Programming Languages and Compilers

Prof. Dr. Uwe Kastens

WS 2013 / 2014

0. Introduction

Objectives

The participants are taught to

- understand properties and notions of programming languages
- understand **fundamental techniques** of language implementation, and to use **generating tools and standard solutions**,
- apply compiler techniques for design and implementation of **specification languages and domain specific languages**

Forms of teaching:

Lectures

Tutorials **Exercises**

Homeworks **Running project**

Contents

Prerequisites

from Lecture	Topic	here needed for
Foundations of Programming Languages:		
	4 levels of language properties	Language specification, compiler tasks
	Context-free grammars	Grammar design, syntactic analysis
	Scope rules	Name analysis
	Data types	Type specification and analysis
Modeling:		
	Finite automata	Lexical analysis
	Context-free grammars	Grammar design, syntactic analysis

References

Material for this course **PLaC**: http://ag-kastens.upb.de/lehre/material/plac
for the Master course **Compilation Methods**: http://ag-kastens.upb.de/lehre/material/compii

Modellierung: http://ag-kastens.upb.de/lehre/material/model
Grundlagen der Programmiersprachen: http://ag-kastens.upb.de/lehre/material/gdp

John C. Mitchell: **Concepts in Programming Languages**, Cambridge University Press, 2003

R. W. Sebesta: **Concepts of Programming Languages**, 4. Ed., Addison-Wesley, 1999

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

A. W. Appel: **Modern Compiler Implementation in Java**, Cambridge University Press, 2nd Edition, 2002 (available for C and for ML, too)

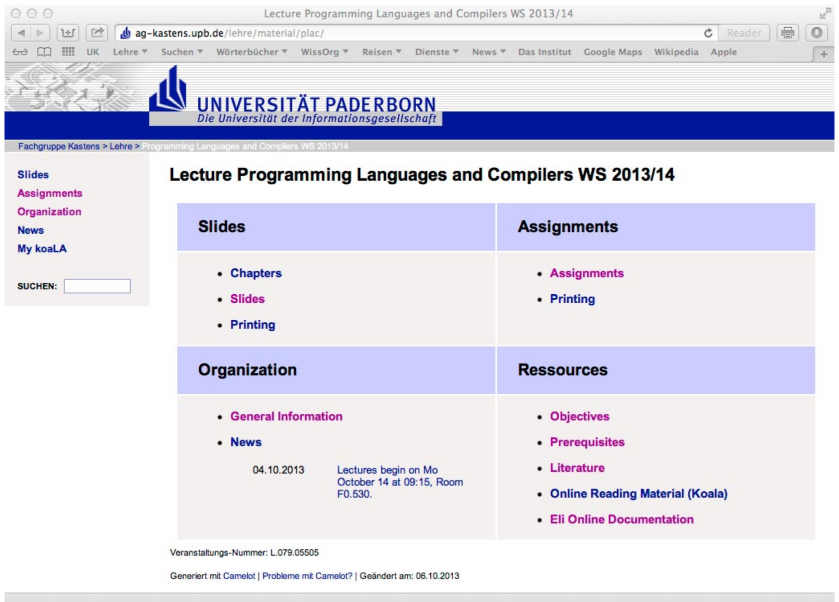
W. M. Waite, L. R. Carter: **An Introduction to Compiler Construction,** Harper Collins, New York, 1993

U. Kastens, A. M. Sloane, W. M. Waite: **Generating Software from Specifications**, Jones and Bartlett Publishers, 2007

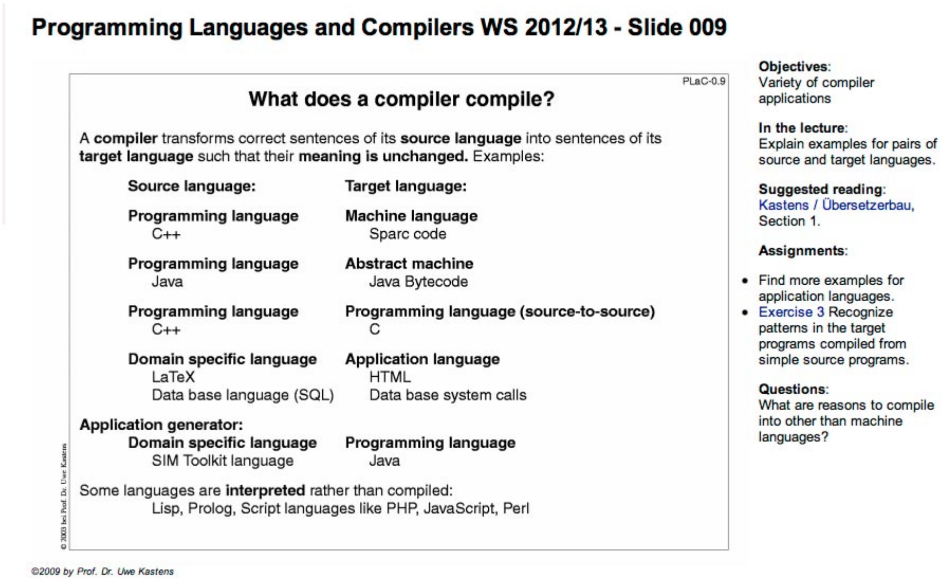
References forReading

Week	Chapter	Kastens	Waite Carter	Eli Doc.
1	0. Introduction			
2	1. Language Properties and Compiler tasks	1, 2	1.1 - 2.1	
3 - 4	2. Symbol Specification and Lexical Analysis	3	2.4 3.1 - 3.3	+
5 - 7	3. Context-free Grammars and Syntactic Analysis	4	4, 5, 6	+
8 - 10	4. Attribute Grammars and Semantic Analysis	5		+
11	5. Binding of Names	6.2	7	+
12	6. Type Specification and Analysis	(6.1)		+
13	7. Specification of Dynamic Semantics			
13	8. Source-to-Source Translation			
	9. Domain Specific Languages			

Course material in the Web



Commented slide in the course material



Organization of the course

Programming Languages and Compilers WS 2013/14 - Organization

Lecturer

Prof. Dr. Uwe Kastens:

Office Hours

- `Wed 16.00 - 17.00   F2.308`
- `Tue 11.00 - 12.00   F2.308`

Hours

Lecture

- `V2    Mo 09.15 - 10.45, F0.530`

Start date: Oct 14, 2013

Excercises

- `Ü1   Mo 11.00 - 11.45, F0.530 / F1.520`

Start date: Oct 14, 2013

Examination

Oral examinations of 20 to 30 min duration. Any topic of the lecture and of the tutorial may be subject of the exam. See also the sequence of questions in Chapter 10.

Two time spans are offered for examinations:

1. Feb 12 to 14 in 2014
2. April 01 to 03 in 2014

Register in PAUL for the one or the other time span; then ask for an appointment by email to my secretary Mrs. Gundelach (sigu@upb.de).

Assignments

- Assignments will be published every week.

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
Domain specific language LaTeX Data base language (SQL)	**Application language** HTML Data base system calls

Application generator:

Domain specific language SIM Toolkit language	**Programming language** Java

Some languages are **interpreted** rather than compiled:
Lisp, Prolog, Script languages like PHP, JavaScript, Perl

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
```


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 */
```

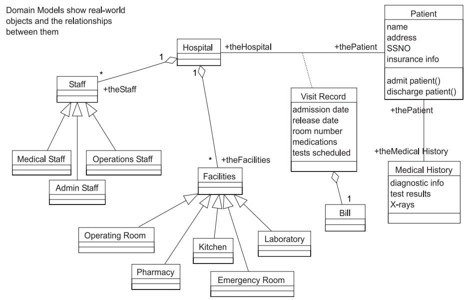
```
\documentstyle[12pt]{article}
\begin{document}
\section{Introduction}
This is a very short document.
It just shows
\begin{itemize}
\item an item, and
\item another item.
\end{itemize}
\end{document}

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

%%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
```

Languages for specification and modeling

SDL (CCITT)
Specification and Description Language:

UML
Unified Modeling 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;
```

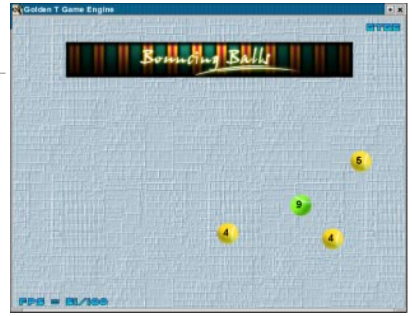


Domain Specific Languages (DSL)

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**
- **Game description language**:




```
game BBall
{  size 640 480;
   background "pics/backgroundbb.png";
   Ball einball; int ballsize;

   initial {
      ballsize=36;
   }

  events {
      pressed SPACE:
      { einball = new Ball(<100,540>, <100,380>);
```


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

Programming languages as target languages:

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

=> Compiler task: Source-to-source compilation

Semester project as running example

SetLan: A Language for Set Computation

SetLan is a domain-specific language for **programming with sets**. Constructs of the the language are dedicated to describe sets and computations using sets. The language allows to define types for sets and variables and expressions of those types. Specific loop constructs allow to iterate through sets. These constructs are embedded in a simple imperative language.

A source-to-source translator **translates SetLan programs into Java** programs.

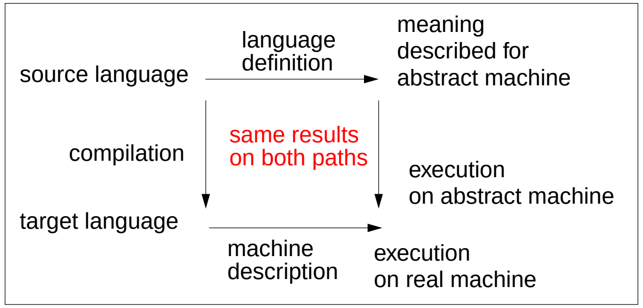
The SetLan translator is implemented using the methods and tools introduced in this course.

The participants of this course get an implementation of a **sub-language of SetLan as a starting point** for their work towards their individual extension of the language and the implementation.

```
{
   set a, b; int i;
   i = 1;
   a = [i, 3, 5];
   b = [3, 6, 8];
   print a+b; printLn;
   print a*b <= b;
   printLn;
}
```

1. Language properties - compiler tasks
Meaning preserving transformation

A **compiler** transforms **any correct sentence** of its **source language** into a sentence of its **target language** such that its **meaning is unchanged**.



A **meaning** is defined only for **all correct** programs => compiler task: error handling

Static language properties are analyzed at **compile time,** e. g. definitions of Variables, types of expressions; => determine the transformation, if the program **compilable**

Dynamic properties of the program are determined and checked at **runtime,** e. g. indexing of arrays => determine the effect, if the program **executable** (However, just-in-time compilation for Java: bytecode is compiled at runtime.)

Levels of language properties - compiler tasks

- **a. Notation of tokens** — **lexical analysis**
  keywords, identifiers, literals
  formal definition: **regular expressions**

- **b. Syntactic structure** — **syntactic analysis**
  formal definition: **context-free grammar**

- **c. Static semantics** — **semantic analysis, transformation**
  binding names to program objects, typing rules
  usually defined by informal texts,
  formal definition: **attribute grammar**

- **d. Dynamic semantics** — **transformation, code generation**
  semantics, effect of the execution of constructs
  usually defined by informal texts
  in terms of an abstract machine**,**
  formal definition: **denotational semantics**

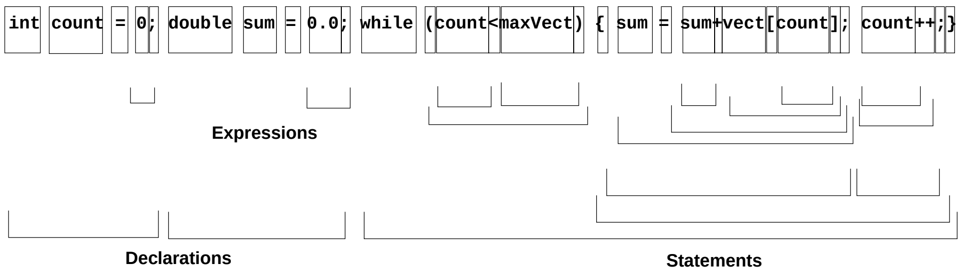
Definition of target language (target machine) — **transformation, code generation assembly**

Example: Tokens and structure

Character sequence

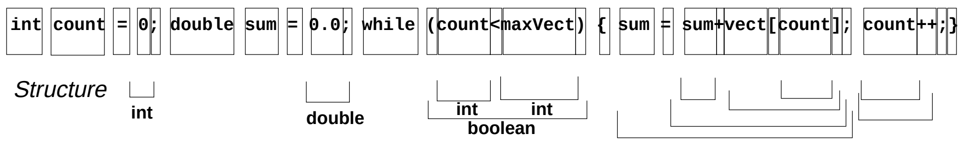
```
int count = 0; double sum = 0.0; while (count<maxVect) { sum = sum+vect[count]; count++;}
```

Tokens



Structure

Example: Names, types, generated code



k1: (count, local variable, int)
k2: (sum, local variable, double)
k3: (maxVect, member variable, int)
k4: (vect, member variable, double array)
...

Static properties: names and types

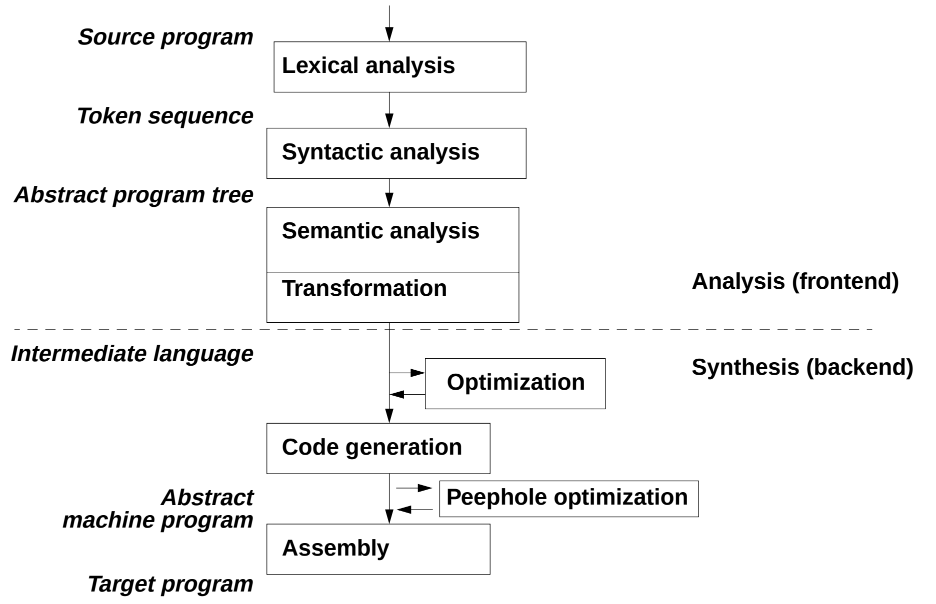
generated Bytecode

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

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

Compiler structure and interfaces



Software qualities of the compiler

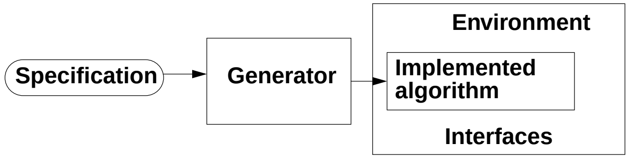
- **Correctness** Compiler translates correct programs correctly; rejects wrong programs and gives error messages
- **Efficiency** Storage and time used by the compiler
- **Code efficiency** Storage and time used by the generated code; compiler task: optimization
- **User support** Compiler task: Error handling (recognition, message, recovery)
- **Robustness** Compiler gives a reasonable reaction on every input; does not break on any program

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**

Generate from specifications

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:

Compiler Frameworks (Selection)

Amsterdam Compiler Kit: (Uni Amsterdam)
The Amsterdam Compiler Kit is fast, lightweight and retargetable compiler suite and toolchain written by Andrew Tanenbaum and Ceriel Jacobs.
Intermediate language EM, set of frontends and backends

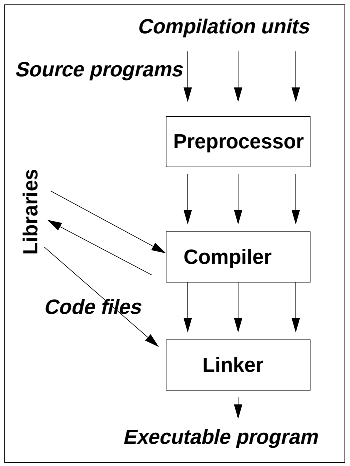
ANTLR: (Terence Parr, Uni San Francisco)
ANother Tool for Language Recognition, (formerly PCCTS) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions

CoCo: (Uni Linz)
Coco/R is a compiler generator, which takes an attributed grammar of a source language and generates a scanner and a parser for this language. The scanner works as a deterministic finite automaton. The parser uses recursive descent.

Eli: (Unis Boulder, Paderborn, Sydney)
Combines a variety of standard tools that implement powerful compiler construction strategies into a domain-specific programming environment called Eli. Using this environment, one can automatically generate complete language implementations from application-oriented specifications.

SUIF: (Uni Stanford)
The SUIF 2 compiler infrastructure project is co-funded by DARPA and NSF.
It is a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers.

Environment of compilers



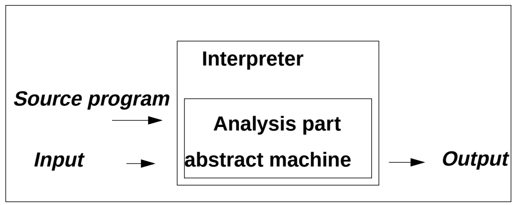
Preprocessor cpp substitutes text macros in source programs, e.g.

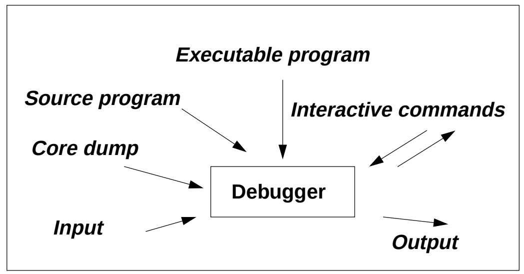
```
#include <stdio.h>
#include "module.h"

