3. Code Generation

Input: Program in intermediate language

Tasks:

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

Output: abstract machine program, stored in a data structure

Design of code generation:

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

Implementation of code generation:

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

Lecture Compilation Methods SS 2011 / Slide 301

Objectives:

Overview on design and implementation

In the lecture:

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

Suggested reading:

Kastens / Übersetzerbau, Section 7

3.1 Storage Mapping

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

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

Design the use of storage areas:

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

Design the mapping of data types (next slides)
Design activation records and translation of function calls (next section)

Lecture Compilation Methods SS 2011 / Slide 302

Objectives:

Design the mapping of the program state on to the machine state

In the lecture:

Explain storage classes and their use

Suggested reading:

Kastens / Übersetzerbau, Section 7.2

Storage Mapping for Data Types

Basic types

arithmetic, boolean, character types

match language requirements and machine properties:
data format, available instructions,
size and alignment in memory

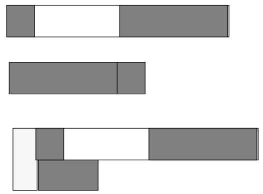
Structured types

for each type — representation in memory and code sequences for operations, e. g. assignment, selection, ...

record — relative address and alignment of components; reorder components for optimization

union — storage overlay, tag field for discriminated union

set — bit vectors, set operations



for **arrays** and **functions** see next slides

Lecture Compilation Methods SS 2011 / Slide 303

Objectives:

Overview on type mapping

In the lecture:

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

- Give examples for mapping of arithmetic types.
- Explain alignment of record fields.
- Explain overlay of union types.
- Discuss a recursive algorithm for type mapping that traverses type descriptions.

Suggested reading:

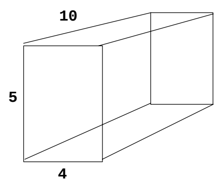
GdP slides on data types

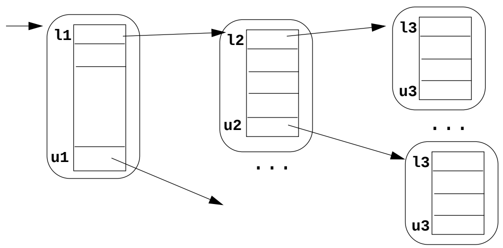
Array Implementation: Pointer Trees

An n-dimensional array

```
a: array[l1..u1, l2..u2, ..., ln..un] of real;
```

is implemented by a **tree of linear arrays**;
n-1 levels of pointer arrays and data arrays on the n-th level





Each single array can be allocated separately, dynamically; scattered in memory

In **Java arrays** are implemented this way.

Lecture Compilation Methods SS 2011 / Slide 304

Objectives:

Understand implementation variant

In the lecture:

Aspects of this implementation variant are explained:

- allocation by need,
- non-orthogonal arrays,
- additional storage for pointers,
- costly indirect access

Assignments:

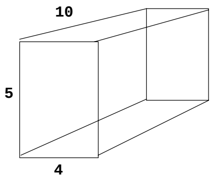
Allocate an array in Java that has the shape of a pyramid. How many pointer and data cells are needed?

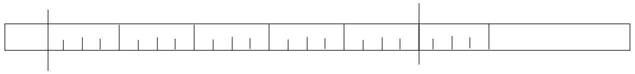
Array Implementation: Contiguous Storage

An n-dimensional array

```
a: array[l1..u1, l2..u2, ..., ln..un] of real;
```

is mapped to **one contiguous storage area linearized in row-major order**:





```
start
store[start] ... store[start + elno*elsz - 1]
```

linear storage map of array a onto byte-array `store` from index `start`:

number of elements	`elno = st1 * st2 * ... * stn`
i-th index stride	`sti = ui - li + 1`
element size in bytes	`elsz`

Index map of `a[i1, i2, ..., in]`:

```
store[start+ (..((i1-l1)*st2 + (i2-l2))*st3 +..)*stn + (in-ln))*elsz]

store[const + (..(i1*st2     + i2)*st3     +..)*stn +  in)*elsz]
```


Lecture Compilation Methods SS 2011 / Slide 305

Objectives:

Understand implementation variant

In the lecture:

Aspects of this implementation variant are explained:

- Give an example for a 3-dimensional array.
- Explain the index function.
- Explain the index function with constant terms extracted.
- Compare the two array implementation variants:
- Allocation in one chunk,
- orthogonal arrays only,
- storage only for data elements,
- efficient direct addressing.
- FORTRAN: column major order!

