |
| Command | ::= | Command ; Command | SequentialCmd |
| | \| | V-name := Expression | AssignCmd |
| | \| | Identifier ( Expression ) | CallCmd |
| | \| | if Expression then Command else Command | IfCmd |
| | \| | while Expression do Command | WhileCmd |
| | \| | let Declaration in Command | LetCmd |
| Expression | ::= | Integer-Literal | IntegerExpr |
| | \| | V-name | VnameExpr |
| | \| | Operator Expression | UnaryExpr |
| | \| | Expression Operator Expression | BinaryExpr |
| V-name | ::= | Identifier | SimpleVname |
| Declaration | ::= | Declaration ; Declaration | SeqDecl |
| | \| | const Identifier ~ Expression | ConstDecl |
| | \| | var Identifier : Type-denoter | VarDecl |
| Type-denoter | ::= | Identifier | SimpleTypeDen |

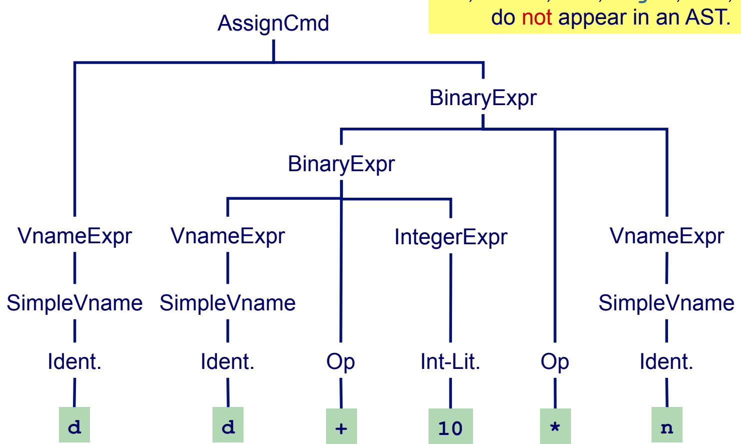




# Abstract Syntax Tree (1)

- Example: `d := d + 10 * n`

Structuring terminals (like `:=`, `if`, `while`, `let`, `begin`, `end`, etc.) do not appear in an AST.








# Abstract Syntax Tree (2)

- In an AST, each node is labelled by a production rule. Consequently, an AST represents the phrase-structure of the program explicitly.
- The AST is a convenient structure for specifying
  - contextual constraints
  - semantics
  - code generation
- ASTs will be used extensively in [Watt & Brown 2000] and in the course of Vertalerbouw.
  - ANTLR, the tool used in the laboratory, works quite elegantly with ASTs.

# Context Constraints (1)

- Apart from the syntax rules, there may be rules which specify that a phrase is well-formed which depend on the context of the phrase: context contraints.
  - scope rules
    - Scope rules are concerned with the visibility of identifiers.
    - Every identifier which is used should first be declared.
    - In other words: every applied occurrence of an identifier is related to a binding occurrence of the same identifier.
  - type rules
    - Each value has a type.
    - Each operation has a type rule, which specifies the types of the operands and the type of the result of the operation.





# Context Constraints (2)

- Example: scope rule

```
let
   const m ~ 2;
   var n: Integer
in begin
   ...
   n := m*2;
   ...
end
```

declaration of n: binding occurrence

use of n: applied occurrence

??

```
let
   var n: Integer
in begin
   ...
   n := m*2;
end
```

applied occurrence

If there is no enclosing scope (i.e. letCmd) where **m** is declared, the binding occurence of **m** is missing: scope error!





# Context Constraints (3)

- Example: type rules

```
let
   var n: Integer
in begin
   ...
   while n > 0 do
      n := n-1;
   ...
end
```

Type rule for $E_1 > E_2$ (GreaterOp):
If $E_1$ and $E_2$ are both of type int, then the result is of type bool.

Type rule for **while** E **do** C (WhileCmd):
E must be a boolean.

Type rule for V:=E (AssignCmd):
The types of V and E must be equivalent.

Type rule for $E_1 - E_2$ (SubOp):
If $E_1$ and $E_2$ are both of type int, then the result is of type int.





