Languages and Compilers
(SProg og Oversættere)

Bent Thomsen
Department of Computer Science
Aalborg University
Lecturer

• Bent Thomsen
  – Associate Professor
    (Database and Programming Technology Research Group)
• Research interests:
  – Programming Language design and implementation
  – Formal foundations
  – Mobile and global systems
  – Distributed systems
  – Concurrency theory
Assistants

• Laurynas Speicys
  – Amanuensis
    (Database & Programming Technology Group)
• Francois Deliege
  – PhD Student
    (Database & Programming Technology Group)
What is the Most Important
Open Problem in Computing?

Increasing Programmer Productivity

- Write programs quickly
- Write programs easily
- Write programs correctly

• Why?
  - Decreases development cost
  - Decreases time to market
  - Decreases support cost
  - Increases satisfaction
Why Programming Languages?

3 ways of increasing programmer productivity:
1. Process (software engineering)
   - Controlling programmers
2. Tools (verification, static analysis, program generation)
   - Important, but generally of narrow applicability
3. Language design --- the center of the universe!
   - Core abstractions, mechanisms, services, guarantees
   - Affect how programmers approach a task (C vs. SML)
   - Multi-paradigm integration
Well …

"Some believe that we lacked the programming language to describe your perfect world"

Agent Smith - The Matrix
SAN FRANCISCO -- Microsoft's Bill Gates cast his company's .Net initiative wide Wednesday, releasing the final version of the long-anticipated developer toolkit, Visual Studio .Net, as well as the underpinnings of its emerging Web-based development platform, called the .Net Framework.

"When we started out we said this could be one of the biggest pieces of work we have to do on a tool," Gates said of Microsoft's efforts to remodel its development tools already used by millions of Visual Basic and C++ developers to add new support for building Web-based applications.

Straying from its typical two-year release cycle, the latest incarnation of Microsoft's application development environment has been in the making for more than three years. New features will allow developers to write applications using more than 20 different programming languages that can run on computers ranging from cell phones to servers and interact with applications written for virtually any computing platform, according to Microsoft.
Sun invites IBM, Cray to collaborate on high-end computer language

By Rick Merritt, EE Times
December 16, 2003 (8:14 p.m. EST)
URL: http://www.eetimes.com/story/OEG20031216S0031

MOUNTAIN VIEW, Calif. — Sun Microsystems is inviting competitors IBM Corp. and Cray Inc. to collaborate on defining a new computer language it claims could bolster performance and productivity for scientific and technical computing. The effort is part of a government-sponsored program under which the three companies are competing to design a petascale-class computer by 2010.
Some new developments in programming languages in 2005

- Java 1.5 (sometimes called J2SE 5.0)
- Ruby
- C# 2.0 and .Net 2.0
- Aspect Oriented Programming
  - AspectJ, Aspect.Net
- Business Process Management
  - BPEL-J, PLEW4WS
- Java 1.6 and C# 3.0 are on the way
What is this course about?

• Programming Language Design
  – Concepts and Paradigms
  – Ideas and philosophy
  – Syntax and Semantics

• Compiler Construction
  – Tools and Techniques
  – Implementations
  – The nuts and bolts
Curricula
(Studie ordning)

The purpose of the course is for the student to gain knowledge of important principles in programming languages and for the student to gain an understanding of techniques for describing and compiling programming languages.
What should you expect to get out of this course

Ideas, principles and techniques to help you
– Design your own programming language or design your own extensions to an existing language
– Tools and techniques to implement a compiler or an interpreter
– Lots of knowledge about programming
Something for everybody

• Design
  – Trade offs
  – Technically feasible
  – Personal taste

• User experience and feedback

• Lots of programming at different levels

• Clever algorithms

• Formal specification and proofs

• History
  – Compiler construction is the oldest CS discipline
Format

• 15 sessions of 4 hours
• Each Lecture will have 3 sessions of 30 min
• 2 hours for exercises
  – Exercises from the previous lecture!
  – Individual exercises
    • Train specific techniques and methods
  – Group exercises
    • Help you discuss concepts, ideas, problems and solutions
• Home reading Literature
Literature


• Some web references
Format (cont.)