#define SIZE 32
#define SEL(ptr,fld) ((ptr)->fld)
```

Separate compilation of compilation units

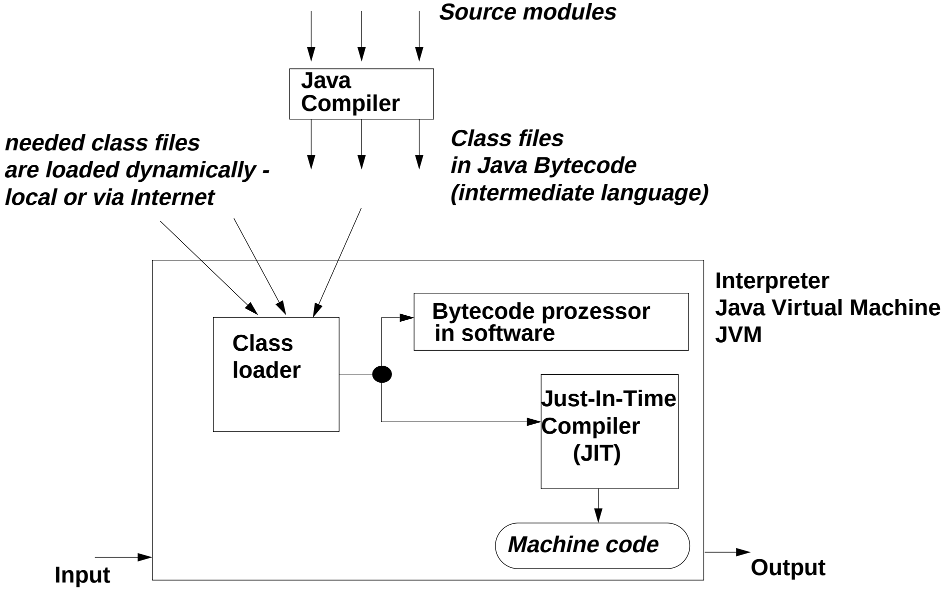
- with interface specification,
  consistency checks,
  and language specific linker:
  Modula, Ada, Java
- without ...;
  checks deferred to system linker:
  C, C++

Interpreter and Debugger





Compilation and interpretation of Java programs



2. Symbol specifications and lexical analysis

Notations of tokens is specified by regular expressions

Token classes: keywords (**`for`**, **`class`**), operators and delimiters (**`+`**, **`==`**, **`;`**, **`{`**), identifiers (**`getSize`**, **`maxint`**), literals (**`42`**, **`'\n'`**)

Lexical analysis isolates tokens within a stream of characters and encodes them:

Tokens

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

Lexical Analysis

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

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

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

Avoid context dependent token specifications

Tokens should be **recognized in isolation**:

e. G. all occurrences of the identifier **`a`** get the same encoding:

`{int a; ... a = 5; ... {float a; ... a = 3.1; ...}}`

distinction of the two different variables would require information from semantic analysis

typedef problem in C:

The C syntax requires **lexical distinction** of type-names and other names:

`typedef int *T; T (*B); X (*Y);`

cause syntactically different structures: declaration of variable **`B`** and call of function **`X`**.
Requires feedback from semantic analysis to lexical analysis.

Identifiers in PL/1 may **coincide with keywords**:

`if` `if = then` **`then`** `then := else` **`else`** `else := then`

Lexical analysis needs feedback from syntactic analysis to distinguish them.

Token separation in FORTRAN:

„Deletion or insertion of blanks does not change the meaning.“

`DO 24 K = 1,5` begin of a loop, 7 tokens

`DO 24 K = 1.5` assignment to the variable **`DO24K`**, 3 tokens

Token separation is determined late.

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

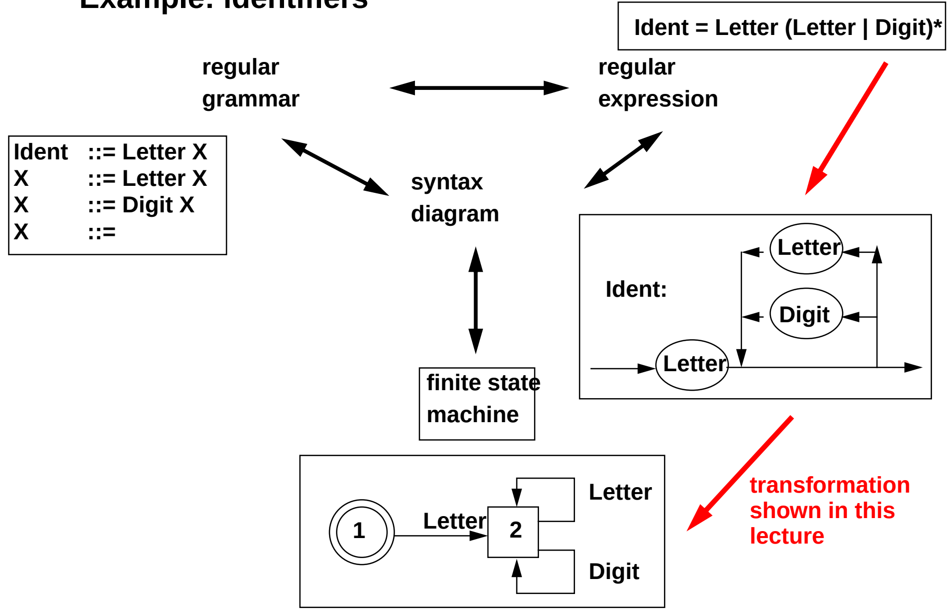
Examples:

```
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

Specification of token notations

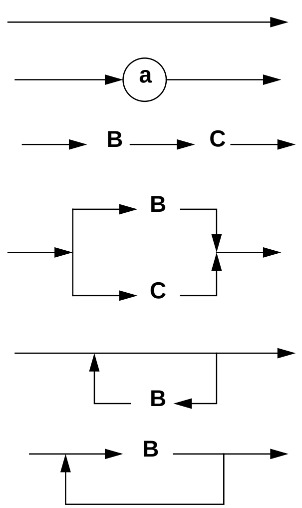
Example: identifiers



Regular expressions mapped to syntax diagrams

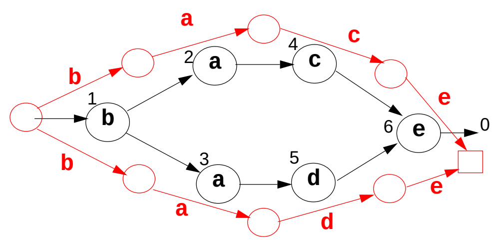
Transformation rules:

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**

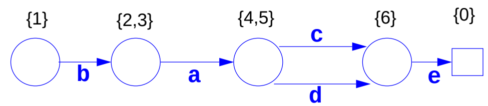


Naive transformation

1. Transform a **syntax diagram into a non-det. FSM** by naively exchanging nodes and arcs



2. Transform a non-det. FSM into a det. FSM:
   Merge equivalents sets of nodes into nodes.



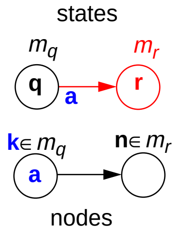
Syntax diagram	**deterministic finite state machine**
set of nodes m_q	state q
sets of nodes m_q and m_r connected with the same character a	transition q ---> r with character a

Construction of deterministic finite state machines

Syntax diagram	**deterministic finite state machine**
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:

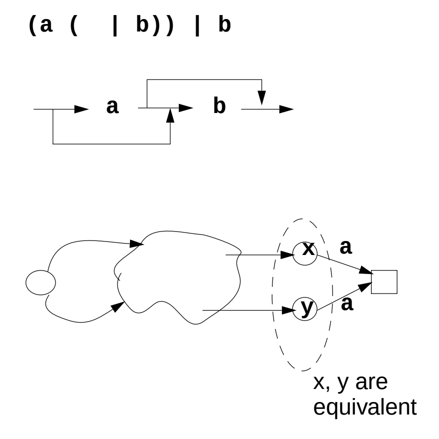
1. **enumerate nodes**; exit of the diagram gets the number *0*
2. **initial set of nodes** m_1 contains all nodes that are reachable from the begin of the diagram; m_1 represents the **initial state 1**.
3. **construct new sets of nodes (states)** and **transitions:**
   - chose state *q* with m_q, chose a character *a*
   - consider the set of nodes with character *a*, s.t. their labels *k* are in m_q.
   - consider all nodes that are directly reachable from those nodes; let m_r be the set of their labels
   - create a state *r* for m_r and a transition from *q* to *r* under *a*.



4. **repeat step 3** until no new states or transitions can be created
5. a state *q* is a **final state** iff *0* is in m_q.

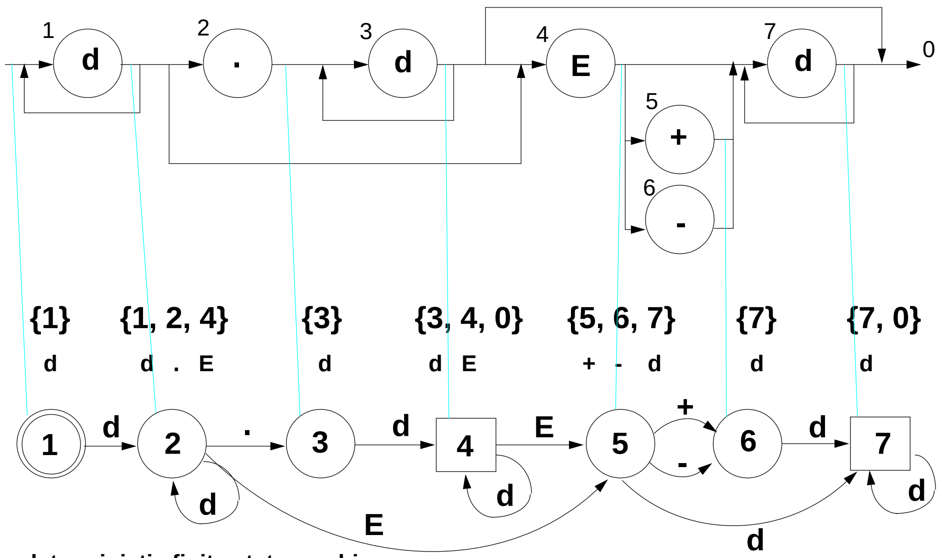
Properties of the transformation

1. **Syntax diagrams** can express languages **more compact** than regular expressions can:
   A regular expression for { a, ab, b} needs more than one occurrence of a or b - a syntax diagram doesn't.
2. The FSM resulting from a transformation of PLaC 2.7a may have m**ore states than necessary**.
3. There are transformations that **minimize the number of states** of any FSM.



Example: Floating point numbers in Pascal

Syntax diagram

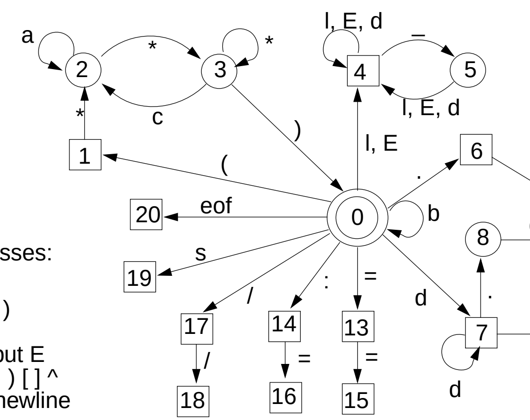


deterministic finite state machine

Composition of token automata

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

- **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)



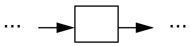
Example: tokens of Lax
[Waite, Goos:
Compiler Construction]

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

Rule of the longest match

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

When does the automaton stop?



Rule of the longest match:

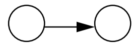
- **The automaton continues as long as there is a transition with the next character.**
- **After having stopped it sets back to the most recently passed final state.**
- **If no final state has been passed an error message is issued.**

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

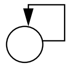
Check the concrete grammar for tokens that may occur adjacent!

Scanner: Aspects of implementation

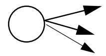
- **Runtime is proportional to the number of characters in the program**
- **Operations per character must be fast** - otherwise the Scanner dominates compilation time
- **Table driven** automata are too **slow:**
  Loop interprets table, 2-dimensional array access, branches
- **Directly programmed**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



sequence



repeat loop



branch, switch

- **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.

