Compiler I

(dt. Übersetzer I)

Prof. Dr. Uwe Kastens

Winter 2001/2002

Lecture Compiler I WS 2001/2002 / Slide 01

In the lecture:

Welcome to the lecture!

Objectives

The participants are taught to

- understand **fundamental techniques** of language implementation,
- use **generating tools and standard solutions**,
- understand compiler construction as a systematic combination of **algorithms, theories** and **software engineering** methods for the solution of a **precisely specified task**,
- apply compiler techniques for languages **other than programming languages**.

Forms of teaching:

Lectures

Tutorials

Homeworks

Exercises

Running project

Lecture Compiler I WS 2001/2002 / Slide 02

Objectives:

Understand the objectives of the course.

In the lecture:

The objectives are explained.

Questions:

- What are your objectives?
- Do they match with these?
- When did you last listen to a talk given in English?

Lectures in English

Some agreements about giving lectures in English:

- I’ll speak English unless someone asks me to explain something in German.
- Stop me or slow me down whenever you get lost.
- I don‘t speak as well as a native speaker; but I’ll do my best ...
- You may ask questions and give answers in English or in German.
- I’ll prepare the slides in English. A German version is available.
- You‘ll have to learn to speak about the material in at least one of the two languages.
- You may vote which language to be used in the tutorials.
- You may chose German or English for the oral exam.

Lecture Compiler I WS 2001/2002 / Slide 03

Objectives:

Clarification about the use of the English language in this course

In the lecture:

The topics on the slide are discussed.

Syllabus

Week	Chapter	Topic
1	**Introduction**	Compiler tasks
2		Compiler structure
3	**Lexical analysis**	Scanning, token representation
4	**Syntactic analysis**	Recursive decent parsing
5		LR Parsing
6		Parser generators
7		Grammar design
8	**Semantic analysis**	Attribute grammars
9		Attribute grammar specifications
10		Name analysis
11		Type analysis
12	**Transformation**	Intermediate language, target trees
13		Target texts
14	**Synthesis**	Overview
15	**Summary**	

Lecture Compiler I WS 2001/2002 / Slide 04

Objectives:

Overview over the topics of the course

In the lecture:

Comments on the topics.

Prerequisites

from Lecture	Topic	here needed for
Foundations of Programming Languages:		
	4 levels of language properties	Compiler tasks, compiler structure
	Context-free grammars	Syntactic analysis
	Scope rules	Name analysis
	Data types	Type analysis
	Lifetime, runtime stack	Storage model, code generation
Modeling:		
	Finite automata	Lexical analysis
	Context-free grammars	Syntactic analysis

Lecture Compiler I WS 2001/2002 / Slide 05

Objectives:

Identify concrete topics of other courses

In the lecture:

Point to material to be used for repetition

Suggested reading:

Course material for *Foundations of Programming Languages*

Course material for *Modeling*

Questions:

- Do you have the prerequisites?
- Are you going to learn or to repeat that material?

References

Material for this course **Compiler I**: http://www.uni-paderborn.de/cs/ag-kastens/compi
in German **Übersetzer I** (1999/2000): http://www.uni-paderborn.de/cs/ag-kastens/uebi
in English **Compiler II**: http://www.uni-paderborn.de/cs/ag-kastens/uebii

Modellierung: http://www.uni-paderborn.de/cs/ag-kastens/model
Grundlagen der Programmiersprachen: http://www.uni-paderborn.de/cs/ag-kastens/gdp

U. Kastens: **Übersetzerbau**, Handbuch der Informatik 3.3, Oldenbourg, 1990
(not available on the market anymore, available in the library of the University)

W. M. Waite, L. R. Carter: **An Introduction to Compiler Construction,**
Harper Collins, New York, 1993

W. M. Waite, G. Goos: **Compiler Construction**, Springer-Verlag, 1983

R. Wilhelm, D. Maurer: **Übersetzerbau - Theorie, Konstruktion, Generierung**,
Springer-Verlag, 1992

A. Aho, R. Sethi, J. D. Ullman: **Compilers - Principles, Techniques and Tools**,
Addison-Wesley, 1986

A. W. Appel: **Modern Compiler Implementation in C**, Cambridge University Press, 1997
(available for Java and for ML, too)

Lecture Compiler I WS 2001/2002 / Slide 06

Objectives:

Useful references for the course

In the lecture:

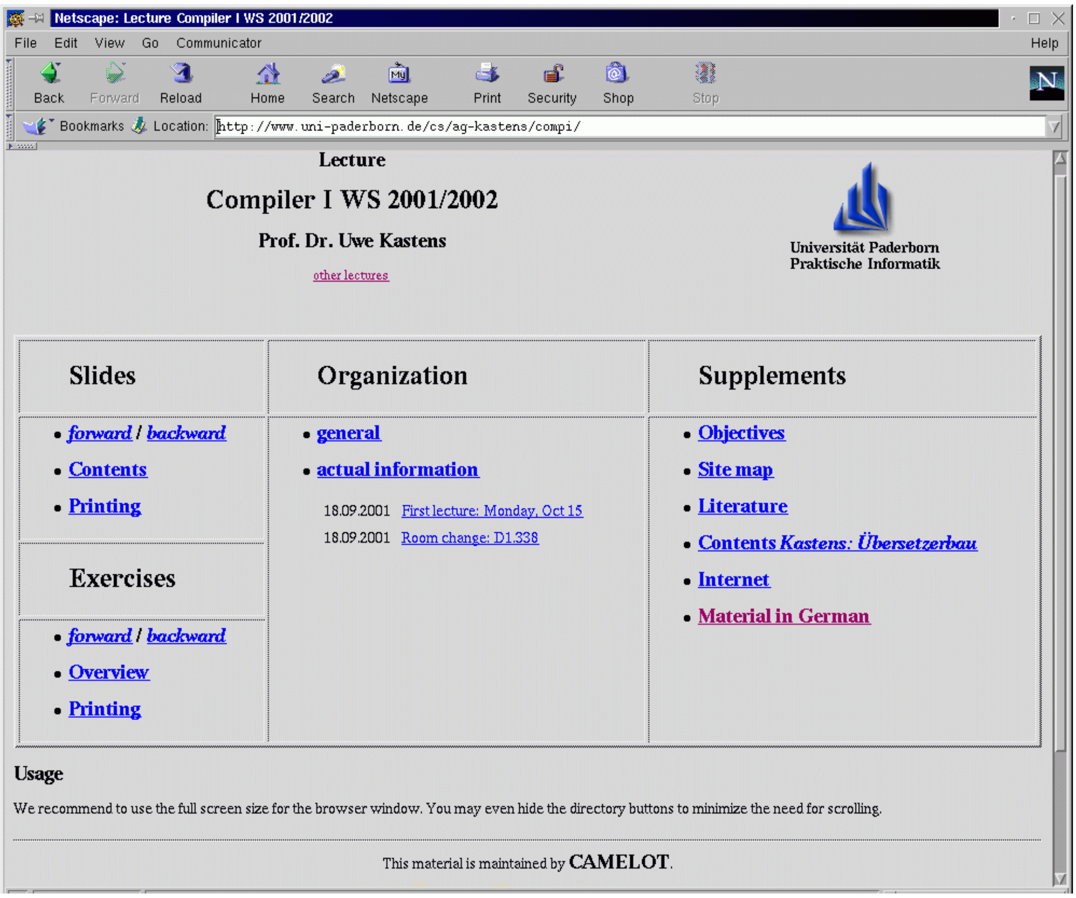
Comments of the course material and books

- The material for this course is being translated from the material of "Übersetzer I (WS 1999/2000)" while the course is given
- The course "Compiler II" will follow next semester.

Questions:

- Find the material in the Web, get used to its structure, place suitable bookmarks.

Course material in the Web



Lecture Compiler I WS 2001/2002 / Slide 07

Objectives:

The root page of the course material.

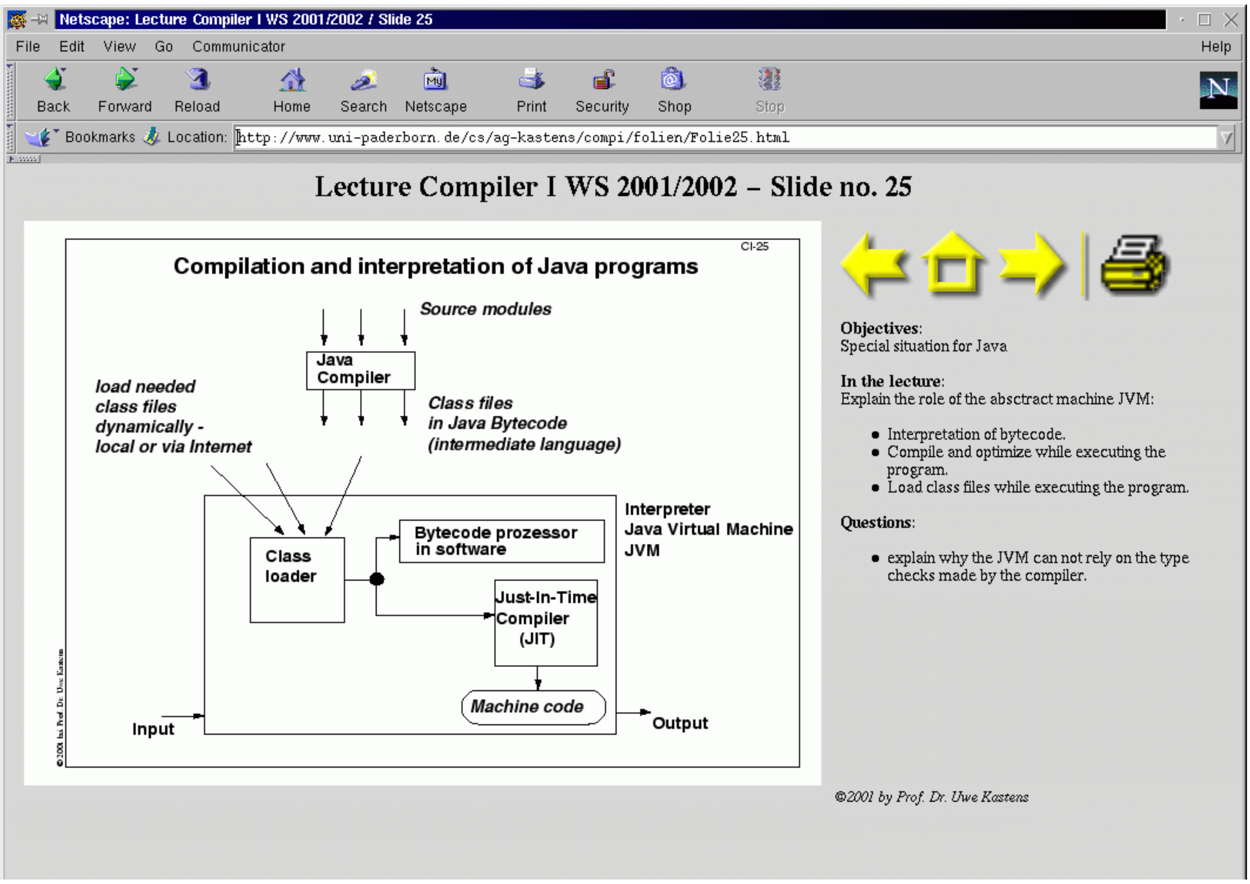
In the lecture:

The navigation structure is explained.

Assignments:

Explore the course material.

Commented slide in the course material



Lecture Compiler I WS 2001/2002 / Slide 07a

Objectives:

A slide of the course material.

In the lecture:

The comments are explained.

Assignments:

Explore the course material.

What does a compiler compile?

A **compiler** transforms correct sentences of its **source language** into sentences of its **target language** such that their **meaning is unchanged.**

Examples:

Source language:	**Target language:**
Programming language C++	**Machine language** Sparc code
Programming language Java	**Abstract machine** Java Bytecode
Programming language C++	**Programming language (source-to-source)** C
Application language LaTeX Data base language (SQL)	**Application language** HTML Data base system calls

Lecture Compiler I WS 2001/2002 / Slide 08

Objectives:

Variety of compiler applications

In the lecture:

Explain examples for pairs of source and target languages.

Suggested reading:

Kastens / Übersetzerbau, Section 1.

Assignments:

- Find more examples for application languages.
- Exercise 3 Recognize patterns in the target programs compiled from simple source programs.

Questions:

What are reasons to compile into other than machine languages?

What is compiled here?

```
class Average
    { private:
        int sum, count;
      public:
        Average (void)
          { sum = 0; count = 0; }
        void Enter (int val)
         { sum = sum + val; count++; }
        float GetAverage (void)
          { return sum / count; }
    };
--------------
_Enter__7Averagei:
            pushl %ebp
            movl %esp,%ebp
            movl 8(%ebp),%edx
            movl 12(%ebp),%eax
            addl %eax,(%edx)
            incl 4(%edx)
    L6:
            movl %ebp,%esp
            popl %ebp
            ret
```

```
class Average
{ private
    int sum, count;
  public
    Average ()
      { sum = 0; count = 0; }
    void Enter (int val)
      { sum = sum + val; count++; }
    float GetAverage ()
      { return sum / count; }
};
---------------
1: Enter: (int) --> void
   Access: []
   Attribute ‚Code‘ (Length 49)
      Code: 21 Bytes Stackdepth: 3 Locals: 2
      0:    aload_0
      1:    aload_0
      2:    getfield cp4
      5:    iload_1
      6:    iadd
      7:    putfield cp4
      10:   aload_0
      11:   dup
      12:   getfield cp3
      15:   iconst_1
      16:   iadd
```


Lecture Compiler I WS 2001/2002 / Slide 09

Objectives:

Recognize examples for compilations

In the lecture:

Anwer the questions below.

Questions:

- Which source and target language are shown here?
- How did you recognize them?

What is compiled here?

```
program Average;
      var sum, count: integer;
          aver: integer;
      procedure Enter (val: integer);
          begin sum := sum + val;
                count := count + 1;
          end;
    begin
      sum := 0; count := 0;
      Enter (5); Enter (7);
      aver := sum div count;
    end.
-----------
void ENTER_5 (char *slnk , int VAL_4)
     {
     {/* data definitions: */
        /* executable code: */
        {
           SUM_1 = (SUM_1)+(VAL_4);
           COUNT_2 = (COUNT_2)+(1);
           ;
        }
     }}/* ENTER_5 */
```

```
\documentstyle[12pt]{article}
\begin{document}
\section{Introduction}
This is a very short document.
It just shows
\begin{itemize}
\item an item, and
\item another item.
\end{itemize}
\end{document}

-------------

%%Page: 1 1
1 0 bop 164 315 a Fc(1)81
b(In)n(tro)r(duction)
164 425 y Fb(This)16
b(is)g(a)h(v)o(ery)e(short)
i(do)q(cumen)o(t.)j(It)c(just)g
(sho)o(ws)237 527 y Fa(\017)24 b
Fb(an)17 b(item,)
c(and)237 628 y Fa(\017)24 b
Fb(another)17 b(item.)
961 2607 y(1)p
eop
```


Lecture Compiler I WS 2001/2002 / Slide 10

Objectives:

Recognize examples for compilations

In the lecture:

Anwer the questions below.

Questions:

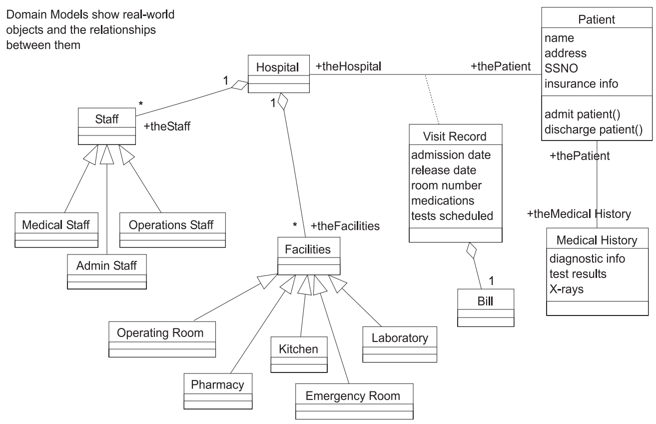
- Which source and target language are shown here?
- How did you recognize them?

Languages for specification and modeling

SDL (CCITT)
Specification and Description Language:

```
block Dialogue;
  signal
    Money, Release, Change, Accept, Avail, Unavail, Price,
    Showtxt, Choice, Done, Flushed, Close, Filled;
  process Coins referenced;
  process Control referenced;
  process Viewpoint referenced;
  signalroute Plop
    from env to Coins
      with Coin_10, Coin_50, Coin_100, Coin_x;
  signalroute Pong
    from Coins to env
      with Coin_10, Coin_50, Coin_100, Coin_x;
  signalroute Cash
    from Coins to Control
      with Money, Avail, Unavail, Flushed, Filled;
    from Control to Coins
      with Accept, Release, Change, Close;

  ...

  connect Pay and Plop;
  connect Flush and Pong;
endblock Dialogue;
```

UML
Unified Modeling Language:



Lecture Compiler I WS 2001/2002 / Slide 11

Objectives:

Be aware of specification languages

In the lecture:

Comments on SDL and UML

Suggested reading:

Text

Questions:

What kind of tools are needed for such specification languages?

Domain Specific Languages (DSL)

A language designed for a **specific application domain.**
Application Generator: Implementation of a DSL by a **program generator**

Examples:

- **Simulation of mechatronic feedback systems**
- **Robot control**
- **Collecting data from instruments**
- **Testing car instruments**
- **Report generator for bibliographies:**

```
string name =   InString "Which author?";
int since =     InInt "Since which year?";
int cnt = 0;

"\nPapers of ", name, " since ", since, ":\n";
[ SELECT name <= Author && since <= Year;
  cnt = cnt + 1;
  Year, "\t", Title, "\n";
]
"\n", name, " published ", cnt, "papers.\n";
```

U. Kastens: Construction of Application Generators Using Eli, Workshop on Compiler Techniques for Application Domain Languages ..., Linköping, April 1996

Lecture Compiler I WS 2001/2002 / Slide 12

Objectives:

Understand DSL by examples

In the lecture:

Explain the examples

Suggested reading:

- C.W. Krueger: Software Reuse, ACM Computing Surveys 24, June 1992
- Conference on DSL (USENIX), Santa Babara, Oct. 1997
- ACM SIGPLAN Workshop on DSL (POPL), Paris, Jan 1997

Questions:

Give examples for tools that can be used for such languages.

Programming languages as source or target languages

Programming languages as source languages:

- **Program analysis**
  call graphs, control-flow graph, data dependencies, e. g. for the year 2000 problem
- **Recognition of structures and patterns**
  e. g. for Reengineering

Program languages as target languages:

- **Specifications (SDL, OMT, UML)**
- **graphic modeling of structures**
- **DSL, Application generator**

=> Compiler task: Source-to-source compilation

Lecture Compiler I WS 2001/2002 / Slide 13

Objectives:

Understand programming languages in different roles

In the lecture:

- Comments on the examples
- Role of program analysis in software engineering
- Role of Source-to-source compilation in software engineering

Questions:

Give examples for the use of program analysis in software engineering.

Semester project as running example

A Structure Generator

We are going to develop a tool that implements **record structures**. In particular, the structure generator takes a set of **record descriptions**. Each specifies a **set of named and typed fields**. For each record a **Java class** declaration is to be generated. It contains a constructor method and access methods for the specified record fields.

The tool will be used in an environment where field description are created by other tools, which for example analyze texts for the occurrence of certain phrases. Hence, the descriptions of fields may occur in arbitrary order, and the same field may be described more than once. The structure generator **accumulates the field descriptions** such that for each record a single class declaration is generated which has all fields of that record.

Design a **domain specific language**.

Implement an **application generator** for it.

Apply all **techniques of the course** that are useful for the task.

Lecture Compiler I WS 2001/2002 / Slide 14

Objectives:

Get an idea of the task

In the lecture:

- Comment the task description.
- Explain the role of the running example.

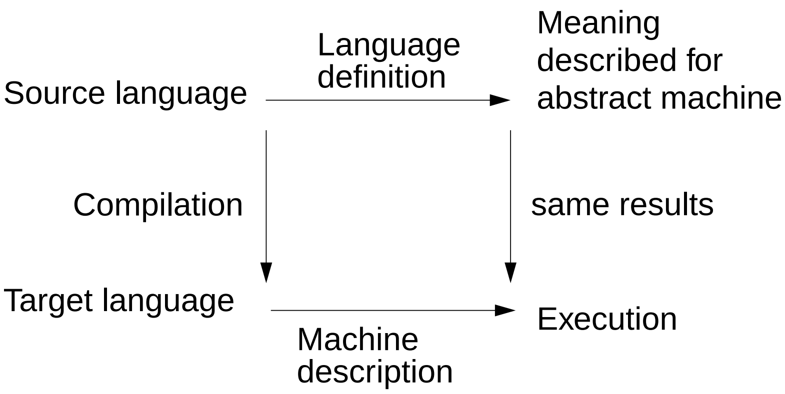
Assignments:

In the tutorial

- Discuss the task description.
- Explain the purpose of such a generator.
- Give examples for its input and output.
- What are the consequences of the second paragraph of the task description?
- Discuss variants of the input.

Meaning preserving transformation

A **compiler** transforms correct sentences of its **source language** into sentences of its **target language** such that their **meaning is unchanged**.



A **meaning** is defined only for **correct** programs. Compiler task: Error handling

The compiler analyses **static** properties of the program at **compile time,**
e. g. definitions of Variables, types of expressions. Decides: Is the program **compilable?**

Dynamic properties of the program are checked at **runtime,**
e. g. indexing of arrays. Decides: Is the program **executable?**

But in Java: Compilation of bytecode at runtime, just in time compilation (JIT)

Lecture Compiler I WS 2001/2002 / Slide 15

Objectives:

Understand fundamental notions of compilation

In the lecture:

The topics on the slide are explained. Examples are given.

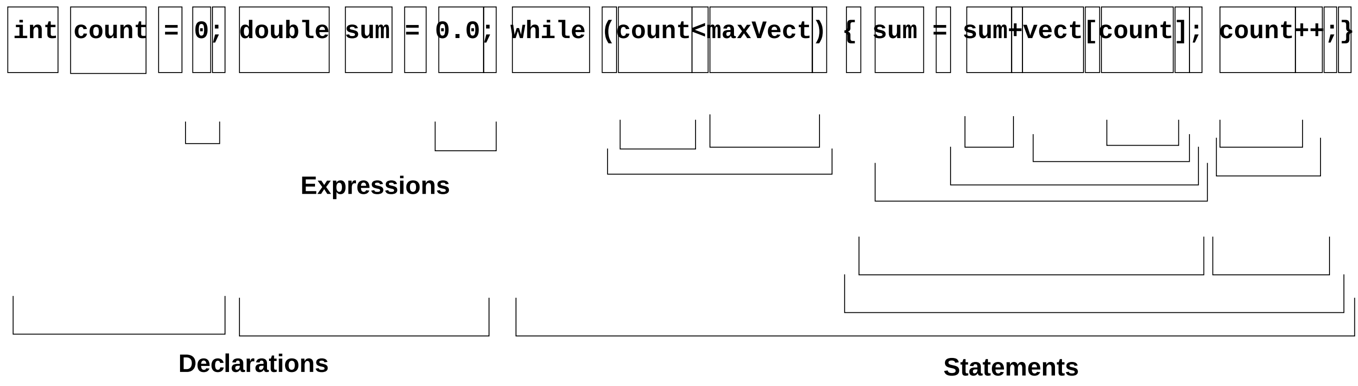
- Explain the role of the arcs in the commuting diagram.
- Distinguish compile time and run-time concepts.
- Discuss examples.

Example: Tokens and structure

Character sequence

```
int count = 0; double sum = 0.0; while (count<maxVect) { sum = sum+vect[count]; count++;}
```

Tokens



Structure

Lecture Compiler I WS 2001/2002 / Slide 16

Objectives:

Get an idea of the structuring task

In the lecture:

Some requirements for recognizing tokens and deriving the program structure are discussed along the example:

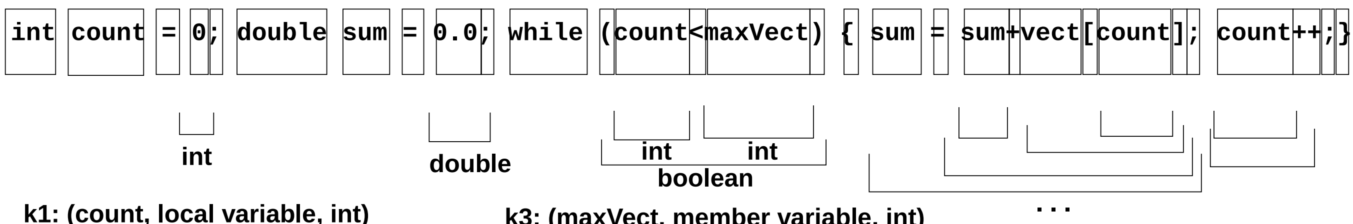
- kinds of tokens,
- characters between tokens,
- nested structure

Questions:

Where do you find the exact requirements for the structuring tasks?

