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.

Lecture Programming Languages and Compilers WS 2010/11 / Slide 701

Objectives:

Introduction of the topic

In the lecture:

The topics on the slide are explained.

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

Lecture Programming Languages and Compilers WS 2010/11 / Slide 702

Objectives:

Introduction of a calculus for formal modelling semantics

In the lecture:

Give an overview on the appoach; the roles of

- semantic domains (cf. lecture on Modelling),
- mappings E and C

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
```


Lecture Programming Languages and Compilers WS 2010/11 / Slide 703

Objectives:

Understand a simple example

In the lecture:

Explain

- the domains of the example,
- the consequences.

Lecture Programming Languages and Compilers WS 2010/11 / Slide 704

Objectives:

Understand the expression functions

In the lecture:

The expression functions on the slide are explained using the given examples.

Questions:

- How would a particular order of evaluation of operands be specified?

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
...
```


Lecture Programming Languages and Compilers WS 2010/11 / Slide 705

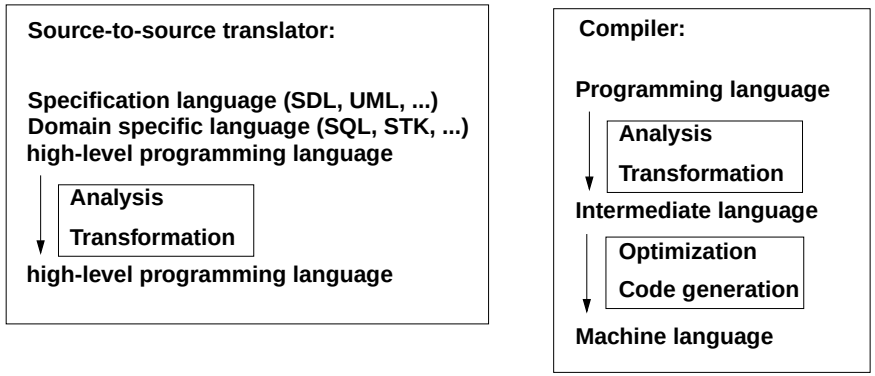
Objectives:
Understand the statement functions

In the lecture:
The domains and functions are explained:

- composition of functions,
- update of the memory,
- alternative functions,
- recursive definition of while-semantics

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

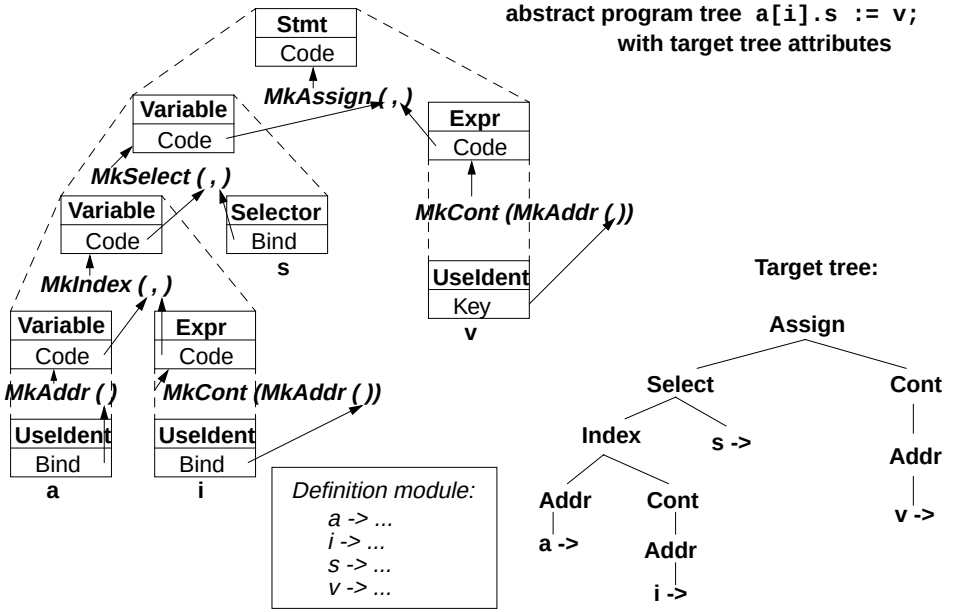
Lecture Programming Languages and Compilers WS 2010/11 / Slide 801

Objectives:
Understand the task

In the lecture:
Explain

- the notion,
- characteristics of source languages,
- comparison with compilers,
- target trees.