# Semantics (1)

- Semantics is concerned with the meaning of programs, i.e., their behaviour when executed.
- Terminology:
  - Commands are executed and perform side effects.
    - Side effects:
      - changing the values of variables
      - perform I/O
  - Declarations are elaborated to produce bindings.
  - Expressions are evaluated and yield values (and may or may not perform side effects).
- The semantics of each specific form of command, expression, declaration, etc. has to specified.

## (Mini) Triangle – Semantics (1)

- AssignCmd: V:=E
  - The expression E is evaluated to yield a value v.
  - Then v is assigned to the variable name V.
- SequentialCmd: $C_1$ ; $C_2$
  - First $C_1$ is executed.
  - Then $C_2$ is executed.
- WhileCmd: **while** E **do** C
  - The expression E is evaluated to yield a truth-value t.
  - If t is true, C is executed, and then the WhileCmd is executed again.
  - If t is false, execution of the WhileCmd is completed.





## (Mini) Triangle – Semantics (2)

- Expression:
  - IntegerExpr: Integer-Literal
    - The expression yields the value of the Integer-Literal.
  - VnameExpr: V-name
    - The expression yields the value of the variable of V-name.
  - UnaryExpr: op E
    - The expression yields the value obtained by applying unary operator op on the value yielded by the expression E.
  - BinaryExpr: $E_1$ op $E_2$
    - The expression yields the value obtained by applying binary operator op to the values yielded by the expression $E_1$ op $E_2$.

Expressions in (Mini)Triangle do not have side effects.





## Triangle

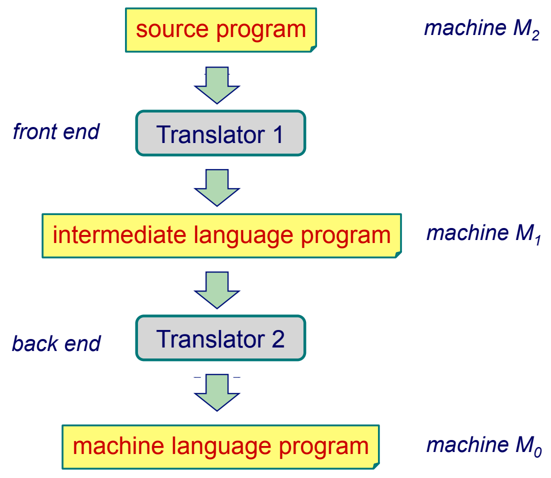
We will practice with Triangle in the laboratory session of week 1.

- Triangle is a small, but realistic, Pascal-like language with **let-in** constructs for local declarations.
  - Triangle supports (user-defined) arrays, records, procedures and functions.
  - Triangle supports value- and reference parameter passing for procedures. Furthermore, procedures and functions can be passed to procedures.
  - Triangle is type complete: no operations are arbitrarily restricted in the types of the language.
  - Triangle expressions are free of side effects.
- See Appendix B for the (informal) specification of Triangle.
- See [Gosling et. al. 1996] for the language definition of Java.