- Lectures
  - Give overview and introduce concepts
  - Will not necessarily follow the books!
- Literature
  - In-depth knowledge
  - A lot to read (two books and some web references)
  - Browse before lecture
  - Read after lecture, but before exercises
- Exercises
  - Do the exercises – they all serve a purpose
  - Help you discuss ideas, concepts, designs, … (groups)
  - Train techniques and tools (sub-groups or individually)
- Project
  - Put it all together

SW4 semester evaluering f05:

Udbyttet kunne have været bedre, og havde vi lavet flere opgaver, ville det have været lettere at lave vores projekt. Generelt lærte vi indholdet af kurset via vores projekt
What is expected of you at the end?

• One goal for this course is for you to be able to explain concepts, techniques, tools and theories to others
  – Your future colleagues, customers and boss
  – (especially me and the examiner at the exam ;-))

• That implies you have to
  – Understand the concepts and theories
  – Know how to use the tools and techniques
  – Be able to put it all together

• I.e. You have to know and know that you know
What you need to know beyond this course

• Know about programming
• Know about machine architectures
• Know about operating systems
• Know about formal syntax and semantics
• So pay attention in those course!
Before we get started

• Tell me if you don’t understand
• Tell me if I am too fast or too slow
• Tell me if you are unhappy with the course
• Tell me before or after the lecture, during exercises, in my office, in the corridors, in the coffee room, by email, …
• Don’t tell me through the semester group minutes
Programming Languages and Compilers are at the core of Computing

All software is written in a programming language.

Learning about compilers will teach you a lot about the programming languages you already know.

Compilers are big – therefore you need to apply all you knowledge of software engineering.

The compiler is the program from which all other programs arise.
What is a Programming Languages

• A programming language is a set of rules that provides a way of telling a computer what operations to perform.
• A programming language is a set of rules for communicating an algorithm
• A programming language provides a linguistic framework for describing computations
What is a Programming Language

- English is a natural language. It has words, symbols and grammatical rules.
- A programming language also has words, symbols and rules of grammar.
- The grammatical rules are called syntax.
- Each programming language has a different set of syntax rules.
Why Are There So Many Programming Languages

- Why does some people speak French?
- Programming languages have evolved over time as better ways have been developed to design them.
  - First programming languages were developed in the 1950s
  - Since then thousands of languages have been developed
- Different programming languages are designed for different types of programs.
# Levels of Programming Languages

<table>
<thead>
<tr>
<th>High-level program</th>
<th>Low-level program</th>
<th>Executable Machine code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class Triangle {</code></td>
<td><code>LOAD r1, b</code></td>
<td>0001001001001000101</td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>LOAD r2, h</code></td>
<td>001001001110110010101101001...</td>
</tr>
<tr>
<td><code>float surface()</code></td>
<td><code>MUL r1, r2</code></td>
<td></td>
</tr>
<tr>
<td><code>return b*h/2;</code></td>
<td><code>DIV r1, #2</code></td>
<td></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>RET</code></td>
<td></td>
</tr>
</tbody>
</table>
What Are the Types of Programming Languages

• First Generation Languages
  Machine
  0000 0001 0110 1110
  0100 0000 0001 0010

• Second Generation Languages
  Assembly
  LOAD x
  ADD R1 R2

• Third Generation Languages
  High-level imperative/object oriented
  public Token scan ( ) {
    while (currentchar == ' ' | currentchar == '
')
    {....}

• Fourth Generation Languages
  Database
  select fname, lname
  from employee
  where department='Sales'
  SQL

• Fifth Generation Languages
  Functional Logic
  fact n = if n==0 then 1 else n*(fact n-1)
  uncle(X,Y) :- parent(Z,Y), brother(X,Z).
  Lisp, SML, Haskel, Prolog
Beyond Fifth Generation Languages

• Some talk about
  – Agent Oriented Programming
  – Aspect Oriented Programming
  – Intentional Programming
  – Natural language programming

