INF225 Notes

Anya Helene Bagge
w/Ralf Lämmel, Vadim Zaytsev

v3, Fall 2016

Contents

Chapter 0

Introduction

Software Language Engineering

What's a *language*? A *software language*? What makes it *engineering*?

Languages

Wikipedia says: *Language is the human capacity for acquiring and using complex systems of communication, and a language is any specific example of such a system. The scientific study of language is called linguistics.*

Anya's definition:

- A language is a system of communication. It carries meaning (*semantics*), and has *abstractions* that allow you to communicate usefully at different levels (i.e., more than just pointing at concrete things or showing a picture of something).

A general-purpose language will allow you to define your own abstraction (e.g., by defining a class in Java, or by defining the term 'language' in English).

Formal Languages

A *formal language* has a formal definition:

- Given an alphabet $\Sigma = \{a, b, \dots\}$ (the basic letters and characters of the language)
- then $\Sigma*$ is the set of all possible strings over the alphabet,
- and $L \subset \Sigma*$ is a language (e.g., those strings over the alphabet that are valid)

For example:

- $\Sigma_{\text{sheep}} = \{\text{“b”}, \text{“æ”}\}$
- $L_{\text{sheep}} = \{\text{“bæ”}, \text{“bææ”}, \text{“bæææææ”}, \text{“bæbæ”}, \text{“bæbæææ”}, \dots\} =$ `/(bæ+)*/` is the language of sheep (bæææ!) (the thing in between the slashes is a *regular expression*, a simple grammar for simple languages, which we'll learn more about later).

A language is typically not defined by listing all valid strings, but rather by defining:

- the lexical syntax (giving the alphabet)
- a grammar for the sentence structure

and possibly also

- typing rules (excludes some sentences as meaningless)
- semantics (defines the meaning of sentences)

Software Languages

A *software language*

- is an *artificial* language (constructed by particular humans, not by cultural evolution)
- which is used in software development.

A language can be a software language even though it's not a programming language (it could be a specification language, for example). A natural language is not a software language, even though it's used in development.

Software languages are also formal languages, though we will only have to deal with some aspects of formal language theory.

There are many kinds of software languages:

- General-purpose programming languages; e.g., Java, Lisp, C, Python, ...
- Domain-specific languages; e.g., SQL, HTML, CSS
- Modelling and meta-modelling languages; e.g. UML, XML Schema
- Data representation; e.g., HTML, markup languages, XML, JSON
- Ontologies
- APIs? For example, Java Collections, Python standard library. Maybe – an API gives you communication, semantics and abstraction, but is typically not "stand-alone"; needs to be embedded in another language.

Engineering

Software Language *Engineering* means we deal with software languages in a way that is:

- systematic,
- disciplined,
- quantifiable,
- informed by science.

What's SLE really about?

What we do, research-wise:

- Development (design, implementation, testing, deployment), and
- maintenance (evolution, recovery, retirement)

of

- formal descriptions, and
- tooling

of/for software languages.

Typical SLE Activities

- Compiler construction [Wikipedia]
- Domain-specific languages [Wikipedia]
- Model-driven engineering [Wikipedia]
- Program generation [Wikipedia]
- Reverse engineering [Wikipedia]
- Reengineering, refactoring [Wikipedia]

Things to learn (maybe) in this course

- languages, grammar – syntax [Lecture 2: Syntax and Grammars]
- trees, parse-trees, abstract syntax trees
- semantics
  - static semantcs / typechecking
  - dynamic semantics / evaluation / compilation
  - scopes, environments, parameter passing
  - state; memory layout, execution model
- typing systems
  - data abstraction, classes, structs
  - dynamic vs static typing
- domain-specific languages
- data-flow analysis
  - optimisation, slicing
- macro systems
- code generation

- intermediate languages / representations
  - three-address code; trees; SSA form
- code transformation
- mapping between different abstraction levels