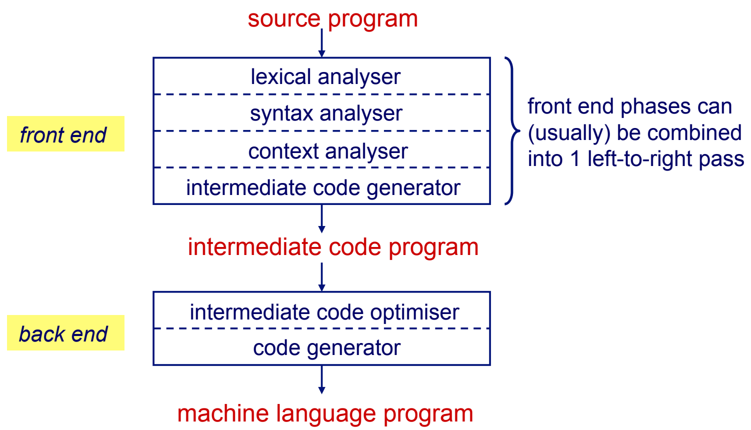




## "General" compiler scheme



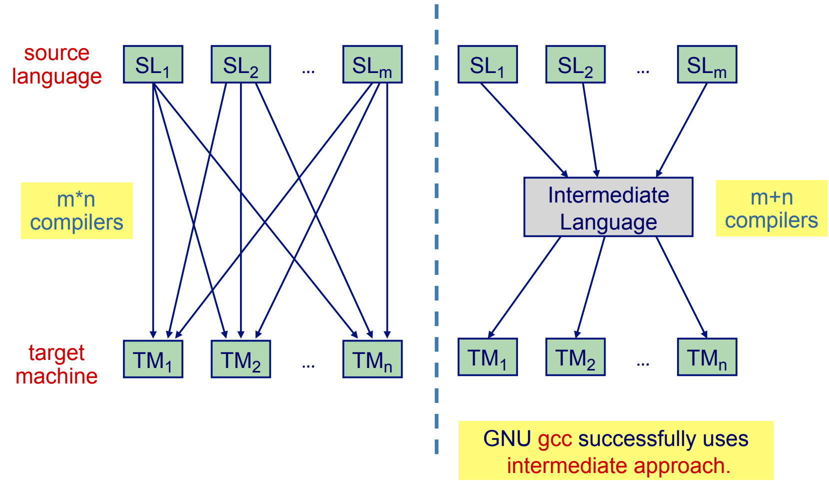
# Overview of "general" compiler








# M+N < M*N








# Ch 2 – Language Processors







# Translators (1)

- A translator accepts a text expressed in a source language and generates semantically-equivalent text expressed in a target language.
- Some translators
  - C → x86 assembly
  - x86 assembly → x86 binary code
  - Java → JVM byte code
  - Java → C
  - JVM byte code → x86 assembly
  - JVM byte code → Java (java disassembler, e.g. `jad`)
  - Dutch → English
  - Natural-language translation/processing is AI-related (see HMI courses)

# Translators (2)

- Terminology:
  - compiler: translates a high-level language into a low-level language.

    *several target instructions per source instruction*
  - assembler: translates from an assembly language into machine code.

    *one machine instruction per source instruction*
  - source program: source language (input) text.
  - object program: target language (output) text.
  - implementation language: programming language in which the translator itself is written.





# Tombstone diagrams (1)

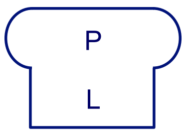
- Tombstone diagrams
  - Set of "puzzle pieces" to reason about language processors and programs.

    *A complete diagram of a translator specifies how the source, target and implementation languages and the underlying machine are related.*
  - four different kinds of pieces
  - combination rules to combine the pieces
    - not all pieces fit together

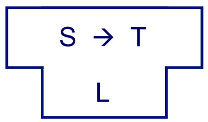




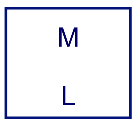
# Tombstone diagrams (2)



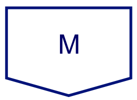

Program P expressed in language L.




Translator implemented in L, which translates programs from source language S to target language T.




Interpreter for language M, implemented in language L.



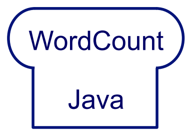

Machine M.



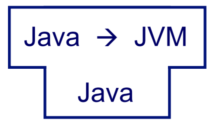


# Tombstone diagrams (3)

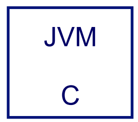
Examples:




WordCount program written in Java.




A compiler from Java to JVM byte code, written in Java.

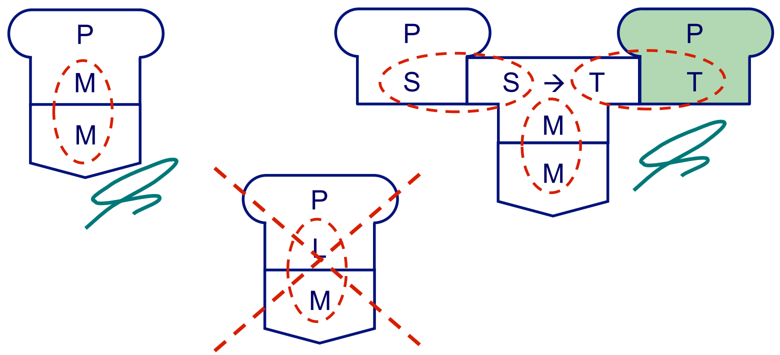



Interpreter for JVM byte code, written in C.