• Maybe you will invent the next big language
The principal paradigms

• Imperative Programming
  – Fortran, Pascal, C
• Object-Oriented Programming
  – Simula, SmallTalk, C++, Java, C#
• Logic/Declarative Programming
  – Prolog
• Functional/Applicative Programming
  – Lisp, Scheme, Haskell, SML, F#
• (Aspect Oriented Programming)
  – AspectJ, AspectC#, Aspect.Net
Programming Language Genealogy

diagram.pdf  Lang_History.htm  Diagram by Peter Sestoft
A language is a language is a language

• Programming languages are languages
• When it comes to mechanics of the task, learning to speak and use a programming language is in many ways like learning to speak a human language
• In both kind of languages you have to learn new vocabulary, syntax and semantics (new words, sentence structure and meaning)
• And both kind of language require considerable practice to make perfect.
But there is a difference!

- Computer languages lack ambiguity and vagueness
- In English sentences such as
  - I saw the man with a telescope
    - Who had the telescope?
  - Take a pinch of salt
    - How much is a pinch?
- In a programming language a sentence either means one thing or it means nothing
What determines a “good” language

• Formerly: Run-time performance  
  – (Computers were more expensive than programmers)
• Now: Life cycle (human) cost is more important  
  – Ease of designing, coding  
  – Debugging  
  – Maintenance  
  – Reusability  
• FADS
Criteria in a good language design

• **Writability**: The quality of a language that enables a programmer to use it to express a computation clearly, correctly, concisely, and quickly.

• **Readability**: The quality of a language that enables a programmer to understand and comprehend the nature of a computation easily and accurately.

• **Orthogonality**: The quality of a language that features provided have as few restrictions as possible and be combinable in any meaningful way.

• **Reliability**: The quality of a language that assures a program will not behave in unexpected or disastrous ways during execution.

• **Maintainability**: The quality of a language that eases errors can be found and corrected and new features added.
Criteria (Continued)

- **Generality**: The quality of a language that avoids special cases in the availability or use of constructs and by combining closely related constructs into a single more general one.

- **Uniformity**: The quality of a language that similar features should look similar and behave similar.

- **Extensibility**: The quality of a language that provides some general mechanism for the user to add new constructs to a language.

- **Standardability**: The quality of a language that allows programs written to be transported from one computer to another without significant change in language structure.

- **Implementability**: The quality of a language that provides a translator or interpreter can be written. This can address to complexity of the language definition.
Different Programming language
Design Philosophies

If all you have is a hammer, then everything looks like a nail.
Programming Language Specification

• Why?
  – A communication device between people who need to have a common understanding of the PL:
    • language designer, language implementor, language user

• What to specify?
  – Specify what is a ‘well formed’ program
    • syntax
    • contextual constraints (also called static semantics):
      – scoping rules
      – type rules
  – Specify what is the meaning of (well formed) programs
    • semantics (also called runtime semantics)
Programming Language Specification

- Why?
- What to specify?
- How to specify?
  - Formal specification: use some kind of precisely defined formalism
  - Informal specification: description in English.
  - Usually a mix of both (e.g. Java specification)
    - Syntax => formal specification using CFG
    - Contextual constraints and semantics => informal
    - Formal semantics has been retrofitted though
Programming Language specification

A Language specification has (at least) three parts:

- Syntax of the language: usually formal: EBNF
- Contextual constraints:
  - scope rules (often written in English, but can be formal)
  - type rules (formal or informal)
- Semantics:
  - defined by the implementation
  - informal descriptions in English
  - formal using operational or denotational semantics

The Syntax and Semantics course will teach you how to read and write a formal language specification – so pay attention!
Important!

- Syntax is the visible part of a programming language
  - Programming Language designers can waste a lot of time discussing unimportant details of syntax
- The language paradigm is the next most visible part
  - The choice of paradigm, and therefore language, depends on how humans best think about the problem
  - There are no right models of computations – just different models of computations, some more suited for certain classes of problems than others
- The most invisible part is the language semantics
  - Clear semantics usually leads to simple and efficient implementations
Syntax Specification

Syntax is specified using “Context Free Grammars”:

- A finite set of **terminal symbols**
- A finite set of **non-terminal symbols**
- A **start symbol**
- A finite set of **production rules**

Usually CFG are written in “Bachus Naur Form” or BNF notation.