Characteristics of Input Data

Table 7
Characteristics of the Input Data

	P4		SYNPUT	
	Occurrences	Characters	Occurrences	Characters
Single spaces	11404	11404	2766	2766
Identifiers	8411	41560	5799	22744
Keywords	4183	15080	2034	7674
>3 spaces	3850	60694	1837	19880
;	2708	2708	1880	1880
:=	1379	2758	966	1932
Integers	1354	2202	527	573
(	1245	1245	751	751
)	1245	1245	751	751
,	1032	1032	842	842
comments	659	13765	675	35066
[	654	654	218	218
]	654	654	218	218
:	635	635	483	483
.	546	546	400	400
Strings	493	2560	303	3017
Space pairs	470	940	39	78
=	438	438	206	206
^	353	353	461	461
<>	213	426	96	192
+	203	203	183	183
-	82	82	61	61
Space triples	56	168	842	2526
..	37	74	21	42
<=	26	52	5	10
>	18	18	27	27
<	14	14	25	25
*	10	10	12	12
>=	5	10	7	14
Reals	0	0	3	14
/	0	0	1	1

significant numbers of characters

W. M. Waite:
The Cost of Lexical Analysis.
Software- Practice and Experience,
16(5):473-488, May 1986.

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

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

3. Context-free Grammars and Syntactic Analysis

Input: **token sequence**

Tasks:

Parsing: construct a derivation according to the **concrete syntax**,
Tree construction: build a structure tree according to the **abstract syntax**,
Error handling: detection of an error, message, recovery

Result: **abstract program tree**

Compiler module parser:

deterministic stack automaton, augmented by actions for tree construction

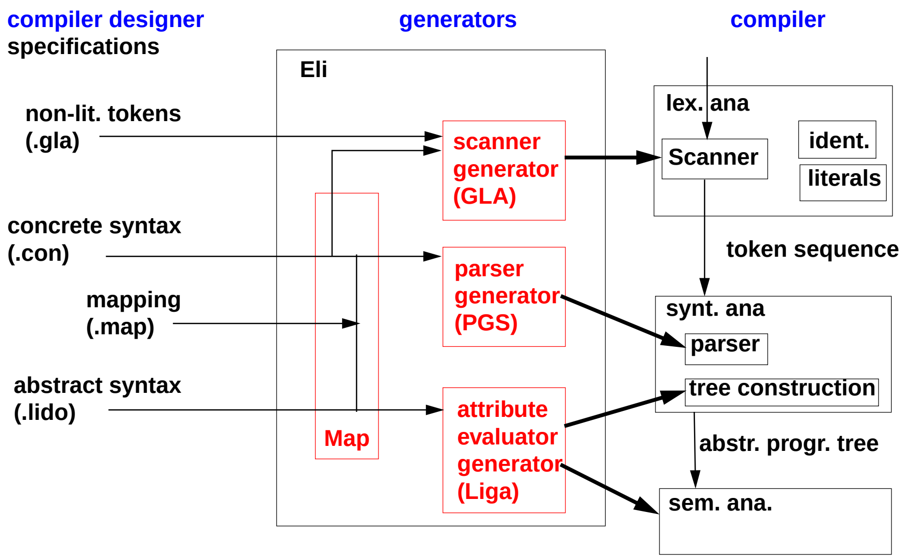
top-down parsers: leftmost derivation; tree construction top-down or bottom-up

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.

Generating the structuring phase from specifications (Eli)



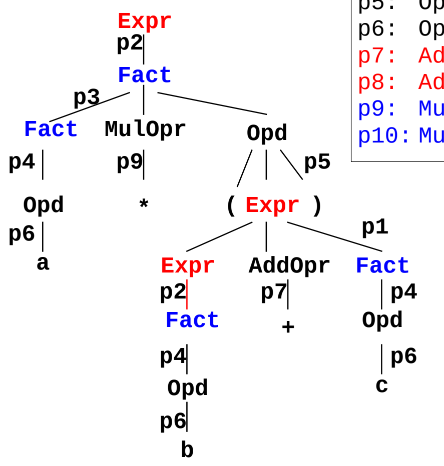
3.1 Concrete and abstract syntax

concrete syntax	**abstract syntax**
- context-free grammar	- context-free grammar
- defines the structure of source programs	- defines abstract program trees
- is unambiguous	- is usually ambiguous
- specifies derivation and parser	- translation phase is based on it
- parser actions specify the tree construction --->	- tree construction
- some chain productions have only syntactic purpose	
`Expr ::= Fact` have no action	no node created
- symbols are mapped `{Expr,Fact}` ->	to one abstract symbol `Exp`
- same action at structural equivalent productions:	- creates tree nodes
`Expr ::= Expr AddOpr Fact &BinEx`	
`Fact ::= Fact MulOpr Opd &BinEx`	
- semantically relevant chain productions, e.g. `ParameterDecl ::= Declaration`	- are kept (tree node is created)
- terminal symbols identifiers, literals, keywords, special symbols	- only semantically relevant ones are kept identifiers, literals
- concrete syntax and symbol mapping specify	- abstract syntax (can be generated)

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)`



`+, - lower precedence`
`*, / higher precedence`

Patterns for expression grammars

Expression grammars are **systematically** constructed, such that **structural properties** of expressions are defined:

one level of precedence, **binary** operator, left-associative:

```
A ::= A Opr B
A ::= B
```

one level of precedence, **binary** operator, right-associative:

```
A ::= B Opr A
A ::= B
```

one level of precedence, **unary Operator**, prefix:

```
A ::= Opr A
A ::= B
```

one level of precedence, **unary Operator**, postfix:

```
A ::= A Opr
A ::= B
```

Elementary operands: **only derived** from the nonterminal of the **highest precedence** level (be H here):

```
H ::= Ident
```

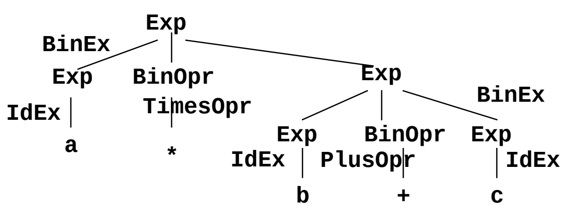
Expressions in parentheses: only derived from the nonterminal of the **highest precedence** level (assumed to be H here); **contain** the nonterminal of the **lowest precedence level** (be A here):

```
H ::= '(' A ')'
```


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

3.2 Design of concrete grammars

Objectives

The concrete grammar 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 provably related to the **documented grammar**;
- can be **mapped to** a suitable **abstract grammar**.

A strategy for grammar development

1. **Examples**: Write at least one example for every intended language construct. Keep the examples for checking the grammar and the parser.
2. **Sub-grammars**: Decompose a non-trivial task into topics covered by sub-gammars, e.g. statements, declarations, expressions, over-all structure.
3. **Top-down**: Begin with the start symbol of the (sub-)grammar, and refine each nonterminal according to steps 4 - 7 until all nonterminals of the (sub-)grammar are refined.
4. **Alternatives**: Check whether the language construct represented by the current nonterminal, say Statement, shall occur in structurally different alternatives, e.g. while-statement, if-statement, assignment. Either introduce chain productions, like
   `Statement ::= WhileStatement | IfStatement | Assignment.`
   or apply steps 5 - 7 for each alternative separately.
5. **Consists of**: For each (alternative of a) nonterminal representing a language construct explain its immediate structure in words, e.g. „A Block is a declaration sequence followed by a statement sequence, both enclosed in curly braces.“ Refine only one structural level. Translate the description into a production. If a sub-structure is not yet specified introduce a new nonterminal with a speaking name for it, e.g.
   `Block ::= '{' DeclarationSeq StatementSeq '}'.`
6. **Natural structure**: Make sure that step 5 yields a „natural“ structure, which supports notions used for static or dynamic semantics, e.g. a range for valid bindings.
7. **Useful patterns**: In step 5 apply patterns for description of sequences, expressions, etc.

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 all modifications**.
  (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**

Grammar design together with language design

Read grammars before writing a new grammar.

Apply **grammar patterns systematically** (cf. GPS-2.5, GPS-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

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 unreasonably complex**:
  e. g. distinction of compile-time expressions from ordinary expressions requires duplication of the expression syntax.

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

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.

3.3 Recursive descent parser

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

Systematic 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 p_i	decision for branch p_i
non-terminal occurrence X ::= ... Y ...	function call Y()
terminal occurrence X ::= ... t ...	accept a token t and read the next token

Productions for `Stmt`:

```
p1: Stmt ::=
     Variable ':=' Expr

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

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


Grammar conditions for recursive descent

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

DecisionSet (A ::= u) $\cap$ **DecisionSet (A ::= v) =** $\emptyset$

with

DecisionSet (A ::= u) := if nullable (u) then **First (u)** $\cup$ **Follow (A)** else **First (u)**

nullable (u) holds iff a derivation u $\Rightarrow^*$ ε exists

First (u) := { t ∈ T | v ∈ V* exists and a derivation u $\Rightarrow^*$ t v }

Follow (A):= { t ∈ T | u,v ∈ V* exist, A ∈ N and a derivation S $\Rightarrow^*$ u A t v }

Example:

	production	DecisionSet
p1:	Prog ::= Block #	**begin**
p2:	Block ::= **begin** Decls Stmts **end**	**begin**
p3:	Decls ::= Decl **;** Decls	**new**
p4:	Decls ::=	Ident **begin**
p5:	Decl ::= **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	**new**	Ident **begin**
Decl	**new**	**;**
Stmts	**begin** Ident	**; end**
Stmt	**begin** Ident	**; end**

Computation rules for nullable, First, and Follow

Definitions:

nullable(u) holds iff a derivation u ⇒* ε exists

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

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

with G = (T, N, P, S); V = T ∪ N; t ∈ T; A ∈ N; u,v ∈ V*

Computation rules:

nullable(ε) = true; nullable(t) = false; nullable(uv) = nullable(u) ∧ nullable(v);
nullable(A) = true iff ∃ A::=u ∈ P ∧ nullable(u)

First(ε) = ∅; First(t) = {t};
First(uv) = if nullable(u) then First(u) ∪ First(v) else First(u)
First(A) = First(u_1) ∪...∪ First(u_n) for all A::=u_i ∈ P

Follow(A):
 if A=S then # ∈ Follow(A)
 if Y::=uAv ∈ P then First(v) ⊆ Follow(A) and if nullable(v) then Follow(Y) ⊆ Follow(A)

Grammar transformations for LL(1)

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

Simple **grammar transformations** that keep the defined **language invariant**:

- **alternative productions that begin with the same symbols:**

left-factorization:

non-LL(1) productions	transformed
A ::= v u A ::= v w	A ::= v X X ::= u X ::= w

- **productions that are directly or indirectly left-recursive:**

elimination of direct recursion:

A ::= A u A ::= v	A ::= v X X ::= u X X ::=

special case empty v:

A ::= A u A ::=	A ::= u A A ::=

u, v, w ∈ V*
X ∈ N does not occur in the original grammar

LL(1) extension for EBNF constructs

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

EBNF construct:	**Option [u]**	**Repetition (u)***
Production:	A ::= v [u] w	A ::= v (u)* w

additional LL(1)-condition:

if nullable(w)
then First(u) ∩ (First(w) ∪ Follow(A)) = ∅
else First(u) ∩ First(w) = ∅

in recursive descent parser:

Option	Repetition
v if (CurrToken in First(u)) { u } w	v while (CurrToken in First(u)) { u } w

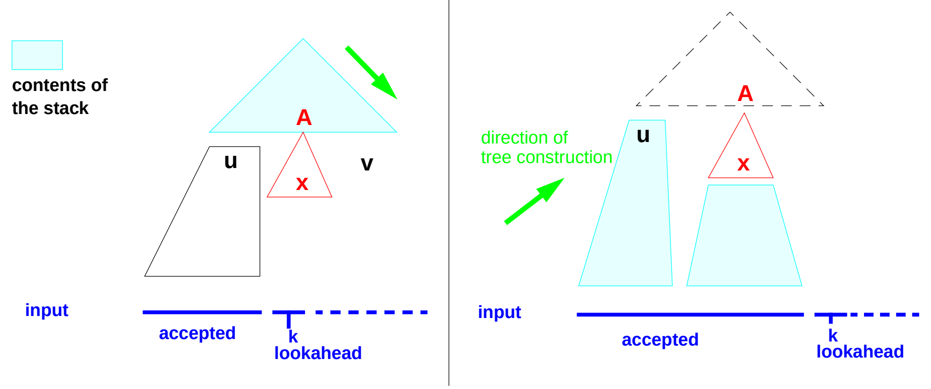
Repetition (u)+ left as exercise

Comparison: top-down vs. bottom-up

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

top-down, predictive
leftmost derivation

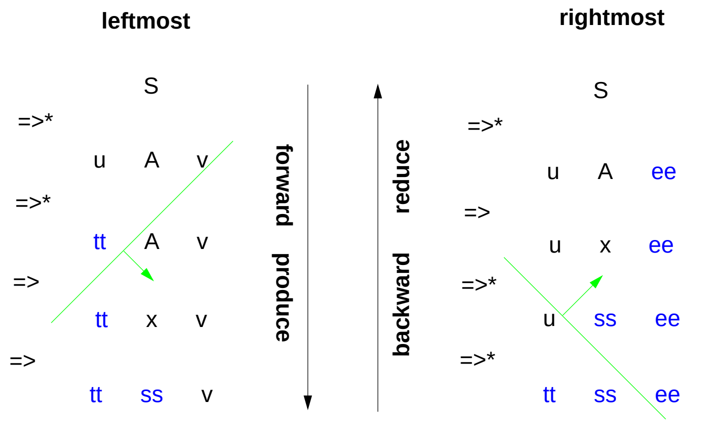
bottom-up
rightmost derivation backwards



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**.

Leftmost and rightmost derivations



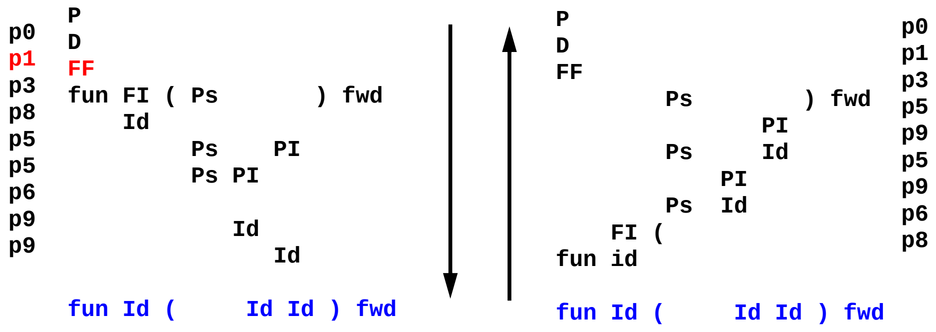
u, v, x ∈ V*

tt, ss, ee ∈ T*

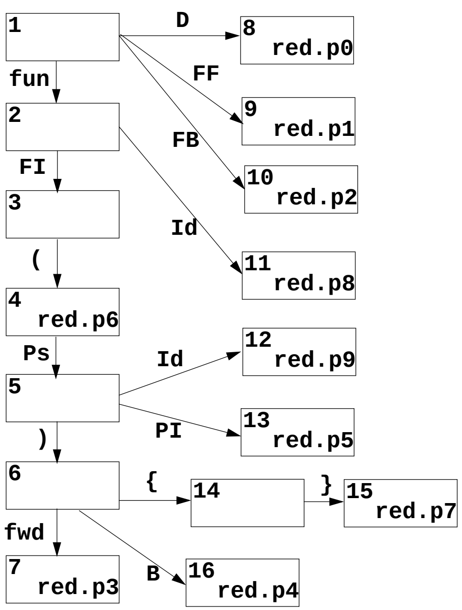
A ∈ N

Derivation tree: top-down vs. bottom-up construction