The x86 processor family.

# Tombstone diagrams (4)

- Combination rule – like "domino":
  - When combining pieces, the sides that touch each other should use the same implementation language.

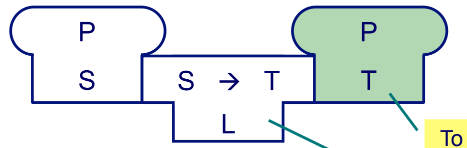







# Tombstone diagrams (5)

- Compilation of a program:
  - When compiling a program P in source language S to a target language T, a new "tombstone piece" is obtained: a program P in language T.




This will only work, if we have a machine on which we can (perhaps indirectly) run programs written in L.

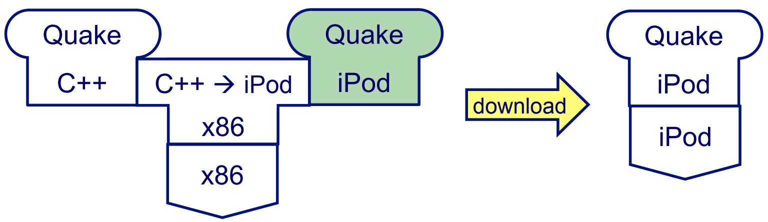
To specify that the program is generated, we use a coloured background for these pieces.





# Cross-compilation

- A cross-compiler runs on one machine (host machine) but generates code for a dissimilar machine (target machine).
  - Useful if the target machine
    - does not have enough memory to compile programs.
    - does not have tools to develop programs.
  - Examples: programs for PDAs, telephones, media players

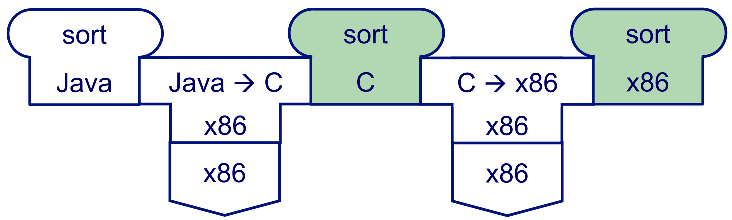







# Two-stage compilation

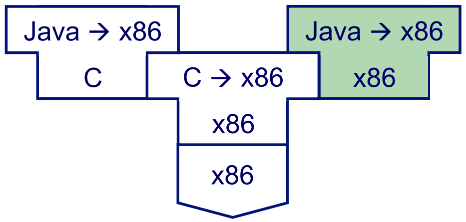
- A two-stage translator is a composition of two translators.
  - With compilers S→T and a T→U, the source program in S is translated to target language U via T.
  - Easily generalized to an n-stage translator.




Compilation of a high-level programming language is usually an n-stage translation, as at least one intermediate language is used for the compilation to executable code.

## Compiling a compiler

- A translator is itself a program, expressed in some language. As such, it can be translated into another language.
  - A compiled translator will usually be faster than its original.








## Interpreters (1)

- An interpreter is a program that accepts a program in a source language and runs that source program immediately.
- Using an interpreter is sensible when
  - the programmer is working in interactive mode, and wishes to see results of each instruction before entering the next instruction, or
  - the program is to be used once and then discarded, or
  - each instruction is expected to be executed only once, or
  - the instructions of the source language have simple formats, and thus can be analyzed easily and efficiently.

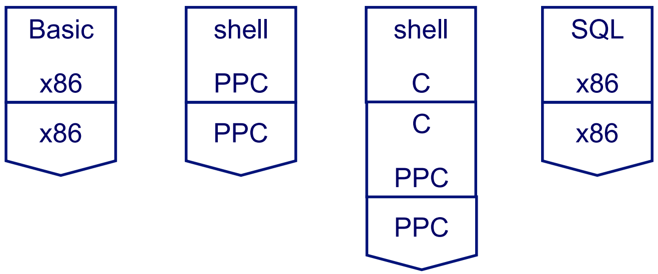
Interpretation is slow. Interpretation can be more than 100 times slower than running an equivalent (but compiled) machine-code program.