Example: Names, types, generated code

Tokens



k1: (count, local variable, int)
k2: (sum, local variable, double)

k3: (maxVect, member variable, int)
k4: (vect, member variable, double array)

Names and types

generated Bytecode

```
0 iconst_0
1 istore_1
2 dconst_0
3 dstore_2
4 goto 19
7 dload_2
8 getstatic #5 <vect[]>
11 iload_1
12 faload
13 f2d
14 dadd
15 dstore_2
16 iinc 1 1
19 iload_1
20 getstatic #4 <maxVect>
23 if_icmplt 7
```


Lecture Compiler I WS 2001/2002 / Slide 17

Objectives:

Get an idea of the name analysis and transformation task

In the lecture:

Some requirements for these tasks are discussed along the example:

- program objects and their properties,
- program constructs and their types
- target program

Questions:

- Why is the name (e.g. count) a property of a program object (e.g. k1)?
- Can you impose some structure on the target code?

Language definition - Compiler task

• **Notation of tokens** keywords, identifiers, literals formal definition: regular expressions	**lexical analysis**
• **Syntactic structure** formal definition: context-free grammar	**syntactic analysis**
• **Static semantics** binding names to program objects, typing rules usually defined by informal texts	**semantic analysis, transformation**
• **Dynamic semantics** semantics, effect of the execution of constructs usually defined by informal texts in terms of an abstract machine	**transformation, code generation**
• **Definition of the target language (machine)**	**transformation, code generation** **assembly**

Lecture Compiler I WS 2001/2002 / Slide 18

Objectives:

Relate language properties to levels of definitions

In the lecture:

- These are prerequisites of the course "Grundlagen der Programmiersprachen" (see course material GdP-13, GdP13a).
- Discuss the examples of the preceding slides under these categories.

Suggested reading:

Kastens / Übersetzerbau, Section 1.2

Assignments:

- Exercise 1 Let the compiler produce error messages for each level.
- Exercise 2 Relate concrete language properties to these levels.

Questions:

Some language properties can be defined on different levels. Discuss the following for hypothetical languages:

- "Parameters may not be of array type." Syntax or static semantics?
- "The index range of an array may not be empty." Static or dynamic semantics?

Compiler tasks

Structuring	Lexical analysis	Scanning Conversion
	Syntactic analysis	Parsing Tree construction
Translation	Semantic analysis	Name analysis Type analysis
	Transformation	Data mapping Action mapping
Encoding	Code generation	Execution-order Register allocation Instruction selection
	Assembly	Instruction encoding Internal Addressing External Addressing

Lecture Compiler I WS 2001/2002 / Slide 19

Objectives:

Task decomposition leads to compiler structure

In the lecture:

- Explain tasks of the rightmost column.
- Relate the tasks to chapters of the course.

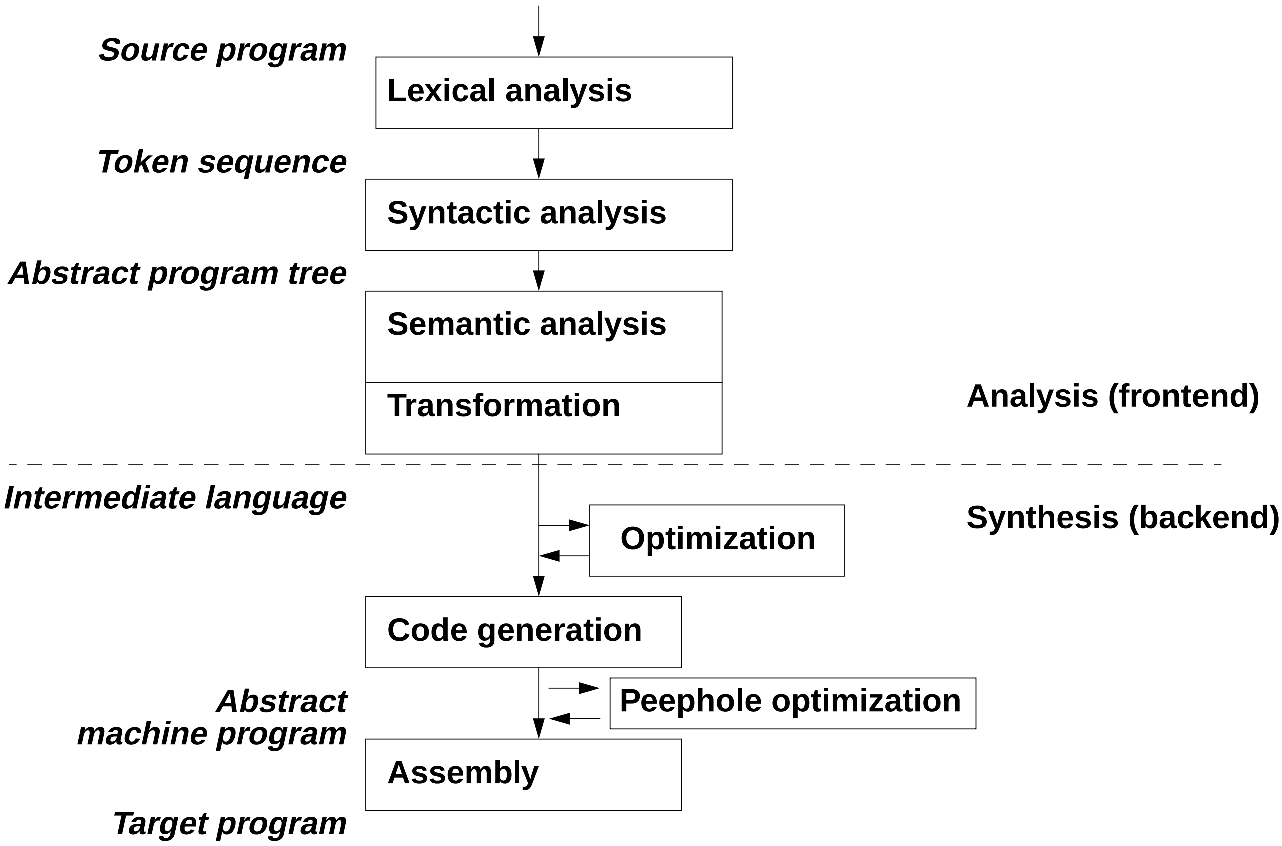
Suggested reading:

Kastens / Übersetzerbau, Section 2.1

Assignments:

Learn the German translations of the technical terms.

Compiler structure and interfaces



Lecture Compiler I WS 2001/2002 / Slide 20

Objectives:

Derive compiler modules from tasks

In the lecture:

In this course we focus on the analysis phase (frontend).

Suggested reading:

Kastens / Übersetzerbau, Section 2.1

Assignments:

Compare this slide with U-08 and learn the translations of the technical terms used here.

Questions:

Use this information to explain the example on slide CI-16

Software qualities of the compiler

- **Correctness** — Translate correct programs correctly.
  Reject wrong programs and give error messages
- **Efficiency** — Storage and time used by the compiler
- **Code efficiency** — Storage and time used by the generated code
  Compiler task: Optimization
- **User support** — Compiler task: Error handling
  (recognition, message, recovery)
- **Robustness** — Give a reasonable reaction on every input

Lecture Compiler I WS 2001/2002 / Slide 21

Objectives:

Consider compiler as a software product

In the lecture:

Give examples for the qualities.

Questions:

Explain: For a compiler the requirements are specified much more precisely than for other software products.

Strategies for compiler construction

- Obey exactly to the language definition
- Use generating tools
- Use standard components
- Apply standard methods
- Validate the compiler against a test suite
- Verify components of the compiler

Lecture Compiler I WS 2001/2002 / Slide 22

Objectives:

Apply software methods for compiler construction

In the lecture:

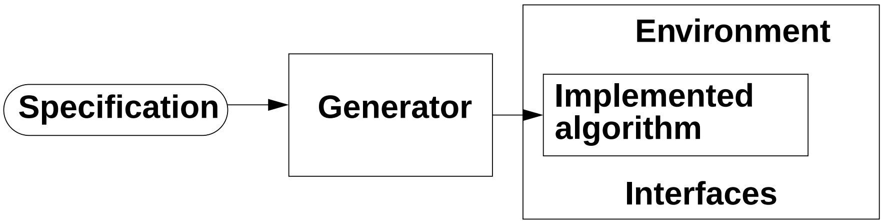
It is explained that effective construction methods exists especially for compilers.

Questions:

What do the specifications of the compiler tasks contribute to more systematic compiler construction?

Generators

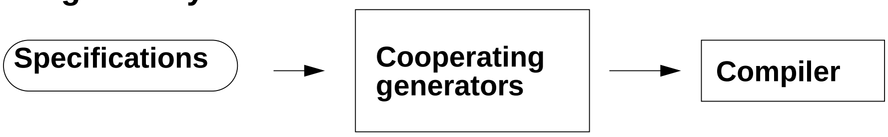
Pattern:



Typical compiler tasks solved by generators:

Regular expressions	**Scanner generator**	Finite automaton
Context-free grammar	**Parser generator**	Stack automaton
Attribute grammar	**Attribute evaluator generator**	Tree walking algorithm
Code patterns	**Code selection generator**	Pattern matching

integrated system Eli:



Lecture Compiler I WS 2001/2002 / Slide 23

Objectives:

Usage of generators in compiler construction

In the lecture:

The topics on the slide are explained. Examples are given.

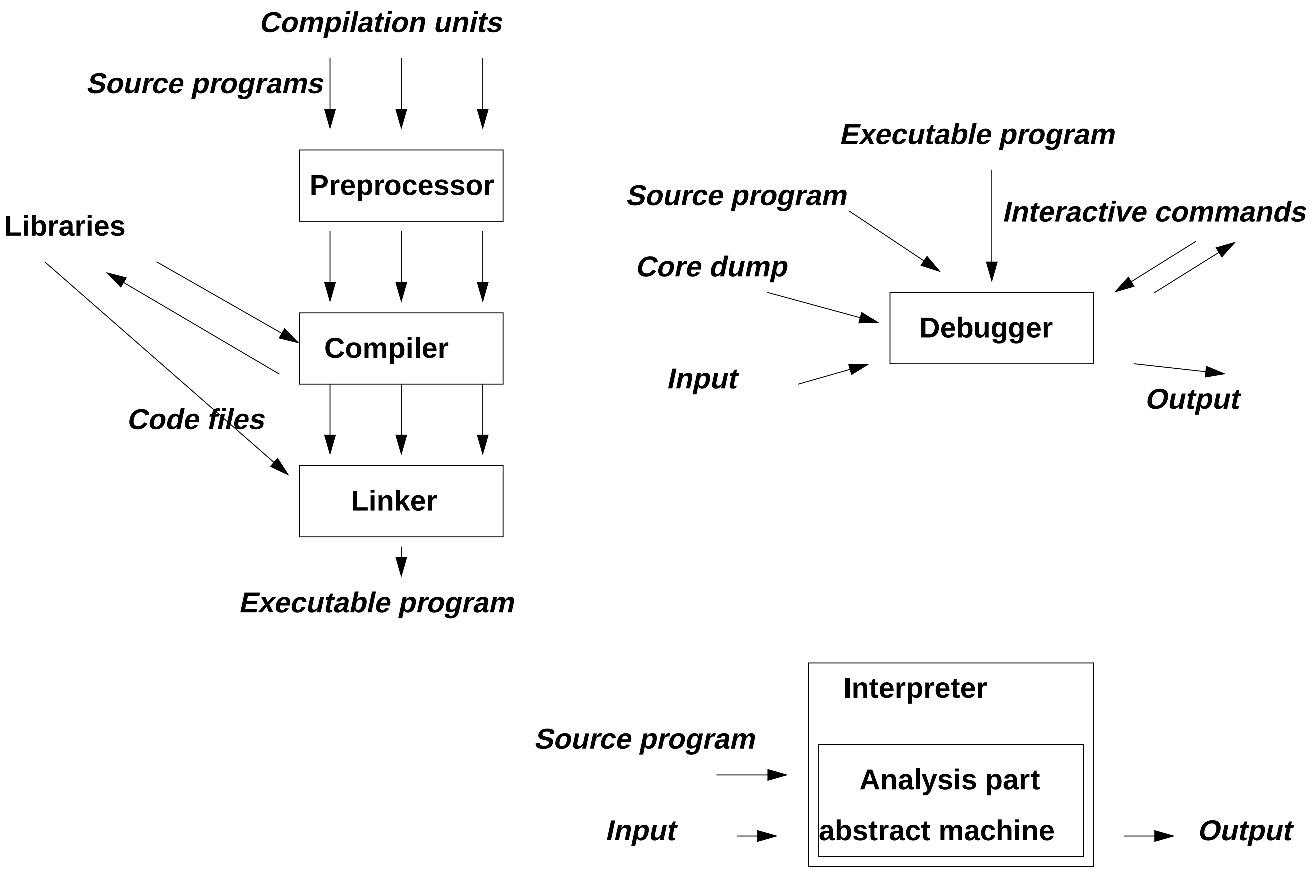
Suggested reading:

Kastens / Übersetzerbau, Section 2.5

Assignments:

- Exercise 5: Find as many generators as possible in the Eli system.

Environment of compilers



Lecture Compiler I WS 2001/2002 / Slide 24

Objectives:

Understand the cooperation between compilers and other language tools

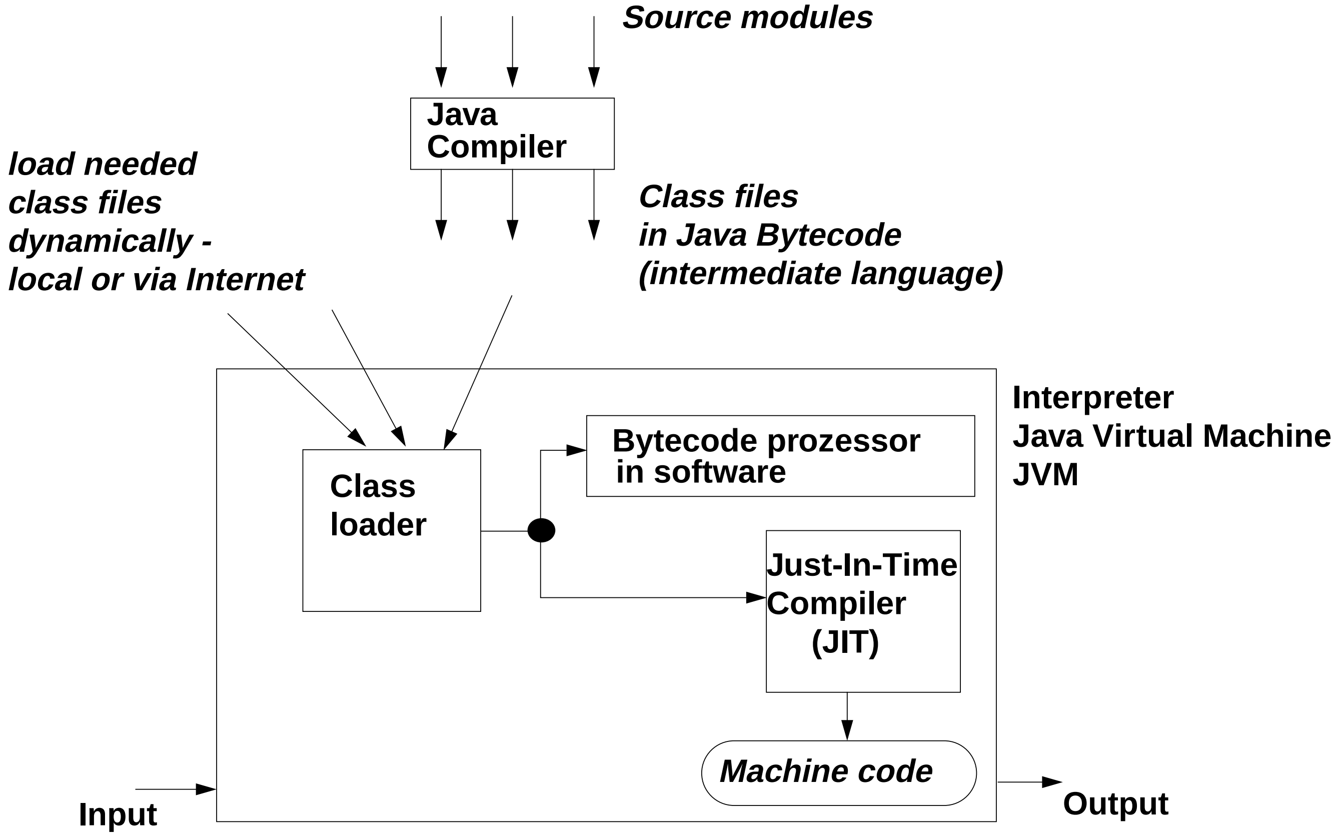
In the lecture:

- Explain the roles of language tools
- Explain the flow of information

Suggested reading:

Kastens / Übersetzerbau, Section 2.4

Compilation and interpretation of Java programs



Lecture Compiler I WS 2001/2002 / Slide 25

Objectives:

Special situation for Java

In the lecture:

Explain the role of the absctract machine JVM:

- Interpretation of bytecode.
- Compile and optimize while executing the program.
- Load class files while executing the program.

Questions:

- explain why the JVM can not rely on the type checks made by the compiler.

Lexical Analysis

Input: ***Program represented by a sequence of characters***

Tasks:	**Compiler modul:**
	Input reader
Recognize and classify tokens	Scanner (central phase, finite state machine)
Skip irrelevant characters	
Encode tokens:	
	Identifier modul
Store token information	Literal modules
Conversion	String storage

Output: ***Program represented by a sequence of encoded tokens***

Lecture Compiler I WS 2001/2002 / Slide 26

Objectives:

Understand lexical analysis subtasks

In the lecture:

Explain

- subtasks and their interfaces using slide CI-16,
- unusual notation of keywords,
- different forms of comments,
- sparation of tokens in FORTRAN,

Suggested reading:

Kastens / Übersetzerbau, Section 3, 3.3.1

Questions:

- Give examples of context dependent information about tokens, which the lexical analysis can not know.
- Some decisions on the notation of tokens and the syntax of a language may complicate lexical analysis. Give examples.
- Explain the typedef problem in C.

Representation of tokens

Uniform encoding of tokens by triples:

Syntax code	attribute	source position
terminal code of the concrete syntax	value or reference into data module	to locate error messages of later compiler phases

Examples:

```
double sum = 5.6e-5;
while (count < maxVect)
{ sum = sum + vect[count];
```

DoubleToken		12, 1
Ident	138	12, 8
Assign		12, 12
FloatNumber	16	12, 14
Semicolon		12, 20
WhileToken		13, 1
OpenParen		13, 7
Ident	139	13, 8
LessOpr		13, 14
Ident	137	13, 16
CloseParen		13, 23
OpenBracket		14, 1
Ident	138	14, 3

Lecture Compiler I WS 2001/2002 / Slide 27

Objectives:

Understand token representation

In the lecture:

Explain the roles of the 3 components using the examples

Suggested reading:

Kastens / Übersetzerbau, Section 3, 3.3.1

Questions:

- What are the requirements for the encoding of identifiers?
- How does the identifier module meet them?
- Can the values of integer literals be represented as attribute values, or do we have to store them in a data module? Explain! Consider also cross compilers!

Specification of token notations

Example: identifiers

Ident = Letter (Letter | Digit)*

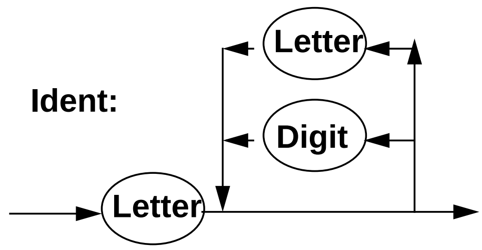
regular grammar

regular expression

Ident ::= Letter X
X ::= Letter X
X ::= Digit X
X ::=

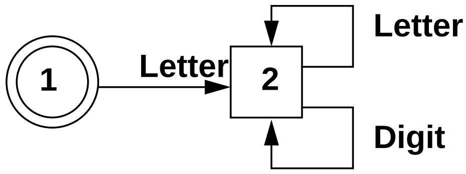
syntax diagram

Ident:



acceptor:

finite state machine



Lecture Compiler I WS 2001/2002 / Slide 28

Objectives:

Equivalent forms of specification

In the lecture:

- Repeat calculi of the lectures "Modellierung" and "Berechenbarkeit und formale Sprachen".
- Our strategy: Specify regular expressions, transform into syntax diagrams, and from there into finite state machines

Suggested reading:

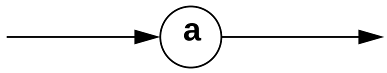
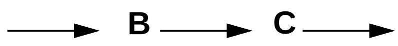
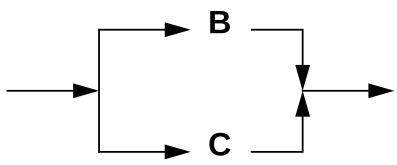
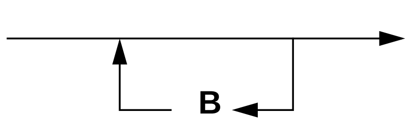
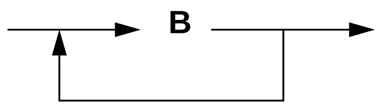
Kastens / Übersetzerbau, Section 3.1

Questions:

- Give examples for Unix tools which use regular expressions to describe their input.

Regular expressions mapped to syntax diagrams

Transformation rules:

regular expression A	syntax diagram for A	
empty		empty
a		single character
B C		sequence
B | C		alternative
B*		repetition, may be empty
B^+		repetition, non-empty

Lecture Compiler I WS 2001/2002 / Slide 29

Objectives:

Construct by recursive substitution

In the lecture:

- Explain the construction for floating point numbers of Pascal.

Suggested reading:

Kastens / Übersetzerbau, Section 3.1

Assignments:

- Apply the technique Exercise 6

Questions:

- If one transforms syntax diagrams into regular expressions, certain structures of the diagram requires duplication of subexpressions. Give examples.
- Explain the analogy to control flows of programs with labels, jumps and loops.

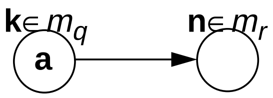
Construction of deterministic finite state machines

Syntax diagram	**deterministic finite state machine**
nodes, arcs	transitions, states
set of nodes m_q	state q
sets of nodes m_q and m_r connected with the same character *a*	transitions q ---> r with character *a*

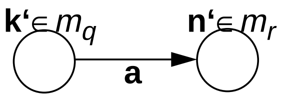
Construction:

1. **enumerate nodes**; exit of the diagram gets the number *0*
2. **initial set of nodes** m_1 contains all nodes that are reachable from the begin of the diagram — **initial state** *1*
3. **construct new sets of nodes (states)** and **transitions:** For a character *a* and a set m_q containing node *k* create set m_r with all nodes *n*, according to the following schema:

for



create



4. **repeat step 3** until no new sets of nodes can be created
5. a state *q* is a **final state** iff *0* is in m_q.

Lecture Compiler I WS 2001/2002 / Slide 30

Objectives:

Understand the method

In the lecture:

- Explain the idea with a small artificial example
- Explain the method using floating point numbers of Pascal (Slide CI-31)

Suggested reading:

Kastens / Übersetzerbau, Section 3.2

Assignments:

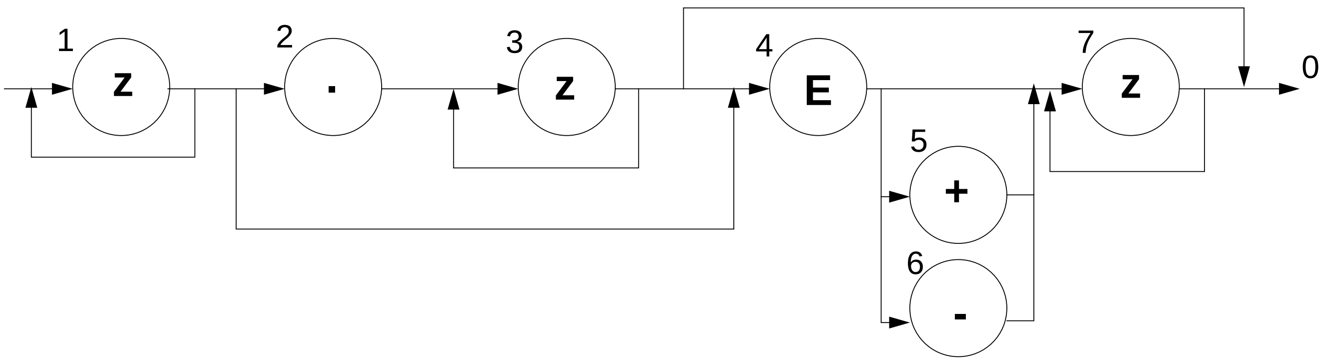
- Apply the method Exercise 6

Questions:

- Why does the method yield deterministic automata?
- Describe roughly a simple technique which may yield non-deterministic automata.