Example: Target tree construction



Lecture Programming Languages and Compilers WS 2010/11 / Slide 802

Objectives:

Recognize the principle of target tree construction

In the lecture:

Explain the principle using the example. Refer to the AG on PLaC-8.3.

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;
```

Lecture Programming Languages and Compilers WS 2010/11 / Slide 803

Objectives:

Attribute grammar specifies target tree construction

In the lecture:

Explain using the example of PLaC-8.2

Generator for creation of structured target texts

Tool PTG: Pattern-based Text Generator

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

1. Specify output pattern with insertion points:

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

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

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

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

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

correspondingly with attribute in the tree

3. Output of the target structure:

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


Lecture Programming Languages and Compilers WS 2010/11 / Slide 804

Objectives:

Principle of producing target text using PTG

In the lecture:

Explain the examples

Questions:

- Where can PTG be applied for tasks different from that of translators?

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>
```


Lecture Programming Languages and Compilers WS 2010/11 / Slide 805

Objectives:

See an application of PTG

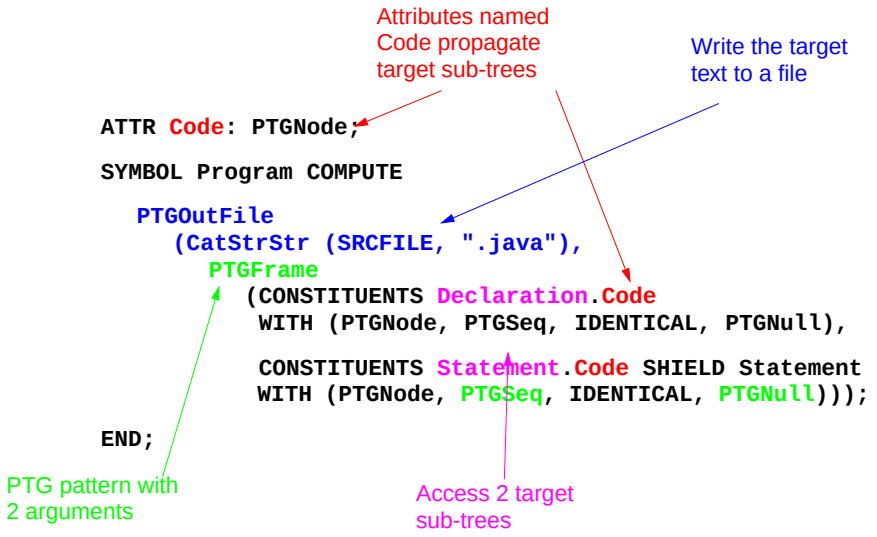
In the lecture:

Explain the patterns

Questions:

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

PTG functions build the target tree (1)



Lecture Programming Languages and Compilers WS 2010/11 / Slide 806

Objectives:

Understand the use of PTG functions for target text creation

In the lecture:

Explain the use of PTG functions in root context.

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;
```


Lecture Programming Languages and Compilers WS 2010/11 / Slide 807

Objectives:

Understand the use of PTG functions for target text creation

In the lecture:

Explain the use of PTG functions to compose the target tree.

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

Lecture Programming Languages and Compilers WS 2010/11 / Slide 808

Objectives:

Understand how to store generated names

In the lecture:

Explain the use of PTG and PDL functions.