## Interpreters (2)

- Examples:
  - Basic interpreter
  - UNIX command language interpreter (shell)
  - SQL interpreter

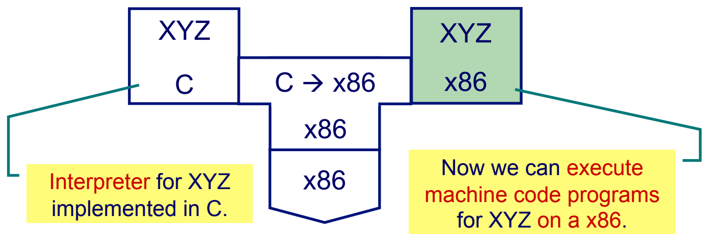




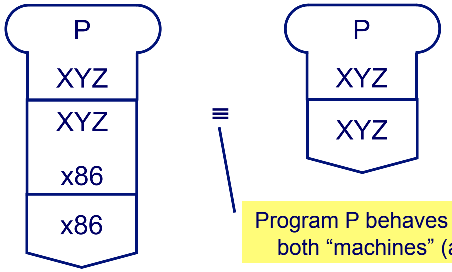



## Hardware emulation

- In the process of designing the architecture and instruction set for a new machine XYZ, an interpreter for machine XYZ is usually implemented.
  - Not only for new machines. E.g., if an "old" machine is not longer available and one needs to run a program for which only the machine code of the "old" machine is available.



## Real/Abstract Machines




Program P behaves exactly the same on both "machines" (apart from speed).

- An interpreter is often called an abstract machine as opposed to its hardware counterpart, which is a real machine.
  - A well-known example of an abstract machine is the Java Virtual Machine (JVM).





## Interpretive Compilers

- Tradeoffs in "executing a program":
  - compiler: long time to compile, but fast execution
  - interpreter: starts running immediately, but will be slow
- An interpretive compiler is a combination of a compiler and an interpreter. The key idea is to translate the source program into an intermediate language (IL).
  - the IL is in level between the source language and the ordinary machine code.
  - the instructions of the IL have simple formats and can therefore be analyzed easily and quickly
  - translation from the source language to IL is easy and fast.

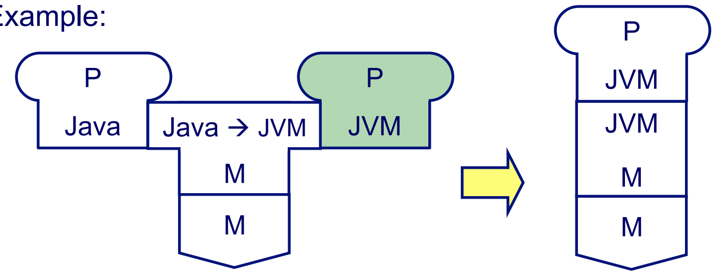




## Java Development Kit (JDK)

- Sun's JDK provides an implementation of an interpretive compiler for Java. Central to the JDK is the JVM.


- Example:





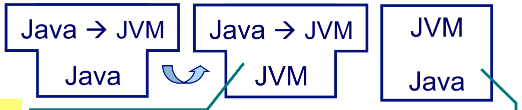



## Portable Compilers

- A program is portable to the extent that it can be (compiled and) run on any machine, without change.
  - Portability is measured in the proportion of code that remains unchanged when moving to a different machine.
  - Application programs in high-level languages should achieve a 95-99% portability.
  - In general, the portability for language processors is much lower, though (about 50%), because a compiler's function is to generate machine code for a particular machine.
    - Unless you are able to parameterize the language processor in (a description of) the machine.
    - A compiler that generates intermediate code is potentially much more portable, though.