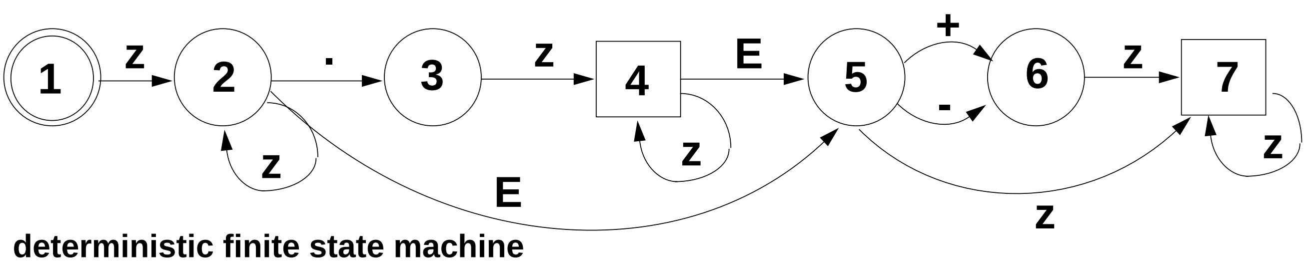
Example: Floating point numbers in Pascal

Syntax diagram



{1} **{1, 2, 4}** **{3}** **{3, 4, 0}** **{5, 6, 7}** **{7}** **{7, 0}**

z **z . E** **z** **z E** **+ - z** **z** **z**



deterministic finite state machine

Lecture Compiler I WS 2001/2002 / Slide 31

Objectives:

Understand the construction method

In the lecture:

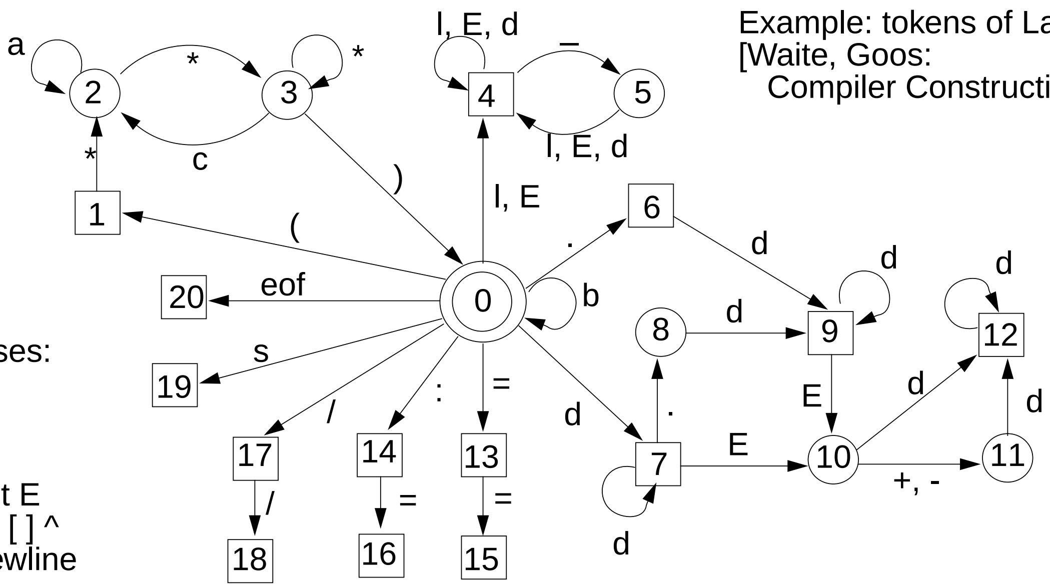
The construction process of slide CI-30 is explained using this example.

Composition of token automata

Construct one finite state machine for each token. Compose them forming a single one:

- **Identify the initial states of the single automata** and identical structures evolving from there (transitions with the same character and states).
- **Keep the final states of single automata distinct,** they classify the tokens.
- **Add automata for comments and irrelevant characters** (white space)

Example: tokens of Lax [Waite, Goos: Compiler Construction]



character classes:

a all but *
c all but * or)
d digits
l all letters but E
s + - * < > ; ,) [ ] ^
b blank tab newline

Lecture Compiler I WS 2001/2002 / Slide 32

Objectives:

Construct a multi-token automaton

In the lecture:

Use the example to

- discuss the composition steps,
- introduce the abbreviation by character classes,
- to see a non-trivial complete automaton.

Suggested reading:

Kastens / Übersetzerbau, Section 3.2

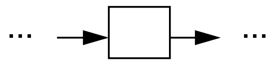
Questions:

Describe the notation of Lax tokens and comments in English.

Rule of the longest match

An automaton may contain **transitions from final states:**

When does the automaton stop?



Rule of the longest match:

- **The automaton continues as long as there is a transition with the next character.**
- **After having stopped it sets back to the most recently passed final state.**
- **If no final state has been passed an error message is issued.**

Consequence: Some kinds of tokens have to be separated explicitly.

Check the concrete grammar for tokens that may occur adjacent!

Lecture Compiler I WS 2001/2002 / Slide 33

Objectives:

Understand the consequences of the rule

In the lecture:

- Discuss examples for the rule of the longest match.
- Discuss different cases of token separation.

Suggested reading:

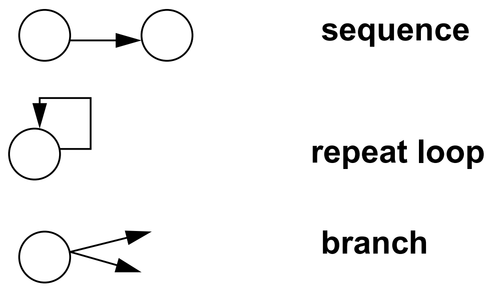
Kastens / Übersetzerbau, Section 3.2

Questions:

- Point out applications of the rule in the Lax automaton, which arose from the composition of sub-automata.
- Which tokens have to be separated by white space?

Scanner: Aspects of implementation

- **Runtime is proportional to the number of characters in the program**
- **Operations per character must be fast** - otherwise the Scanner dominates compilation time
- **Table driven** automata are too **slow:**
  Loop interprets table, 2-dimensional array access, branches
- **Directly programmed** automata is **faster;** transform **transitions into control flow:**



- **Fast loops** for sequences of irrelevant **blanks**.
- Implementation of **character classes**:
  bit pattern or indexing - avoid slow operations with sets of characters.
- **Do not copy characters** from input buffer - maintain a pointer into the buffer, instead.

Lecture Compiler I WS 2001/2002 / Slide 34

Objectives:

Runtime efficiency is important

In the lecture:

- Advantages of directly programmed automata. Compare to table driven.
- Measurements on occurrences of symbols: Single spaces, identifiers, keywords, squences of spaces are most frequent. Comments contribute most characters.

Suggested reading:

Kastens / Übersetzerbau, Section 3.3

Assignments:

- Generate directly programmed automata Exercise 7

Questions:

- Are there advantages for table-driven automata? Check your arguments carefully!

Identifier module and literal modules

- **Uniform interface for all scanner support modules:**
  Input parameters: pointer to token text and its length;
  Output parameters: syntax code, attribute

- **Identifier module encodes identifier occurrences bijective (1:1), and recognizes keywords**
  Implementation: hash vector, extensible table, collision lists

- **Literal modules for floating point numbers, integral numbers, strings**

  Variants for representation in memory:
  token text; value converted into compiler data; value converted into target data

  Caution:
  Avoid overflow on conversion!
  Cross compiler: compiler representation may differ from target representation

- **Character string memory:**
  stores strings without limits on their lengths,
  used by the identifier module and the literal modules

Lecture Compiler I WS 2001/2002 / Slide 35

Objectives:

Safe and efficient standard implementations are available

In the lecture:

- Give reasons for the implementation techniques.
- Show different representations of floating point numbers.
- Escape characters in strings need conversion.

Suggested reading:

Kastens / Übersetzerbau, Section 3.3

Questions:

- Give examples why the analysis phase needs to know values of integral literals.
- Give examples for representation of literals and their conversion.

Scanner generators

generate the central function of lexical analysis

GLA	University of Colorado, Boulder; component of the Eli system
Lex	Unix standard tool
Flex	Successor of Lex
Rex	GMD Karlsruhe

Token specification: regular expressions

GLA	library of precoined specifications; recognizers for some tokens may be programmed
Lex, Flex, Rex	transitions may be made conditional

Interface:

GLA	as described in this chapter; cooperates with other Eli components
Lex, Flex, Rex	actions may be associated with tokens (statement sequences) interface to parser generator Yacc

Implementation:

GLA	directly programmed automaton in C
Lex, Flex, Rex	table-driven automaton in C
Rex	table-driven automaton in C or in Modula-2
Flex, Rex	faster, smaller implementations than generated by Lex

Lecture Compiler I WS 2001/2002 / Slide 36

Objectives:

Know about some common generators

In the lecture:

Explain specific properties mentioned here.

Suggested reading:

Kastens / Übersetzerbau, Section 3.4

Assignments:

Use GLA and Lex Exercise 7

Syntactic analysis

Input: token sequence

Tasks:
Parsing: construct derivation according to **concrete syntax**,
Tree construction according to **abstract syntax**,
Error handling (detection, message, recovery)

Result: abstract program tree

Compiler module parser:

deterministic stack automaton, augmented by actions for tree construction

top-down parsers: leftmost derivation; tree construction top-down or bottom-up

bottom-up parsers: rightmost derivation backwards; tree construction bottom-up

Abstract program tree (condensed derivation tree):

represented by a **data structure in memory** for the translation phase to operate on,
linear **sequence of nodes on a file** (costly in runtime),
sequence of calls of functions of the translation phase.

Lecture Compiler I WS 2001/2002 / Slide 37

Objectives:

Relation between parsing and tree construction

In the lecture:

- Explain the tasks, use example on CI-16.
- Sources of prerequisites:
- context-free grammars: "Grundlagen der Programmiersprachen (2nd Semester), or "Berechenbarkeit und formale Sprachen" (3rd Semester),
- Tree representation in prefix form, postfix form: "Modellierung" (st Semester); see CI-5.

Suggested reading:

Kastens / Übersetzerbau, Section 4.1

Concrete and abstract syntax

concrete syntax	**abstract syntax**
context-free grammar	context-free grammar
defines the structure of source programs	defines abstract program trees
unambigous	usually ambiguous
specifies derivation and parser	translation phase is based on it
parser actions specify the --->	tree construction
some chain productions only for syntactic purpose `Expr ::= Fact` have no action	keep only semantically relevant ones no node created

symbols of syntactic chain productions comprised in symbol classes `Exp={Expr,Fact}`

same action at structural equivalent productions:

```
Expr ::= Expr AddOpr Fact &BinEx
Fact ::= Fact MulOpr Opd    &BinEx
```

terminal symbols	keep only semantically relevant ones as tree nodes
given the concrete syntax and the actions and	the symbol classes the abstract syntax can be generated

Lecture Compiler I WS 2001/2002 / Slide 38

Objectives:

Distinguish roles and properties of concrete and abstract syntax

In the lecture:

- Use the expression grammar of CI-39, CI-40 for comparison.
- Construct abstract syntax systematically.
- Context-free grammars specify trees - not only strings! Is also used in software engineering to specify interfaces.

Suggested reading:

Kastens / Übersetzerbau, Section 4.1

Assignments:

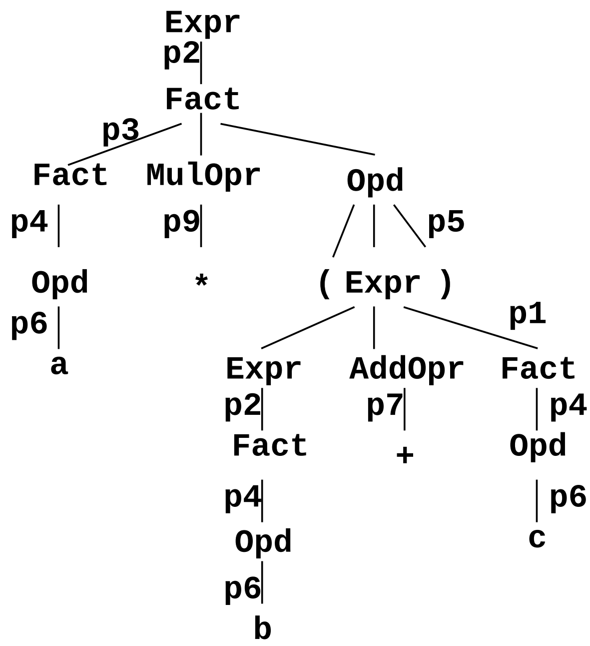
- Generate abstract syntaxes from concrete syntaxes and symbol classes.
- Use Eli for that task. Exercise 10

Questions:

- Why is no information lost, when an expression is represented by an abstract program tree?
- Give examples for semantically irrelevant chain productions outside of expressions.
- Explain: XML-based languages are defined by context-free grammars. Their sentences are textual representations of trees.

Example: concrete expression grammar

```
name  production                   action
p1:  Expr   ::= Expr AddOpr Fact   BinEx
p2:  Expr   ::= Fact
p3:  Fact   ::= Fact MulOpr Opd    BinEx
p4:  Fact   ::= Opd
p5:  Opd    ::= '(' Expr ')'
p6:  Opd    ::= Ident              IdEx
p7:  AddOpr ::= '+'                PlusOpr
p8:  AddOpr ::= '-'                MinusOpr
p9:  MulOpr ::= '*'                TimesOpr
p10: MulOpr ::= '/'                DivOpr
```



derivation tree for `a * (b + c)`

Lecture Compiler I WS 2001/2002 / Slide 39

Objectives:

Illustrate comparison of concrete and abstract syntax

In the lecture:

- Repeat concepts of "GdP" (slide GdP-2.5): Grammar expresses operator precedences and associativity.
- The derivation tree is constructed by the parser - not necessarily stored as a data structure.
- Chain productions have only one non-terminal symbol on their right-hand side.

Suggested reading:

Kastens / Übersetzerbau, Section 4.1

Suggested reading:

slide GdP-2.5

Questions:

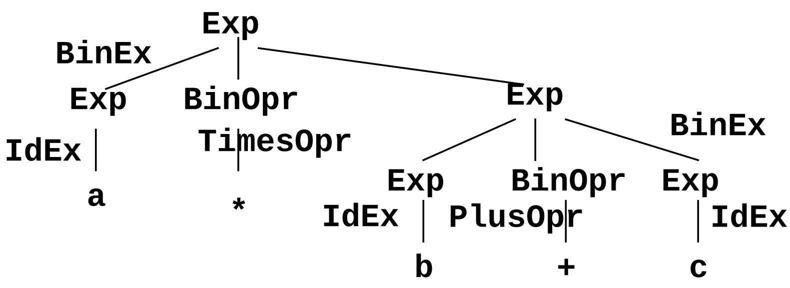
- How does a grammar express operator precedences and associativity?
- What is the purpose of the chain productions in this example.
- What other purposes can chain productions serve?

Example: abstract expression grammar

name **production**

```
BinEx:     Exp    ::= Exp BinOpr Exp
IdEx:      Exp    ::= Ident
PlusOpr:   BinOpr ::= '+'
MinusOpr:  BinOpr ::= '-'
TimesOpr:  BinOpr ::= '*'
DivOpr:    BinOpr ::= '/'
```

abstract program tree for `a * (b + c)`



symbol classes: `Exp = { Expr, Fact, Opd }, BinOpr = { AddOpr, MulOpr }`

Actions of the concrete syntax: **productions** of the abstract syntax to create tree nodes for **no action** at a concrete chain production: **no tree node** is created

Lecture Compiler I WS 2001/2002 / Slide 40

Objectives:

Illustrate comparison of concrete and abstract syntax

In the lecture:

- Repeat concepts of "GdP" (slide GdP-2.9):
- Compare grammars and trees.
- Actions create nodes of the abstract program tree.
- Symbol classes shrink node pairs that represent chain productions into one node

Suggested reading:

Kastens / Übersetzerbau, Section 4.1

Suggested reading:

slide GdP-2.9

Questions:

- Is this abstract grammar unambiguous?
- Why is that irrelevant?

Recursive descent parser

top-down (construction of the **derivation** tree), **predictive** method

Sytematic transformation of a context-free grammar into a set of functions:

non-terminal symbol X	function X
alternative productions for X	branches in the function body
decision set of production pi	decision for branch pi
non-terminal occurrence X ::= ... Y ...	function call Y()
terminal occurrence X ::= ... t ...	accept a token t an read the next token

Example:

```
p1: Stmt ::= Variable ':=' Expr  p2: Stmt ::= 'while' Expr 'do' Stmt
```

Function:

```
void Stmt ()
{  switch (CurrSymbol)
   {
case decision set for p1:           case decision set for p2:
   Variable();                         accept(whileSym);
   accept(assignSym);                  Expr();
   Expr();                             accept(doSym);
   break;                              Stmt();
                                       break;

   default: Fehlerbehandlung();
}  }
```


Lecture Compiler I WS 2001/2002 / Slide 41

Objectives:

Understand the construction schema

In the lecture:

Explanation of the method:

- Relate grammar constructs to function constructs.
- Each function plays the role of an acceptor for a symbol.
- accept function for reading and checking of the next token (scanner).
- Computation of decision sets on CI-42.
- Decision sets must be pairwise disjoint!

Suggested reading:

Kastens / Übersetzerbau, Section 4.2

Questions:

- A parser algorithm is based on a stack automaton. Where is the stack of a recursive descent parser? What corresponds to the states of the stack automaton?
- Where can actions be inserted into the functions to output production sequences in postfix or in prefix form?

Grammar conditions for recursive descent

A context-free grammar is **strong LL(1)**, if for any pair of productions that have the same symbol on their left-hand sides, the **decision sets are disjoint**:

productions:	A ::= u		A ::= v	
decision sets:	First (u Follow(A))	$\cap$	First (v Follow(A))	$= \emptyset$

First set and **follow set:**

First (u) := { t ∈ T \| v ∈ V* exists and a derivation u ⇒* t v } and ε ∈ First (u) if u ⇒* ε exists

Follow (A) := { t ∈ T \| u,v ∈ V* exist, A ∈ N and a derivation S ⇒* u A v such that t ∈ First (v) }

Example:

	production		decision set
p1:	Prog	::= Block #	begin
p2:	Block	::= begin Decls Stmts end	begin
p3:	Decls	::= Decl ; Decls	new
p4:	Decls	::=	Ident begin
p5:	Decls	::= new Ident	new
p6:	Stmts	::= Stmts ; Stmt	begin Ident
p7:	Stmts	::= Stmt	begin Ident
p8:	Stmt	::= Block	begin
p9:	Stmt	::= Ident := Ident	Ident

non-terminal X	First(X)	Follow(X)
Prog	begin	
Block	begin	# ; end
Decls	ε new	Ident begin
Decl	new	;
Stmts	begin Ident	; end
Stmt	begin Ident	; end

Lecture Compiler I WS 2001/2002 / Slide 42

Objectives:

Strong LL(1) can easily be checked

In the lecture:

- Explain the definitions using the example.
- First set: set of terminal symbols, which may begin some token sequence that is derivable from u.
- Follow set: set of terminal symbols, which may follow an A in some derivation.

Suggested reading:

Kastens / Übersetzerbau, Section 4.2, LL(k) conditions, computation of First sets and Follow sets

Questions:

The example grammar is not strong LL(1).

- Show where the condition is violated.
- Explain the reason for the violation.
- What would happen if we constructed a recursive descent parser although the condition is violated?

Grammar transformations for LL(1)

Consequences of strong LL(1) condition: A strong LL(1) grammar can not have

- **alternative productions that begin with the same symbols**
- **productions that are directly or indirectly left-recursive.**

Simple grammar transformations that keep the defined language invariant:

	non-LL(1) productions	transformed
• **left-factorization:** u, v, w ∈ V* X ∈ N does not occur in the original grammar	A ::= v u A ::= v w	A ::= v X X ::= u X ::= w
• **elimination of direct recursion :**	A ::= A u A ::= v	A ::= v X X ::= u X X ::=

EBNF constructs can avoid violation of strong LL(1) condition:

for example repetition of u: A ::= v (u)* w
additional condition: First(u) ∩ First(w Follow(A)) = ∅
branch in the function body: v **while (CurrToken in First(u)) { u }** w
correspondingly for EBNF constructs u^+, [u]

Lecture Compiler I WS 2001/2002 / Slide 43

Objectives:

Understand transformations and their need

In the lecture:

- Argue why strong LL(1) grammars can not have such productions.
- Show why the transformations remove those problems.
- Replacing left-recursion by right recursion would usually distort the structure.
- There are more general rules for indirect recursion.
- Show EBNF productions in recursive descent parsers.

Questions:

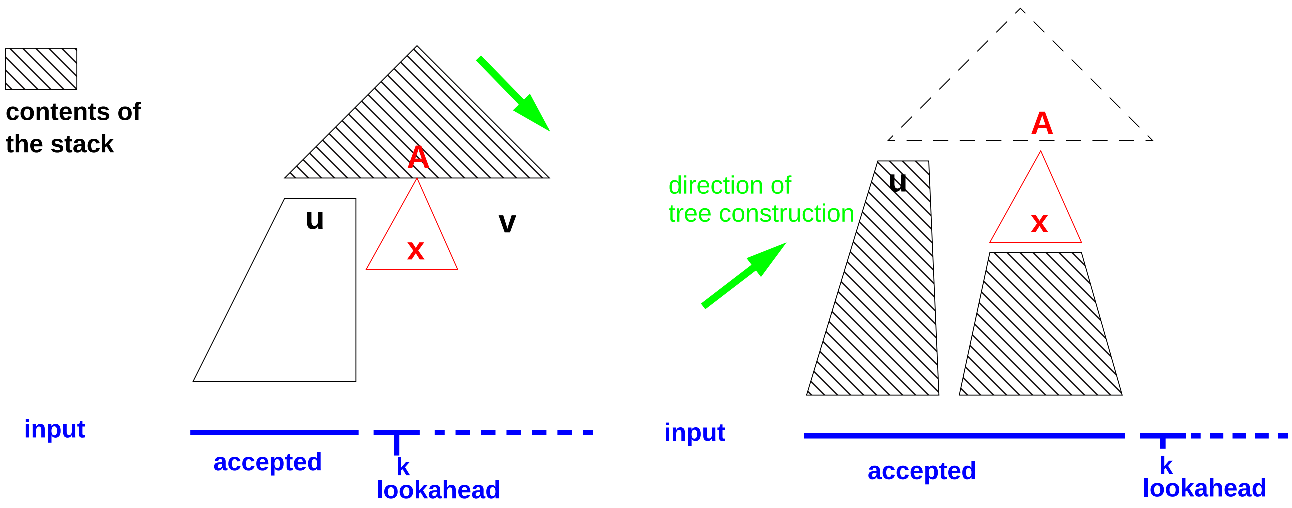
- Apply recursion elimination for expression grammars.
- Write a strong LL(1) expression grammar using EBNF.

Comparison: top-down vs. bottom-up

Information a stack automata has when it decides to apply production A ::= x :

top-down, predictive
leftmost derivation

bottom-up
rightmost derivation backwards



A bottom-up parser has seen more of the input when it decides to apply a production.

Consequence: **bottom-up** parsers and their grammar classes are more **powerful**.

Lecture Compiler I WS 2001/2002 / Slide 44

Objectives:

Understand the decision basis of the automata

In the lecture:

Explain the meaning of the graphics:

- role of the stack: contains states of the automaton,
- accepted input: will not be considered again,
- lookahead: the next k symbols, not yet accepted
- leftmost derivation: leftmost non-terminal is derived next; rightmost correspondingly,
- consequences for the direction of tree construction,

Abbreviations

- LL: (L)eft-to-right, (L)eftmost derivation,
- LR: (L)eft-to-right, (R)ightmost derivation,
- LALR: (L)ook(A)head LR

Suggested reading:

Kastens / Übersetzerbau, Section Text zu Abb. 4.2-1, 4.3-1

Questions:

Use the graphics to explain why a bottom-up parser without lookahead (k=0) is reasonable, but a top-down parser is not.

LR(1) automata

LR(k) grammars introduced 1965 by Donald Knuth; non-practical until subclasses were defined.

LR parsers construct the derivation tree **bottom-up,** a right-derivation backwards.

LR(k) grammar condition can not be checked directly, but
a context-free grammar is LR(k), iff the (canonical) **LR(k) automaton is deterministic**.

We consider only **1 token lookahead: LR(1).**

The **stacks** of LR(k) (and LL(k)) automata **contain states**.
The construction of LR and LL states is based on the notion of **items** (also called situations):

An **item** represents the progress of analysis with respect to one production:

[A ::= u . v R] z. B. **[B ::= (. D ; S) {#}]**

. position of analysis

R expected **right context,** i. e. a set of terminals which may follow after the application of the complete production.
(for general k: R contains terminal sequences not longer than k)

Reduce item:

[A ::= u v . R] z. B. **[B ::= (D ; S) . {#}]**

characterizes a reduction using this production if the next input token is in R.