```
p0:  P  ::= D
P1:  D  ::= FF
P2:  D  ::= FB
P3:  FF ::= 'fun' FI '(' Ps ')' 'fwd'
P4:  FB ::= 'fun' FI '(' Ps ')' B
P5:  Ps ::= Ps PI
P6:  Ps ::=
p7:  B  ::= '{' '}'
p8:  FI ::= Id
p9:  PI ::= Id
```



LR(0) -Automaton



reduction	stack	input
	1	fun Id(Id Id)fwd
	1 2	Id(Id Id)fwd
	1 2 11	(Id Id)fwd
p8		
	1 2 3	(Id Id)fwd
	1 2 3 4	Id Id)fwd
p6		
	1 2 3 4 5	Id Id)fwd
	1 2 3 4 5 12	Id)fwd
p9		
	1 2 3 4 5 13	Id)fwd
p5		
	1 2 3 4 5	Id)fwd
	1 2 3 4 5 12	)fwd
p9		
	1 2 3 4 5 13	)fwd
p5		
	1 2 3 4 5	)fwd
	1 2 3 4 5 6	fwd
	1 2 3 4 5 6 7	#
p3		
	1 9	#
p1		
	1 8	#
p0		

3.4 LR parsing

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).**

Comparison of LL and LR states:

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** (see next slide).

Each **state** of an automaton represents **LL: one item** **LR: a set of items**
An LL item corresponds to a position in a case branch of a recursive function.

LR(1) items

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

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

. marks the position of analysis: *accepted and reduced* . *to be accepted*

R expected **right context:**
a **set of terminals** which may follow in the input
when the complete production is accepted.
(general k>1: R contains sequences of terminals not longer than k)

Items can distinguish different right contexts: [A ::= u . v R] and [A ::= u . v R']

Reduce item:

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

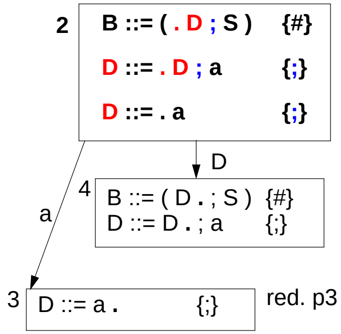
characterizes a reduction using this production if the next input token is in R.

The automaton uses R only for the decision on reductions!

A **state** of an LR automaton represents **a set of items**

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.



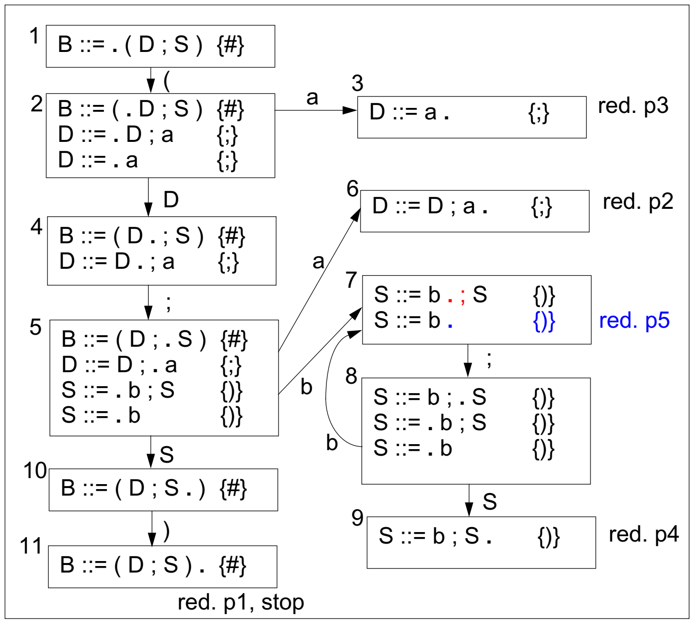
A **shift transition** is made under
a **token read** from input or
a **non-terminal** symbol
obtained from a **preceding reduction.**
The state is pushed.

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

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



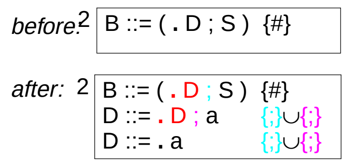
In state 7 a decision is required on next input:
- if ; then shift
- if) then reduce p5

In states 3, 6, 9, 11 a decision is not required:
- reduce on any input

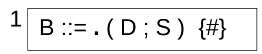
Construction of LR(1) automata

Algorithm: 1. Create the start state.
2. For each created state compute the transitive closure of its items.
3. Create transitions and successor states as long as new ones can be created.

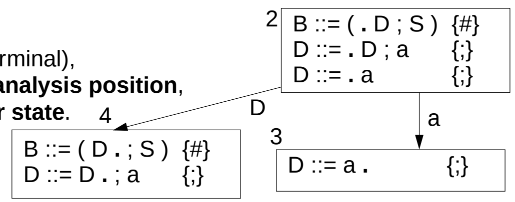
Transitive closure is to be applied to each state q:
Consider all items in q with the analysis position before a non-terminal B:
[A_1 ::= u_1 . B v_1 R_1] ... [A_n ::= u_n . B v_n R_n],
then for each production **B ::= w**
[B ::= . w   First (v_1 R_1)∪...∪First (v_n R_n)]
has to be added to state q.



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



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 corresponding items
with the **analysis position**
advanced behind the x occurrence.



Operations of LR(1) automata

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:
reduce start production,
see # in the input

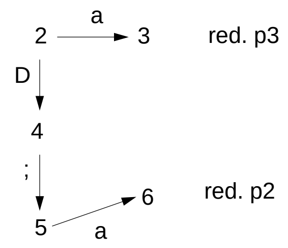
Example:

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	#	

Left recursion versus right recursion

left recursive productions:

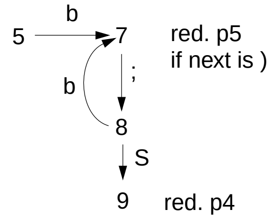
p2: D ::= D ; a
p3: D ::= a



reduction immediately after
each **; a** is accepted

right recursive productions:

p4: S ::= b ; S
p5: S ::= b



the states for all **; b** are
pushed before the first reduction

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 ∈ **R2**

Shift-reduce conflict for „dangling else" ambiguity

State 1:

S ::= **.** Stmt	{#}
Stmt ::= **.** if ... then Stmt	{#}
Stmt ::= **.** if ... then Stmt else Stmt	{#}
Stmt ::= **.** a	{#}

Transitions: Stmt →, a →, if ... then ↓

State 3:

Stmt ::= if ... then **.** Stmt	{#}
Stmt ::= if ... then **.** Stmt else Stmt	{#}
Stmt ::= **.** if ... then Stmt	{# else}
Stmt ::= **.** if ... then Stmt else Stmt	{# else}
Stmt ::= **.** a	{# else}

Transitions: Stmt →, a →, if ... then ↓

State 5:

Stmt ::= if ... then **.** Stmt	{# else}
Stmt ::= if ... then **.** Stmt else Stmt	{# else}
Stmt ::= **.** if ... then Stmt	{# else}
Stmt ::= **.** if ... then Stmt else Stmt	{# else}
Stmt ::= **.** a	{# else}

Transitions: if →, a →, Stmt ↓

State 6:

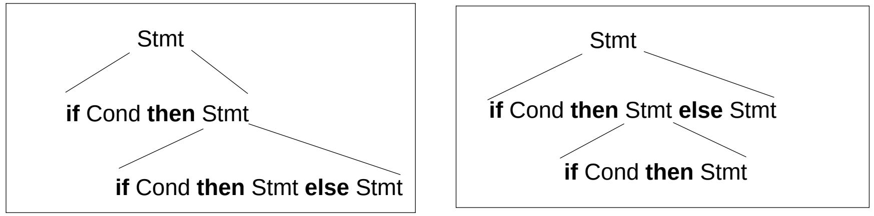
Stmt ::= if ... then Stmt **.**	{# else}
Stmt ::= if ... then Stmt **.** else Stmt	{# else}

Transition: else →

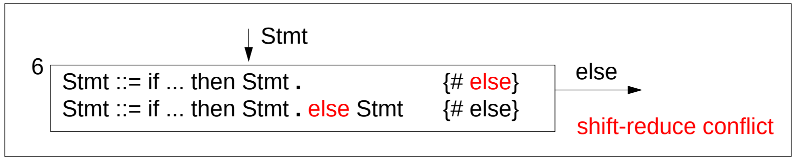
shift-reduce conflict

Decision of ambiguity

dangling else ambiguity:



desired solution for Pascal, C, C++, Java



State 6 of the automaton can be modified such that
an input token **else is shifted** (instead of causing a reduction);
yields the desired behaviour.

Some parser generators allow such modifications.

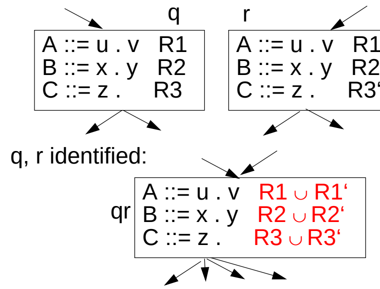
Simplified LR grammar classes

LR(1):
too many states for practical use**,** because right-contexts distinguish many states.
Strategy: simplify right-contexts sets; **fewer states**; grammar classes less powerful

LALR(1):
construct LR(1) automaton,
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)



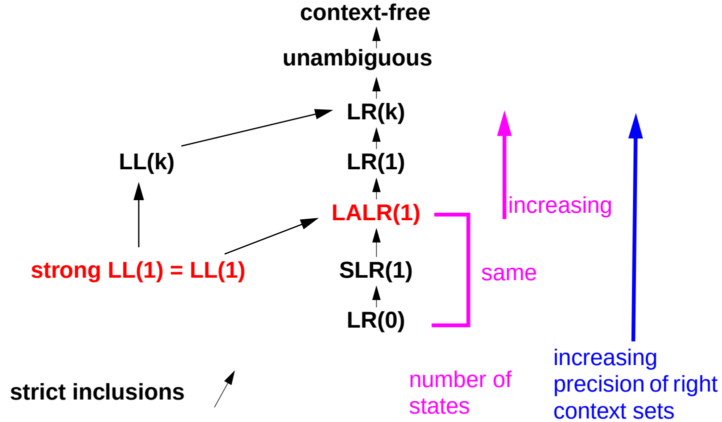
SLR(1):
LR(0) states; in reduce items
use larger right-context sets for decision:
[A ::= u . Follow (A)]

A ::= u . v
B ::= x . y
C ::= z .   **Follow(C)**

LR(0):
all items **without right-context**
Consequence: reduce items only in singleton sets

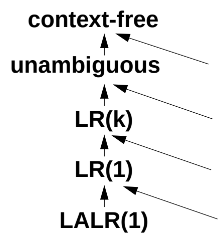
C ::= z .

Hierarchy of grammar classes



Reasons for LALR(1) conflicts

Grammar condition does not hold:



ambiguous — most cases

unbounded lookahead needed

fixed length lookahead > 1 needed

merge of LR(1) states introduces **conflicts** — rare cases

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

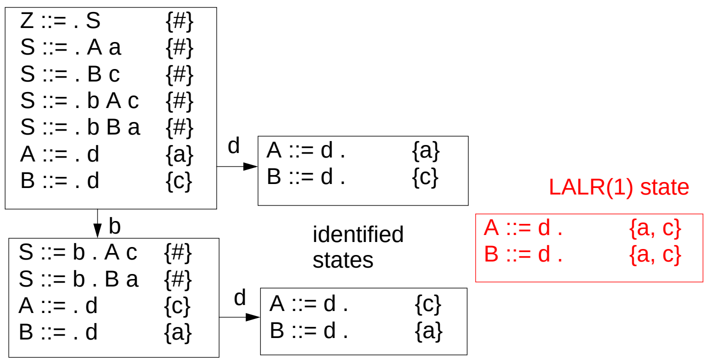
LR(1) but not LALR(1)

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.

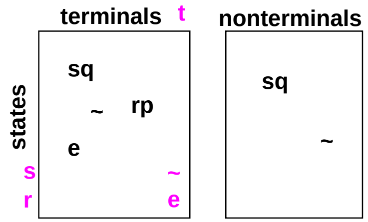
LR(1) states



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

Table driven implementation of LR automata

LR parser tables



sq: shift into state q

rp: reduce production p

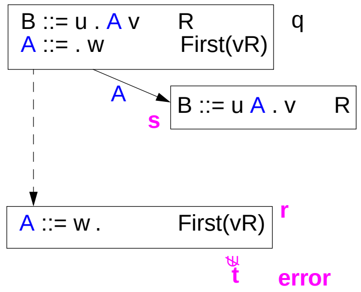
e: error

~: not reachable
don't care

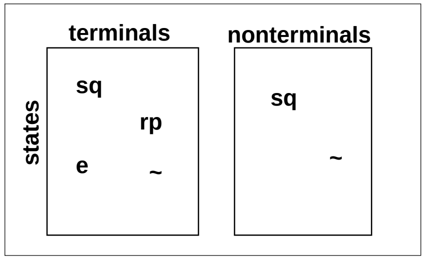
nonterminal table

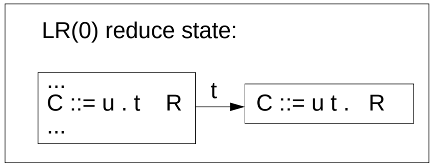
- has **no reduce entries** and **no error entries** (only **shift** and **don't-care** entries)
  reason:
  a reduction to A reaches a state from where a shift under A exists (by construction)

unreachable entries in terminal table:
if t is erroneus input in state r, then state s will not be reached with input t



Implementation of LR automata





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.

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

3.5 Syntax Error Handling
General criteria

- **recognize error as early as possible**
  LL and LR can do that:
  no transitions after error position
- **report the symptom in terms of the source text**
  rather than in terms of the state of the parser
- **continue parsing short after the error position**
  analyze as much as possible
- **avoid avalanche errors**
- **build a tree that has a correct structure**
  later phases must not break
- **do not backtrack, do not undo actions,**
  not possible for semantic actions
- **no runtime penalty for correct programs**

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;}`

(w = `int compute (int i) { a = i *`, t = `/`)

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

Error recovery

Continuation point:
A 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 $d \in T$ and w, y, v, $z \in T^*$.

error position
w t x =
w y d z
w v d z
continuation

Examples:

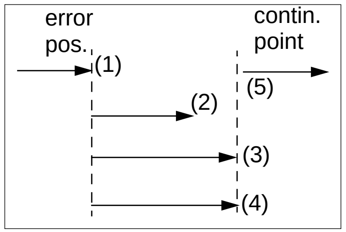
w y d z	w yd z	w y d z
`a = i * / c;...`	`a = i * / c;...`	`a = i * / c;...`
`a = i *   c;...`	`a = i *e/ c;...`	`a = i * e  ;...`
delete `/`	**insert** error identifier e	**delete** `/ c` and **insert** error id. **e**

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 D of tokens as potential continuation points.

Steps of the method:

1. **Save the contents of the parse stack** when an error is recognized.
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 deletion/insertion of elements in D.

4. Attribute grammars and semantic analysis

Input: **abstract program tree**

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

Output: **attributed program tree**

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: a **tree walking algorithm** that calls functions of semantic modules **in specified contexts** and in an **admissible order**

4.1 Attribute grammars

Attribute grammar (AG): specifies **dependent computations in abstract program trees; declarative**: explicitly specified dependences only; a suitable order of execution is computed

Computations solve the tasks of semantic analysis (and transformation)

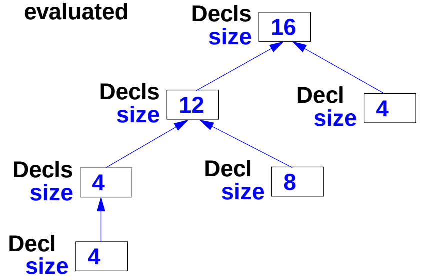
Generator produces **a plan for tree walks**
that execute calls of the computations,
such that the specified dependences are obeyed,
computed values are propagated through the tree

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

Example: AG specifies size of declarations

```
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 = Type.size;
END;
```

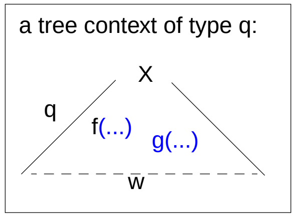
tree with dependent attributes evaluated



Basic concepts of attribute grammars (1)

An AG specifies **computations in trees** expressed by **computations associated to productions** of the abstract syntax

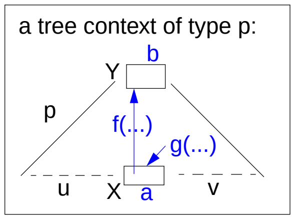
```
RULE q: X ::= w COMPUTE
   f(...); g(...);
END;
```



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

An AG specifies **dependences between computations**: expressed by **attributes associated to grammar symbols**

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

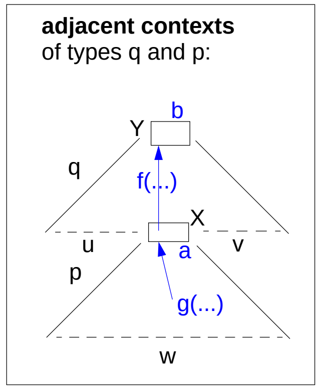


Attributes represent: **properties of symbols** and
pre- and post-conditions of computations:
post-condition = f (pre-condition)
f(X.a) uses the result of g(...); hence
X.a = g(...) is specified to be executed before f(X.a)

Basic concepts of attribute grammars (2)

dependent computations in adjacent contexts:

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



attributes may specify
dependences without propagating any value;
specifies the order of effects of computations:

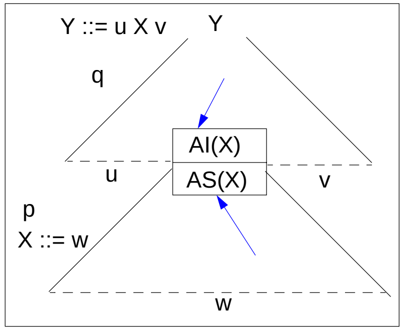
```
X.GotType = ResetTypeOf(...);
Y.Type = GetTypeOf(...) <- X.GotType;
```

`ResetTypeOf` will be called before `GetTypeOf`

Definition of attribute grammars

An **attribute grammar** AG = (G, A, C) is defined by

- a **context-free grammar G** (abstract syntax)
- for each **symbol X** of G a set of **attributes A(X)**, written X.a if a ∈ 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: Y::= ... X... has exactly one computation
for each attribute of AS(Y), for the symbol on the left-hand side of p, and
for each attribute of AI(X), for each symbol occurrence on the right-hand side of p

AG Example: Compute expression values

The AG specifies: The value of each expression is computed and printed at the root:

```
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;
```

A (Expr) = AS(Expr) = {value}

AS(Opr) = {value}
AI(Opr) = {left, right}

A(Opr) = {value, left, right}

AG Binary numbers

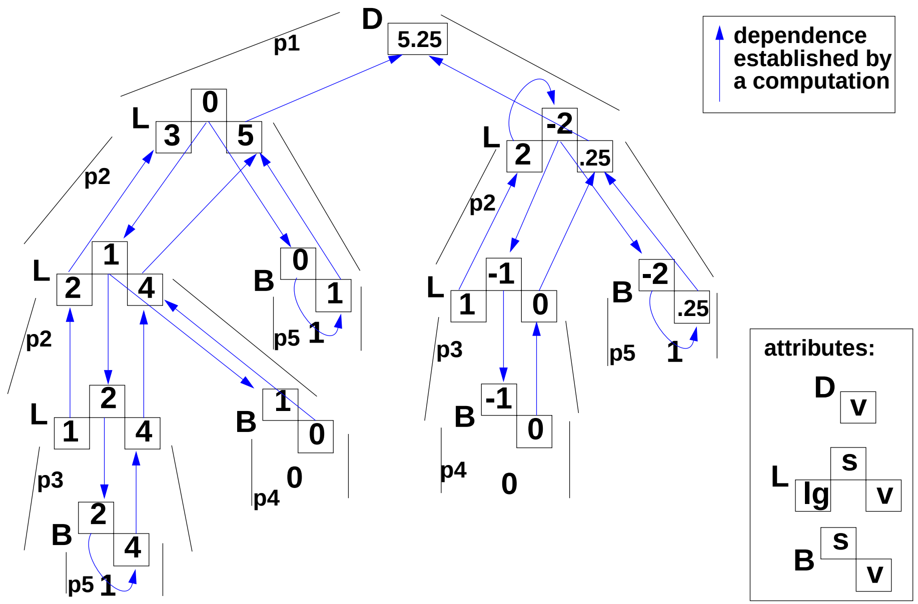
Attributes:	`L.v, B.v`	value
	`L.lg`	number of digits in the sequence L
	`L.s, B.s`	scaling of B or the least significant digit of L

```
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;
```

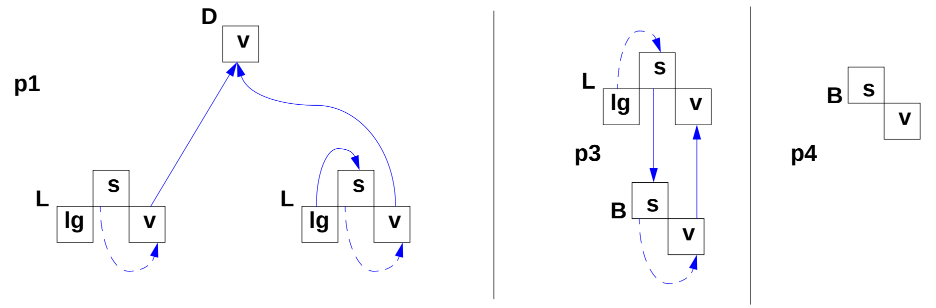
scaled binary value:

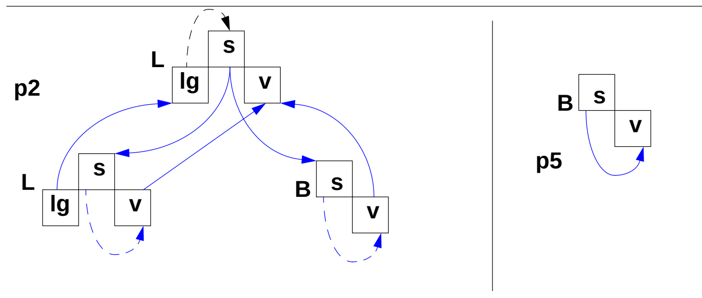
$B.v = 1 * 2^{B.s}$

An attributed tree for AG Binary numbers



Dependence graphs for AG Binary numbers





If a tree exists, that has a path from X.a to X.b at some node of Type X, the graphs have an **indirect dependence**

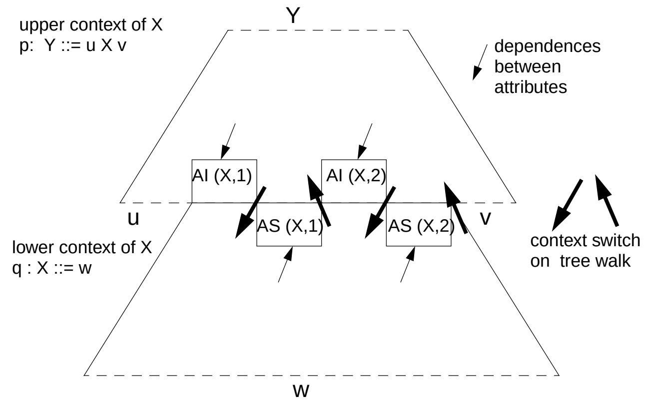
X.a - - - -► X.b

Attribute partitions

The sets AI(X) and AS(X) are **partitioned** each 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 dependences that contradict the evaluation order** of the sequence of sets:AI (X, 1), AS (X, 1), ..., AI (X, k), AS (X, k)

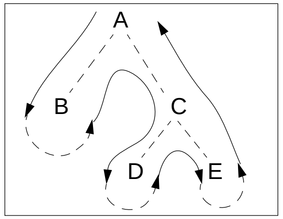
Construction of attribute evaluators

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** in visited contexts.
- The execution order obeys the **attribute dependences**.

Pass-oriented strategies for the tree walk: **AG class:**

k times **depth-first left-to-right**	**LAG (k)**
k times depth-first right-to-left	RAG (k)
alternatingly left-to-right / right-to left	**AAG (k)**
once **bottom-up (synth. attributes only)**	**SAG**

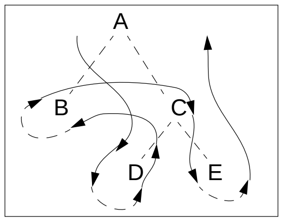


AG is checked if attribute dependences
fit to desired pass-oriented strategy; see LAG(k) check.

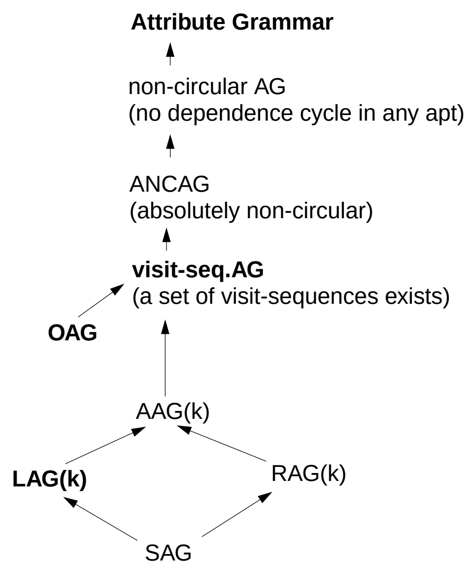
non-pass-oriented strategies:

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

A generator fits the plans to the dependences of the AG.



Hierarchy of AG classes



Visit-sequences

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

$p: X_0 ::= X_1 \dots X_i \dots X_n$

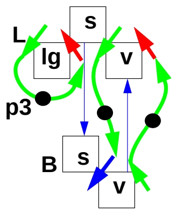
A visit-sequence is a **sequence of operations**:

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

Example out of the AG for binary numbers:

vs_{p3}: L ::= B

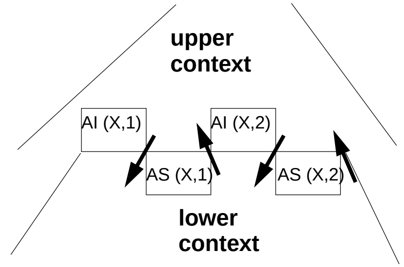
L.lg=1; ↑1; B.s=L.s; ↓B,1; L.v=B.v; ↑2



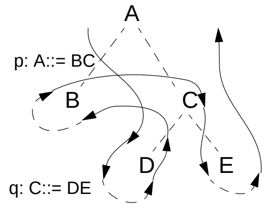
Interleaving of visit-sequences

Visit-sequences for adjacent contexts are executed interleaved.

The **attribute partition** of the common nonterminal specifies the **interface** between the upper and lower visit-sequence:



Example in the tree:



interleaved visit-sequences:

vs_p: ... ↓C,1 ...↓B,1 ...↓C,2 ...↑1

vs_q: ... ↓D,1 ... ↑1 ... ↓E,1 ... ↑2

Implementation:one **procedure for each section of a visit-sequence upto** ↑
a **call** with a switch over applicable productions for ↓

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

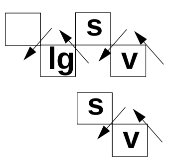
L.lg=1; ↑1; B.s=L.s; ↓B,1; L.v=B.v; ↑2

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

B.v=0; ↑1

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

B.v=Power2(B.s); ↑1

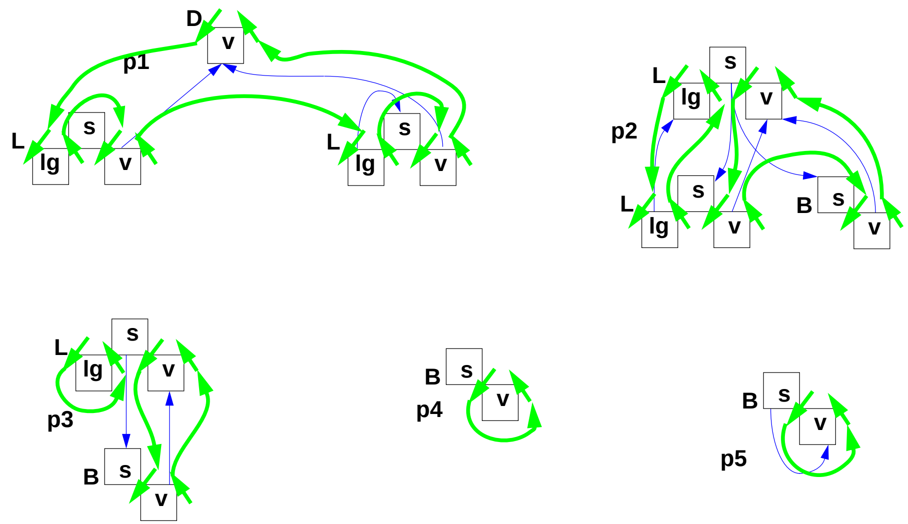


L visited twice

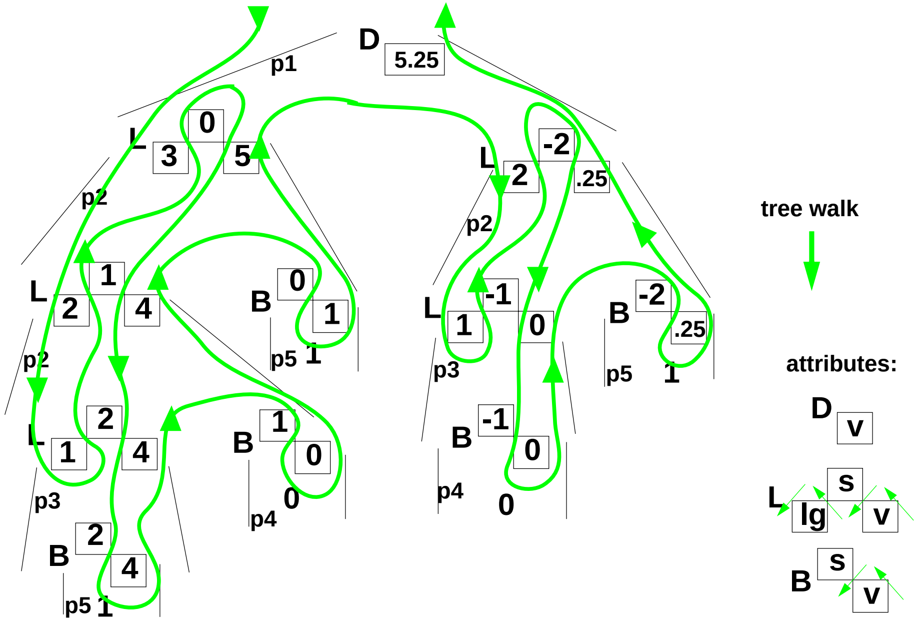
B visited once

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

Visit-Sequences for AG Binary numbers (tree patterns)



Tree walk for AG Binary numbers



LAG (k) condition

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**.

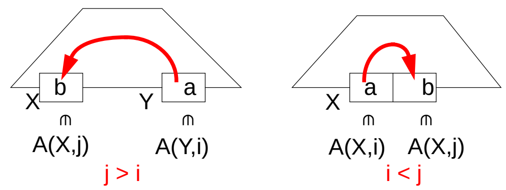
Crucial dependences:

In every dependence graph every dependence

- **Y.a -> X.b** where X and Y occur on the **right-hand side** and **Y is right of X** implies that **Y.a belongs to an earlier pass than X.b**, and
- **X.a -> X.b** where X occurs on the **right-hand side** implies that **X.a belongs to an earlier pass than X.b**

Necessary and sufficient condition over dependence graphs - expressed graphically:

A dependency **from right to left**



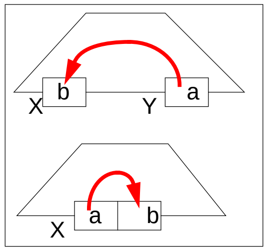
A dependence **at one symbol** on the right-hand side

LAG (k) algorithm

Algorithm checks whether there is a **k>=1** such that an **AG is LAG(k)**.

Method:
compute iteratively **A(1)**, ..., **A(k)**;
in each iteration try to allocate all remaining attributes to the current pass, i.e. **A(i)**;
remove those which can not be evaluated in that pass

Algorithm:



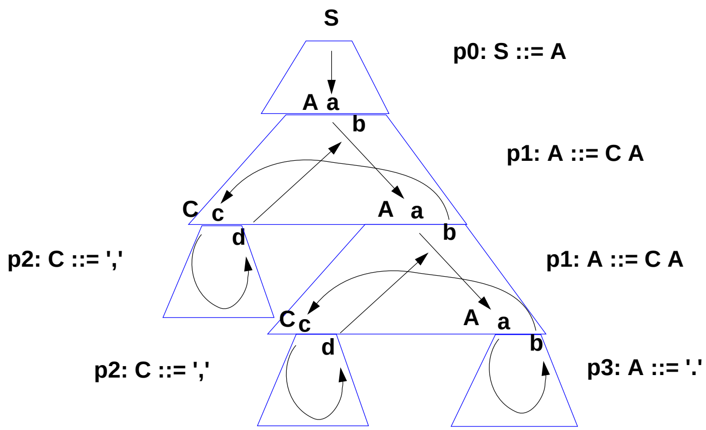
Set **i=1** and **Cand=** all attributes

repeat
set **A(i) = Cand**; set **Cand** to empty;

while still attributes can be removed from **A(i)** do
remove an attribute X.b from A(i) and add it to **Cand** if
- there is a **crucial dependence**
Y.a -> **X.b** s.t.
X and **Y** are on the right-hand side, **Y** to the right of **X** and **Y.a** in **A(i)** or
X.a -> **X.b** s.t. **X** is on the right-hand side and **X.a** is in **A(i)**
- **X.b** depends on an attribute that is not yet in any **A(i)**

if **Cand** is empty: exit: the AG **is LAG(k)** and all attributes are assigned to their passes
if **A(i)** is empty: exit: the AG **is not LAG(k) for any k**
else: set i = i + 1

AG not LAG(k) for any k



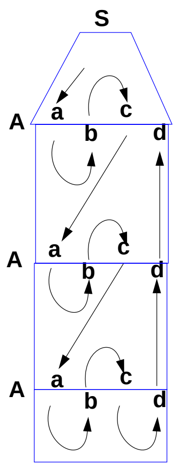
A.a can be allocated to the first left-to-right pass.
C.c, C.d, A.b can not be allocated to any pass.

The AG is RAG(1), AAG(2) and
can be evaluated by visit-sequences.

AG not evaluable in passes

No attribute can be allocated to any pass for any strategy.

The AG can be evaluated by visit-sequences.



p0: S ::= A

p1: A ::= ',' A

p1: A ::= ',' A

p2: A ::= '.'

Generators for attribute grammars

LIGA	University of Paderborn	OAG
FNC-2	INRIA	ANCAG (superset of OAG)
CoCo	Universität Linz	LAG(k)

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**

Explicit left-to-right depth-first propagation