Suggested reading:

GdP slides on data types

Questions:

- What information is needed in an array descriptor for a dynamically allocated multi-dimensional array?

Functions as Data Objects

Functions may occur **as data objects**:

- variables
- parameters
- function results
- lambda expressions (in functional languages)

Functions that are defined on the **outermost program level** (non-nested)

can be implemented by just the **address of the code**.

Functions that are **defined in nested structures** have to be implemented by a **pair: (closure, code)**

The **closure** contains all **bindings** of names to variables or values that are valid when the **function definition is executed**.

In **run-time stack** implementations the **closure is a sequence of activation records on the static predecessor chain.**

Lecture Compilation Methods SS 2011 / Slide 306

Objectives:

Understand the concept of closure

In the lecture:

The topics on the slide are explained:

- examples for functions as data objects,
- recall functional programming (GdP),
- closures as a sequence of activation records,
- relate closures to run-time stacks

Suggested reading:

GdP slides on run-time stack

Questions:

- Why must a functional parameter in Pascal be represented by a pair (closure, code)?

3.2 Run-Time Stack
Activation Records

Run-time stack contains one **activation record** for each active function call.

Activation record:
provides storage for the data of a function call.

dynamic link:
link from callee to caller,
to the preceding record on the stack

static link:
link from callee c to the record s where c is defined

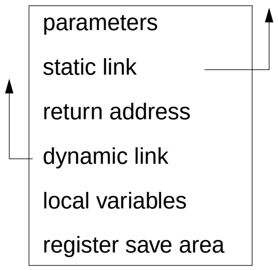
s is a call of a function which contains the definition of the function, the call of which created c.

Variables of surrounding functions are accessed via the static predecessor chain.

Only relevant for languages which allow **nested functions**, classes, objects.

closure of a function call:
the **activation records on the static predecessor chain**

activation record:



Lecture Compilation Methods SS 2011 / Slide 307

Objectives:
Understand activation records

In the lecture:
Explain

- static and dynamic links,
- Explain nesting and closures,
- return address.

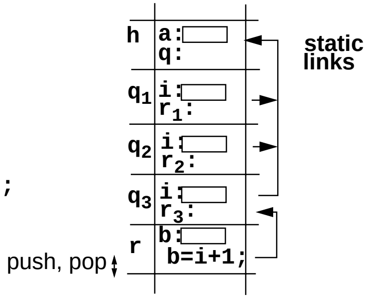
See C-3.10 for relation to call code.

Example for a Run-Time Stack

Run-time stack:
A call creates an activation record and pushes it onto the stack.
It is popped on termination of the call.

nested functions

```
h float a;
  q int i;
    r int b;
      b=i+1;
    if(..) q();
    r();
  q();
```



The **static link** points to the activation record where the called function is defined, e. g. r_3 in q_3

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

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

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

Lecture Compilation Methods SS 2011 / Slide 308

Objectives:
Understand run-time stacks

In the lecture:

- Explain static links.
- Explain nesting and closures.

Questions:

- Why do threads need a separate run-time stack?

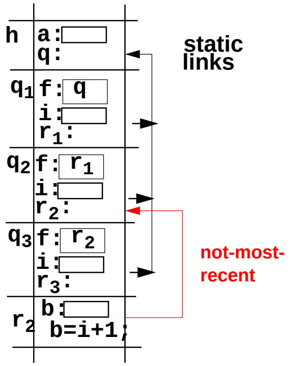
Not-Most-Recent Property

The **static link** of an activation record c for a function r
points to an activation record d for a function q where r is defined in.
If there are activation records for q on the stack, that are more recently created than d,
the **static link to d is not-most-recent**.

That effect can be achieved by using functional parameters or variables.
Example:

nested functions

```
h  float a;

   q(funct f)
     int i;

     r  int b;
        b=i+1;

     if(..) q(r);

     *f();

   q(q);
```



Lecture Compilation Methods SS 2011 / Slide 309

Objectives:

Really understand static links

In the lecture:

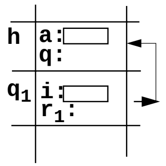
- Explain not-most-recent property.
- r[1] and r[2] must be represented by different values, because they have different closures.