## Portable Compiler Kit for Java (1)

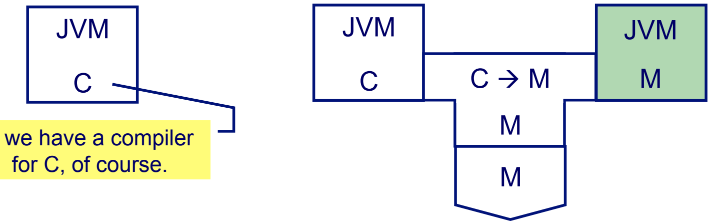
- Portable JDK:




We cannot compile Java programs until we have an implementation of the JVM.

We cannot use the JVM interpreter until we can compile Java programs.

- Solution: rewrite the interpreter for JVM for the target machine.



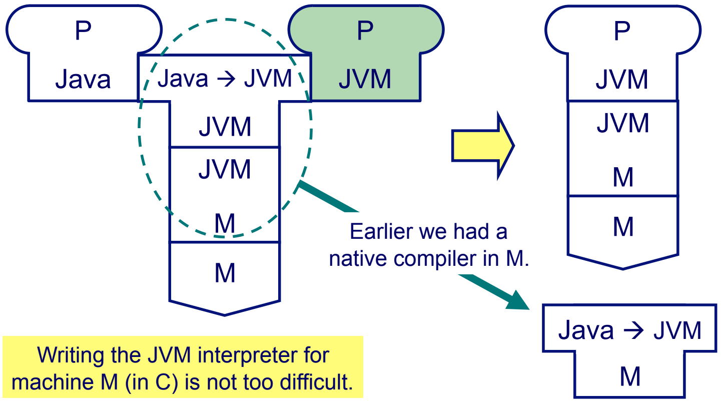

If we have a compiler for C, of course.





## Portable Compiler Kit for Java (2)

- Now we are able to compile Java programs using this JVM interpreter:




Earlier we had a native compiler in M.

Writing the JVM interpreter for machine M (in C) is not too difficult.





## Vertalerbouw 2010/2011

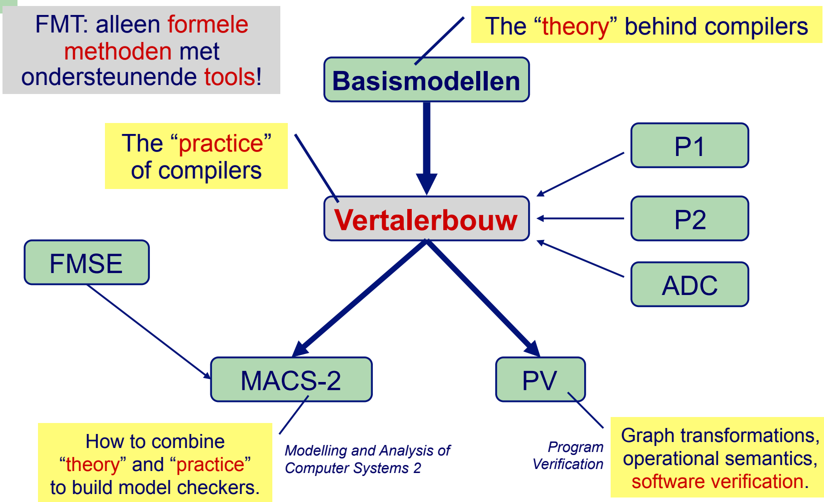
- Plaats van VB in INF/CS curriculum
- Organisatie
  - hoorcolleges
  - practica
  - beoordeling
- VB 2010/2011 kw4 - rooster

VB 2010/2011 is ten opzichte van vorig jaar inhoudelijk nauwelijks veranderd.





## VB binnen INF/CS Curriculum (2)



## Organisatie (1)

Zie website:
http://fmt.cs.utwente.nl/courses/vertalerbouw/

met name: "Inleiding" uit practicumhandleiding (vb-intro.pdf).