You should learn enough that you can get started on

- designing / evolving SLE technologies
- test and evaluate your work
- figure out what went wrong and do better next time

You *won't* learn:

- Automatons, NFAs, DFAs
- Constructing LR or LL parsers or lexers

Futher reading

- SLE conference – definition of the term software language engineering

Chapter 1

Concrete Syntax

Grammars

A *grammar* describes the syntactic aspect of a language. Formally, a grammar is a tuple $G = \langle T, N, P, S \rangle$, where P is a set of *grammar productions*; T is the set of *terminals* in P (the alphabet), N is the set of *non-terminals* in P, and S is the *start-symbol*.

Syntactially, a *language* L is the set of strings over the alphabet T that conform to the grammar G. L is a subset of $T*$ – the (infinite) set of strings over the alphabet T. Alternatively, we can define a language as the set of strings generated by the grammar G (by starting at the start-symbol, and generating all permutations).

In a *regular grammar*, the productions are of the form (with a being a terminal and A and B being non-terminals):

$$A \rightarrow aB \tag{1.1}$$

or

$$A \rightarrow Ba \tag{1.2}$$

A *linear grammar* allows both forms within the same grammar, but a regular grammar must use one or the other.

However, instead of writing productions as above, regular grammars often written using *regular expressions* (abbreviated *re*, *regex* or *regexp*). They are commonly used to define the structure of the basic words in a programming language.

For any regular grammar, we can define a *finite automaton* which recognises sentences in the regular language. Commonly, such automata are used to divide a program text into individual words or *tokens*.

In *context-free languages*, the productions are of the form:

$$A \rightarrow (a|B)* \tag{1.3}$$

Context-free grammars are used to defined the structure of the sentences of a programming language.

Concrete and Abstract Syntax

Concrete Syntax

Concrete syntax is described by a grammar; typically a context-free grammar (with the lexical sub-parts possibly described by a regular grammar). It is the syntax used when you write programs or text in a language.

A grammar is a particular implementation of a syntax – a formal set of rules that describe the syntax. The same syntax can be described in many different ways: there are many C++ grammars (and nearly all of them are wrong), but just one standard syntax.

In designing a concrete syntax, important considerations are:

- Ease of use for programmers (making it feel "natural")
- Familiarity / similarity to other languages (easier learning)
- Robustness (small typos shouldn't drastically change the meaning of a program without warning; syntax errors should be easy to identify and recover from)

For example, Java is designed to have a fairly easy to use syntax (e.g., the syntax "class A implements B" is easy to read, understand and remember) and to be familiar to C and C++ programmers. The C familiarity has a negative impact on robustness, because of C's terse syntax – parse error recovery is made difficult by the low level of redundancy, and some simple mistakes can have big consequences:

```
1 for(int i = 0; i < 10; i++); // whoops! extra semicolon
    System.out.println(i);
```

Syntax design is a language design issue. Grammar design is a language implementation issue (though the language specification will typically also come with a grammar, specifying the syntax). Important grammar design considerations are:

- Technology. If the grammar is to be used by a parser generator, it is typically limited by what the parser generator supports. These limitations can be quite severe (for example, left-recursive grammars (i.e., with $A \rightarrow AB$) can't be used directly with LL parsers).
- Readability. Various tricks to avoid technological deficiencies can make a grammar really hard to read (e.g., avoiding left recursion); using features such as priorities and the `*`, `?`, `+`, `|` and `{}` operators can make a grammar easier to read.
- Structure. Some grammars will tend to produce deeply nested trees on parsing; this can be annoying depending on how your compiler is implemented.
- Simplicity and compactness. Should every syntactical pattern that repeats itself be factored out into its own production? Or should one try to minimise the number of productions at the cost of bigger productions? This will typically impact both readability and structure.