Each **state** of an automaton represents **LL: one item** **LR: a set of items**

Lecture Compiler I WS 2001/2002 / Slide 45

Objectives:

Fundamental notions of LR automata

In the lecture:

Explain

- meaning of an item,
- lookahead in the input and right context in the automaton.
- There is no right context set in case of an LR(0) automaton.

Suggested reading:

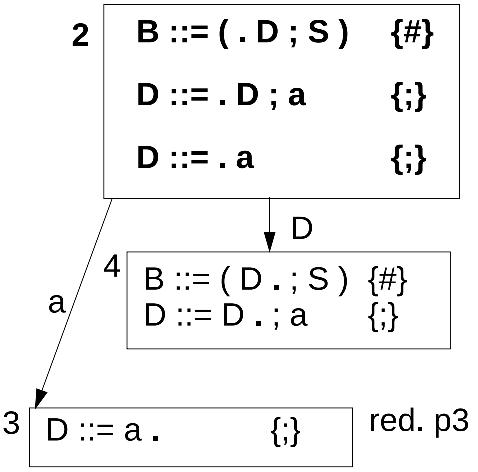
Kastens / Übersetzerbau, Section 4.3

Questions:

- What contains the right context set in case of a LR(3) automaton?

LR(1) states and operations

A **state of an LR automaton represents a set of items**
Each item represents a way in which analysis may proceed from that state.



A shift transition is made under
a **token read** from input or
a **non-terminal** symbol
obtained from a **preceding reduction.**
The state is pushed.

A **reduction** is made according to a reduce item.
n states are popped for a production of length n.

Operations:	**shift**	read and push the next state on the stack
	reduce	reduce with a certain production, pop n states from the stack
	error	error recognized, report it, recover
	stop	input accepted

Lecture Compiler I WS 2001/2002 / Slide 46

Objectives:

Understand LR(1) states and operations

In the lecture:

Explain

- Sets of items,
- shift transitions,
- reductions.

Suggested reading:

Kastens / Übersetzerbau, Section 4.3

Questions:

- Explain: A state is encoded by a number. A state represents complex information which is important for construction of the automaton.

Example for a LR(1) automaton

Grammar:

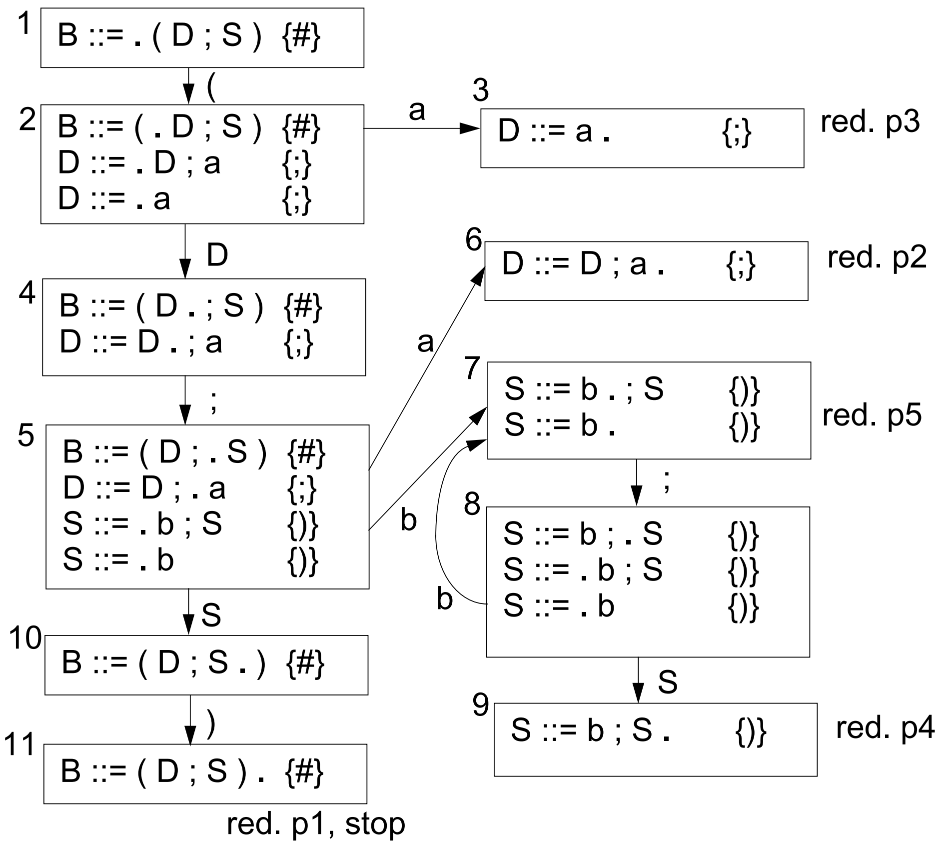
p1 B ::= (D ; S)

p2 D ::= D ; a

p3 D ::= a

p4 S ::= b ; S

p5 S ::= b



Lecture Compiler I WS 2001/2002 / Slide 47

Objectives:

Example for states, transitions, and automaton construction

In the lecture:

Use the example to explain

- the start state,
- the creation of new states,
- transitions into successor states,
- transitive closure of item set,
- push and pop of states,
- consequences of left-recursive and right-recursive productions,
- use of right context to decide upon a reduction,

erläutern.

Suggested reading:

Kastens / Übersetzerbau, Section 4.3

Questions:

- Describe the subgraphs for left-recursive and right-recursive productions. How do they differ?
- How does a LR(0) automaton decide upon reductions?

Construction of LR(1) automata

Create the start state; create transitions and states as long as new ones can be created.

Transitive closure is to be applied to each state:
If **[A ::= u . B v R]** is in state q,
with the analysis position before a non-terminal B,
then for each production **B ::= w**
[B ::= . w First (v R)]
has to be added to state q.

before:

B ::= (. D ; S) {#}

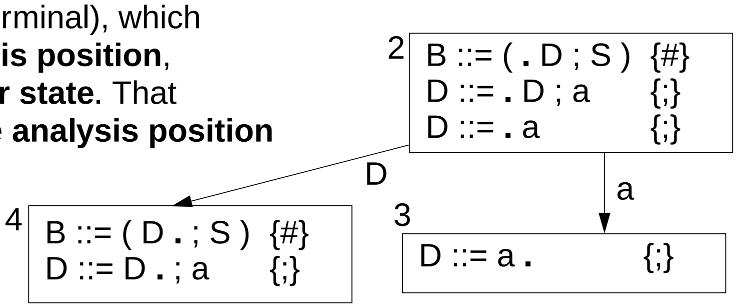
after:

2
B ::= (. D ; S) {#}
D ::= . D ; a {;}
D ::= . a {;}

Start state:
Closure of **[S ::= . u {#}]**
S ::= u is the **unique start production**,
is an **artificial end symbol** (eof)

1
B ::= . (D ; S) {#}

Successor states:
For each **symbol x** (terminal or non-terminal), which occurs in some items **after the analysis position**, a **transition** is created **to a successor state**. That contains a corresponding item with the **analysis position advanced behind the x** occurrence.



Lecture Compiler I WS 2001/2002 / Slide 48

Objectives:

Understand the method

In the lecture:

Explain using the example on CI-47:

- transitive closure,
- computation of the right context sets,
- relation between the items of a state and those of one of its successor

Suggested reading:

Kastens / Übersetzerbau, Section 4.3

Questions:

- Explain the role of the right context.
- Explain its computation.

Operations of the LR(1) automaton

shift x (terminal or non-terminal):
from current state q
under x into the**successor state q‘** ,
push q‘

reduce p:
apply production p B ::= u ,
pop as many states,
as there are **symbols in u**, from the
new current state make a **shift with B**

error:
the current state has no transition
under the next input token,
issue a **message** and **recover**

stop:
recuce start production,
see # in the input

Example:

stack	**input**	**reduction**
1	(a ; a ; b ; b) #	
1 2	a ; a ; b ; b) #	
1 2 3	; a ; b ; b) #	p3
1 2	; a ; b ; b) #	
1 2 4	; a ; b ; b) #	
1 2 4 5	a ; b ; b) #	
1 2 4 5 6	; b ; b) #	p2
1 2	; b ; b) #	
1 2 4	; b ; b) #	
1 2 4 5	b ; b) #	
1 2 4 5 7	; b) #	
1 2 4 5 7 8	b) #	
1 2 4 5 7 8 7	) #	p5
1 2 4 5 7 8	) #	
1 2 4 5 7 8 9	) #	p4
1 2 4 5	) #	
1 2 4 5 10	) #	
1 2 3 5 10 11	#	p1
1	#	

Lecture Compiler I WS 2001/2002 / Slide 49

Objectives:

Understand how the automaton works

In the lecture:

Explain operations

Questions:

- Why does the automaton behave differently on a-sequences and b-sequences?
- Which behaviour is better?

LR conflicts

An **LR(1) automaton that has conflicts is not deterministic**. Its **grammar is not LR(1)**; correspondingly defined for any other LR class.

2 kinds of conflicts:

reduce-reduce conflict:
A state contains two reduce items, the **right context sets** of which are **not disjoint**:

```
...
A ::= u .   R1
B ::= v .   R2
...
```

R1, R2 not disjoint

shift-reduce conflict:
A state contains a **shift item** with the **analysis position in front of a t** and a **reduce item with t in its right context set**.

```
...
A ::= u .t v   R1
B ::= w .      R2
...
```

t $\in$ **R2**

Lecture Compiler I WS 2001/2002 / Slide 50

Objectives:

Understand LR conflicts

In the lecture:

Explain: In certain situations the given input token t can not determine

- Reduce-reduce: which reduction is to be taken;
- Shift-reduce: whether the next token is to be shifted, a reduction is to be made.

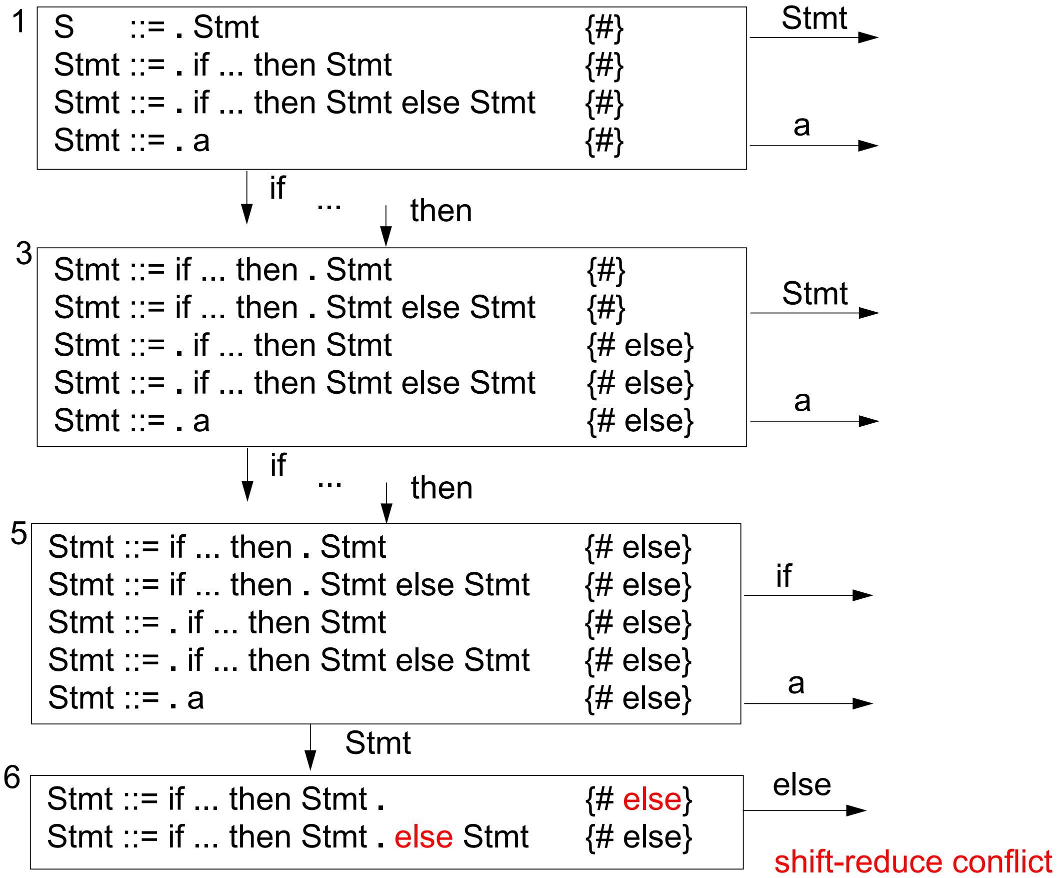
Suggested reading:

Kastens / Übersetzerbau, Section 4.3

Questions:

- Why can a shift-shift conflict not exist?
- In LR(0) automata items do not have a right-context set. Explain why a state with a reduce item may not contain any other item.

Shift-reduce conflict for „dangling else“ ambiguity



Lecture Compiler I WS 2001/2002 / Slide 51

Objectives:

See a conflict in an automaton

In the lecture:

Explain

- the construction
- a solution of the conflict: The automaton can be modified such that in state 6, if an else is the next input token, it is shifted rather than a reduction is made. In that case the ambiguity is solved such that the else part is bound to the inner if. That is the structure required in Pascal and C. Some parser generators can be instructed to resolve conflicts in this way.

Suggested reading:

Kastens / Übersetzerbau, Section 4.3

Simplified LR grammar classes

LR(1):

too many states for practical use
Reason: right-contexts distinguish many states
Strategy: simplify right-contexts sets,
fewer states, grammar classes are less powerful

LR(0):

all items **without right-context**
Consequence: reduce items only in singleton sets

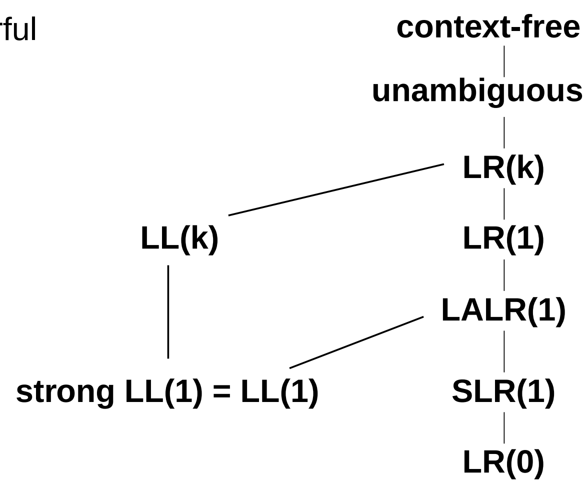
SLR(1):

LR(0) states; in reduce items
use larger right-context sets for decision:
[A ::= u . Follow (A)]

LALR(1):

identify LR(1) states if their items differ only
in their right-context sets, unite the sets for those items;
yields the states of the **LR(0) automaton**
augmented by the "exact“ LR(1) right-context.
State-of-the-art parser generators accept LALR(1)

Grammar hierarchy:
(strict inclusions)



Lecture Compiler I WS 2001/2002 / Slide 52

Objectives:

Understand relations between LR classes

In the lecture:

Explain:

- LALR(1), SLR(1), LR(0) automata have the same number of states,
- compare their states,
- discuss the grammar classes for the example on slide CI-47.

Suggested reading:

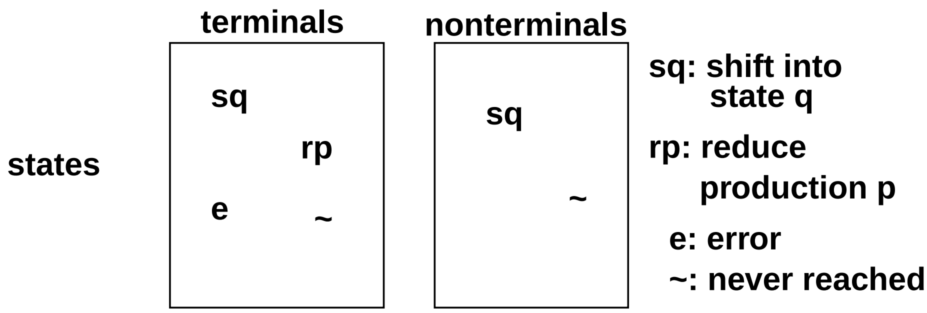
Kastens / Übersetzerbau, Section 4.3

Questions:

- Assume that the LALR(1) contruction for a given grammar yields conflicts. Classify the potential reasons using the LR hierarchy.

Implementation of LR automata

Table-driven:



Compress tables:

- **merge rows or columns** that differ only in irrelevant entries; method: graph coloring
- extract a **separate error matrix** (bit matrix); increases the chances for merging
- **normalize the values of rows or columns**; yields smaller domain; supports merging
- **eliminate LR(0) reduce states**; new operation in predecessor state: **shift-reduce** eliminates about 30% of the states in practical cases

About 10-20% of the original table sizes can be achieved!

Directly programmed LR-automata are possible - but usually too large.

Lecture Compiler I WS 2001/2002 / Slide 53

Objectives:

Implementation of LR tables

In the lecture:

Explanation of

- pair of tables and their entries,
- unreachable entries,
- compression techniques, derived from general table compression,
- Singleton reduction states yield an effective optimization.

Questions:

- Why are there no error entries in the nonterminal part?
- Why are there unreachable entries?
- Why does a parser need a shift-reduce operation if the optimization of LR(0)-reduction states is applied?

Error handling: general criteria

- **recognize error as early as possible**
  LL and LR can do that
- **report the symptom in terms of the source text**
- **continue parsing short after the error position**
- **avoid avalanche errors**
- **build a tree that has a correct structure**
- **do not backtrack, do not undo actions**
- **no runtime penalty for correct programs**

Lecture Compiler I WS 2001/2002 / Slide 54

Objectives:

Accept strong requirements

In the lecture:

- The reasons for and the consequences of the requirements are discussed.
- Some of the requirements hold for error handling in general - not only that of the syntactic analysis.

Error position

Error recovery: Means that are taken by the parser after recognition of a syntactic error in order to continue parsing

Correct prefix: The token sequence $w \in T^*$ is a correct prefix in the language L(G), if there is an $u \in T^*$ such that **$w\ u \in L(G)$**; i. e. w can be extended to a sentence in L(G).

Error position: t is the (first) error position in the **input w t x** , where $t \in T$ and $w, x \in T^*$, if **w is a correct prefix** in L(G) and **w t is not a correct prefix**.

Example:

```
int compute (int i) { a = i * / c; return i;}
```

$\underbrace{\texttt{int compute (int i) \{ a = i *}}_{w}\ \underbrace{\texttt{/}}_{t}$

LL and LR parsers recognize an error at the error position; they can not accept t in the current state.

Lecture Compiler I WS 2001/2002 / Slide 55

Objectives:

Error position from the view of the parser

In the lecture:

Explain the notions with respect to parser actions using the examples.

Questions:

Assume the programmer omitted an opening parenthesis.

- Where is the error position?
- What is the symptom the parser recognizes?

Error recovery

Continuation point:
The token d at or behind the error position t such that **parsing of the input continues at d**.

Error repair
with respect to a consistent derivation - regardless the intension of the programmer!

Let the input be w t x with the error position at t and let w t x = w y d z,
then the recovery (conceptually) **deletes y** and **inserts v**,
such that **w v d is a correct prefix** in L(G), with d ∈ T and w, y, v, z ∈ T*.

Examples:

```
    w     y d  z
a = i * / c;...
a = i *   c;...
```

delete `/`

```
    w     yd  z
a = i * / c;...
a = i *e/ c;...
```

insert error id. e

```
    w      y d z
a = i * / c;...
a = i * e  ;...
```

delete `/ c`
and **insert** error id. **e**

Lecture Compiler I WS 2001/2002 / Slide 56

Objectives:

Understand error recovery

In the lecture:

Explain the notions with respect to parser actions using the examples.

Questions:

Assume the programmer omitted an opening parenthesis.

- What could be a suitable repair?

Recovery method: simulated continuation

Problem: Determine a continuation point close to the error position and reach it.

Idea: Use parse stack to determine a set of tokens as potential continuation points.

Steps of the method:

1. **Save the contents of the parse stack** when an error is recognized. Skip the error token.
2. **Compute a set D $\subseteq$ T of tokens that may be used as continuation point** (**anchor set**)
   Let a modified parser run to completion:
   Instead of reading a token from input it is inserted into D; (modification given below)
3. **Find a continuation point d**: Skip input tokens until a token of D is found.
4. **Reach the continuation point d**:
   Restore the saved parser stack as the current stack.
   Perform dedicated transitions until d is acceptable.
   Instead of reading tokens (conceptually) insert tokens. Thus a correct prefix is constructed.
5. **Continue normal parsing**.

Augment parser construction for steps 2 and 4:
For each parser state select a transition and its token,
such that the parser empties its stack and terminates as fast as possible.
This selection can be **generated automatically**.
The quality of the recovery can be improved by influence on the computation of D.

Lecture Compiler I WS 2001/2002 / Slide 57

Objectives:

Error recovery can be generated

In the lecture:

- Explain the idea and the steps of the method.
- The method yields a correct parse for any input!
- Other, less powerful methods determine sets D statically at parser construction time, e. g. semicolon and curly bracket for errors in statements.

Questions:

- How does this method fit to the general requirements for error handling?

Parser generators

PGS	Univ. Karlsruhe; in Eli	**LALR(1), table-driven**
Cola	Univ. Paderborn; in Eli	**LALR(1), optional: table-driven or directly programmed**
Lalr	Univ. / GMD Karlsruhe	**LALR(1), table-driven**
Yacc	Unix tool	**LALR(1), table-driven**
Bison	Gnu	**LALR(1), table-driven**
Llgen	Amsterdam Compiler Kit	**LL(1), recursive descent**
Deer	Univ. Colorado, Bouder	**LL(1), recursive descent**

Form of grammar specification:

EBNF: Cola, PGS, Lalr; **BNF**: Yacc, Bison

Error recovery:

simulated continuation, automatically generated:	Cola, PGS, Lalr
error productions, hand-specified:	Yacc, Bison

Actions:

statements in the implementation language

at the end of productions:	Yacc, Bison
anywhere in productions:	Cola, PGS, Lalr

Conflict resolution:

modification of states (reduce if ...)	Cola, PGS, Lalr
order of productions:	Yacc, Bison
rules for precedence and associativity:	Yacc, Bison

Implementation languages:

C: Cola, Yacc, Bison **C, Pascal, Modula-2, Ada**: PGS, Lalr

Lecture Compiler I WS 2001/2002 / Slide 58

Objectives:

Overview over parser generators

In the lecture:

- Explain the significance of properties

Suggested reading:

Kastens / Übersetzerbau, Section 4.5

Design of concrete grammars

Objectives

The concrete grammars for **parsing**

- is parsable **-** fulfills the **grammar condition** of the chosen parser generator;
- specifies the **intended language -** or a small super set of it;
- is provable related to the **documented grammar**;
- can be **mapped to** a suitable **abstract grammar**.

Lecture Compiler I WS 2001/2002 / Slide 59

Objectives:

Guiding objectives

In the lecture:

The objectives are explained.

Grammar design for an existing language

- Take the grammar of the **language specification literally**.
- Only **conservative modifications** for parsability or for mapping to abstract syntax.
- **Describe any modification**.
  (see ANSI C Specification in the Eli system description
  http://www.uni-paderborn.de/fachbereich/AG/agkastens/eli/examples/eli_cE.html)

  - **Java** language specification (1996):
    Specification grammar is not LALR(1).
    5 problems are described and how to solve them.
  - **Ada** language specification (1983):
    Specification grammar is LALR(1)
    - requirement of the language competition
  - **ANSI C, C++**:
    several ambiguities and LALR(1) conflicts, e.g.
    „**dangling else**",
    „**typedef problem**":
    ```
    A (*B);
    ```
    is a declaration of variable **B**, if **A** is a type name,
    otherwise it is a call of function **A**

Lecture Compiler I WS 2001/2002 / Slide 60

Objectives:

Avoid document modifications

In the lecture:

- Explain the conservative strategy.
- Java gives a solution for the dangling else problem.
- Explain the typedef problem.

Grammar design together with language design

Read grammars before writing a new grammar.

Apply **grammar patterns systematically** (cf. GdP-2.5, GdP-2.8)

- repetitions
- optional constructs
- precedence, associativity of operators

Syntactic structure should reflect semantic structure:

E. g. a range in the sense of scope rules should be represented by a single subtree of the derivation tree (of the abstract tree).

Violated in Pascal:

functionDeclaration ::= functionHeading block
functionHeading ::= ‘function‘ identifier formalParameters ‘:‘ resultType ‘;‘

formalParameters together with block form a range,
but identifier does not belong to it

Lecture Compiler I WS 2001/2002 / Slide 61

Objectives:

Grammar design rules

In the lecture:

- Refer to GdP slides.
- Explain semantic structure.
- Show violation of the example.