```
ATTR pre, post: int;
RULE: Root ::= Block COMPUTE
  Block.pre = 0;
END;
RULE: Block ::= '{' Constructs '}' COMPUTE
  Constructs.pre = Block.pre;
  Block.post = Constructs.post;
END;
RULE: Constructs ::= Constructs Construct COMPUTE
  Constructs[2].pre = Constructs[1].pre;
  Construct.pre = Constructs[2].post;
  Constructs[1].post = Construct.post;
END;
RULE: Constructs ::= COMPUTE
  Constructs.post = Constructs.pre;
END;
RULE: Construct ::= Definition COMPUTE
  Definition.pre = Construct.pre;
  Construct.post = Definition.post;
END;
RULE: Construct ::= Statement COMPUTE
  Statement.pre = Construct.pre;
  Construct.post = Statement.post;
END;
RULE:Definition ::= 'define' Ident ';' COMPUTE
  Definition.printed =
     printf ("Def %d defines %s in line %d\n",
               Definition.pre, StringTable (Ident), LINE);
  Definition.post =
     ADD (Definition.pre, 1) <- Definition.printed;
END;
RULE: Statement ::= 'use' Ident ';' COMPUTE
  Statement.post = Statement.pre;
END;
RULE: Statement ::= Block COMPUTE
  Block.pre = Statement.pre;
  Statement.post = Block.post;
END;
```

`Definitions` are enumerated and printed from left to right.

The next `Definition` number is propagated by a pair of attributes at each node:

`pre` (inherited)
`post` (synthesized)

The value is initialized in the `Root` context and

incremented in the `Definition` context.

The computations for propagation are systematic and redundant.

Left-to-right depth-first propagation using a CHAIN

```
CHAIN count: int;

RULE: Root ::= Block COMPUTE
   CHAINSTART Block.count = 0;
END;

RULE: Definition ::= 'define' Ident ';'
COMPUTE
   Definition.print =
      printf ("Def %d defines %s in line %d\n",
              Definition.count, /* incoming */
              StringTable (Ident), LINE);

   Definition.count = /* outgoing */
      ADD (Definition.count, 1)
      <- Definition.print;
END;
```

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

One CHAIN name; attribute pairs are generated where needed.

`CHAINSTART` initializes the CHAIN in the root context of the CHAIN.

Computations on the CHAIN are **strictly bound** by dependencies.

Trivial computations of the form X.pre = Y.pre in CHAIN order can be **omitted**. They are generated where needed.

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

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

The nesting depths of **`Blocks`** are computed.

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

Propagation from computation to the uses are generated as needed.

No explicit computations or attributes are needed for the remaining rules and symbols.

Dependency pattern CONSTITUENTS

```
RULE: Root ::= Block COMPUTE
   Root.DefDone =
      CONSTITUENTS Definition.DefDone;
END;
RULE: Definition ::= 'define' Ident ';'
COMPUTE
   Definition.DefDone =
      printf ("%s defined in line %d\n",
              StringTable (Ident), LINE);
END;
RULE: Statement ::= 'use' Ident ';' COMPUTE
   printf ("%s used in line %d\n",
           StringTable (Ident), LINE)
   <- INCLUDING Root.DefDone;
END;
```

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

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

Propagation from computation to the **`CONSTITUENTS`** construct is generated where needed.

The shown combination with **`INCLUDING`** is a common dependency pattern.

All **`printf`** calls in **`Definition`** contexts are done before any in a **`Statement`** context.

5. Binding of Names
5.1 Fundamental notions

Program entity: An **identifiable** entity that has **individual properties**, is used potentially at **several places in the program**. Depending on its **kind** it may have one or more runtime instances; e. g. type, function, variable, label, module, package.

Identifiers: a class of tokens that are used to **identify program entities; e. g. `minint`**

Name: a **composite construct** used to **identify a program entity**, usually contains an identifier; e. g. **`Thread.sleep`**

Static binding: A binding is established **between a name and a program entity**. It is **valid** in a certain area of the **program text**, the **scope of the binding.** There the name identifies the program entity. Outside of its scope the name is unbound or bound to a different entity. Scopes are expressed in terms of program constructs like blocks, modules, classes, packets

Dynamic binding: Bindings are established in the **run-time** environment; e. g. in Lisp.

A **binding may be established**

- **explicitly by a definition**; it usually **defines properties** of the program entity; we then destinguish **defining and applied occurrences** of a name; e. g. in C: **`float x = 3.1; y = 3*x;`** or in JavaScript: **`var x;`**
- **implicitly by using the name**; properties of the program entity may be defined by the context; e. g. bindings of global and local variables in PHP

5.2 Scope rules

Scope rules: a set of rules that specify for a given language how bindings are established and where they hold.

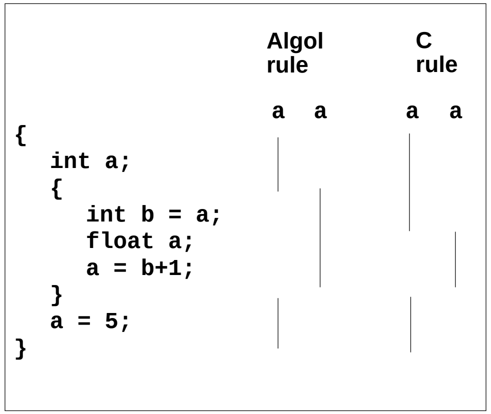
2 variants of fundamental **hiding rules** for languages with nested structures. Both are based on **definitions that explicitly introduce bindings**:

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. in Algol 60, Pascal, Java

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 to the end.

e. g. in C, C++, Java



Defining occurrence before applied occurrences

The **C rule** enforces the defining occurrence of a binding precedes all its applied occurrences.

In Pascal, Modula, Ada the **Algol rule** holds. An **additional rule** requires that the defining occurrence of a binding precedes all its applied occurrences.

Consequences:

- specific constructs for **forward references of functions** which may call each other recursively:
  `forward` function declaration in Pascal;
  function declaration in C before the function definition,
  exemption form the def-before-use-rule in Modula
- specific constructs for **types** which may contain **references** to each other **recursively**:
  forward type references allowed for pointer types in Pascal, C, Modula
- specific rules for labels to allow **forward jumps**:
  label declaration in Pascal before the label definition,
  Algol rule for labels in C
- (Standard) **Pascal** requires **declaration parts** to be structured as a sequence of declarations for constants, types, variables and functions, such that the former may be used in the latter. **Grouping by coherence criteria** is not possible.

Algol rule is **simpler, more flexible** and allows for **individual ordering** of definitions according to design criteria.

Multiple definitions

Usually a **definition** of an identifier is required to be **unique** in each range. That rule guarantees that at most one binding holds for a given (plain) identifier in a given range.

Deviations from that rule:

- Definitions for the same binding are allowed to be repeated, e. g. in C
  `external int maxElement;`
- Definitions for the same binding are allowed to accumulate properties of the program entity, e. g. AG specification language LIDO: association of attributes to symbols:
  ```
  SYMBOL AppIdent: key: DefTableKey;
  ...
  SYMBOL AppIdent: type: DefTableKey;
  ```
- **Separate name spaces** for bindings of different kinds of program entities. Occurrences of identifiers are syntactically distinguished and associated to a specific name space, e. g. in Java bindings of packets and types are in different name spaces:
  `import Stack.Stack;`
  in C labels, type tags and other bindings have their own name space each.
- **Overloading** of identifiers: **different program entities are bound to one identifier** with overlapping scopes. They are **distinguished by static semantic information** in the context, e. g. overloaded functions distinguished by the signature of the call (number and types of actual parameters).

Explicit Import and Export

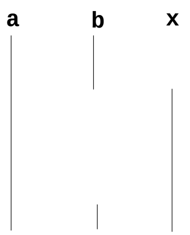
Bindings may be **explicitly imported to or exported from a range** by specific language constructs. Such features have been introduced in languages like Modula-2 in order to support **modular decomposition and separate compilation**.

Modula-2 defines two different import/export features
1. **Separately compiled modules**:

```
DEFINITION MODULE Scanner;           interface of a separately compiled module
   FROM Input IMPORT Read, EOL;                          imported bindings
   EXPORT QUALIFIED Symbol, GetSym;                      exported bindings
   TYPE Symbol = ...;                        definitions of exported bindings
   PROCEDURE GetSym;
END Scanner;
IMPLEMENTATION MODULE Scanner BEGIN ... END Scanner;
```

2. **Local modules, embedded in the block structure** establish scope boundaries:

```
VAR a, b: INTEGER;
...
MODULE m;
   IMPORT a;
   EXPORT x;
   VAR x: REAL;
BEGIN ... END m;
...
```



Bindings as properties of entities

Program entities may have a property that is a set of bindings,
e. g. the entities exported by a module interface or the fields of a struct type in C:

```
typedef struct {int x, y;} Coord;

Coord anchor[5];
anchor[0].x = 42;
```

The type `Coord` has the bindings of its fields as its property; `anchor[0]` has the type `Coord`; `x` is bound in its set of bindings.

Language constructs like the `with`-statement of Pascal insert such sets of bindings into the bindings of nested blocks:

```
type Coord = record x, y: integer; end;
var  anchor: array [0..4] Coord;
     a, x: real;
begin ...
  with anchor[0] do
    begin ...
      x := 42;
    end;
  ...
end;
```

Bindings of the type `Coord` are inserted into the textually nested scopes; hence the field `x` hides the variable `x`.

Inheritance with respect to binding

Inheritance is a **relation between object oriented classes**. It defines the basis for **dynamic binding of method calls**. However, **static binding rules** determine the **candidates for dynamic binding** of method calls.

A class has a **set of bindings as its property**.

It consists of the bindings **defined in the class** and those **inherited** from classes and interfaces.

An **inherited binding may be hidden** by a local definition.

That set of bindings is used for identifying qualified names (cf. **`struct`** types):

```
D d = new D; d.f();
```

A class may be **embedded in a context** that provides bindings. An unqualified name as in **`f()`** is bound in the **class's local and inherited** sets, and **then** in the **bindings of the textual context** (cf. **`with`**-statement).

```
class E
{  void f(){...}
   void h(){...}
   ...
}
```

```
class D
   extends E
{  void f(){...}
   void g(){...}
   ...
}
```

```
interface I
{  public void k();
}
```

```
class A
{ void f(){...}
   class C
      extends D implements I
   { void tr(){ f(); h();}
   }
}
```


5.3 An environment module for name analysis

The compiler represents a **program entity by a key**. It references a description of the entity's properties.

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

Bindings that have a **common scope** are composed to **sets**.

An **environment** is a **linear sequence of sets of bindings** e_1, e_2, e_3, ... that are connected by a **hiding relation**: a binding (a, k) in e_i hides a binding (a,h) in e_j if i < j.

Scope rules can be modeled using the concept of **environments**.

The **name analysis task** can be **implemented** using a **module** that implements **environments** and operations on them.

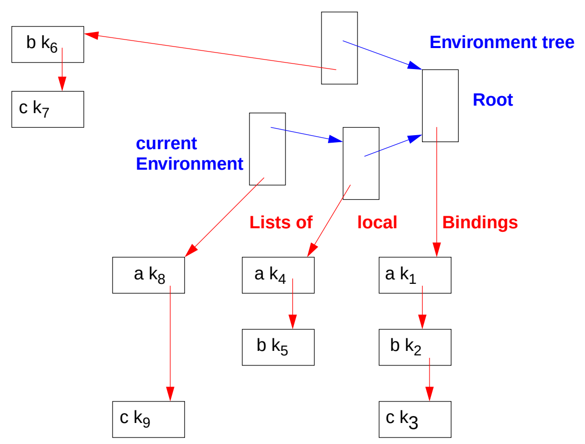
Environment module

Implements the abstract data type **Environment**:
hierarchically nested sets of **Binding**s **(identifier, environment, key)**
(The binding pair (i,k) is extended by the environment to which the binding belongs.)

Functions:

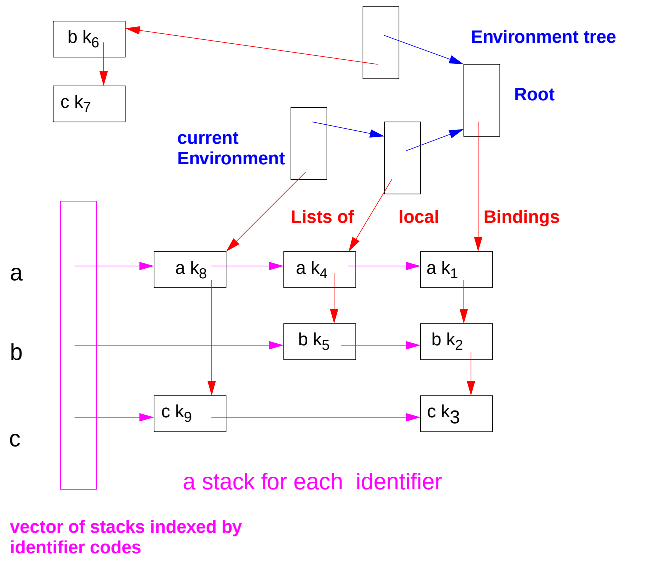
NewEnv ()	creates a new Environment e, to be used as root of a hierarchy
NewScope (e_1)	creates a new Environment e_2 that is nested in e1. Each binding of e_1 is also a binding of e_2 if it is not hidden there.
BindIdn (e, id)	introduces a binding (id, e, k) if e has no binding for id; then k is a new 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.

Data structure of the environment module (1)



k_i: key of the defined entity

Data structure of the environment module (2)



k_i: key of the defined entity

Environment operations in tree contexts

Operations in tree contexts and the order they are called can **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);`

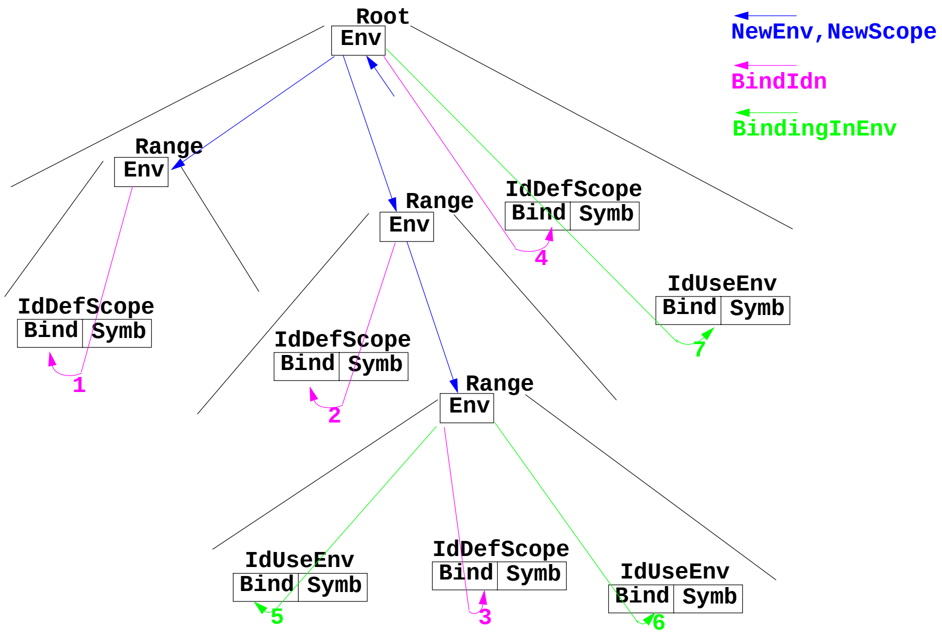
Preconditions for specific scope rules:

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).

Attribute computations for binding of names



6. Type specification and type analysis

A **type** characterizes a set of (simple or structured) values and the applicable operations.

The language design constrains the way how values may interact.

Strongly typed language:
The implementation can guarantee that all type constraints can be checked

- **at compile time (static typing):** compiler finds type errors (developer), or
- **at run time (dynamic typing):** run time checks find type errors (tester, user).

static typing (plus run time checks): Java (strong); C, C++, Pascal, Ada (almost strong)
dynamic: script languages like Perl, PHP, JavaScript
no typing: Prolog, Lisp

Statically typed language:
Programmer declares type property - compiler checks (most languages)
Programmer uses typed entities - compiler infers their type properties (e.g. SML)

Compiler keeps track of the type of any

- **defined entity that has a value** (e. g. variable); stores type property in the definition module
- **program construct** elaborates to a value (e. g. expressions); stores type in an attribute

Concepts for type analysis

Type: characterization of a subset of the values in the universe of operands available to the program. „a triple of int values“

Type denotation: a source-language construct used to denote a user-defined typ (language-defined types do not require type denotations).