Closures on Run-Time Stacks

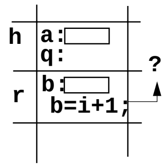
Function calls can be implemented by a run-time stack if the
closure of a function is still on the run-time stack when the function is called.

```
h  float a;

   q  int i;

      r  int b;
         b=i+1;

      return r;

   *(q()) ();
```

Example for violation:



during the call of q



the closure for the call of r is missing

Language conditions to guarantee run-time stack discipline:

Pascal:	functions not allowed as function results, or variables
C:	no nested functions
Modula-2:	nested functions not allowed as values of variables

Functional languages maintain activation records on the heap instead of the run-time stack

Lecture Compilation Methods SS 2011 / Slide 310

Objectives:

Language condition for run-time stacks

In the lecture:

- Explain language restrictions to ensure that necessary closures are on the run-time stack.

Questions:

- Explain why C, Pascal, and Modula-2 obey the requirement on stack discipline?

Activation Records and Call Code

activation record:

result
parameters
static link
return address
dynamic link
local variables
register save area

\- +

0 base address

\+ -

call code

push parameter values
push static link
subroutine jump

function code

push dynamic link
stack register := top of stack
increment top of stack
for local variables
save registers
...
function body
...
restore registers
deallocate local variables
pop stack register
return jump

pop static link
pop parameter area
use and pop result

Lecture Compilation Methods SS 2011 / Slide 311

Objectives:

Relation between activation record and call code

In the lecture:

Explain

- contents of records,
- how to save registers,
- relative addresses of data in the activation record
- register windowing related to run-time stacks

Suggested reading:

Kastens / Übersetzerbau, Section 7.2.2, 7.3.1

Questions:

- How would you design the layout of activation records for a processor that provides register windowing?

3.3 Code Sequences for Control Statements

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

Notation of the **`Code`** constructs:

`Code (S)`	generate code for statements **`S`**
`Code (C, true, M)`	generate code for condition **`C`** such that it branches to M if **`C`** is true, otherwise control continues without branching
`Code (A, Ri)`	generate code for expression **`A`** such that the result is in register **`Ri`**

Code sequence for if-else statement:

```
if (cond) ST; else SE;:

        Code (cond, false, M1)
        Code (ST)
        goto M2
   M1:  Code (SE)
   M2:
```


Lecture Compilation Methods SS 2011 / Slide 312

Objectives:

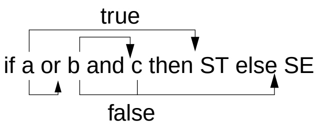
Concept of code sequences for control structures

In the lecture:

- Explain the notation.
- Explain the code sequence for if-else statements.

Short Circuit Translation of Boolean Expressions

Boolean expressions are translated into **sequences of conditional branches**.
Operands are evaluated from left to right until the result is determined.



2 code sequences for each operator; applied to condition tree on a top-down traversal:

Code (A and B, true, M):	Code (A, false, N) Code (B, true, M) N:
Code (A and B, false, M):	Code (A, false, M) Code (B, false, M)
Code (A or B, true, M):	Code (A, true, M) Code (B, true M)
Code (A or B, false, M):	Code (A, true, N) Code (B, false, M) N:

Code (not A, X, M):	Code (A, not X, M)
Code (A < B, true, M):	Code (A, Ri); Code (B, Rj) cmp Ri, Rj braLt M
Code (A < B, false, M):	Code (A, Ri); Code (B, Rj) cmp Ri, Rj braGe M
Code for a leaf:	conditional jump

Lecture Compilation Methods SS 2011 / Slide 313

Objectives:

Special technique for translation of conditions

In the lecture:

- Explain the transformation of conditions.
- Use the example of C-3.14
- Use 2 inherited attributes for the target label and the case when to branch.
- Discuss whether the technique may be applied for C, Pascal, and Ada.