Syntactic restrictions versus semantic conditions

Express a restriction **syntactically** only if
it can be **completely covered with reasonable complexity**:

- **Restriction can not be decided syntactically**:
  e.g. type check in expressions:
  BoolExpression ::= IntExpression ‘<‘ IntExpression

- **Restriction can not always be decided syntactically**:
  e. g. disallow array type to be used as function result
  Type ::= ArrayType | NonArrayType | Identifier
  ResultType ::= NonArrayType
  If a type identifier may specify an array type,
  a semantic condition is needed, anyhow

- **Syntactic restriction is unreasonable complex**:
  e. g. distinction of compile-time expressions from ordinary expressions requires duplication of the expression syntax.

Lecture Compiler I WS 2001/2002 / Slide 62

Objectives:

How to express restrictions

In the lecture:

- Examples are explained.
- Semantic conditions are formulated with attribute grammar concepts, see next chapter.

Assignments:

Discuss further examples for restrictions.

Reasons of LALR(1) conflicts

Grammar condition does not hold:

context-free		
unambiguous	**ambiguous**	most cases
LR(k)	**unbounded lookahead** needed	
LR(1)	**fixed length lookahead > 1** needed	
LALR(1)	**merge of LR(1) states** introduces **conflicts**	rare cases

LALR(1) parser generator can not distinguish these cases.

Lecture Compiler I WS 2001/2002 / Slide 63

Objectives:

Distinguish cases

In the lecture:

The cases are explained.

Eliminate ambiguities

unite syntactic constructs - distinguish them semantically

Examples:

- Java:

```
ClassOrInterfaceType   ::= ClassType | InterfaceType
InterfaceType          ::= TypeName
ClassType              ::= TypeName
```

replace first production by

```
ClassOrInterfaceType   ::= TypeName
```

semantic analysis distinguishes between class type and interface type

- Pascal:

```
factor               ::= variable | ... | functionDesignator
variable             ::= entireVariable | ...
entireVariable       ::= variableIdentifier
variableIdentifier   ::= identifier                                  (**)
functionDesignator   ::= functionIdentifier                          (*)
                     |   functionIdentifer '(' actualParameters ')'
functionIdentifier   ::= identifier
```

eliminate marked (*) alternative

semantic analysis checks whether (**) is a function identifier

Lecture Compiler I WS 2001/2002 / Slide 64

Objectives:

Typical ambiguities

In the lecture:

- Same notation with different meanings;
- ambiguous, if they occur in the same context.
- Conflicting notations may be separated by several levels of productions (Pascal example)

Questions:

Unbounded lookahead

The decision for a **reduction** is determined by a **distinguishing token** that may be **arbitrarily far to the right**:

Example, forward declarations as could have been defined in Pascal:

functionDeclaration ::=
‘function‘ forwardIdent formalParameters ‘:‘ resultType ‘;‘ ‘forward‘

| ‘function‘ functionIdent formalParameters ‘:‘ resultType ‘;‘ block

The distinction between forwardIdent and functionIdent would require to see the forward or the begin token.

Replace forwardIdent and functionIdent by the same nonterminal; distinguish semantically.

Lecture Compiler I WS 2001/2002 / Slide 65

Objectives:

Typical situation

In the lecture:

Explain the problem and the solution using the example

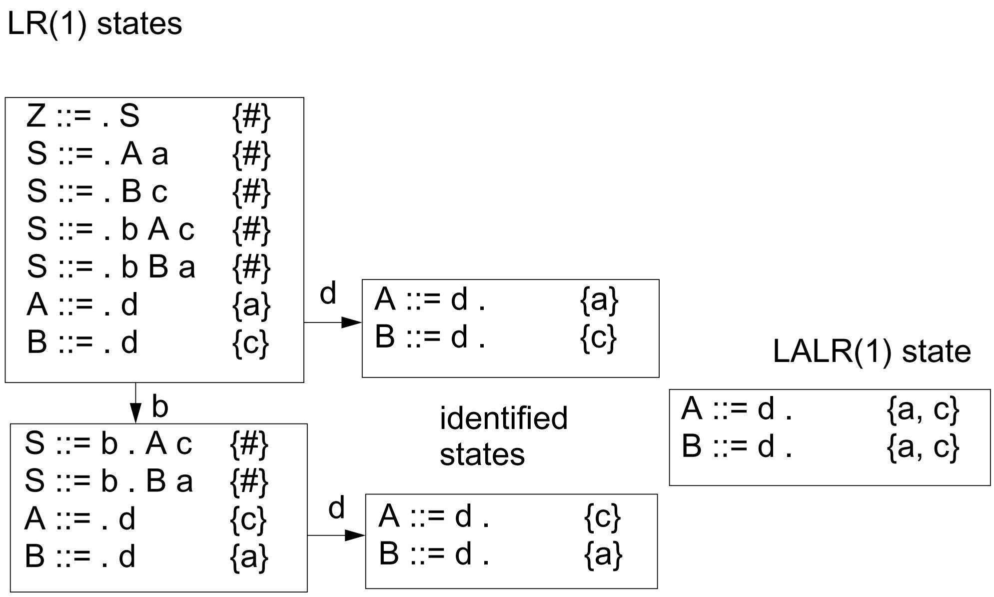
Questions:

LR(1) but not LALR(1)

Identification of LR(1) states causes non-disjoint right-context sets.

Artificial example:

Grammar:
Z ::= S
S ::= A a
S ::= B c
S ::= b A c
S ::= b B a
A ::= d.
B ::= d.



Avoid the distinction between A and B - at least in one of the contexts.

Lecture Compiler I WS 2001/2002 / Slide 66

Objectives:

Understand source of conflicts

In the lecture:

Explain grammar the pattern, and why identification of states causes a conflict.

4. Semantic analysis and transformation

Input: **abstract program tree**

Tasks:	**Compiler module:**
name analysis	environment module
properties of program entities	definition module
type analysis, operator identification	signature module
transformation	tree generator

Output: **target tree, intermediate code, target program in case of source-to-source**

Standard implementations and generators for compiler modules

Operations of the compiler modules are called at nodes of the abstract program tree

Model: dependent computations in trees

Specification: **attribute grammars**

generated: tree walking algorithm that calls operations
in specified contexts and in an admissable order

Lecture Compiler I WS 2001/2002 / Slide 67

Objectives:

Tasks and methods of semantic analysis

In the lecture:

Explanation of the

- tasks,
- compiler modules,
- principle of dependent computations in trees.

Suggested reading:

Kastens / Übersetzerbau, Section Introduction of Ch. 5 and 6

4.1 Attribute grammars

Attribute grammar (AG) specifies **dependent computations in the abstract program tree**
declarative: explicit dependencies only; a suitable order of execution is computed

Computations solve the tasks of semantic analysis and transformation

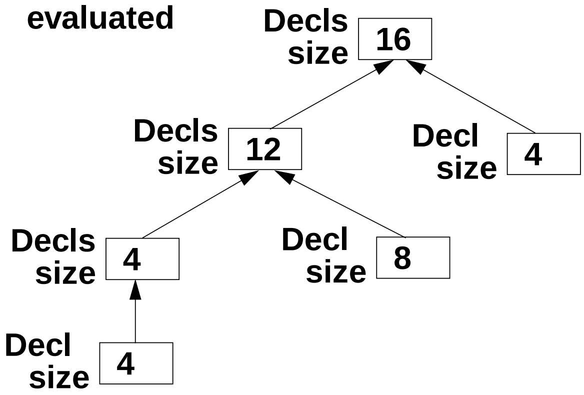
Generator produces **a plan for tree walks**
that execute calls of the computations,
such that the specified dependencies are obeyed,
computed values are propagated through the tree

Result: attribute evaluator; applicable for any tree specified by the AG

Example: **attribute grammar**

```
RULE Decls ::= Decls Decl COMPUTE
    Decls[1].size =
        Add (Decls[2].size, Decl.size);
END;
RULE Decls ::= Decl COMPUTE
    Decls.size = Decl.size;
END;
RULE Decl ::= Type Name COMPUTE
    Decl.size = ...;
END;
```

tree with dependent attributes evaluated



Lecture Compiler I WS 2001/2002 / Slide 68

Objectives:

Get an informal idea of attribute grammars

In the lecture:

Explain computations in tree contexts using the example

Suggested reading:

Kastens / Übersetzerbau, Section 5, 5.1

Questions:

Why is it useful NOT to specify an evaluation order explicitly?

Basic concepts of attribute grammars

An AG specifies computations in tree:

expressed by **computations associated to productions of the abstract syntax**

```
RULE p: Y ::= u COMPUTE f(...); g(...); END;
```

computations f(...) and g(...) are executed in every tree context of type p

An AG specifies dependencies between computations:

expressed by **attributes associated to grammar symbols**

```
RULE p: X ::= u Y v COMPUTE      X.b =   f(Y.a);
                                 Y.a =   g(...);
END;                        post-condition   pre-condition
```

`f(Y.a)` uses the result of `g(...)`; hence `Y.a=g(...)` will be executed before `f(Y.a)`

dependent computations in adjacent contexts:

```
RULE r: X ::= v Y w COMPUTE X.b = f(Y.a); END;
RULE p: Y ::= u    COMPUTE Y.a = g(...); END;
```

attributes may specify dependencies without propagating any value:

```
X.GotType = ResetTypeOf(...);
Y.Type = GetTypeOf(...) <- X.GotType;
```

`ResetTypeOf` will be called before `GetTypeOf`

Lecture Compiler I WS 2001/2002 / Slide 69

Objectives:

Get a basic understanding of AGs

In the lecture:

Explain

- the AG notation,
- dependent computations,
- adjacent contexts in trees

Suggested reading:

Kastens / Übersetzerbau, Section 5, 5.1

Assignments:

- Read and modify examples in Lido notation to introduce AGs

Definition of attribute grammars

An **attribute grammar** is defined by

a **context-free grammar G**, (abstract syntax, tree grammar)

for each **symbol X** of G a set of **attributes A(X)**, written X.a if a $\in$ A(X)

for each **production (rule) p** of G a set of **computations** of one of the forms

X.a = f (... Y.b ...)   or   g (... Y.b ...)   where X and Y occur in p

Consistency and completeness of an AG:

Each A(X) is partitioned into two disjoint subsets: AI(X) and AS(X)

AI(X): **inherited attributes** are computed in rules p where X is on the **right**-hand side of p

AS(X): **synthesized attributes** are computed in rules p where X is on the **left**-hand side of p

Each rule p: X ::= ... Y ... has exactly one computation
for all attributes of AS(X), and
for all attributes of AI(Y), for all symbol occurrences on the right-hand side of p

Lecture Compiler I WS 2001/2002 / Slide 69a

Objectives:

Formal view on AGs

In the lecture:

The completeness and consistency rules are explained using the example of CI-69b

AG Example: Compute expression values

The AG specifies: The value of an expression is computed and printed:

```
ATTR value: int;

RULE: Root ::=  Expr  COMPUTE
  printf ("value is %d\n",
          Expr.value);
END;

TERM Number: int;

RULE: Expr ::= Number COMPUTE
  Expr.value = Number;
END;

RULE: Expr ::= Expr Opr Expr
COMPUTE
  Expr[1].value = Opr.value;
  Opr.left  = Expr[2].value;
  Opr.right = Expr[3].value;
END;
```

```
SYMBOL Opr: left, right: int;

RULE: Opr ::=  '+'  COMPUTE
  Opr.value  =
     ADD (Opr.left, Opr.right);
END;

RULE: Opr ::=  '*'  COMPUTE
  Opr.value =
     MUL (Opr.left, Opr.right);
END;
```


Lecture Compiler I WS 2001/2002 / Slide 69b

Objectives:

Exercise formal definition

In the lecture:

- Show synthesized, inherited attributes.
- Check consistency and completeness.

Questions:

- Add a computation such that a pair of sets AI(X), AS(X) is no longer disjoint.
- Add a computation such that the AG is inconsistent.
- Which computations can be omitted whithout making the AG incomplete?
- What would the effect be if the order of the three computations on the bottom left of the slide was altered?

AG Binary numbers

Attributes:

`L.v, B.v`	value
`L.lg`	number of digits in the sequence L
`L.s, B.s`	scaling of B or the least significant digit of L

```
RULE p1:   D ::= L '.' L   COMPUTE
   D.v = ADD (L[1].v, L[2].v);
   L[1].s = 0;
   L[2].s = NEG (L[2].lg);
END;
RULE p2:   L ::= L B       COMPUTE
   L[1].v = ADD (L[2].v, B.v);
   B.s = L[1].s;
   L[2].s = ADD (L[1].s, 1);
   L[1].lg = ADD (L[2].lg, 1);
END;
RULE p3:   L ::= B         COMPUTE
   L.v = B.v;
   B.s = L.s;
   L.lg = 1;
END;
RULE p4:   B ::= '0'       COMPUTE
   B.v = 0;
END;
RULE p5:   B ::= '1'       COMPUTE
   B.v = Power2 (B.s);
END;
```


Lecture Compiler I WS 2001/2002 / Slide 70

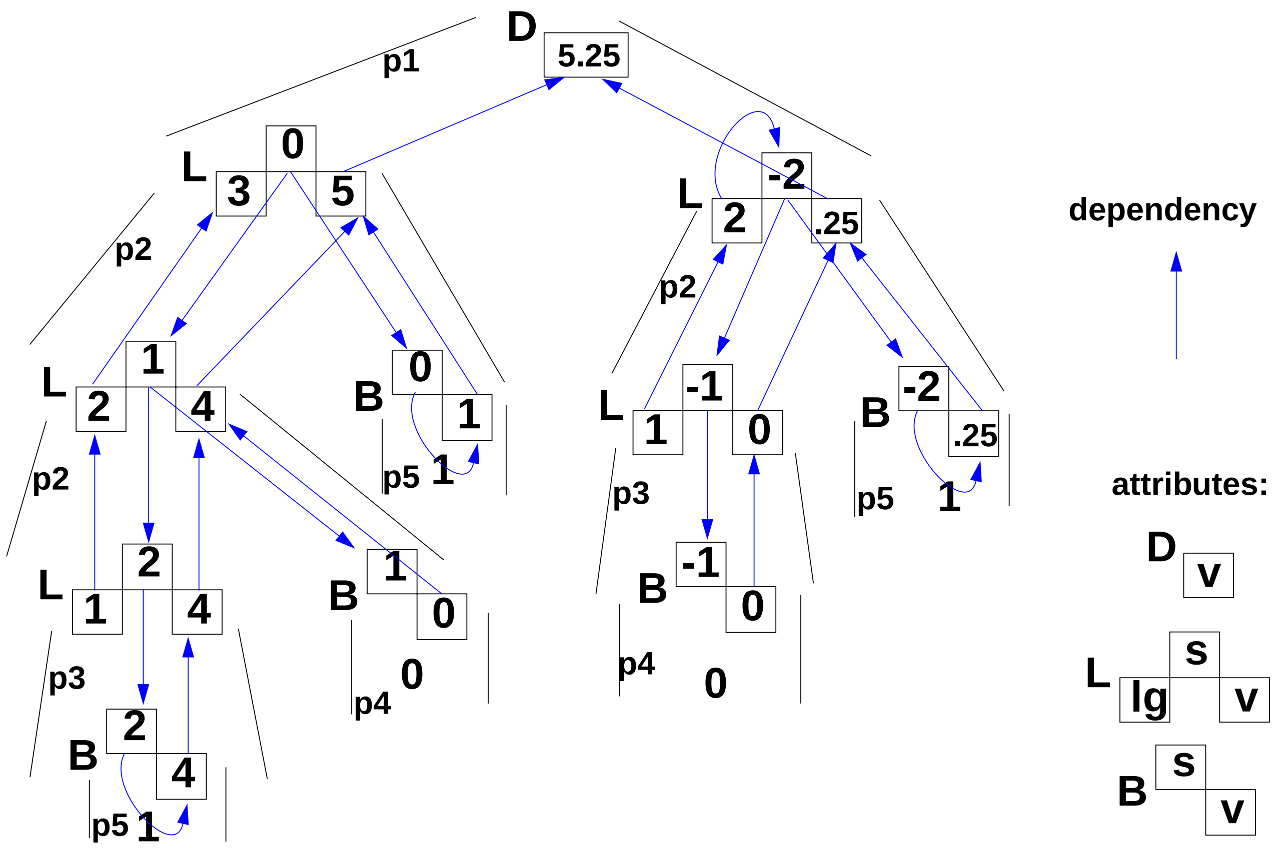
Objectives:

A complete example for an AG

In the lecture:

- Explain the task.
- Explain the role of the attributes.
- Explain the computations in tree contexts.
- Show a tree with attributes and dependencies (CI-71)

An attributed tree for AG Binary numbers



Lecture Compiler I WS 2001/2002 / Slide 71

Objectives:

An attributed tree

In the lecture:

- Show a tree with attributes.
- Show tree contexts specified by grammar rules.
- Relate the dependencies to computations.
- Evaluate the attributes.

Questions:

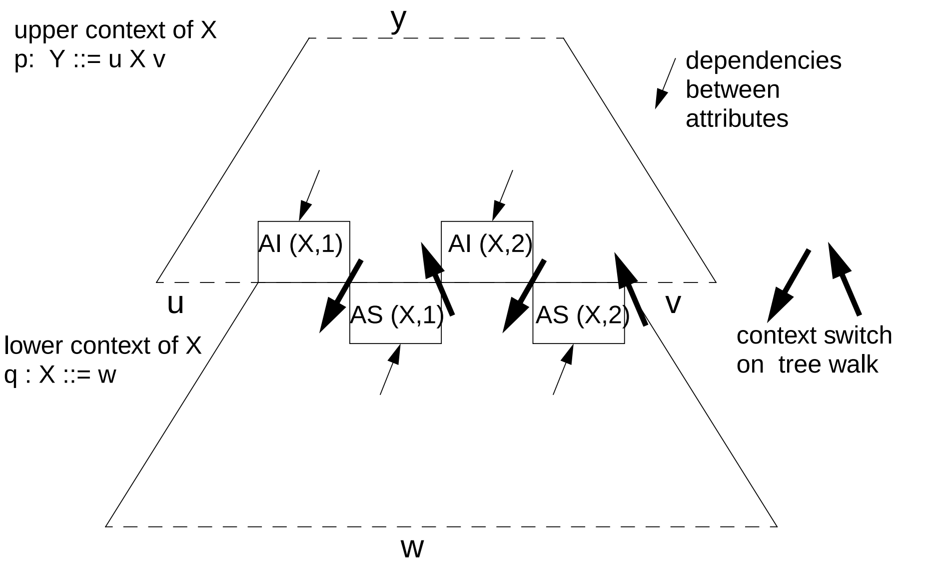
- Some attributes do not have an incoming arc. Why?
- Show that the attribues of each L node can be evaluated in the order lg, s, v.

Dependency analysis for AGs

2 disjoint sets of attributes for each symbol X:

AI (X) : inherited (dt. erworben), **computed in upper contexts** of X

AS (X): synthesized (dt. abgeleitet), **computed in lower contexts** of X.



Objective: Partition of attribute sets, such that

AI (X, i) is computed **before the i-th visit** of X

AS (X, i) is computed **during the i-th visit** of X

Necessary precondition for the existence of such a partition:
No node in any tree has direct or indirect dependencies that contradict the evaluation order of the sequence of sets: AI (X, 1), AS (X, 1), ..., AI (X, k), AS (X, k)

Lecture Compiler I WS 2001/2002 / Slide 72

Objectives:

Understand the concept of attribute partitions

In the lecture:

Explain the concepts

- sets of synthesized and inherited attributes,
- upper and lower context,
- context switch,
- attribute partitions: sequence of disjoint sets which alternate between synthesized and inherited

Suggested reading:

Kastens / Übersetzerbau, Section 5.2

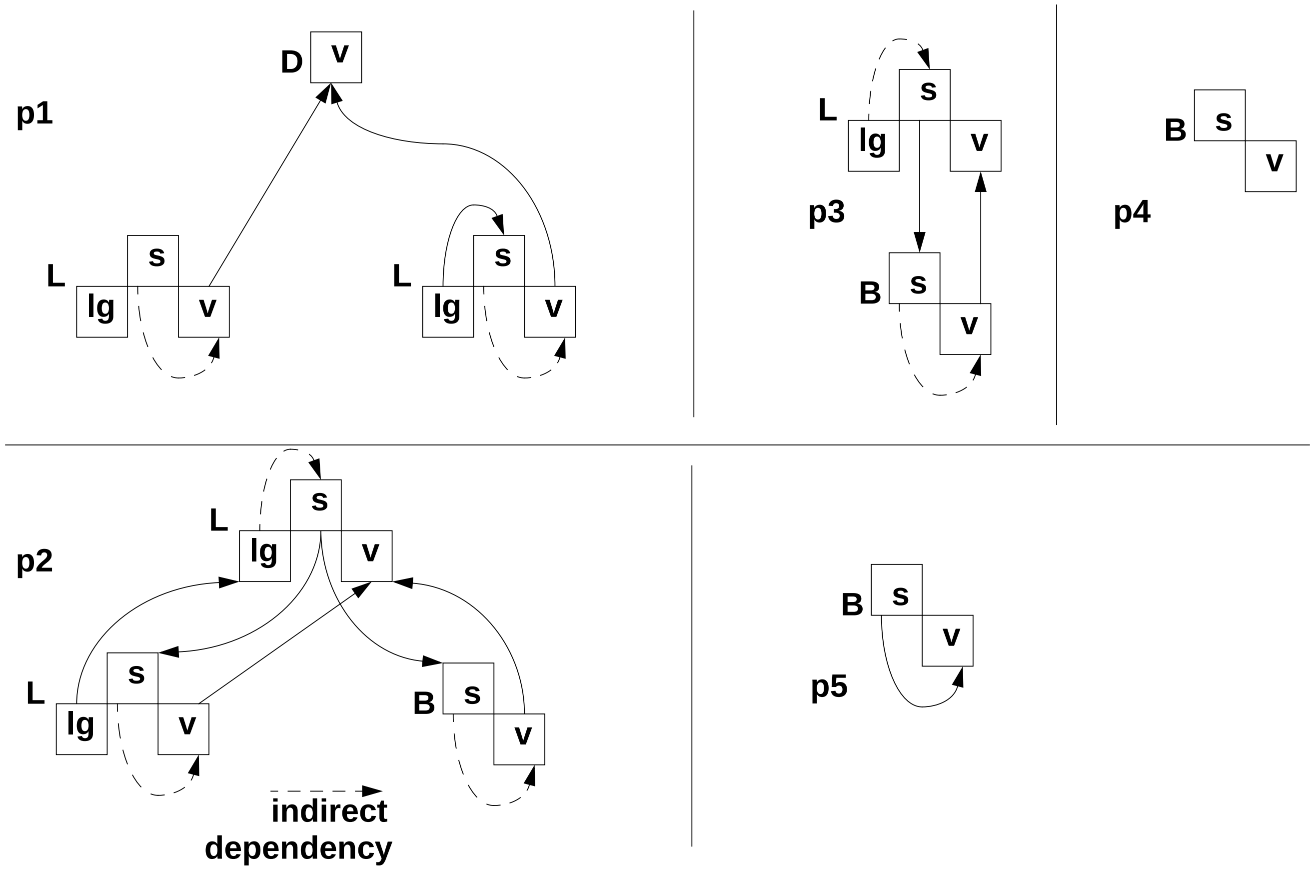
Assignments:

Construct AGs that are as simple as possible and each exhibits one of the following properties:

- There are some tree that have a dependency cycle, other trees don’t.
- The cycles extend over more than one context.
- There is an X that has a partition with k=2 but not with k=1.
- There is no partition, although no tree exists that has a cycle. (caution: difficult puzzle!)

(Exercise 22)

Dependency graphs for AG Binary numbers



Lecture Compiler I WS 2001/2002 / Slide 73

Objectives:

Represent dependencies

In the lecture:

- graph representation of dependencies that are specified by computations,
- compose the graphs to yield a tree with dependencies,
- explain indirect dependencies
- Use the graphs as an example for partitions (CI-72)
- Use the graphs as an example for LAG(k) algorithm (CI-77)

Construction of attribute evaluators

For a given attribute grammar an attribute evaluator is constructed:

- It is **applicable to any tree** that obeys the abstract syntax specified in the rules of the AG.
- It performs a **tree walk** and
  executes computations when visiting a context for which they are specified.
- The execution order obeys the **attribute dependencies**.

Pass-oriented strategies for the tree walk:	**AG class**
k times **depth-first left-to-right**	**LAG (k)**
k times depth-first **alternatingly left-to-right / right-to left**	**AAG (k)**
once **bottom-up**	**SAG**

The attribute dependencies of the AG are checked
whether the desired pass-oriented strategy is applicable; see LAG(k) algorithm.

non-pass-oriented strategies:
visit-sequences: **OAG**
an individual plan for each rule of the abstract syntax

Generator fits the plans to the dependencies.