```
typedef struct {int year, month, day;} Date;
```

sameType: a partition defining type denotations that might denote the same type.

Type identifier: a name used in a source-language program to specify a type.

```
typedef struct {int year, month, day;} Date;
```

Typed identifier: a name used in a source-language program to specify an entity (such as a variable) that can take any value of a given type.

```
int count;
```

Operator: an entity having a signature that relates operand types to a result type.

```
iAdd: int x int -> int
```

Indication: a set of operators with different signatures.

```
{iAdd, fAdd, union, concat}
```

acceptableAs: a partial order defining the types that can be used in a context where a specific type is expected. `short -> int -> long`

Taxonomy of type systems

[Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. ACM Computing Surveys, 17(4):471–523, 1985.]

- **monomorphism**: Every entity has a unique type. Consequence: different operators for similar operations (e.g. for `int` and `float` addition)
- **polymorphism**: An operand may belong to several types.
  - -- **ad hoc polymorphism**:
    - --- **overloading**: a construct may different meanings depending on the context in which it appears (e.g. + with 4 different signatures in Algol 60)
    - --- **coercion**: implicit conversion of a value into a corresponding value of a different type, which the compiler can insert wherever it is appropriate (only 2 add operators)
  - -- **universal polymorphism**: operations work uniformly on a range of types that have a common structure
    - --- **inclusion polymorphism**: sub-typing as in object-oriented languages
    - --- **parametric polymorphism**: **polytypes** are type denotations with type parameters, e.g. `('a x 'a)`, `('a list x ('a -> 'b) -> 'b list)`
      All types derivable from a polytype have the **same type abstraction**.
      Type parameters are substituted by type **inference** (SML, Haskell) or by **generic instantiation** (C++, Java)
      see GPS 5.9 - 5.10

© 2011 bei Prof. Dr. Uwe Kastens

Monomorphism and ad hoc polymorphism

```
monomorphism                          (1)
polymorphism
├── ad hoc polymorphism
│       overloading                   (2)
│       coercion                      (3)
└── universal polymorphism
    ├── inclusion polymorphism        (4)
    └── parametric polymorphism       (5)
```

monomorphism (1):
4 different names for addition:

```
addII: int   x int   -> int
addIF: int   x float -> float
addFI: float x int   -> float
addFF: float x float -> float
```

overloading (2):
1 name for addition **+**;
4 signatures are distinguished by actual operand and result types:

```
+: int   x int   -> int
+: int   x float -> float
+: float x int   -> float
+: float x float -> float
```

coercion (3):
`int` is acceptableAs **`float`**,
2 names for two signatures:

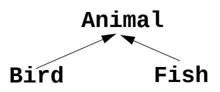
```
addII: int   x int   -> int
addFF: float x float -> float
```

© 2011 bei Prof. Dr. Uwe Kastens

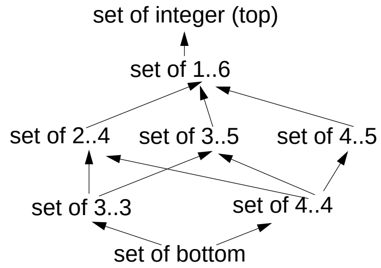
Examples for inclusion polymorphism (4)

Sub-typing:
S ist a **sub-type of** type T, S **<:** T, if each value of S is acceptable where a value of type T is expected.

Sub-type relation established by classes in **object-oriented languages**

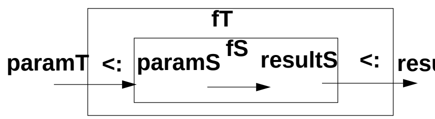


Lattice of set types in Pascal:



A **function** of type **fS** can be called where a function of type **fT** is expected, i.e. **fS <: fT,** if

fT = paramT -> resultT   **paramT <: paramS**
fS = paramS -> resultS   **resultS <: resultT**



© 2011 bei Prof. Dr. Uwe Kastens

Compiler's 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 this 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 value d, if P has not been set

Operations are **called in tree contexts**, dependences control accesses, e. g. SetP before GetP

Implementation of data structure:a property list for every key

Definition module is generated from specifications of the form

Property name : property type;

```
ElementNumber: int;
```

Generated functions: `ResetElementNumber`, `SetElementNumber`, `GetElementNumber`

Language defined entities

Language-defined types, operators, and indications are represented by **known keys** - definition table keys, created by initialization and made available as **named constants**.

Eli's specification language OIL can be used to specify language defined types, operators, and indications, e.g.:

```
OPER
   iAdd (intType,intType):intType;
   rAdd (floatType,floatType):floatType;

INDICATION
   PlusOp: iAdd, rAdd;

COERCION
   (intType):floatType;
```

It results in known keys for two types, two operators, and an indication. The following identifiers can be used to name those keys in tree computations:

```
intType, floatType, iAdd, rAdd, PlusOp

RULE: Operator ::= '+' COMPUTE Operator.Indic = PlusOp;END;
```

The coercion establishes the language-defined relation

```
intType acceptableAs floatType
```


Language-defined and user-defined types

A **language-defined type** is represented by a keyword in a program. The compiler determines sets an attribute `Type.Type`:

```
RULE: Type ::= 'int' COMPUTE
   Type.Type = intType;
END;
```

The type analysis modules of Eli export a computational role for **user-defined types**:

`TypeDenotation`: denotation of a user-defined type. The Type attribute of the symbol inheriting this role is set to a new definition table key by a module computation.

```
RULE: Type ::= ArrayType COMPUTE
   Type.Type = ArrayType.Type;
END;

SYMBOL ArrayType INHERITS TypeDenotation END;

RULE: ArrayType ::= Type '[' ']' END;
```


Classification of identifiers (1)

The type analysis modules export four **computational roles to classify identifiers**:

`TypeDefDefId`: definition of a type identifier. The designer must write a computation setting the Type attribute of this symbol to the type bound to the identifier.

`TypeDefUseId`: reference to a type identifier defined elsewhere. The Type attribute of this symbol is set by a module computation to the type bound to the identifier.

`TypedDefId`: definition of a typed identifier. The designer must write a computation setting the Type attribute of this symbol to the type bound to the identifier.

`TypedUseId`: reference to a typed identifier defined elsewhere. The Type attribute of this symbol is set by a module computation to the type bound to the identifier.

```
SYMBOL ClassBody INHERITS TypeDenotation END;
SYMBOL TypIdDef INHERITS TypeDefDefId END;
SYMBOL TypIdUse INHERITS TypeDefUseId END;

RULE: ClassDecl ::=
   OptModifiers 'class' TypIdDef OptSuper OptInterfaces ClassBody
COMPUTE TypIdDef.Type = ClassBody.Type;
END;

RULE: Type ::= TypIdUse COMPUTE
Type.Type = TypIdUse.Type;
END;
```

Classification of identifiers (2)

A declaration introduces typed entities; it plays the role **TypedDefinition**.

TypedDefId is the role for identifiers in a context where the type of the bound entity is determined

TypedUseId is the role for identifiers in a context where the type of the bound entity is used. The role **ChkTypedUseId** checks whether a type can be determined for the particular entity:

```
RULE: Declaration ::= Type VarNameDefs ';' COMPUTE
   Declaration.Type = Type.Type;
END;

SYMBOL Declaration INHERITS TypedDefinition END;
SYMBOL VarNameDef  INHERITS TypedDefId END;
SYMBOL VarNameUse  INHERITS TypedUseId, ChkTypedUseId END;
```


Type analysis for expressions (1): trees

An **expression** node represents a **program construct that yields a value**, and an **expression tree** is a subtree of the AST made up **entirely of expression nodes**. Type analysis within an expression tree is uniform; additional specifications are needed only at the roots and leaves.

The type analysis modules export the role **ExpressionSymbol** to classify expression nodes. It carries two attributes that characterize the node inheriting it:

Type: the type of value delivered by the node. It is always set by a module computation.

Required: the type of value required by the context in which the node appears.
The designer may write a computation to set this inherited attribute in the upper context if the node is the root of an expression tree; otherwise it is set by a module computation.

A node **n** is type-correct if (**n.Type acceptableAs n.Required**).

PrimaryContext expands to attribute computations that set the Type attribute of an expression tree leaf. The first argument must be the grammar symbol representing the expression leaf, which must inherit the **ExpressionSymbol** role. The second argument must be the result type of the expression leaf.

DyadicContext characterizes expression nodes with two operands. All four arguments of DyadicContext are grammar symbols: the result expression, the indication, and the two operand expressions. The second argument symbol must inherit the **OperatorSymbol** role; the others must inherit **ExpressionSymbol**.

Type analysis for expressions (2): leaves, operators

The nodes of expression trees are characterized by the roles **ExpressionSymbol** and **OperatorSymbol**. The tree contexts are characterized by the roles **PrimaryContext** (for leaf nodes), **MonadicContext**, **DyadicContext**, **ListContext** (for inner nodes), and **RootContext**:

```
SYMBOL Expr       INHERITS ExpressionSymbol END;
SYMBOL Operator   INHERITS OperatorSymbol END;
SYMBOL ExpIdUse   INHERITS TypedUseId END;

RULE: Expr ::= Integer COMPUTE
   PrimaryContext(Expr, intType);
END;
RULE: Expr ::= ExpIdUse COMPUTE
   PrimaryContext(Expr, ExpIdUse.Type);
END;
RULE: Expr ::= Expr Operator Expr COMPUTE
   DyadicContext(Expr[1], Operator, Expr[2], Expr[3]);
END;
RULE: Operator ::= '+' COMPUTE
   Operator.Indic = PlusOp;
END;
```


Type analysis for expressions (3): Balancing

The conditional expression of C is an example of a **balance context**: The type of each branch (**Expr[3],Expr[4]**) has to be acceptable as the type of the whole conditional expression (**Expr[1]**):

```
RULE: Expr ::= Expr '?' Expr ':' Expr COMPUTE
   BalanceContext(Expr[1],Expr[3],Expr[4]);
END;
```

For the condition the pattern of slide PLaC-6.10 applies.

Balancing can also occur with an **arbitrary number of expressions** the type of which is balanced to yield a **common type at the root node** of that list, e.g. in

```
SYMBOL CaseExps INHERITS BalanceListRoot, ExpressionSymbolEND;
SYMBOL CaseExp  INHERITS BalanceListElem, ExpressionSymbolEND;

RULE: Expr ::= 'case' Expr 'in' CaseExps 'esac' COMPUTE
  TransferContext(Expr[1],CaseExps);
END;

RULE: CaseExps LISTOF CaseExp END;
RULE: CaseExp ::= Expr COMPUTE
  TransferContext(CaseExp,Expr);
END;
```

Type analysis for expressions (4)

Each **expression tree** has a **root**. The the RULE context in which the expression root in on the left-hand side specifies which requirements are imposed to the type of the expression. In the context of an assignment statement below, both occurrences of **`Expr`** are expression tree roots:

```
RULE: Stmt ::= Expr ':=' Expr COMPUTE
   Expr[2].Required = Expr[2].Type;
END;
```

In principle there are 2 different cases how the context states requirements on the type of the Expression root:

- no requirement: **`Expr.Required = NoKey;`** (can be omitted, is set by default) **`Expr[1]`** in the example above
- a specific type: **`Expr.Required =`** computation of some type**;** **`Expr[2]`** in the example above

Operators of user-defined types

User-defined types may introduce operators that have operands of that type, e.g. the indexing operator of an array type:

```
SYMBOL ArrayType INHERITS OperatorDefs END;

RULE: ArrayType ::= Type '[' ']' COMPUTE
   ArrayType.GotOper =
      DyadicOperator(
         ArrayAccessor, NoOprName,
         ArrayType.Type, intType, Type.Type);
END;
```

The above introduces an operator definition that has the signature
`ArrayType.Type x intType -> Type.Type`
and adds it to the operator set of the indication **`ArrayAccessor`**.
The context below identifies an operator in that set, using the types of **`Expr[2]`** and **`Subscript`**. Instead of an operator nonterminal the **`Indication`** is given.

```
SYMBOL Subscript  INHERITS ExpressionSymbol END;
RULE: Expr ::= Expr '[' Subscript ']' COMPUTE
   DyadicContext(Expr[1], , Expr[2], Subscript);
   Indication(ArrayAccessor);
   IF(BadOperator,
      message(ERROR,"Invalid array reference",0,COORDREF));
END;
```


Functions and calls

Functions (methods) can be considered as operators having *n => 0* operands (parameters). Roles: **`OperatorDefs`**, **`ListOperator`**, and **`TypeListRoot`**:

```
SYMBOL MethodHeader INHERITS OperatorDefs END;
SYMBOL Parameters INHERITS TypeListRoot END;

RULE: MethodHeader ::=
   OptModifiers Type FctIdDef '(' Parameters ')' OptThrows COMPUTE
   MethodHeader.GotOper =
      ListOperator(
            FctIdDef.Key, NoOprName,
            Parameters, Type.Type);
END;
```

A call of a function (method) with its arguments is then considered as part of an expression tree. The function name (**`FctIdUse`**) contributes the **`Indication`**:

```
SYMBOL Arguments INHERITS OperandListRoot END;
RULE: Expr ::= Expr '.' FctIdUse '(' Arguments ')' COMPUTE
   ListContext(Expr[1], , Arguments);
   Indication(FctIdUse.Key);
   IF(BadOperator,message(ERROR, "Not a function", 0, COORDREF));
END;
```

The specification allows for overloaded functions.

Type equivalence: name equivalence

Two types *t* and *s* are **name equivalent** if their names *tn* and *sn* are the same or if *tn* is defined to be *sn* or sn defined to be *tn.* An anonymous type is different from any other type.

Name equivalence is applied for example in **Pascal**, and for classes and interfaces in **Java**.