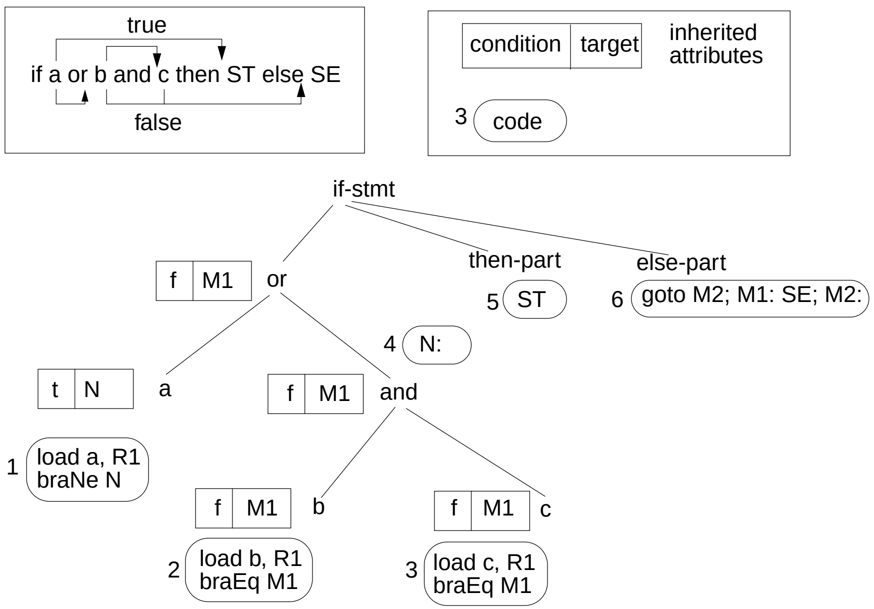
Suggested reading:

Kastens / Übersetzerbau, Section 7.3.3

Questions:

- Why does the transformation of conditions reduce code size?
- How is the technique described by an attribute grammar?
- Why is no instruction generated for the operator *not* ?
- Discuss whether the technique may or must be applied for C, Pascal, and Ada.

Example for Short Circuit Translation



Lecture Compilation Methods SS 2011 / Slide 314

Objectives:

Illustrate short circuit translation

In the lecture:

Discuss together with C-3.13

Suggested reading:

Kastens / Übersetzerbau, Section 7.3.3

Code Sequences for Loops

While-loop variant 1:

```
while (Condition) Body

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

While-loop variant 2:

```
while (Condition) Body

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

Pascal for-loop unsafe variant:

```
for i:= Init to Final do Body

     i = Init
  L: if (i>Final) goto M
     Code (Body)
     i++
     goto L
  M:
```

Pascal for-loop safe variant:

```
for i:= Init to Final do Body

     if (Init==minint) goto L
     i = Init - 1
     goto N
  L: Code (Body)
  N: if (i>= Final) goto M
     i++
     goto L
  M:
```


Lecture Compilation Methods SS 2011 / Slide 315

Objectives:

Understand loop code

In the lecture:

- Explain the code sequences for while-loops.
- Discuss the two variants.
- Explain the code sequences for for-loops.
- Variant 1 may cause an exception if Final evaluates to maxint.
- Variant 2 avoids that problem.
- Variant 2 needs further checks to avoid an exception if Init evaluates to minint.
- Both variants should not evaluate the Final expression on every iteration.

Questions:

- What are the advantages or problems of each alternative?

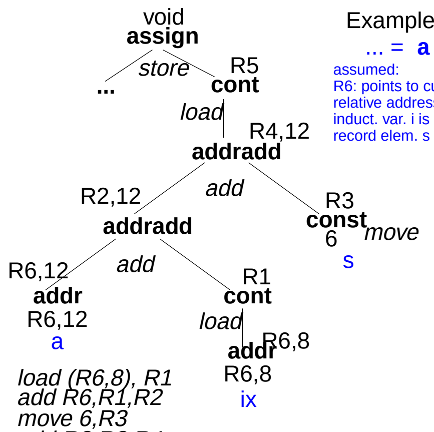
3.4 Code Selection

- Given: target tree in intermediate language.
- **Optimizing selection: Select patterns** that translate single nodes or small subtrees into machine instructions; cover the whole tree with as few instructions as possible.
- Method: **Tree pattern matching,** several techniques