Lecture Compiler I WS 2001/2002 / Slide 74

Objectives:

Tree walk strategiees

In the lecture:

- Show the relation between tree walk strategies and attribute dependencies.

Suggested reading:

Kastens / Übersetzerbau, Section 5, 5.1

Questions:

A grammar class is more powerful if it covers AGs with more complex dependencies.

- Arrange the AG classes in a hierarchy according to that property.

Visit-sequences

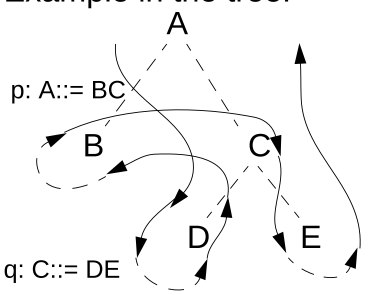
A **visit-sequence** (dt. Besuchssequenz) vs_p **for each production** of the tree grammar:

p: X_o ::= X_1 ... X_i ... X_n

A visit-sequence is a **sequence of operations**:

↓ i, j	j-th **visit of the i-th subtree**
↑ j	j-th r**eturn to the ancestor** node
$eval_c$	execution of a **computation** c associated to p

Example in the tree:

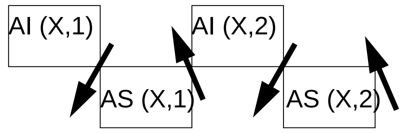


visit-sequences

vs_p: ... ↓C,1 ...↓B,1 ...↓C,2 ...↑1

vs_q: ... ↓D,1 ... ↑1 ... ↓E,1 ... ↑2

attribute partitions
guaranty
correct interleaving:



Implementation:

one procedure for each section of a visit-sequence upto ↑

a call with a switch over applicable productions for ↓

Lecture Compiler I WS 2001/2002 / Slide 75

Objectives:

Understand the concept of visit-sequences

In the lecture:

Explain

- context switch,
- interleaving of visit-sequences for adjacent contexts,
- partitions are "interfaces" for context switches,
- implementation using procedures and calls

Suggested reading:

Kastens / Übersetzerbau, Section 5.2.2

Assignments:

- Construct a set of visit-sequences for a small tree grammar, such that the tree walk solves a certain task.
- Find the description of the design pattern "Visitor" and relate it to visit-sequences.

Questions:

- Describe visit-sequences which let trees being traversed twice depth-first left-to-right.

Visit-sequences for the AG Binary numbers

vs_{p1}: D ::= L '.' L

↓L[1],1; L[1].s=0; ↓L[1],2; ↓L[2],1; L[2].s=NEG(L[2].lg);

↓L[2],2; D.v=ADD(L[1].v, L[2].v); ↑1

vs_{p2}: L ::= L B

↓L[2],1; L[1].lg=ADD(L[2].lg,1); ↑1

L[2].s=ADD(L[1].s,1); ↓L[2],2; B.s=L[1].s; ↓B,1; L[1].v=ADD(L[2].v, B.v); ↑2

vs_{p3}: L ::= B

L.lg=1; ↑1; B.s=L.s; ↓B,1; L.v=B.v; ↑2

vs_{p4}: B ::= '0'

B.v=0; ↑1

vs_{p5}: B ::= '1'

B.v=Power2(B.s); ↑1

Implementation:

Procedure vs<i><p> for each section of a vs_p to a ↑i
a call with a switch over alternative rules for ↓X,i

Lecture Compiler I WS 2001/2002 / Slide 76

Objectives:

Example for visit-sequences (CI-75)

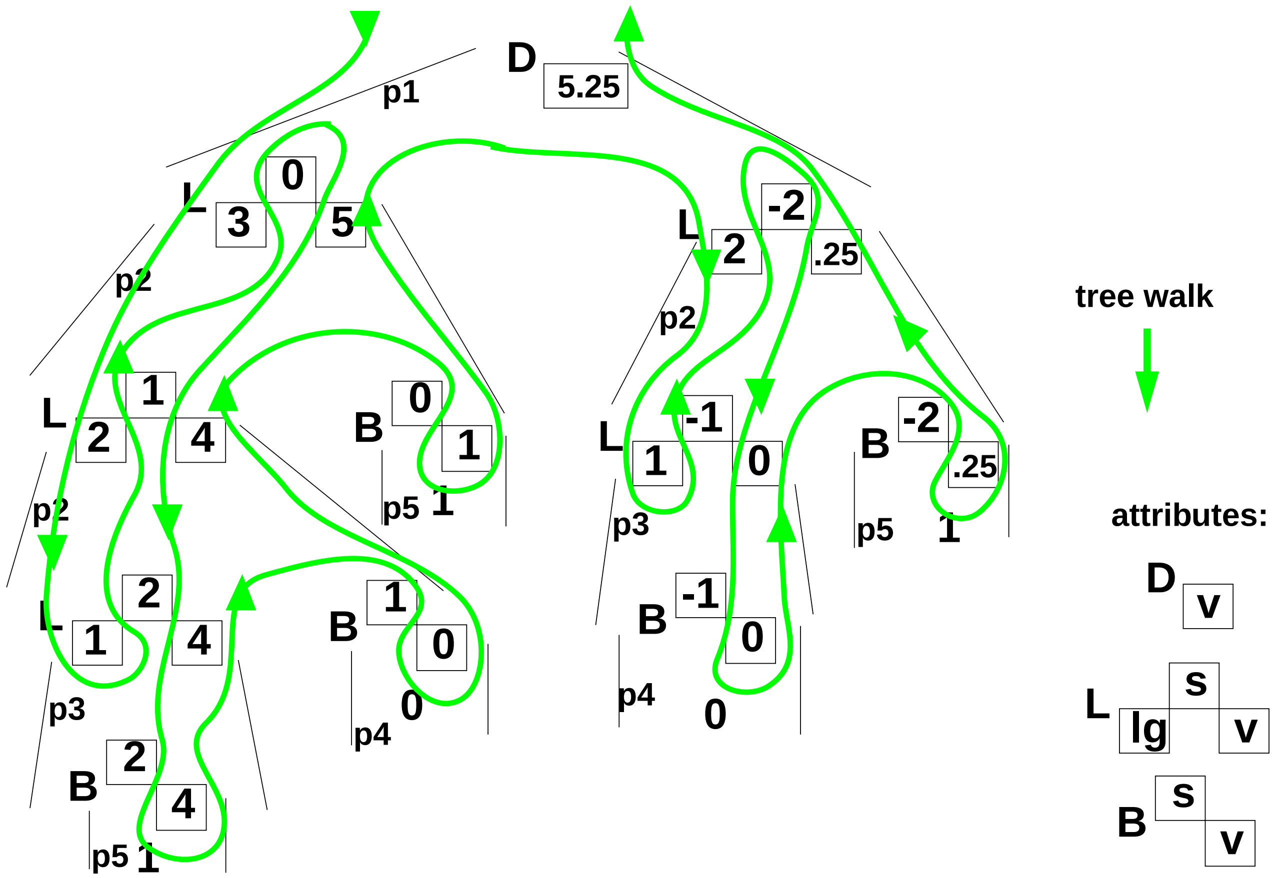
In the lecture:

- Show tree walk

Questions:

- Check that adjacent visit-sequences interleave correctly.
- Check that all dependencies between computations are obeyed.
- Write procedures that implement these visit-sequences.

Tree walk for AG Binary numbers



Lecture Compiler I WS 2001/2002 / Slide 76a

Objectives:

See a concrete tree walk

In the lecture:

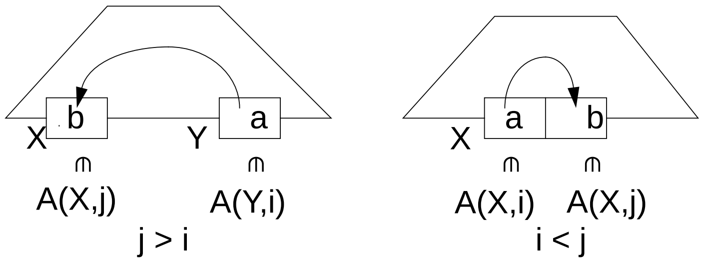
Show that the visit-sequences of CI-76 produce this tree walk for the tree of CI-71.

LAG (k) condition and algorithm

An AG is a LAG(k), if: For each symbol X there is an attribute partition A (X,1), ..., A (X, k), such that the attributes in A (X, i) can be computed in the i-th depth-first left-to-right pass.

Necessary and sufficient condition over dependency graphs - expressed graphically:

A dependency
from right to left



A dependency
at one symbol
on the right-hand
side

Algorithm: computes A (1), ..., A (k), if the AG is LAG(k), for i = 1, 2, ...

A (i) := all attributes that are not yet assigned

remove attributes from A(i) as long as the following rules are applicable:

- remove X.b, if there is a context where it depends on an attribute of A (i) according to the pattern given above,
- remove Z.c, if it depends on a removed attribute

Finally: all attributes are assigned to a passes i = 1, ..., k — the AG **is LAG(k)**
all attributes are removed from A(i) — the AG **is not LAG(k) for any k**

Lecture Compiler I WS 2001/2002 / Slide 77

Objectives:

Understand the LAG condition

In the lecture:

- Explain the LAG(k) condition,
- motivate it by depth-first left-to-right tree walks,
- explain the algorithm using the example of CI-73.

Suggested reading:

Kastens / Übersetzerbau, Section 5.2.3

Assignments:

- Check LAG(k) condition for AGs (Exercise 20)

Questions:

- At the end of each iteration of the i-loop one of three conditions hold. Formulate them.

Generators for attribute grammars

LIGA	University of Paderborn	OAG
FNC-2	INRIA	ANCAG (Oberklasse von OAG)
Synthesizer Generator	Cornell University	OAG, inkrementell
CoCo	Universität Linz	LAG(1)

Properties of the generator LIGA

- integrated **in the Eli system**, cooperates with other Eli tools
- **high level specification language** Lido
- modular and **reusable AG components**
- object-oriented constructs usable for **abstraction of computational patterns**
- computations are **calls of functions** implemented outside the AG
- **side-effect computations** can be controlled by dependencies
- notations for **remote attribute access**
- **visit-sequence** controlled attribute evaluators, implemented in C
- **attribute storage optimization**

Lecture Compiler I WS 2001/2002 / Slide 78

Objectives:

See what generators can do

In the lecture:

- Explain the generators
- Explain properties of LIGA

Suggested reading:

Kastens / Übersetzerbau, Section 5.4

State attributes without values

```
RULE: Root ::= Expr COMPUTE
  Expr.print = "yes";
  printf ("\n") <- Expr.printed;
END;
RULE: Expr ::= Number COMPUTE
  Expr.printed =
    printf ("%d ", Number) <- Expr.print;
END;
RULE: Opr  ::= '+' COMPUTE
  Opr.printed = printf ("+ ") <- Opr.print;
END;
RULE: Opr  ::= '*' COMPUTE
  Opr.printed = printf ("* ") <- Opr.print;
END;
RULE: Expr  ::= Expr Opr Expr COMPUTE
  Expr[2].print = Expr[1].print;
  Expr[3].print = Expr[2].printed;
  Opr.print = Expr[3].printed;
  Expr[1].printed = Opr.printed;
END;
```

The attributes **`print`** and **`printed`** do not have a value

They just describe pre- and post-conditions of computations:

`Expr.print:`
postfix output has been done up to not including this node

`Expr.printed:`
postfix output has been done up to including this node

Lecture Compiler I WS 2001/2002 / Slide 78a

Objectives:

Understand state attributes

In the lecture:

Explain

- attributes without values,
- representing only dependencies between computations.

Questions:

How would the output look like if we had omitted the state attributes and their dependencies?

Dependency pattern CHAIN

```
CHAIN print: VOID;

RULE: Root ::= Expr COMPUTE
  CHAINSTART HEAD.print = "yes";
  printf ("\n ") <- TAIL.print;
END;

RULE: Expr ::= Number COMPUTE
  Expr.print =
    printf ("%d ", Number) <- Expr.print;
END;

RULE: Opr ::= '+' COMPUTE
  Opr.post = printf ("+") <- Opr.pre;
END;

RULE: Expr ::= Expr Opr Expr COMPUTE
  Opr.pre = Expr[3].print;
  Expr[1].print = Opr.post;
END;
```

A CHAIN specifies a **left-to-right depth-first** dependency through a subtree.

Trivial computations of the form X.a = Y.b in the CHAIN order can be **omitted**. They are added as needed.

Lecture Compiler I WS 2001/2002 / Slide 78b

Objectives:

See LIDO construct CHAIN

In the lecture:

- Explain the CHAIN pattern.
- Compare the example with CI-78a

Dependency pattern INCLUDING

```
ATTR depth: int;

RULE: Root ::= Block COMPUTE
  Block.depth = 0;
END;

RULE: Statement ::= Block COMPUTE
  Block.depth =
     ADD (INCLUDING Block.depth, 1);
END;

TERM Ident: int;

RULE: Definition ::= 'define' Ident  COMPUTE
  printf ("%s defined on depth %d\n ",
           StringTable (Ident),
           INCLUDING Block.depth);
END;
```

`INCLUDING Block.depth`
accesses the **`depth`** attribut of the next upper node of type **`Block.`**

An **attribute** at the root of a subtree is **used from within the subtree**.

Propagation through the contexts in between is **omitted**.

Lecture Compiler I WS 2001/2002 / Slide 78c

Objectives:

See LIDO construct INCLUDING

In the lecture:

Explain the use of the INCLUDING construct.

Dependency pattern CONSTITUENTS

```
RULE: Block ::= '{' Sequence '}' COMPUTE
  Block.DefDone =
    CONSTITUENTS Definition.DefDone;
END;
RULE: Definition ::= 'Define' Ident COMPUTE
  Definition.DefDone =
    printf ("%s defined in line %d\n",
            StringTable(Ident), LINE);
END;
RULE: Usage ::= 'use' Ident COMPUTE
   printf ("%s used in line %d\n ",
           StringTable(Ident), LINE),
   <- INCLUDING BLOCK.DefDone;
END;
```

`CONSTITUENTS Definition.DefDone` accesses the **`DefDone`** attributes of all **`Definition`** nodes in the subtree below this context

A computation **accesses attributes from the subtree below** its context.

Propagation through the contexts in between is **omitted**.

The shown combination with INCLUDING is a common dependency pattern.

Lecture Compiler I WS 2001/2002 / Slide 78d

Objectives:

See LIDO construct CONSTITUENTS

In the lecture:

Explain the use of the CONSTITUENTS construct.

4.2 Definition module

Central data structure, stores properties of program entities
e. g. *type of a variable, element type of an array type*

A program entity is identified by the **key** of its entry in the data structure.

Operations:

NewKey ( )	yields a new key
ResetP (k, v)	sets the property P to have the value v for key k
SetP (k, v, d)	as ResetP; but the property is set to d if it has been set before
GetP (k, d)	yields the value of the Property P for the key k; yields the default-Wert d, if P has not been set

Operations are called as dependent computations in the tree

Implementation: a property list for every key, for example

Generation of the definition module: From specifications of the form

Property name : property type;
```
ElementNumber: int;
```

functions ResetElementNumber, SetElementNumber, GetElementNumber are generated.

Lecture Compiler I WS 2001/2002 / Slide 79

Objectives:

Properties of program entities

In the lecture:

- Explain the operations,
- explain the generator,
- give examples.

Suggested reading:

Kastens / Übersetzerbau, Section S. 130 unten

Assignments:

- Use the PDL tool of Eli

Questions:

- Give examples where calls of the operations are specified as computations in tree contexts. Describe how they depend on each other.

4.3 Type analysis

Task: Compute and check types of program entities and constructs at compile time

- **defined entities** (e. g. variables)
  have a type property, stored in the definition module
- **program constructs** (e. g. expressions)
  have a type attribute, associated to their symbol resp. tree node
  special task: **resolution of overloaded operators** (functions, methods)
- **types themselves are program entities**
  represented by keys;
  named using type definitions; **unnamed** in complex type notations
- **types have properties**
  e. g. the element type of an array type
- **type checking for program entities and for program constructs**
  a type must / may not have certain properties in certain contexts
  compare expected and given type; **type relations**: equal, compatible;
  compute type **coercion**

Lecture Compiler I WS 2001/2002 / Slide 80

Objectives:

Learn to categorize the tasks

In the lecture:

- Motivate type analysis tasks with typical properties of strongly typed languages;
- give examples

Suggested reading:

Kastens / Übersetzerbau, Section 6.1

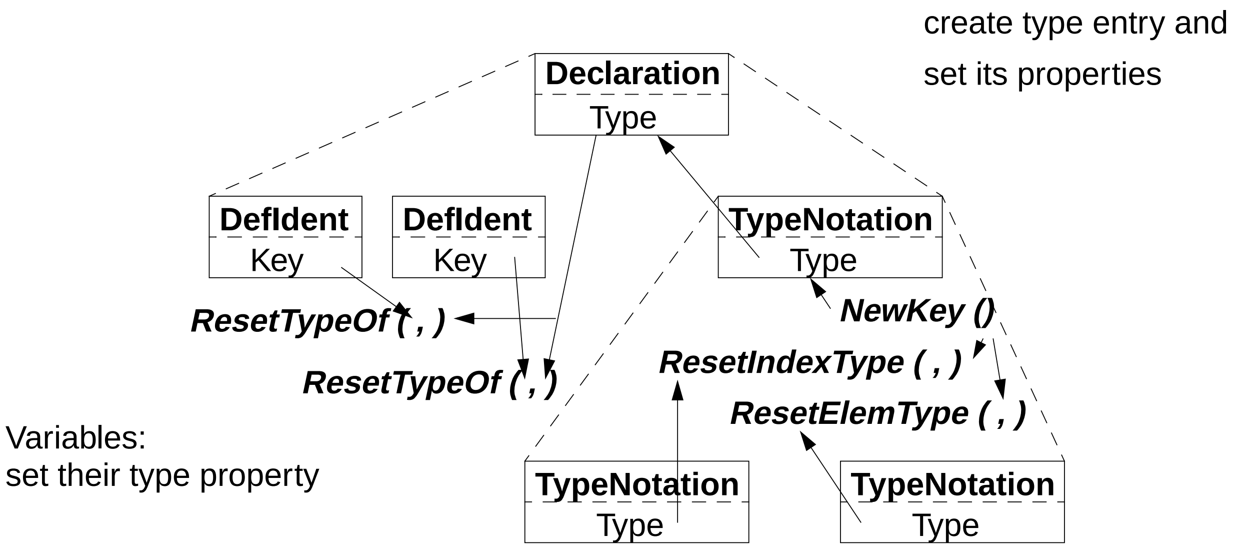
Questions:

- Give examples for program entities that have a type property and for others which don't.
- Enumerate at least 5 properties of types in Java, C or Pascal.
- Give an example for a recursively defined type, and show its representation using keys.

Declarations and type notations

operations in the tree for the construct:

```
a, b: array [1..10] of real;
```



Lecture Compiler I WS 2001/2002 / Slide 81

Objectives:

Understand type analysis for declarations

In the lecture:

- Types as properties of program entities,
- types as attributes of program constructs,
- explain attributes and computations in the tree,
- explain the dependencies between the computations.

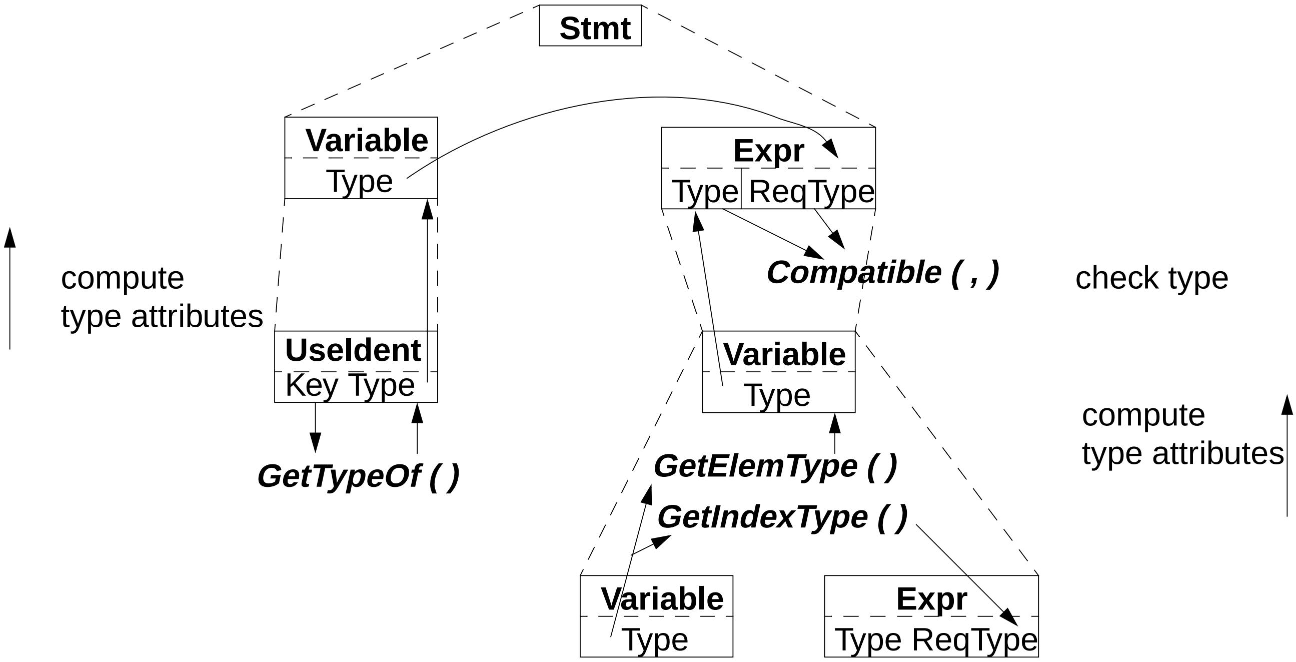
Suggested reading:

Kastens / Übersetzerbau, Section 6.1

Types of expressions required by context

operations in the tree for:

```
x := a[i];
```



Lecture Compiler I WS 2001/2002 / Slide 82

Objectives:

Example for computation and check of types

In the lecture:

- Types as properties of program entities,
- types as attributes of program constructs,
- explain attributes and computations in the tree,
- explain the dependencies between the computations.

Suggested reading:

Kastens / Übersetzerbau, Section 6.1

Assignments:

- Compose the trees of CI-81 and CI-82 into a complete tree. Find an evaluation order for the operations. State for each operation the weakest precondition with respect to the execution of other operations.

(see also Exercise 24)

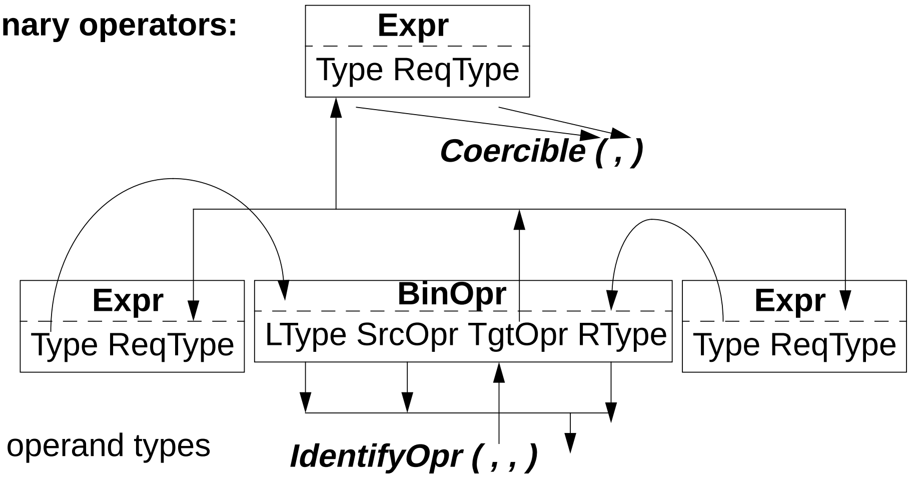
Overloading resolution for operators

Overloading: **same operator symbol** (source operator) is used for **several target operators** having **different signatures** and **different meanings**, e. g. specified by a table like:

symbol	**signature**	**meaning**
+	int ✕ int -> int	addition of integral numbers
+	real ✕ real -> real	floating point addition
+	set ✕ set -> set	union of sets
=	t ✕ t -> boolean	comparison for values of type t

Coercion: implicitly applicable type conversion: e. g. int -> real, char -> string, ...