```
type  a = record x: char; y: real end;
      b = record x: char; y: real end;
      c = b;

      e = record x: char; y: ↑ e end;
      f = record x: char; y: ↑ g end;
      g = record x: char; y: ↑ f end;


var   s, t: record x: char; y: real end;
      u: a; v: b; w: c;
      k: e; l: f; m: g;
```

Which types are equivalent?
The value of which variable may be assigned to which variable?

Type equivalence: structural equivalence

In general, two types *t* and *s* are **structurally equivalent** if their definitions become the same when all type identifiers in the definitions of *t* and in *s* are recursively substituted by their definitions. (That may lead to infinite trees.)
Structural equivalence is applied for example in **Algol-68**, and for array types in **Java**.

The example of the previous slide is interpreted under structural equivalence:

```
type  a = record x: char; y: real end;
      b = record x: char; y: real end;
      c = b;

      e = record x: char; y: ↑ e end;
      f = record x: char; y: ↑ g end;
      g = record x: char; y: ↑ f end;

var   s, t: record x: char; y: real end;
      u: a; v: b; w: c;
      k: e; l: f; m: g;
```

Which types are equivalent?
The value of which variable may be assigned to which variable?

Algorithms determine structural equivalence by decomposing the whole set of types into maximal partitions, which each contain only equivalent types.

Type analysis for object-oriented languages (1)

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;
```

Analyze dynamic method binding; try to decide it statically:

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

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


Type analysis for object-oriented languages (2)

Check signature of overriding methods:

calls must be **type safe**

Java requires the **same signature**

weaker requirements would be 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; } }`

Language Eiffel requires **covariant parameter types**: type unsafe!

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;
     (i)          (ii)      (iii)
```

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

```
(i)     'c= bool
(i)     'b= 'a
(ii)    'd= 'a
(iii)   'a= int
```

solve the system of equations:

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

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)
```


Semantic error handling

Design rules:

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

- Any explicit or implicit **requirement of the language definition** needs to be checked by an operation in the tree, e. g. `if (IdUse.Bind == NoBinding) message (...)`
- Checks have to be associated to the **smallest relevant context** yields precise source position for the report; information is to be propagated to that context. **wrong**: „some arguments have wrong types“
- **Meaningfull error reports. wrong**: „type error“
- **Different reports for different violations**; do not connect symptoms 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`**

7. Specification of Dynamic Semantics

The **effect of executing a program** is called its dynamic semantics. It can be described by **composing the effects** of executing the elements of the program, according to its **abstract syntax**. For that purpose the **dynamic semantics of executable language constructs** are specified.

Informal specifications are usually formulated in terms of an abstract machine, e. g.

> *Each **variable has a storage cell**, suitable to store values of the type of the variable. An **assignment v := e** is **executed** by the following steps: determine the storage cell of the variable v, **evaluate the expression** e yielding a value x, an storing x in the storage cell of v.*

The effect of common operators (like arithmetic) is usually not further defined (pragmatics).

The effect of an **erroneous program construct is undefined**. An erroneous program is not executable. The language specification often does not explicitly state, what happens if an erroneous program construct is executed, e. g.

> *The **execution of an input statement is undefined** if the next value of the the input is **not a value of the type** of the variable in the statement.*

A **formal calculus** for specification of dynamic semantics is **denotational semantics**.
It **maps language constructs to functions**, which are then **composed** according to the abstract syntax.

Denotational semantics

Formal calculus for specification of dynamic semantics.

The executable constructs of the **abstract syntax are mapped on functions**, thus defining their effect.

For a given structure tree the functions associated to the tree nodes are **composed** yielding a semantic function of the whole program - **statically**!

That calculus allows to

- **prove dynamic properties** of a program formally,
- reason about the **function of the program** - rather than about is operational execution,
- reason about **dynamic properties of language constructs** formally.

A **denotational specification** of dynamic semantics of a programming language consists of:

- specification of **semantic domains**: in imperative languages they model the program state
- a function **`E`** **that maps all expression constructs** on semantic functions
- a function **`C`** **that maps all statement contructs** on semantic functions

Semantic domains

Semantic domains describe the **domains and ranges of the semantic functions** of a particular language. For an imperative language the central semantic domain describes the **program state**.

Example: semantic domains of a very **simple imperative language**:

`State = Memory × Input × Output`		program state
`Memory = Ident → Value`		storage
`Input = Value*`		the input stream
`Output = Value*`		the output stream
`Value = Numeral | Bool`		legal values

Consequences for the language specified using these semantic domains:

- The language can allow **only global variables**, because a 1:1-mapping is assumed between identifiers and storage cells. In general the storage has to be modelled:

  `Memory = Ident → (Location → Value)`
- **Undefined values** and an **error state** are not modelled; hence, behaviour in **erroneous cases** and **exeption handling** can not be specified with these domains.

Mapping of expressions

Let **`Expr`** be the set of all **constructs of the abstract syntax** that represent expressions, then the function **`E`** maps **`Expr`** on functions which describe **expression evaluation**:

```
E: Expr → (State → Value)
```

In this case the semantic expression functions **compute a value in a particular state**.
Side-effects of expression evaluation can not be modelled this way. In that case the evaluation function had to return a potentially changed state:

```
E: Expr → (State → (State × Value))
```

The mapping **`E`** is **defined by enumerating the cases of the abstract syntax** in the form

```
E[ abstract syntax construct ]state  = functional expression
E[ X]                         s      = F s
```

for example:

```
E [e1 + e2] s = (E [e1] s) + (E [e2] s)
...
E [Number] s = Number
E [Ident] (m, i, o) = m Ident
```

the memory map applied to the identifier

Mapping of statements

Let **`Command`** be the set of all **constructs of the abstract syntax** that represent statements, then the function **`C`** maps **`Command`** on functions which describe **statement execution:**

```
C: Command → (State → State)
```

In this case the semantic statement functions **compute a state transition.**
Jumps and labels in statement execution can not be modelled this way. In that case an additional functional argument would be needed, which models the continuation after execution of the specified construct, **continuation semantics**.

The mapping **`C`** is defined by enumerating the cases of the abstract syntax in the form

```
C[ abstract syntax construct] state  = functional expression
C[ X]                         s      = F s
```

for example:

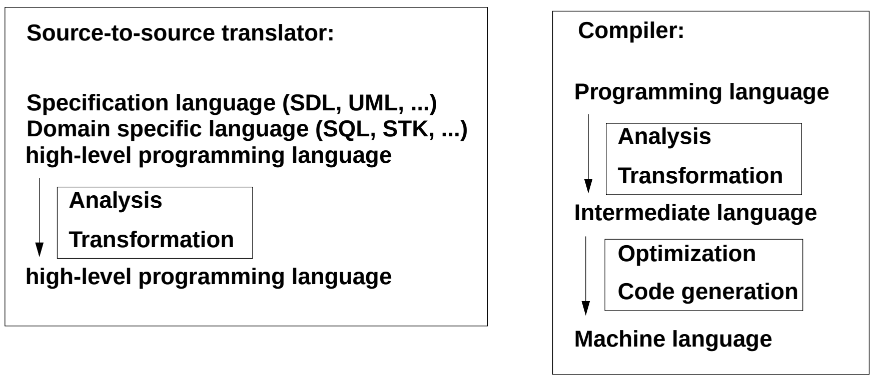
```
C [stmt1; stmt2] s = (C [stmt2] o C [stmt1]) s        function composition
C [v := e] (m, i, o) = (M [(E [e] (m, i, o)) / v], i, o)
```

e is evaluated in the given state and the memory map is changed at the cell of v

```
C [if ex then stmt1 else stmt2] s = E[ex]s -> C[stmt1]s, C[stmt2]s
C [while ex do stmt] s =
   E[ex]s -> (C[while ex do stmt] o C[stmt])s, s
...
```


8. Source-to-source translation

Source-to-source translation:
Translation of a **high-level source language** into a **high-level target language**.

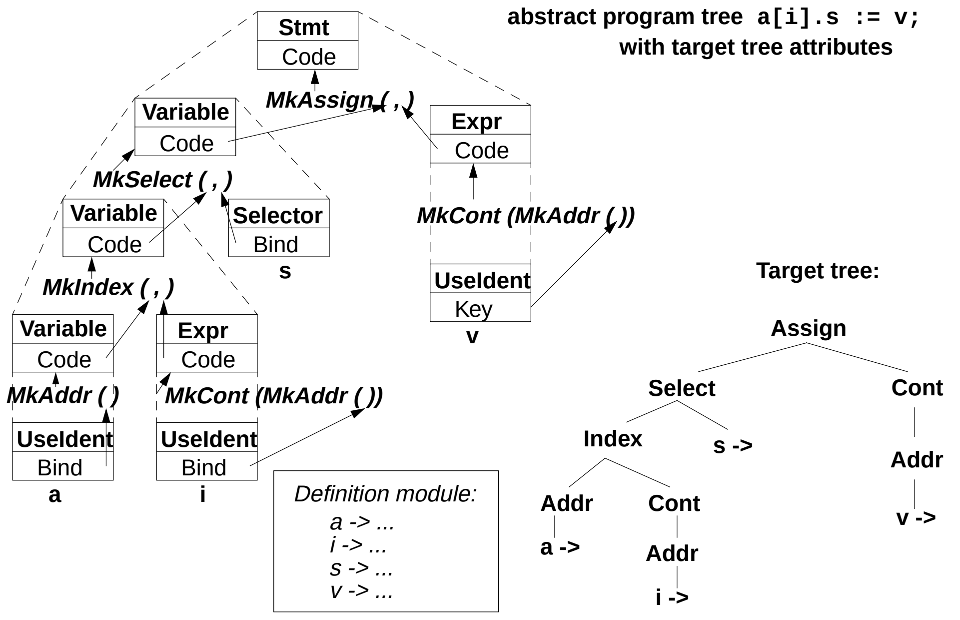


Transformation task:

input: structure tree + properties of constructs (attributes), of entities (def. module)

output:**target tree** (attributes) in textual representation

Example: Target tree construction



Attribute grammar for target tree construction

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;

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);`

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>
<LI> <A HREF="#position_Train">Train</A></LI>
</UL>
```

PTG functions build the target tree (1)

Attributes named Code propagate target sub-trees

Write the target text to a file

```
ATTR Code: PTGNode;

SYMBOL Program COMPUTE

   PTGOutFile
      (CatStrStr (SRCFILE, ".java"),
         PTGFrame
            (CONSTITUENTS Declaration.Code
             WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull),

             CONSTITUENTS Statement.Code SHIELD Statement
             WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull)));

END;
```

PTG pattern with 2 arguments

Access 2 target sub-trees

PTG functions build the target tree (2)

```
RULE: Declaration ::= Type VarNameDefs ';' COMPUTE
   Declaration.Code =
      CONSTITUENTS VarNameDef.Code
      WITH (PTGNode, PTGSeq, IDENTICAL, PTGNull);
END;

SYMBOL VarNameDef COMPUTE
   SYNT.Code =
      IF (EQ (INCLUDING TypedDefinition.Type, intType),
         PTGIntDeclaration (SYNT.NameCode),
      ...
      PTGNULL))));
END;
```


Generate and store target names

```
SYMBOL VarNameDef: NameCode: PTGNode;

SYMBOL VarNameDef COMPUTE
   SYNT.NameCode =
      PTGAsIs
         (StringTable
            (GenerateName (StringTable (TERM))));

   SYNT.GotTgtName =
      ResetTgtName (THIS.Key, SYNT.NameCode);
END;

SYMBOL VarNameUse COMPUTE
   SYNT.Code = GetTgtName (THIS.Key, PTGNULL)
      <- INCLUDING Program.GotTgtName;
END;

SYMBOL Program COMPUTE
   SYNT.GotTgtName =
      CONSTITUENTS VarNameDef.GotTgtName;
END;
```

Create a new name from the source name

Store the name in the definition module

Access the name from the definition module

All names are stored before any is accessed

9. Domain Specific Languages (DSL)

(under construction)

10. Summary
Questions to check understanding

1. Language properties - compiler tasks

1.1. Associate the compiler tasks to the levels of language definition.

1.2. Describe the structure of compilers and the interfaces of the central phases.

1.3. For each phase of compiler frontends describe its task, its input, its output.

1.4. For each phase of compiler frontends explain how generators can contribute to its implementation.

1.5. What specifications do the generators of (1.4) take and what do they generate?

1.6. What data structures are used in each of the phases of compiler frontends?

1.7. Give examples for feedback between compiler phases.

1.8. Java is implemented differently than many other languages, e.g. C++, what is the main difference?

2. Symbol specification and lexical analysis

2.1. Which formal methods are used to specify tokens?

2.2. How are tokens represented after the lexical analysis phase?

2.3. Which information about tokens is stored in data structures?

2.4. How are the components of the token representation used in later phases?

2.5. Describe a method for the construction of finite state machines from syntax diagrams.

2.6. What does the rule of the longest match mean?

2.7. Compare table-driven and directly programmed automata.

2.8. Which scanner generators do you know?

3. Context-free grammars and syntactic analysis

3.1. Which roles play concrete and abstract syntax for syntactic analysis?

3.2. Describe the underlying principle of recursive descent parsers. Where is the stack?

3.3. What is the grammar condition for recursive descent parsers?

3.4. Explain systematic grammar transformations to achieve the LL(1) condition.

3.5. Why are bottom-up parsers in general more powerful than top-down parsers?

3.6. Which information does a state of a LR(1) automaton represent?

3.7. Describe the construction of a LR(1) automaton.

3.8. Which kinds of conflicts can an LR(1) automaton have?

3.9. Characterize LALR(1) automata in contrast to those for other grammar classes.

3.10. Describe the hierarchy of LR and LL grammar classes.

3.11. Which parser generators do you know?

3.12. Explain the fundamental notions of syntax error handling.

3.13. Describe a grammar situation where an LR parser would need unbounded lookahead.

3.14. Explain: the syntactic structure shall reflect the semantic structure.

4. Attribute grammars and semantic analysis

4.1. What are the fundamental notions of attribute grammars?

4.2. Under what condition is the set of attribute rules complete and consistent?

4.3. Which tree walk strategies are related to attribute grammar classes?

4.4. What do visit-sequences control? What do they consist of?

4.5. What do dependence graphs represent?

4.6. What is an attribute partition; what is its role for tree walking?

4.7. Explain the LAG(k) condition.

4.8. Describe the algorithm for the LAG(k) check.

4.9. Describe an AG that is not LAG(k) for any k, but is OAG for visit-sequences.

4.10. Which attribute grammar generators do you know?

4.11. How is name analysis for C scope rules specified?

4.12. How is name analysis for Algol scope rules specified?

4.13. How is the creation of target trees specified?

5. Binding of names

5.1. How are bindings established explicitly and implicitly?

5.2. Explain: consistent renaming according to scope rules.

5.3. What are the consequences if defining occurence before applied occurence is required?

5.4. Explain where multiple definitions of a name could be reasonable?

5.5. Explain class hierarchies with respect to static binding.

5.6. Explain the data structure for representing bindings in the environment module.

5.7. How is the lookup of bindings efficiently implemented?

5.8. How is name analysis for C scope rules specified by attribute computations?

5.9. How is name analysis for Algol scope rules specified by attribute computations?

6. Type specification and analysis

6.1. What does „statically typed“ and „strongly typed“ mean?

6.2. Distinguish the notions „type“ and „type denotation“?

6.3. Explain the taxonomy of type systems.

6.4. How is overloading and coercion specified in Eli?

6.5. How is overloading resolved?

6.6. Distinguish Eli’s four identifier roles for type analysis?

6.7. How is type analysis for expressions specified in Eli?

6.8. How is name equivalence of types defined? give examples.

6.9. How is structural equivalence of types defined? give examples.

6.10. What are specific type analysis tasks for object-oriented languages?

6.11. What are specific type analysis tasks for functional languages?

7. , 8. Dynamic semantics and transformation

7.1. What are denotational semantics used for?

7.2. How is a denotational semantic description structured?

7.3. Describe semantic domains for the denotational description of an imperative language.

7.4. Describe the definition of the functions E and C for the denotational description of an imperative language.

7.5. How is the semantics of a while loop specified in denotational semantics?

7.6. How is the creation of target trees specified by attribute computations?

7.7. PTG is a generator for creating structured texts. Explain its approach.