Example: assignment

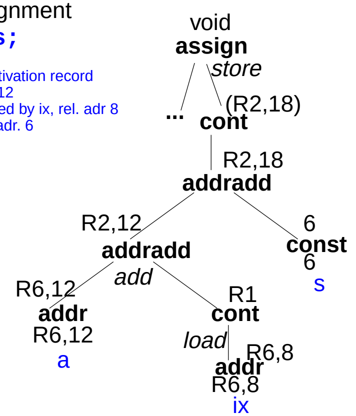
`... = a[i].s;`

assumed:
R6: points to current activation record
relative address of a is 12
induct. var. i is substituted by ix, rel. adr 8
record elem. s has rel. adr. 6



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

cost: 6 instructions



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

cost: 3 instructions

Lecture Compilation Methods SS 2011 / Slide 316

Objectives:

Understand the task

In the lecture:

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

- The task is explained.
- Example: Code of different cost for the same tree.

Selection Technique: Value Descriptors

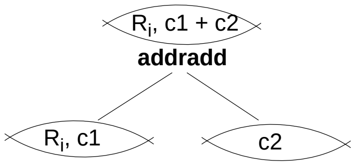
Intermediate language **tree node operators**; e.g.:

addr	address of variable
const	constant value
cont	load contents of address
addradd	address + value

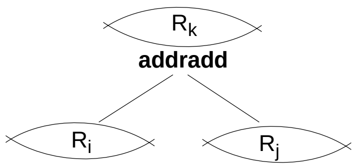
Value descriptors state how/where the value of a tree node is represented, e. g.

$\mathbf{R_i}$	value in register R_i
c	constant value c
$\mathbf{R_i,c}$	address R_i + c
(adr)	contents at the address adr

alternative **translation patterns** to be selected context dependend:



addradd R_i, c1 c2 -> R_i, c1 + c2 ./.



addradd R_i R_j -> R_k add R_i, R_j, R_k

Lecture Compilation Methods SS 2011 / Slide 317

Objectives:

Notion of value descriptors

In the lecture:

- Explain value descriptors
- Explain alternative translation patterns
- Concept of deferred operations
- Different costs of translations
- Compare with the concept of overloaded operators: here, selection by kind of value descriptor.

Suggested reading:

Kastens / Übersetzerbau, Section 7.3.4

Questions:

- How is the technique related to overloaded operators in source languages?

Example for a Set of Translation Patterns

#	operator	operands		result	code
1	addr	R_i, c		-> R_i,c	./.
2	const	c		-> c	./.
3	const	c		-> R_i	move c, R_i
4	cont	R_i, c		-> (R_i, c)	./.
5	cont	R_i		-> (R_i)	./.
6	cont	R_i, c		-> R_j	load (R_i, c), R_j
7	cont	R_i		-> R_j	load (R_i), R_j
8	addradd	R_i	c	-> R_i, c	./.
9	addradd	R_i, c1	c2	-> R_i, c1 + c2	./.
10	addradd	R_i	R_j	-> R_k	add Ri, R_j, R_k
11	addradd	R_i, c	R_j	-> R_k, c	add R_i, R_j, R_k
12	assign	R_i	R_j	-> void	store R_j, R_i
13	assign	R_i	(R_j, c)	-> void	store (R_j,c), R_i
14	assign	R_i,c	R_j	-> void	store R_j, R_i,c

Lecture Compilation Methods SS 2011 / Slide 318

Objectives:

Example

In the lecture:

- Explain the meaning of the patterns.
- Use the example for the tree of C-3.19

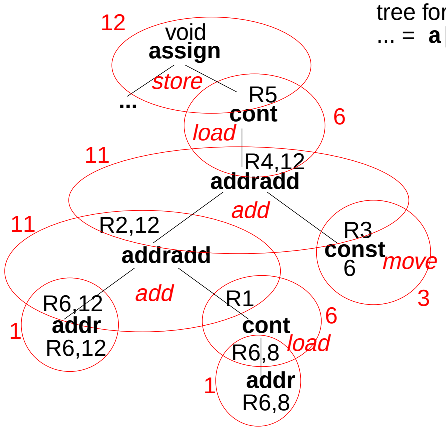
Suggested reading:

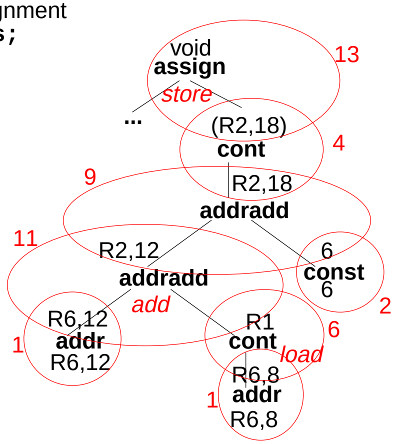
Kastens / Übersetzerbau, Section 7.3.4

C-3.19

Tree Covered with Translation Patterns

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





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

cost: 6 instructions

6
application of pattern #6

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

cost: 3 instructions

Lecture Compilation Methods SS 2011 / Slide 319

Objectives:

Example for pattern applications

In the lecture:

- Show applications of patterns.
- Show alternatives and differences.
- Explain costs accumulated for subtrees.
- Compose code in execution order.

C-3.20

Pattern Selection

Pass 1 bottom-up:

Annotate the nodes with sets of pairs
{ (v, c) | v is a kind of value descriptor that an applicable pattern yields,
c are the accumulated subtree costs}

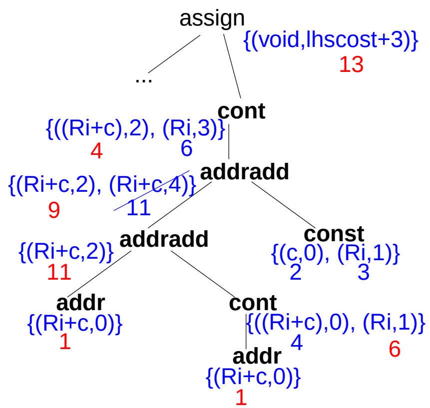
If (v, c1), (v, c2) keep only the cheaper pair.

Pass 2 top-down:

Select for each node the cheapest pattern, that fits to the selection made above.

Pass 3 bottom-up:

Emit code.



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

cost: 3 instructions

Improved technique:

relative costs per sets =>
finite number of potential sets
integer encoding of the sets at generation time

Lecture Compilation Methods SS 2011 / Slide 320

Objectives:

2-pass selection algorithm

In the lecture:

- Explain the role of the pairs and sets.
- Show the selection using the following pdf file: an example for pattern selection
- Overloading resolution in Ada is performed in a similar way (without costs).

Pattern Matching in Trees: Bottom-up Rewrite

Bottom-up Rewrite Systems (BURS) :
a general approach of the pattern matching method:

Specification in form of tree patterns, similar to C-3.18 - C-3.20

Set of patterns is **analyzed at generation** time.

Generator produces a **tree automaton** with a finite set of states.

On the bottom-up traversal it annotates each tree node with
a **set of states**:
those selection decisions which may lead to an optimal solution.

Decisions are made on the base of the **costs of subtrees**
rather than costs of nodes.

Generator: BURG

Lecture Compilation Methods SS 2011 / Slide 321

Objectives:

Get an idea of the BURS method

In the lecture:

- Explain the basic ideas of BURS.
- Compare it to the previous technique.
- Decides on the base of subtree costs.
- Very many similar patterns are needed.

Suggested reading:

Kastens / Übersetzerbau, Section 7.4.3

Questions:

- In what sense must the specification be complete?

Tree Pattern Matching by Parsing

The tree is represented in prefix form.

Translation patterns are specified by tuples (CFG production, code, cost),
Value descriptors are the nonterminals of the grammar, e. g.

8	RegConst ::= **addradd** Reg Const	nop	0
11	RegConst ::= **addradd** RegConst Reg	add R_i, R_j, R_k	1

Deeper patterns allow for more effective optimization:

Void ::= **assign** RegConst **addradd** Reg Const	store (Ri, c1),(Rj, c2)	1

Parsing for an ambiguous CFG:
application of a production is decided on the base of the production costs
rather than the accumulated subtree costs!

Technique „Graham, Glanville“
Generators: GG, GGSS

Lecture Compilation Methods SS 2011 / Slide 322

Objectives:

Understand the parsing approach

In the lecture:

Explain

- how a parser performs a tree matching,
- that the parser decides on the base of production costs,
- that the grammar must be complete,
- that very many similar patterns are needed.

Suggested reading:

Kastens / Übersetzerbau, Section 7.4.3

Questions:

- In what sense must the grammar be complete? What happens if it is not?
- Why is it desirable that the grammar is ambiguous?
- Why is BURS optimization more effective?