Context of overloaded binary operators:



given: source operator and operand types
find: target operator

Lecture Compiler I WS 2001/2002 / Slide 83

Objectives:

Understand the task of overloading resolution

In the lecture:

Explain

- overloaded operators, functions, and methods,
- attribute computations,
- Eli tool OIL

Suggested reading:

Kastens / Übersetzerbau, Section 6.1

Assignments:

- overloading resolution as in C (Exercise 23)

Type analysis for object-oriented languages

Class hierarchy is a type hierarchy:

implicit type coercion: class -> super class
explicit type cast: class -> subclass

Variable of class type may contain
an object (reference) of its subclass

```
Circle k = new Circle (...);

GeometricShape f = k;

k = (Circle) f;
```

Check signature of overriding methods:

calls must be type safe; Java requires the same signature;
following weaker requirements are sufficient (*contra variant parameters*, language Sather):

call of dynamically bound method: `a = x.m (p);`

Variable: `X x; A a; P p; C c; B b;`

super class `class X { C m (Q q) {` use of `q;... return c; } }`

subclass `class Y { B m (R r) {` use of `r;... return b; } }`

Analyse dynamic methode binding; try to decide it statically:

static analysis tries to further restrict the run-time type:

```
GeometricShape f;...; f = new Circle(...);...; a = f.area();
```


Lecture Compiler I WS 2001/2002 / Slide 84

Objectives:

Understand classes as types

In the lecture:

Explain

- class hierarchy - type coercion
- type checking for dynamically bound methods calls
- predict the runtime classs of objects

Questions:

- Why would overridden methods not be type safe if they had "covariant" parameters (all 3 arrows between the classes X and Y would point up)? That is the situation in Eiffel.

Type analysis for functional languages (1)

Static typing and type checking without types in declarations

Type inference: Types of program entities are inferred from the context where they are used

Example in ML:

```
fun choice (cnt, fct) =
  if fct cnt then cnt else cnt - 1;
```

describe the types of entities using type variables:

```
cnt:    'a,
fct:    'b->'c,
choice: ('a * ('b->'c)) -> 'd
```

form equations that describe the uses of typed entities

```
'c = bool
'b = 'a
'd = 'a
'a = int
```

solve the system of equations:

```
choice: (int * (int->bool)) -> int
```


Lecture Compiler I WS 2001/2002 / Slide 85

Objectives:

Understand type inference

In the lecture:

Explain how types are computed from the operations without having typed declarations

Questions:

- How would type inference find type errors?

Type analysis for functional languages (2)

Parametrically polymorphic types: types having type parameters

Example in ML:

```
fun map (l, f) =
          if null l
          then nil
          else (f (hd l)) :: map (tl l, f)
```

polymorphic signature:

```
map: ('a list * ('a -> 'b)) -> 'b list
```

Type inference yields **most general type** of the function,
such that all uses of entities in operations are correct;

i. e. **as many unbound type parameters as possible**

calls with different concrete types, consistently substituted for the type parameter:

```
map([1,2,3], fn i => i*i)         'a = int, 'b = int
map([1,2,3], even)                'a = int, 'b = bool
map([1,2,3], fn i =(i,i))         'a = int, 'b = ('a*'a)
```


Lecture Compiler I WS 2001/2002 / Slide 86

Objectives:

Understand polymorphic types

In the lecture:

- Explain analysis with polymorphic types.
- Explain the difference of polymorphic types and generic types from the view of type analysis.

4.4 Name analysis

Identifiers identify program entities in the program text (**statically**).

The **definition** of an identifier b introduces a **program entity** and **binds** it to the **identifier**.
The binding is valid in a certain range of the program text: the **scope of the definition**.

Name analysis task: Associate the **key of a program entity to each occurrence of an identifier** (consistent renaming) according to **scope rules** of the language.

Hiding rules for languages with nested structures:

- **Algol rule**: The definition of an identifier b is valid in the **whole smallest enclosing range**; but not in inner ranges that have a definition of b, too.
(e. g. Algol 60, Pascal, Java, ... with additional rules)
- **C rule**: The definition of an identifier b is valid in the **smallest enclosing range from the position of the definition** to the end; but not in inner ranges that have another definition of b from the position of that definition. (e. g. C, C++, Java, ... with additional rules)

Ranges are syntactic constructs like **blocks, functions, modules, classes, packets**
- as defined for the particular language.

Implementation of name analysis:
Operations of the environment module are called in suitable tree contexts.

Lecture Compiler I WS 2001/2002 / Slide 87

Objectives:

Understand task of name analysis

In the lecture:

Explanations and examples for

- hiding rules (see "Grundlagen der Programmiersprachen"),
- name analysis task: consistent renaming

Suggested reading:

Kastens / Übersetzerbau, Section 6.2, 6.2.2

Questions:

- Assume consistent renaming has been applied to a program. Why are scope rules irrelevant for the resulting program?

Environment module

Implements the abstract data type **Environment**:
hierarchically nested sets of **Binding**s **(identifier, environment, key)**

Functions:

NewEnv ()	creates a new Environment e, to be used as root of a hierarchy
NewScope (e_1)	creates a new Environment e_2 that is nested in e1. Each binding of e_1 is also a binding of e_2 if it is not hidden there.
BindIdn (e, id)	introduces a binding (id, e, k) if e has no binding for id; then k is a new key representing a new entity; in any case the result is the binding triple (id, e, k)
BindingInEnv (e, id)	yields a binding triple (id, e_1, k) of e or a surrounding environment of e; yields NoBinding if no such binding exists.
BindingInScope (e, id)	yields a binding triple (id, e, k) of e, if contained directly in e, NoBinding otherwise.

Lecture Compiler I WS 2001/2002 / Slide 88

Objectives:

Learn the interface of the Environment module

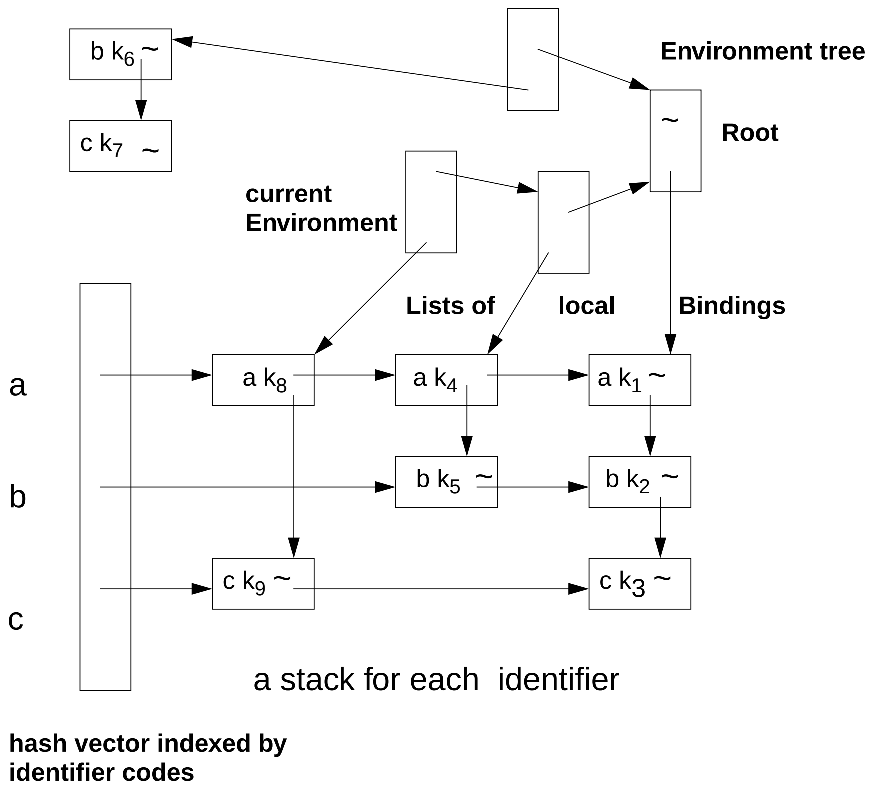
In the lecture:

- Explain the notion of Environment,
- Explain the example of CI-89,
- show that the module is generally applicable.

Suggested reading:

Kastens / Übersetzerbau, Section 6.2.2

Data structure of the environment module



k_i: key of the defined entity

Lecture Compiler I WS 2001/2002 / Slide 89

Objectives:

An efficient data structure

In the lecture:

Explanations and examples for

- Explain the concept of identifier stacks.
- Demonstrate the effect of the operations.
- O(1) access instaed of linear search.
- Explain how the current environment is changed using operations Enter and Leave, which insert a set of bindings into the stacks or remove it.

Suggested reading:

Kastens / Übersetzerbau, Section 6.2.2

Questions:

- In what sense is this data structure efficient?
- Describe a program for which a linear search in definition lists is more efficient than using this data structure.
- The efficiency advantage may be lost if the operations are executed in an unsuitable order. Explain!
- How can the current environment be changed without calling Enter and Leave explicitly?

Environment operations in tree contexts

Operations in tree contexts and the order they are called model scope rules.

Root context:

```
Root.Env = NewEnv ();
```

Range context that may contain definitions:

```
Range.Env = NewScope (INCLUDING (Range.Env, Root.Env);
```

accesses the next enclosing Range or Root

defining occurrence of an identifier `IdDefScope`:

```
IdDefScope.Bind = BindIdn (INCLUDING Range.Env, IdDefScope.Symb);
```

applied occurrence of an identifier `IdUseEnv`:

```
IdUseEnv.Bind = BindingInEnv (INCLUDING Range.Env, IdUseEnv.Symb);
```

Preconditions for specific scope rules:

Algol rule: all **`BindIdn()`** of all surrounding ranges before any **`BindingInEnv()`**
C rule: **`BindIdn()`** and **`BindingInEnv()`** in textual order

The resulting **bindings are used for checks and transformations**, e. g.

- no applied occurrence without a valid defining occurrence,
- at most one definition for an identifier in a range,
- no applied occurrence before its defining occurrence (Pascal).

Lecture Compiler I WS 2001/2002 / Slide 90

Objectives:

Apply environment module in the program tree

In the lecture:

- Explain the operations in tree contexts.
- Show the effects of the order of calls.

Suggested reading:

Kastens / Übersetzerbau, Section 6.2.1

Assignments:

Use Eli module for a simple example.

Questions:

- How do you check the requirement "definition before application"?
- How do you introduce bindings for predefined entities?
- Assume a simple language where the whole program is the only range. There are no declarations, variables are implicitly declared by using their name. How do you use the operations of the environment module for that language?

Semantic error handling

Design rules:

Error reports **related to the source code**:

- any explicit or implicit **requirement of the language definitions** needs to be checked by an operation in the tree
- check has to be associated to the **smallest relevant context** yields precise source position for the report; propagate information to that context if necessary
- **meaningfull error report**
- **different reports for different violations**, do not connect texts by **or**

All **operations specified for the tree are executed**, even if errors occur:

- introduce **error values**, e. g. **`NoKey, NoType, NoOpr`**
- operations that **yield results** have to yield a reasonable one in case of error,
- operations have to accept **error values as parameters**,
- **avoid messages for avalanche errors** by suitable extension of relations, e. g. every type is compatible with **`NoType`**

Lecture Compiler I WS 2001/2002 / Slide 91

Objectives:

Design rules for error handling

In the lecture:

Explanations and examples

Suggested reading:

Kastens / Übersetzerbau, Section 6.3

5. Transformation

Create **target tree** to represent the program in the intermediate language.

Intermediate language spcified externally or designed for the abstract source machine.

Design rules:

- **simplify the structure**
  only those constructs and properties that are needed for the synthesis phase;
  omit declarations and type denotations - they are kept in the definition module
- **unify constructs**
  e. g. standard representation of loops, or translation into jumps and labels
- **distinguished target operators for overloaded operators**
- **explicit target operators for implicit source operations**
  e. g. type coercion, contents operation for variable access, run-time checks

Transfer **target tree and definition module to synthesis phase**
as data structure, file, or sequence of function calls

For **source-to-source translation** the target tree represents the **target program**.
The target text is produced from the tree by **recursive application of text patterns**.

Lecture Compiler I WS 2001/2002 / Slide 92

Objectives:

Properties of intermediate languages

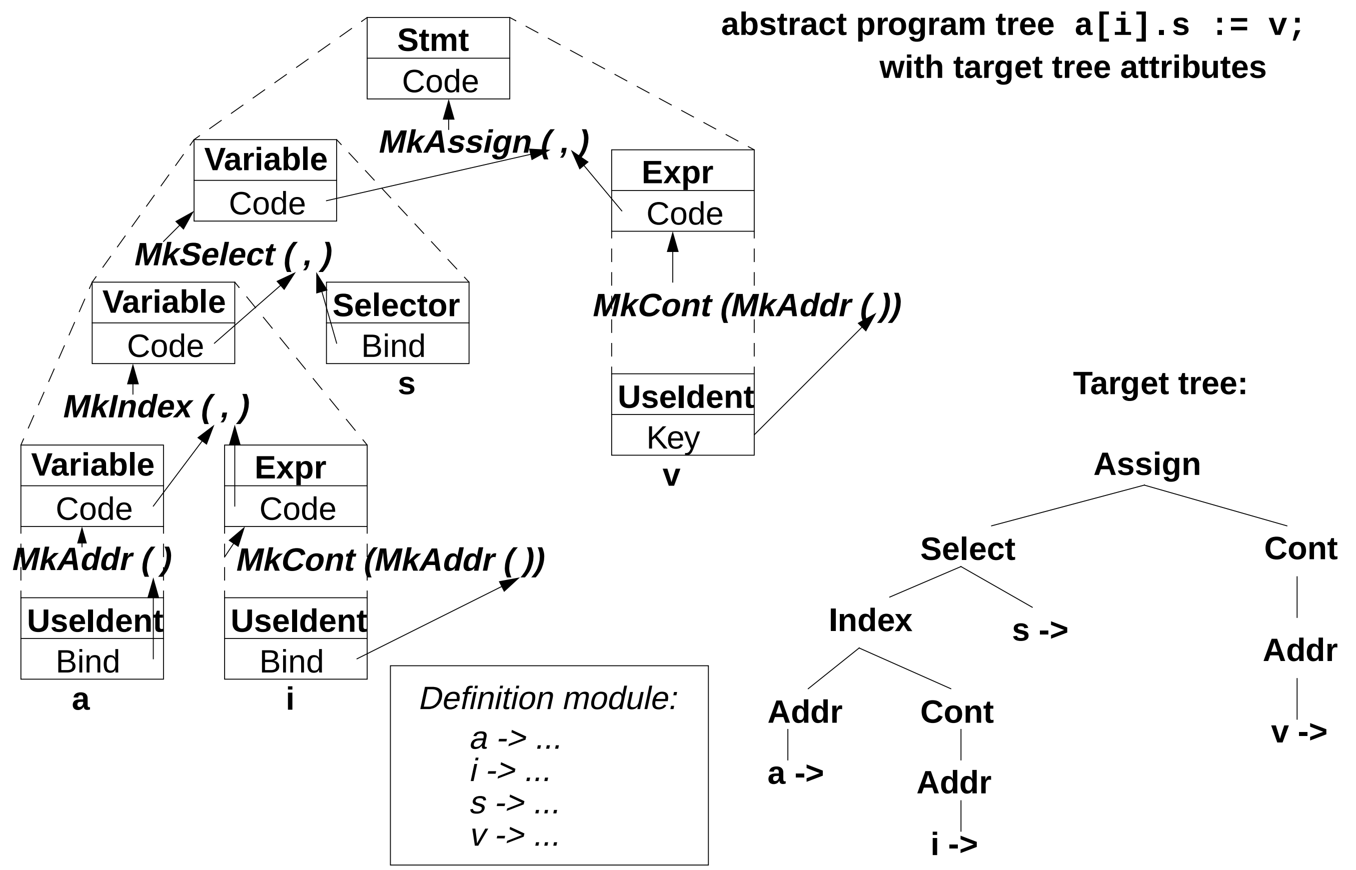
In the lecture:

Example for a target tree on CI-93

Suggested reading:

Kastens / Übersetzerbau, Section 6.4

Example: Target tree construction



Lecture Compiler I WS 2001/2002 / Slide 93

Objectives:

Recognize the principle of target tree construction

In the lecture:

Explain the principle using the example.

Attribute grammar for target tree construction (CI-93)

```
RULE: Stmt ::= Variable ':=' Expr            COMPUTE
    Stmt.Code = MkAssign (Variable.Code, Expr.Code);
END;
RULE: Variable ::= Variable '.' Selector   COMPUTE
    Variable[1].Code = MkSelect (Variable[2].Code, Selector.Bind);
END;
RULE: Variable ::= Variable '[' Expr ']'     COMPUTE
    Variable[1].Code = MkIndex (Variable[2].Code, Expr.Code);
END;
RULE: Variable ::= UseIdent                  COMPUTE
    Variable.Code = MkAddr (UseIdent.Bind);
END;
RULE: Expr ::= UseIdent                      COMPUTE
    Expr.Code = MkCont (MkAddr (UseIdent.Bind));
END;
```

Lecture Compiler I WS 2001/2002 / Slide 94

Objectives:

Attribute grammar specifies target tree construction

In the lecture:

Explain using the example of CI-93

Generator for creation of structured target texts

Tool PTG: Pattern-based Text Generator

Creation of structured texts in arbitrary languages. Used as computations in the abstract tree, and also in arbitrary C programs. Principle shown by examples:

1. Specify output pattern with insertion points:

```
ProgramFrame:   $
                "void main () {\n"
                $
                "}\n"

Exit:           "exit (" $ int ");\n"

IOInclude:      "#include <stdio.h>"
```

2. PTG generates a function for each pattern; calls produce target structure:

```
PTGNode a, b, c;
a = PTGIOInclude ();
b = PTGExit (5);
c = PTGProgramFrame (a, b);
```

correspondingly with attribute in the tree

3. Output of the target structure:

`PTGOut (c);` or `PTGOutFile ("Output.c", c);`

Lecture Compiler I WS 2001/2002 / Slide 95

Objectives:

Principle of producing target text using PTG

In the lecture:

Explain the examples

Questions:

- Where can PTG be applied for tasks different from compilers?

PTG Patterns for creation of HTML-Texts

concatenation of texts:
```
    Seq:         $ $
```

large heading:
```
    Heading:     "<H1>" $1 string "</H1>\n"
```

small heading:
```
    Subheading:  "<H3>" $1 string "</H3>\n"
```

paragraph:
```
    Paragraph:   "<P>\n" $1
```

Lists and list elements:
```
    List:        "<UL>\n" $ "</UL>\n"
    Listelement: "<LI>" $ "</LI>\n"
```

Hyperlink:
```
    Hyperlink:   "<A HREF=\"" $1 string "\">" $2 string "</A>"
```

Text example:

```
<H1>My favorite travel links</H1>
<H3>Table of Contents</H3>
<UL>
<LI> <A HREF="#position_Maps">Maps</A>
<LI> <A HREF="#position_Train">Train</A>
</UL>
```


Lecture Compiler I WS 2001/2002 / Slide 96

Objectives:

See an application of PTG

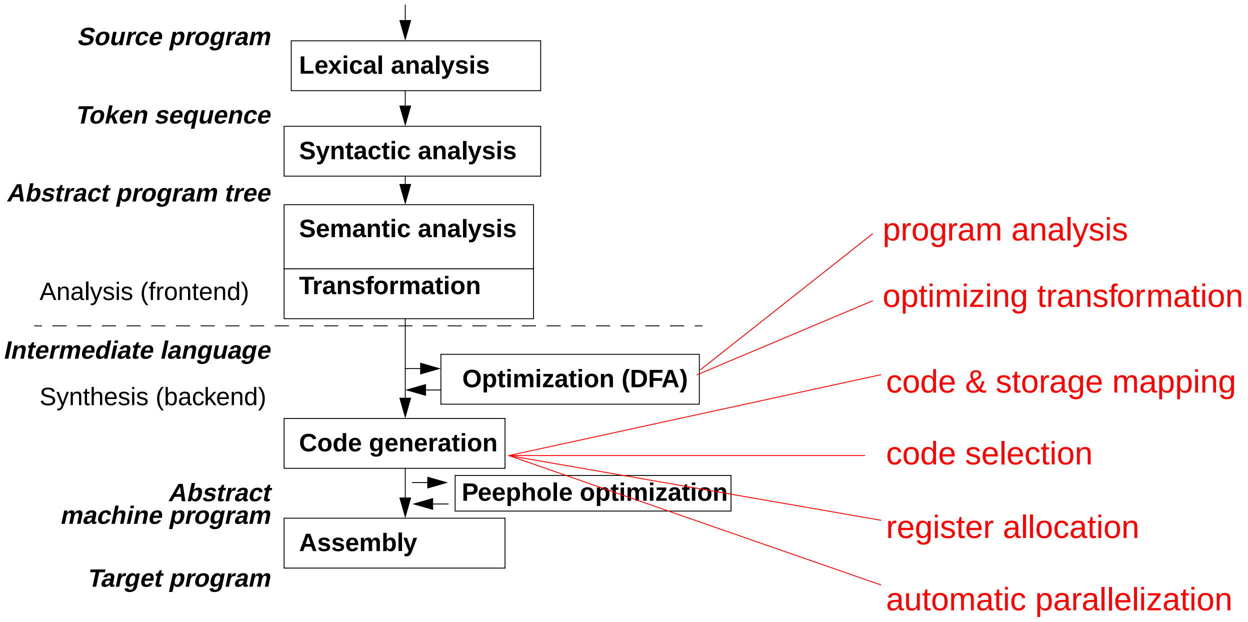
In the lecture:

Explain the patterns

Questions:

- Which calls of pattern functions produce the example text given on the slide?

6. Synthesis: An Overview



Lecture Compiler I WS 2001/2002 / Slide 97

Objectives:

Relate synthesis topics to compiler structure

In the lecture:

- This chapter addresses only a selection of synthesis topics.
- Only a rough idea is given for each topic.
- The topics are treated completely in the lecture "Compiler II".

Optimization

Objective: Reduce run-time and/or code size of the program, without changing its effect.
Eliminate redundant computations, simplify computations.

Input:	Program in intermediate language
Task:	**Analysis** (find redundancies), apply **transformations**
Output:	Improved program in intermediate language

Program analysis:
static properties of program structure and execution
safe, pessimistic assumptions where input and dynamic execution paths are not known

Context of analysis:

Expression	local optimization	
Basic block	local optimization	
Control flow graph (procedure)	global	intra-procedural optimization
Control flow graph, call graph	global	inter-procedural optimization

Lecture Compiler I WS 2001/2002 / Slide 98

Objectives:

Overview over optimization

In the lecture:

- Program analysis computes safe assumptions at compile time about execution of the program.
- The larger the analysis context, the better the information.
- Conventionally this phase is called "Optimization", although in most cases a formal optimum can not be defined or achieved with practical effort.

Suggested reading:

Kastens / Übersetzerbau, Section 8

Optimizing Transformations

Name of transformation:	**Example for its application:**
• **Algebraic simplification of expressions**	`2*3.14   x+0   x*2   x**2`
• **Constant propagation (dt. Konstantenweitergabe)**	`x = 2; ... y = x * 5;`
• **Common subexpressions (Gemeinsame Teilausdrücke)**	`x=a*(b+c);...y=(b+c)/2;`
• **Dead variables (Überflüssige Zuweisungen)**	`x = a + b; ... x = 5;`
• **Copy propagation (Überflüssige Kopieranweisungen)**	`x = y; ... ; z = x;`
• **Dead code (nicht erreichbarer Code)**	`b = true;...if (b) x = 5; else y = 7;`
• **Code motion (Code-Verschiebung)**	`if (c) x = (a+b)*2; else x = (a+b)/2;`
• **Function inlining (Einsetzen von Aufrufen)**	`int Sqr (int i) { return i * i; }`
• **Loop invariant code**	`while (b) {... x = 5; ...}`
• **Induction variables in loops**	`i = 1; while (b) { k = i*3; f(k); i = i+1;}`

Analysis checks **preconditions for safe application** of each transformation;
more applications, if preconditions are analysed in **larger contexts**.

Interdependences:
Application of a transformation may **enable or inhibit** another application of a transformation.

Order of transformations is relevant.

Lecture Compiler I WS 2001/2002 / Slide 99

Objectives:

Get an idea of important transformations

In the lecture:

- Some transformations are explained.
- The preconditions are discussed for some of them.