A production rule in BNF notation is written as:

\[ N ::= \alpha \]

where \( N \) is a non terminal

and \( \alpha \) a sequence of terminals and non-terminals

\[ N ::= \alpha | \beta | ... \]

is an abbreviation for several rules with \( N \) as left-hand side.
Syntax Specification

A CFG defines a set of strings. This is called the language of the CFG.

Example:

Start ::= Letter
      | Start Letter
      | Start Digit
Letter ::= a | b | c | d | ... | z
Digit ::= 0 | 1 | 2 | ... | 9

Q: What is the “language” defined by this grammar?
**Example: Syntax of “Mini Triangle”**

Mini triangle is a very simple Pascal-like programming language.

An example program:

```plaintext
!This is a comment.
let const m ~ 7;
  var n
in
  begin
    n := 2 * m * m  ;putint(n)
  end
```

- **Declarations**
- **Expression**
- **Command**
Example: Syntax of “Mini Triangle”

Program ::= single-Command
single-Command
  ::= 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
Command ::= single-Command
  | Command ; single-Command
...
Example: Syntax of “Mini Triangle” (continued)

Expression
   ::= primary-Expression
     | Expression Operator primary-Expression
primary-Expression
   ::= Integer-Literal
     | V-name
     | Operator primary-Expression
     | ( Expression )
V-name ::= Identifier
Identifier ::= Letter
   | Identifier Letter
   | Identifier Digit
Integer-Literal ::= Digit
   | Integer-Literal Digit
Operator ::= + | - | * | / | < | > | =
Example: Syntax of “Mini Triangle” (continued)

Declaration
  ::= single-Declaration
     | Declaration ; single-Declaration
single-Declaration
  ::= const Identifier ~ Expression
     | var Identifier : Type-denoter
Type-denoter ::= Identifier

Comment ::= ! CommentLine eol
CommentLine ::= Graphic CommentLine
Graphic ::= any printable character or space
Syntax Trees

A syntax tree is an ordered labeled tree such that:

a) terminal nodes (leaf nodes) are labeled by terminal symbols

b) non-terminal nodes (internal nodes) are labeled by non terminal symbols.

c) each non-terminal node labeled by \( N \) has children \( X_1, X_2, \ldots X_n \) (in this order) such that \( N := X_1, X_2, \ldots X_n \) is a production.
Syntax Trees

Example:

Expression ::= Expression Op primary-Exp

```
Expression ::= Expression Op primary-Exp
  Expression ::= Expression Op primary-Exp
    primary-Exp ::= V-name
      V-name ::= Ident
        Ident ::= Ident
          Ident ::= Ident
            d
```
Contextual Constraints

Syntax rules alone are not enough to specify the format of well-formed programs.

**Example 1:**
```plaintext
let const m~2
in m + x
```
**Undefined!**

**Example 2:**
```plaintext
let const m~2;
var n:Boolean
in begin
  n := m<4;
  n := n+1
end
```
**Type error!**
Scope Rules

Scope rules regulate visibility of identifiers. They relate every **applied occurrence** of an identifier to a **binding occurrence**

**Example 1**
```plaintext
let const m~2;
  var r:Integer
in
  r := 10*m
```

**Example 2:**
```plaintext
let const m~2
in
  m + x
```

**Terminology:**

*Static binding vs. dynamic binding*
Type Rules

Type rules regulate the expected types of arguments and types of returned values for the operations of a language.

Examples

Type rule of $<$:
- $E_1 < E_2$ is type correct and of type $\text{Boolean}$
- if $E_1$ and $E_2$ are type correct and of type $\text{Integer}$

Type rule of $\text{while}$:
- $\text{while } E \text{ do } C$ is type correct
- if $E$ of type $\text{Boolean}$ and $C$ type correct

Terminology:

*Static typing vs. dynamic typing*
Semantics

Specification of semantics is concerned with specifying the “meaning” of well-formed programs.