- Hoorcolleges: 9x (ma 3+4 en di 6+7)
  - 7x theorie uit [Watt & Brown 2000]
  - 2x ANTLR (tool voor practicum)
- Practicum: 3 groepen
  - clusters: Zi 4054 A, B en C
  - indeling bij eerste practicum op woensdag 27 april 2011
  - verplicht: aanwezigheid wordt gecontroleerd
  - iedere groep heeft eigen studentassistent
- Onderwijsmateriaal:
  - [Watt & Brown 2000]
  - practicumhandleiding (via website)

[Watt & Brown 2000] en Triangle compiler bevatten fouten. Zie de website voor errata en bugfixes.





## Organisatie (2)

- Beoordeling
  - twee opgavenseries (individueel) OS
  - eindopdracht (in tweetallen) P
  - eindcijfer = (OS + P) / 2
    *mits OS en P beiden >= 5.0, anders 4*

Cijfers voor OS en P blijven alleen dit jaar nog staan. Aparte delen van OS echter niet.

- Opgavenseries
  - eerste serie komt (uiterlijk) ma 02 mei 2011 beschikbaar
  - stricte deadlines
    *overschrijden van de deadline kost punten.*
  - worden nagekeken door eigen studentassistent
  - strict individueel: fraude levert 1.0 voor het vak en uitsluiting voor dit studiejaar





## Organisatie (3)

- Practicum deel 1 - introductie
  - 5 weken
  - oefenen met stof van hoorcollege
  - oefenen met ANTLR 3
  - succesvolle afronding noodzakelijk voor deel 2
  - advies: voorbereiden van de practica (met name wk 1 en 2)
- Practicum deel 2 - zelf een vertaler bouwen
  - 3 weken (+ 2 tentamenweken + 1 uitloopweek)
  - gebruikmaken van ANTLR
  - verschillende moeilijksheidsgraad
    - (maximum) cijfer hangt af van gekozen opdracht
  - deadline: woensdag 6 juli 2011 (= woensdag na tentamens)





## Tijdsbesteding

5 ECTS = 140 uur

| | | |
|---|---|---|
| hoorcolleges: contacturen | 9 x 2 = | 18 |
| zelfstudie naast hoorcolleges | 7 x 3 = | 21 |
| practicum deel 1: contacturen | 5 x 4 = | 20 |
| voorbereidingen practicum blok 1 | 5 x 2 = | 10 |
| opgavenseries | 2 x 8 = | 16 |
| practicum deel 2: contacturen | 3 x 6 = | 18 |
| eindopdracht (buiten practicumuren) | 5 x 5 = | 25 |
| verslag | | 12 |
| TOTAAL | | **140** |

# VB 2010/2011 kw4 - rooster

| | | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
| ma | 1+2 | | | | | S2 | | | | | | |
| | 3+4 | | H2 | H4 | H6 | H8 | H9 | HX | | | | |
| | 6+7 | | | | | | | | | | | |
| | 8+9 | | S1 | | S1 | | | S2 | | | | |
| di | 1+2 | | | | | | | | | | | |
| | 3+4 | H1 | H3 | H5 | H7 | xtra | xtra | xtra | xtra | | | |
| | 6+7 | | | | | | | | | | | |
| | 8+9 | | | | | | | | | | | |
| wo | 1+2 | | | | | | | | | | | |
| | 3+4 | | | | | | | | | | | |
| | 6+7 | P1 | P2 | P3 | P4 | P5 | P6 | P7 | P8 | | | |
| | 8+9 | | | | | | | | | | | Eind |





# Hoorcolleges

| | | | |
|---|---|---|---|
| 1 | di | 26 april | Ch. 1 – Introduction & Ch. 2 – Language Processors |
| 2 | ma | 02 mei | Ch. 3 – Compilation & Ch. 4 – Syntactic Analysis |
| 3 | di | 03 mei | Ch. 5 – Contextual Analysis |
| 4 | ma | 09 mei | ANTLR 1 – Introduction |
| 5 | di | 10 mei | Ch. 6 – Run-Time Organization |
| 6 | ma | 16 mei | Ch. 7 – Code Generation |
| 7 | di | 17 mei | Code Optimization |
| 8 | ma | 23 mei | Garbage Collection |
| 9 | ma | 30 mei | ANTLR 2 - ASTs & Error Handling |
| X | ma | 06 juni | Invited Talk (TBA) |