Suggested reading:

Kastens / Übersetzerbau, Section 8.1

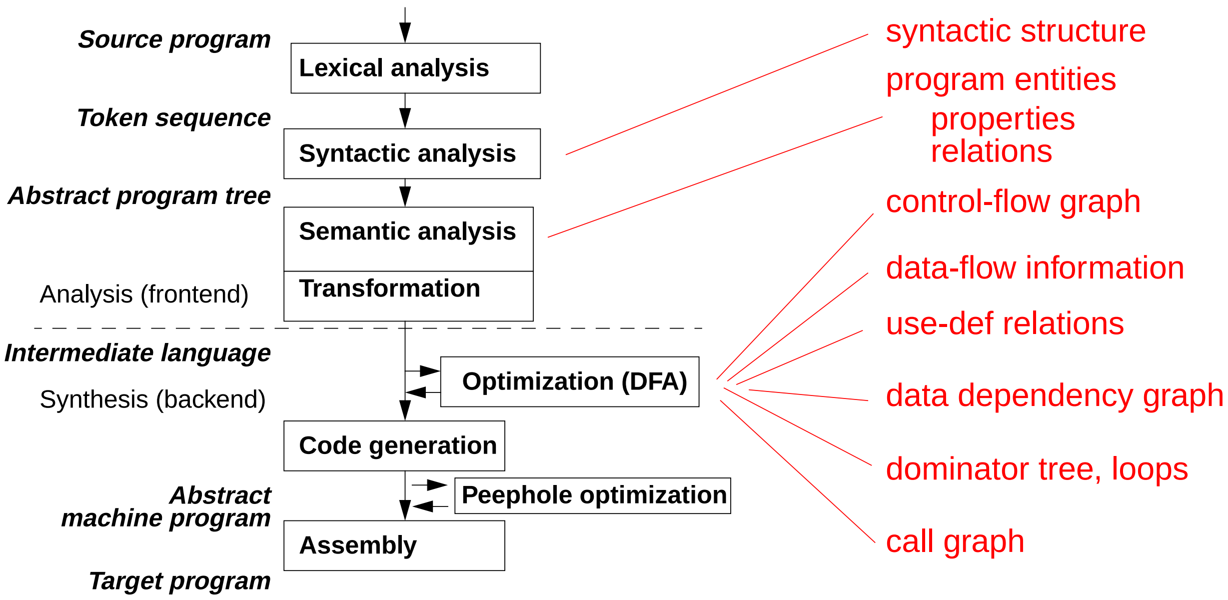
Assignments:

- Apply some transformations in a given example program.

Questions:

- Which of the transformations need to analyze pathes through the program?
- Give an example for a pair of transformations, such that an application of the first one enables an application of the second.

Analysis in Compilers



Lecture Compiler I WS 2001/2002 / Slide 100

Objectives:

See some methods of program analysis

In the lecture:

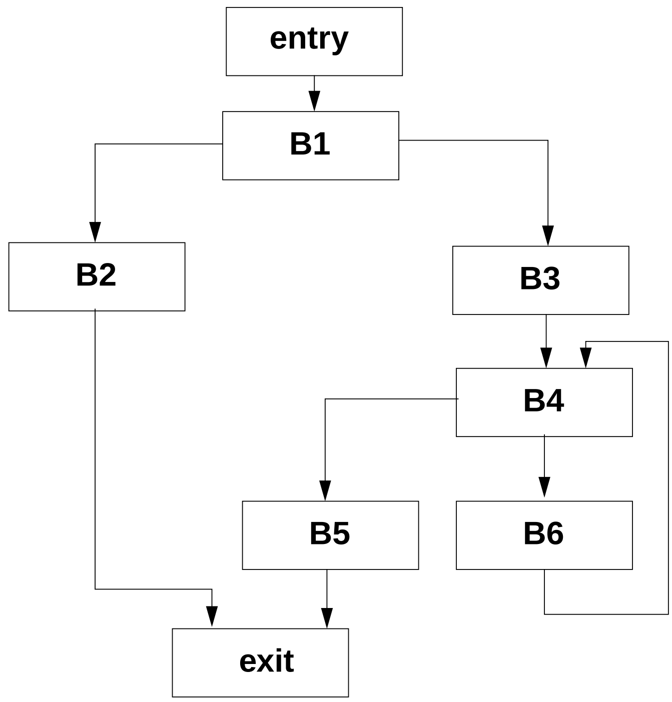
Give brief explanations of the methods

Example for a Control-flow Graph

Intermediate code with basic blocks:

```
     receive m                      B1
     f0 <- 0
     f1 <- 1
     if m <= 1 goto L3

     i <- 2                         B3

L1:  if i <= m goto L2              B4

     return f2                      B5

L2:  f2 <- f0 + f1                  B6
     f0 <- f1
     f1 <- f2
     i <- i + 1
     goto L1

L3:  return m                       B2
```

Control-flow graph:
[Muchnick, p. 172]



Lecture Compiler I WS 2001/2002 / Slide 101

Objectives:

Example for a control-flow graph

In the lecture:

- The control-flow graph represents the basic blocks and their branches.
- See Lecture "Modellierung", Mod-4.27 ("Programmablaufgraphen")

Data-Flow Analysis

Data-flow analysis (DFA) provides information about how the execution of a program may manipulate its data.

Many different problems can be formulated as **data-flow problems**, for example:

- Which assignments to variable **v** may influence a use of **v** at a certain program position?
- Is a variable **v** used on any path from a program position **p** to the exit node?
- The values of which expressions are available at program position **p**?

Data-flow problems are stated in terms of

- **paths through the control-flow graph** and
- **properties of basic blocks**.

Data-flow analysis provides information for **global optimization.**

Data-flow analysis does **not** know

- input values provided at run-time,
- branches taken at run-time.

Its results are to be interpreted **pessimistic**.

Lecture Compiler I WS 2001/2002 / Slide 102

Objectives:

Goals and ability of data-flow analysis

In the lecture:

- The topics on the slide are explained.
- Examples for the use of DFA information are given.
- Examples for pessimistic information are given.

Suggested reading:

Kastens / Übersetzerbau, Section 8.2.4

Questions:

- What's wrong about optimistic information?
- Why can pessimistic information be useful?

Specification of a DFA Problem

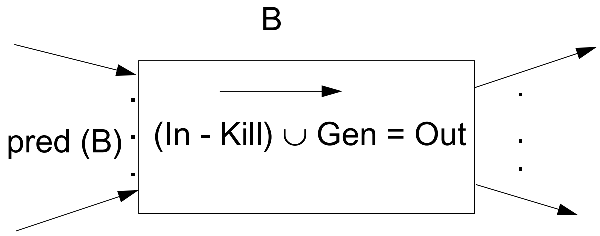
Specification of reaching definitions:

- **Description**:
  A definiton **d** of a variable **v** reaches the begin of a block **B**
  if **there is a path** from **d** to **B** on which **v** is not assigned again.
- It is a **forward problem**.
- The **meet operator** is union.
- The **analysis information** in the sets are assignments at certain program positions.
- **Gen (B)**:
  contains all definitions `d: v = e;` in **B**,
  such that **v** is not defined after **d** in **B**.
- **Kill (B)**:
  if v is assigned in B, then Kill(B)
  contains all definitions `d: v = e;`
  in blocks different from **B**,
  such that **B** has a definition of **v**.

2 equations for each basic block:

$$Out(B) = Gen(B) \cup (In(B) - Kill(B))$$

$$In(B) = \Theta_{h \in pred(B)} Out(h)$$



Lecture Compiler I WS 2001/2002 / Slide 103

Objectives:

Get an idea of DFA problems

In the lecture:

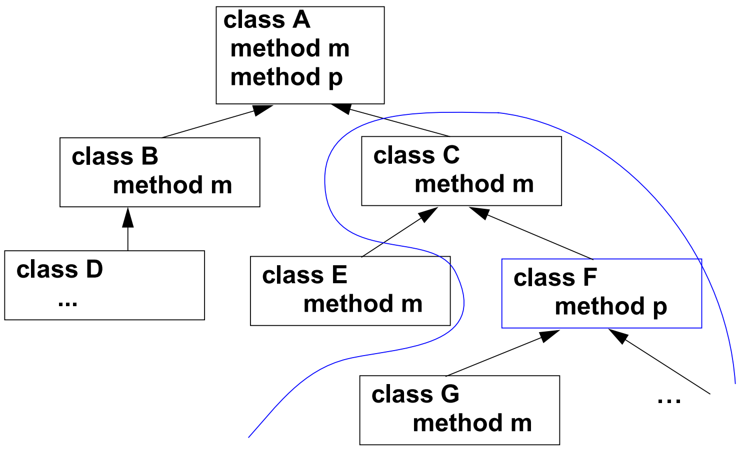
Explain how DFA problems are specified by a set of equations.

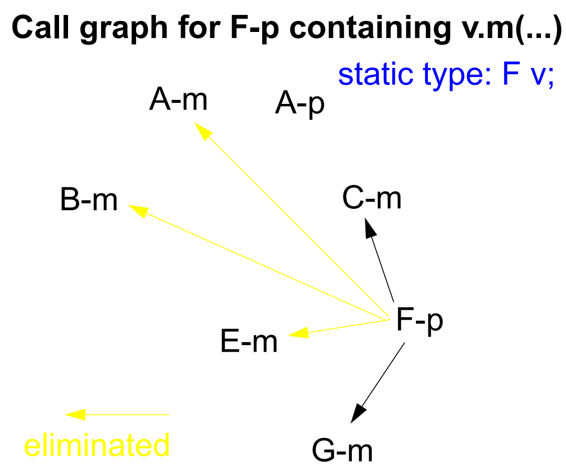
Call Graphs for object-oriented programs

The call graph is reduced to a set of **reachable methods** using the **class hierarchy** and the **static type of the receiver** expression in the call:

If a method F-p **is reachable** and
if it contains a **dynamically bound call v.m(...)** and
T is the static type of v,

then every method **m that is inherited by T or by a subtype of T is also reachable**, and arcs go from F-p to them.





Lecture Compiler I WS 2001/2002 / Slide 104

Objectives:

See a typical object-oriented analysis

In the lecture:

- Dynamically bound method calls contribute significantly to the cost of object-oriented programs.
- Static resolution as far as possible is very effective.

Code Generation

Input: Program in intermediate language

Tasks:

Storage mapping	properties of program objects (size, address) in the definition module
Code selection	generate instruction sequence, optimizing selection
Register allocation	use of registers for intermediate results and for variables

Output: abstract machine program, stored in a data structure

Design of code generation:

- analyze **properties of the target processor**
- plan **storage mapping**
- design at least one **instruction sequence** for each operation of the intermediate language

Implementation of code generation:

- Storage mapping:
  a traversal through the program and the definition module computes
  sizes and addresses of storage objects
- Code selection: use a generator for pattern matching in trees
- Register allocation:
  methods for expression trees, basic blocks, and for CFGs

Lecture Compiler I WS 2001/2002 / Slide 105

Objectives:

Overview on design and implementation

In the lecture:

- Identify the 3 main tasks.
- Emphasize the role of design.

Suggested reading:

Kastens / Übersetzerbau, Section 7

Storage Mapping

Objective:
for each storable program object compute storage class, relative address, size

Implementation:
use properties in the definition module, travers defined program objects

Design the use of storage areas:

code storage	progam code
global data	to be linked for all compilation units
run-time stack	activation records for function calls
heap	storage for dynamically allocated objects, garbage collection
registers for	addressing of storage areas (e. g. stack pointer) function results, arguments local variables, intermediate results (**register allocation**)

Design the type mapping ... C-29

Lecture Compiler I WS 2001/2002 / Slide 106

Objectives:

Design the mapping of the program state onto the machine state

In the lecture:

Explain storage classes and their use

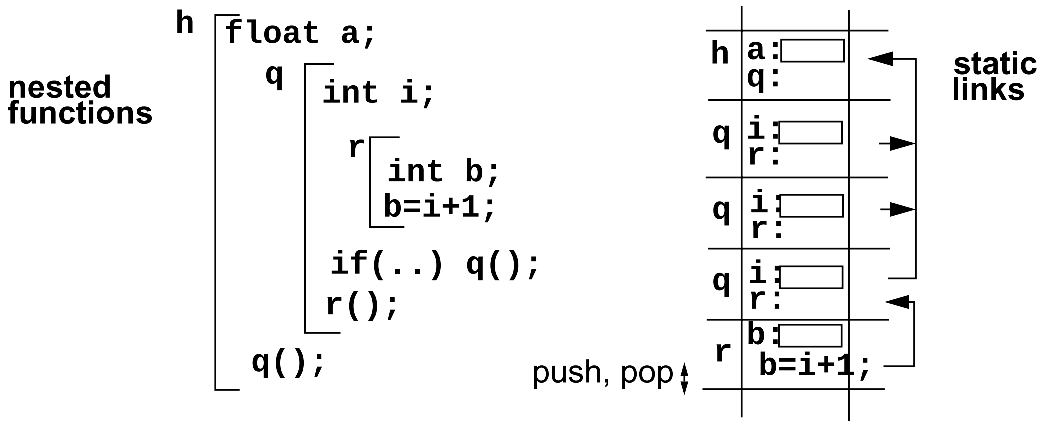
Suggested reading:

Kastens / Übersetzerbau, Section 7.2

Run-Time Stack

Run-time stack contains one **activation record** for each active function call.
Activation record provides storage local data of a function call. (see C-31)

Nested functions (nested classes and objects): static predecessor chain links the accessible activation records, **closure of a function**



Requirement: The closure of a function is still on the run-time stack when the function is called.

Languages without recursive functions (FORTRAN) do not use a run-time stack.

Optimization: activation records of **non-recursive functions** may be allocated statically.

Parallel processes, threads, coroutines need a **separate run-time stack** each.

Lecture Compiler I WS 2001/2002 / Slide 107

Objectives:

Understand the concept of run-time stacks

In the lecture:

The topics on the slide are explained. Examples are given.

- Explain static and dynamic links.
- Explain nesting and closures.
- Different language restrictions to ensure that necessary closures are on the run-time stack.

Questions:

- How do C, Pascal, and Modula-2 obey the requirement on stack discipline?
- Why do threads need a separate run-time stack?

Code Sequences for Control Statements

A **code sequence** defines how a **control statement** is transformed into jumps and labels.

Several variants of code sequences may be defined for one statement.

Example:

```
while (Condition) Body      M1:  Code (Condition, false, M2)
                                 Code (Body)
                                 goto M1
                            M2:
```

variant:

```
                                 goto M2
                            M1:  Code (Body)
                            M2:  Code (Condition, true, M1)
```

Meaning of the **Code** constructs:

Code (S):	generate code for statements **S**
Code (C, true, M)	generate code for condition **C** such that it branches to M if **C** is true, otherwise control continues without branching

Lecture Compiler I WS 2001/2002 / Slide 108

Objectives:

Concept of code sequences for control structures

In the lecture:

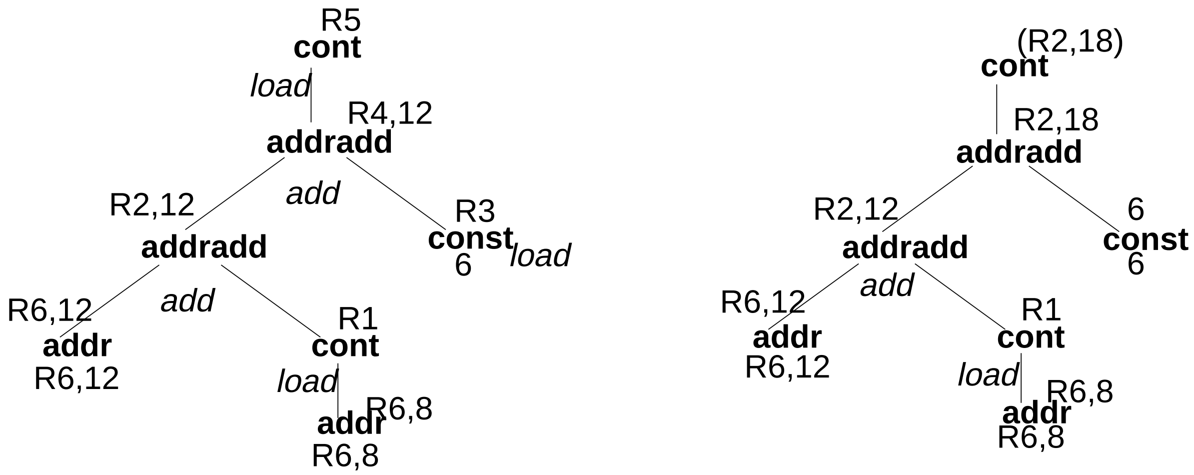
- Explain the code sequence for while statements.
- Explain the transformation of conditions.
- Discuss the two variants.
- Develop a code sequence for for statements.

Questions:

- What are the advantages of each alternative?
- Give a code sequence for do-while statements.

Example for Code Selection

tree for assignment ... = **`a[i].s;`**



load (R6,8), R1
add R6,R1,R2
load 6,R3
add R2,R3,R4
load (R4,12),R5
store R5, ...

cost: 6 instructions

load (R6,8), R1
add R6,R1,R2
store (R2,18),...

cost: 3 instructions

Lecture Compiler I WS 2001/2002 / Slide 109

Objectives:

Get an idea of code selection by tree patterns

In the lecture:

- Show application of patterns.
- Explain code costs.

Register Allocation

Use of registers:

intermediate results of expression evaluation

reused results of expression evaluation (CSE)

contents of frequently used variables

parameters of functions, function result (cf. register windowing)

stack pointer, frame pointer, heap pointer, ...

Number of registers is limited - for each register class: address, integer, floting point

register allocation aims at ruduction of

- number of memory accesses
- spill code, i. e. instructions that store and reload the contents of registers

specific allocation methods for different context ranges:

- expression trees (Sethi, Ullman)
- basic blocks (Belady)
- control flow graphs (graph coloring)

useful technique: defer register allocation until a later phase,
use an unbound set of **symbolic registers** instead

Lecture Compiler I WS 2001/2002 / Slide 110

Objectives:

Overview on register allocation

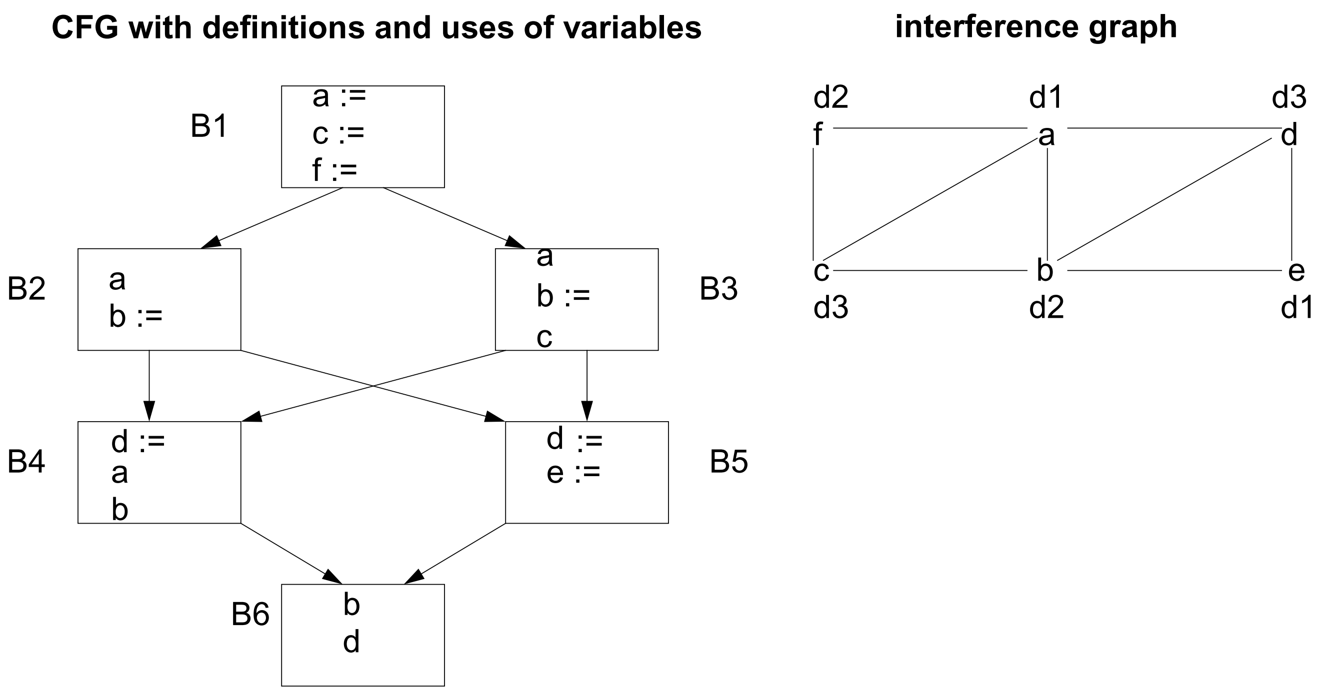
In the lecture:

Explain the use of registers for different purposes.

Suggested reading:

Kastens / Übersetzerbau, Section 7.5

Example for Graph Coloring



Lecture Compiler I WS 2001/2002 / Slide 111

Objectives:

Get an idea of register allocation by graph coloring

In the lecture:

- Explain the example.
- Refer to lecture "Modellierung" Mod-4.21

Suggested reading:

Kastens / Übersetzerbau, Section 7.5.4, Fig. 7.5-6

Assignments:

- Apply the technique for another example.

Questions:

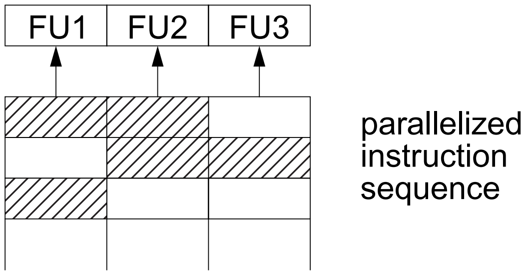
- Why is variable b in block B5 alive?

Code Parallelization

Target processor executes several instructions in parallel.
Compiler arranges instruction sequence for shortest execution time: **instruction scheduling**

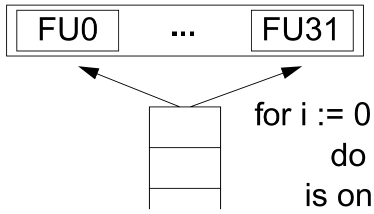
Principles of parallelism in processors:

Parallel functional units (FU)
super scalar, VLIW:



Data parallel processor
vector processor

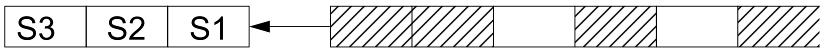
all FUs execute the same instruction on individual data (SIMD)



for i := 0 to 31
do c[i] := a[i] + b [i];
is one instruction!

Analyze and transform loops

Pipeline processor



sequential code scheduled for pipelining

Lecture Compiler I WS 2001/2002 / Slide 112

Objectives:

3 abstractions of processor parallism

In the lecture:

- Explain the abstract models,
- relate them to real processors,
- explain the instruction scheduling tasks.

Suggested reading:

Kastens / Übersetzerbau, Section 8.5

Questions:

- What has to be known about instruction execution in order to solve the instruction scheduling problem in the compiler?

Software Pipelining

Technique for parallelization of loops.

A single loop body does not exhibit enough parallelism => sparse schedule.

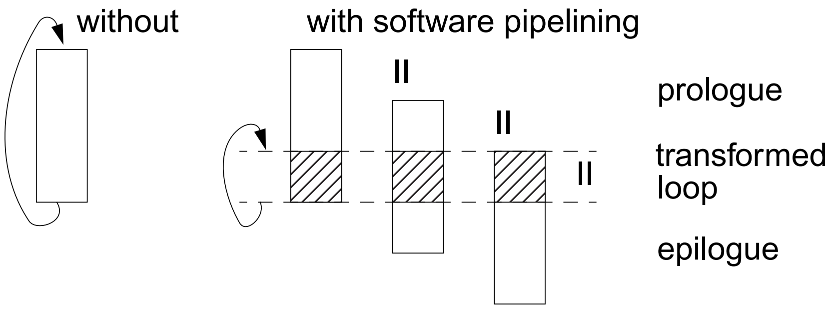
Idea of software pipelining:

transformed loop body executes several loop iterations in parallel,
iterations are shifted in time => compact schedule

Prologue, epilogue: initiation and finalization code

Technique:

1. **DDG** for loop body with dependencies into later iterations
2. Find a schedule such that iterations can begin with a **short initiation interval II**
3. Construct new loop, prologue, and epilogue



II: Initiation Interval

Lecture Compiler I WS 2001/2002 / Slide 113

Objectives:

Increase parallelism in loops

In the lecture:

- Explain the underlying idea

Questions:

Explain:

- The shorter the initiation interval is, the greater is the parallelism, and the compacter is the schedule.
- The transformed loop contains each instruction of the loop body exactly once.

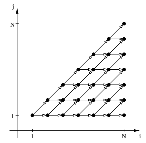
Loop Parallelization

Compilation steps:

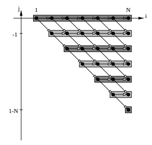
- **nested loops** operating on **arrays**,
  sequentiell execution of iteration space

```
DECLARE B[0..N,0..N+1]

FOR I := 1 ..N
    FOR J := 1 .. I
        B[I,J] :=
            B[I-1,J]+B[I-1,J-1]
    END FOR
END FOR
```

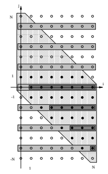
- analyze **data dependencies**
  data-flow: definition and use of array elements



- **transform loops**
  keep data dependencies intact
- **parallelize inner loop(s)**
  map onto field or vector of processors



- **map arrays onto processors**
  such that many acceses are local,
  transform index spaces



Lecture Compiler I WS 2001/2002 / Slide 114

Objectives:

Overview on regular loop parallelization

In the lecture:

Explain

- Application area: scientific computations,
- goals: execute inner loops in parallel with efficient data access,
- transformation steps.