Terminology:

Expressions are evaluated and yield values (and may or may not perform side effects)

Commands are executed and perform side effects.

Declarations are elaborated to produce bindings

Side effects:
- change the values of variables
- perform input/output
Semantics

Example: The (informally specified) semantics of commands in mini Triangle.

Commands are executed to update variables and/or perform input output.

The assignment command $V := E$ is executed as follows:

1. first the expression $E$ is evaluated to yield a value $v$
2. then $v$ is assigned to the variable named $V$

The sequential command $C_1; C_2$ is executed as follows:

1. first the command $C_1$ is executed
2. then the command $C_2$ is executed

etc.
**Semantics**

**Example:** The semantics of expressions.

*An expression is evaluated to yield a value.*

An (integer literal expression) $IL$ yields the integer value of $IL$

The (variable or constant name) expression $V$ yields the value of the variable or constant named $V$

The (binary operation) expression $E_1 \ O \ E_2$ yields the value obtained by applying the binary operation $O$ to the values yielded by (the evaluation of) expressions $E_1$ and $E_2$

etc.
Semantics

Example: The semantics of declarations.

A declaration is elaborated to produce bindings. It may also have the side effect of allocating (memory for) variables.

The constant declaration `const I~E` is elaborated by binding the identifier value `I` to the value yielded by `E`.

The constant declaration `var I:T` is elaborated by binding `I` to a newly allocated variable, whose initial value is undefined. The variable will be deallocated on exit from the let containing the declaration.

The sequential declaration `D1;D2` is elaborated by elaborating `D1` followed by `D2` combining the bindings produced by both. `D2` is elaborated in the environment of the sequential declaration overlaid by the bindings produced by `D1`.
Structured operational semantics

\[ \text{[ass]} \quad \langle x := a, s \rangle \rightarrow s[x \mapsto a]s \]

\[ \text{[skip]} \quad \langle \text{skip}, s \rangle \rightarrow s \]

\[ \text{[comp]} \quad \frac{\langle S_1, s \rangle \rightarrow s', \langle S_2, s' \rangle \rightarrow s''}{\langle S_1; S_2, s \rangle \rightarrow s''} \]

\[ \text{[if]} \quad \frac{\langle S_1, s \rangle \rightarrow s', \langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s' \text{ if } B[h]s = \text{tt} \}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'} \]

\[ \frac{\langle S_2, s \rangle \rightarrow s' \text{ if } B[h]s = \text{ff} \}{\langle \text{if } b \text{ then } S_1 \text{ else } S_2, s \rangle \rightarrow s'} \]

\[ \text{[while]} \quad \frac{\langle S, s \rangle \rightarrow s', \langle \text{while } b \text{ do } S, s' \rangle \rightarrow s'' \text{ if } B[h]s = \text{tt} \}{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s''} \]

\[ \frac{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s \text{ if } B[h]s = \text{ff} \}{\langle \text{while } b \text{ do } S, s \rangle \rightarrow s} \]
Language Processors: Why do we need them?

Programmer

Compute surface area of a triangle?

How to bridge the “semantic gap”?

0101001001...

Hardware

Programmer

Concepts and Ideas

Java Program

JVM Assembly code

JVM Binary code

JVM Interpreter

X86 Processor

Hardware
Language Processors: What are they?

A programming language processor is any system (software or hardware) that manipulates programs.

Examples:

– Editors
  • Emacs
– Integrated Development Environments
  • Borland jBuilder
  • Eclipse
  • Visual Studio .Net
– Translators (e.g. compiler, assembler, disassembler)
– Interpreters
Interpreter

1. Source Code
2. Pure Interpreter
3. Execution Results
You use lots of interpreters everyday!

Several languages are used to add dynamics and animation to HTML. Many programming languages are executed (possibly simultaneously) in the browser!
And also across the web

Web-Client

Web-Browser

HTML-Form (+JavaScript)

Submit Data

Response

Reply

Web-Server

WWW

PHP Script

Call PHP interpreter

Response

LAN

Data

Call PHP interpreter

Response

LAN

Database Server

DBMS

SQL commands

Database Output
Compilation

- **Compilation** is at least two-step process, in which the original program (source program) is input to the compiler, and a new program (target program) is output from the compiler. The compilation steps can be visualized as the following.
Compiler (simple view)
Hybrid compiler / interpreter

Diagram:

- Source code
- Lexical analyzer
- Syntax analyzer
- Semantic analyzer
- Code generator
- Intermediate language code
- Interpreter
- External libraries
- Execution results
The “Phases” of a Compiler

Source Program

Syntax Analysis  →  Error Reports

Abstract Syntax Tree

Contextual Analysis  →  Error Reports

Decorated Abstract Syntax Tree

Code Generation

Object Code
Multi Pass Compiler

A multi pass compiler makes several passes over the program. The output of a preceding phase is stored in a data structure and used by subsequent phases.

Dependency diagram of a typical Multi Pass Compiler:
Different Phases of a Compiler

The different phases can be seen as different transformation steps to transform source code into object code.

The different phases correspond roughly to the different parts of the language specification:

- Syntax analysis <-> Syntax
- Contextual analysis <-> Contextual constraints
- Code generation <-> Semantics
Tools and Techniques

• Front-end: Syntax analysis
  – How to build a Scanner and Lexer
    • By hand in Java
    • Using Tools
      – JavaCC
      – SableCC
      – Lex and Yacc (JLex and JavaCUP)
      – (lg and pg – compiler tools for .Net)

• Middle-part: Contextual Analysis

• Back-end: Code Generation
  – Target Machines
    • TAM
    • JVM
    • .Net CLR
Q: Which programming languages play a role in this picture?

A: All of them!
Important

• At the end of the course you should …
• know
  – Which techniques exists
  – Which tools exists
• Be able to choose “the right ones”
  – Objective criteria
  – Subjective criteria
• Be able to argue and justify your choices!
How does the course fit with my project?

That is a good question!

For SW4 SPO is a PE course
For DAT2 and F6S there is a choice
SPO as PE course for SW4

3.2 Projektenheden på 4. semester, SW4

_Tema: Sprogteknologi / Language Technology_

_Målbeskrivelse:_ Efter projektenheden skal den studerende kunne anvende væsentlige principper i programmeringssprog og teknikker til beskrivelse og oversættelse af sprog generelt.

_Indhold:_ Projektet består i en analyse af en softwareteknisk problemstilling, hvis løsning kan beskrives i form af et design af væsentlige begreber for et konkret programmeringssprog. I tilknytning hertil skal konstrueres en oversætter/fortolker for sproget, som viser dels at man kan vurdere anvendelsen af kendte parsertværktøjer og/eller -teknikker, dels at man har opnået en forståelse for hvordan konkrete sproglige begreber repræsenteres på køretidspunktet.

_PE-kurser:_ Der udbydes normalt projektenhedskurser indenfor emnerne: Sprog og oversættelse (SPO, 3 ECTS) samt Syntaks og semantik (SS, 3 ECTS).

Studieenhedskurser: DNA og DBS.
SPO as PE course on DAT2/F6S

6.3.2.1 Projektenhed DAT2A

Tema: Sprog og oversættelse / Language and Compilation.
Omfang: 22 ECTS.

Formål: At kunne anvende væsentlige principper i programmeringssprog og teknikker til beskrivelse og oversættelse af sprog generelt.

Indhold: Projektet består i en analyse af en datalogisk problemstilling, hvis løsning naturligt kan beskrives i form af et design af væsentlige begreber for et konkret programmeringssprog. I tilknytning hertil skal konstrueres en oversætter/fortolker for sproget, som viser dels at man kan vurdere anvendelsen af kendte parseværktøjer og/eller -teknikker, dels at man har opnået en forståelse for hvordan konkrete sproglige begreber repræsenteres på køretidspunktet.

PE-kurser: MVP, SPO

Studieenhedskurser: DNA, SS og PSS.
SS as PE course on DAT2/F6S

6.3.2.3 Projektenhed DAT2C
Tema: Syntaks og semantik / Formal Languages - Syntax and Semantics.
Omfang: 22 ECTS.

Formål: At kunne anvende modeller for beskrivelse af syntaktiske og semantiske aspekter af programmeringssprog og anvende disse i implementation af sprog og verifikation/analyse af programmer.

Indhold: Et typisk projekt vil bl.a. indeholde præcis definition af de væsentlige dele af et sprogs syntaks og semantik og anvendelser af disse definitioner i implementation af en oversætter/fortolker for sproget og/eller verifikation.

PE-kurser: MVP, SS.

Studieenhedskurser: DNA, SPO og PSS.
SPO or SS as PE course

• Choose SPO as PE course
  – If your focus is on language design and/or implementation of a compiler/interpreter
  – If you like to talk about SS at the course exam

• Choose SS as PE course
  – If your focus is on language definition and/or correctness proofs of implementation
  – If you like to talk about SPO at the course exam
Programming Language Projects

- A good DAT2F6S/SW4 project group can
  - Design a language (or language extensions)
  - Define the language syntax using CFG
  - Define the language semantics using SOS
  - Implement a compiler/interpreter
    - in Java (or C/C++, C#, SML, …)
    - Using front-end tools such as JavaCC or SableCC
    - Do code generation for abstract machine
      - TAM, JVM (PerlVM or .Net CLR) or new VM
    - Or code generation to some high level language
      - C, Java, C#, SQL, XML
  - Prove correctness of compiler
    - Using SOS for Prg. Lang. and VM
Some advice

- A language design and compiler project is easy to structure.
  - Design phase
  - Front-end development
  - Contextual analysis
  - Code generation or interpretation

- You will learn the techniques and tools you need in time for you to apply them in your project
Programming Language Life cycle

- Design
- Specification
- Manuals, Textbooks
- Compiler
- Prototype
The course in a snapshot

• Lecture 1 – overview + language specification concepts
• Lecture 2 – programming language concepts and design issues
• Lecture 3 – Syntax analysis – recursive decent parsers
• Lecture 4 – Syntax analysis – JavaCC, JLex+CUP
• Lecture 5 – Syntax analysis – LR parsing - SableCC
• Lecture 6 – Contextual Analysis
• Lecture 7 – Type systems
• Lecture 8 – More programming language design issues
• Lecture 9 – Interpretation and virtual machines
• Lecture 10 – Code generation
• Lecture 11 - Code generation
• Lecture 12 – Run-time organisation and garbage collection
• Lecture 13 - Design issues for OO languages
• Lecture 14 – Design issues for concurrent and distributed languages
• Lecture 15 – Compiler optimizations and Programming Language life cycle
Some advice on Project Prosals

• The most successful DAT2/SW4/F6S projects are those that manage to use the SPO, SS and DNA courses
  – Usually that means designing, specifying and implementing a ”traditional” block structured PASCAL or C like language or extensions of such languages

• Project that in the past have had problems are:
  – Extensions to SQL or other DB languages
  – Projects targeting low-level or odd-hardware
  – Anything XML
Summary

• Programming Language Design
  – New features
  – History, Paradigm, philosophy

• Programming Language Specification
  – Syntax
  – Contextual constraints
  – Meaning (semantics and code generation)

• Programming Language Implementation
  – Compiler
  – Interpreter
  – Hybrid system
Finally

Keep in mind, the compiler is the program from which all other programs arise. If your compiler is under par, all programs created by the compiler will also be under par. No matter the purpose or use -- your own enlightenment about compilers or commercial applications -- you want to be patient and do a good job with this program; in other words, don't try to throw this together on a weekend.

Asking a computer programmer to tell you how to write a compiler is like saying to Picasso, "Teach me to paint like you."

*Sigh* Nevertheless, Picasso shall try.