Concrete Syntax in Rascal

Once you have defined a grammar in Rascal (or imported an existing grammar), you can write code fragments in concrete syntax within your Rascal program. Concrete syntax fragments are enclosed in back-quotes (`` `...` ``), and optionally preceded by the name of the appropriate non-terminal in parenthesis:

```
(Expr)`a + 2`
```

Pattern variables in concrete syntax fragments are enclosed in angle brackets (<...>). It may be necessary to type the variables with their corresponding non-terminal, in order to help out the Rascal parser:

```
(Expr)`<Expr a> + 2`
```

Parse Trees

The value of a concrete syntax fragment is its *parse tree* (also known as a concrete syntax tree):

```
Expr: `a + 2`
Tree: appl(prod(sort("Expr"),[sort("Expr"),...
```

Parse tree values can be printed (which yields the original code text), and manipulated just like any other Rascal data value. Translating a parse tree back to a text is called *unparsing*. You can do this in Rascal by interpolating the value in a string: `"<myTree>"`, or calling the `unparse()` function (defined in the `ParseTree` module). The parse tree type `Tree` is a subtype of the `Node` type, as are all user-defined algebraic data types.

The structure of a parse tree follows directly from the grammar and the parsing process: for every parse tree node, there is a corresponding production in the grammar that was used to parse the text for that node, and each parse tree node has one child for every symbol (terminal or non-terminal) on the right-hand side of a production rule.

The parse tree traces the sequence of productions that was used during the parse – different parsing techniques may produce different trees. In our case, Rascal uses a *generalised parsing* technique, which produces all possible trees – a parse forest (the ordering of the trees in the forest may still depend on the parser; in Rascal, the trees are unordered (in a set)).

In Rascal, the parse tree includes all the information needed to reconstruct the original input text, as well as information on all the productions used during parsing, and the source code locations of all the nodes.

Abstract Syntax

The *abstract syntax tree* (AST) of a program encodes just the information necessary to preserve the meaning of the original program text. The *abstract syntax* describes the structure of the abstract syntax tree – it can be defined using a regular tree grammar, or an algebraic data type or term (in Rascal, ML, Prolog, ...), or an object-oriented inheritance hierarchy of node classes (Java, C++, ...), or as an S-expression (in Lisp languages).

The abstract syntax tree can be used as an internal representation in a language processor, but it is not the only possible representation.

An abstract syntax can be generated by a grammar in the following way: For every non-terminal type, there is a corresponding abstract syntax type. Each type has one constructor (or node type) corresponding to each production in the grammar, with one child for every symbol in the production that is not a literal token (e.g, punctation, keywords or spaces). If a constructor has only one child, of the same type, it can be removed (e.g., this would be the case for a parenthesis expression). You can do this process entirely based on the information contained in a parse tree. Translating a parse tree into a corresponding abstract syntax tree is called *imploding* the parse tree.

Given an abstract syntax tree, it is possible to reconstruct a parse tree or program text, given the original grammar – though the resulting program may be slightly different in terms of spaces and punctation. This is called *unparsing* or *pretty-printing* (particularly if the output is nicely formatted). Parsing, imploding, pretty-printing and then reparsing may not yield the exact same parse tree as the original tree, but it should still implode to the same abstract syntax tree (otherwise there is a bug in your tool chain!).

Although the abstract syntax may be derived from a grammar, it can actually be useful to define it yourself, to capture the core constructs in the language. Multiple cases in the concrete syntax may be folded into the same constructor in the abstract syntax, and the abstract syntax may distinguish between cases that aren't syntactically distinct in the concrete syntax (e.g., in the case of overloaded operators).

It may be entirely sensible to design a language around the abstract syntax first, and then later on add a concrete syntax – this was done in the case of Lisp (where they basically ended up using a representation of the abstract syntax as a concrete syntax).

Various phases in a language processor may change the abstract syntax tree, or use slightly different versions of the abstract syntax (e.g., after type checking, the nodes for variables include the type of the variable) – it is also possible to *decorate* or *annotate* the AST as processing proceeds. This adds extra information to the nodes in the AST, without impacting the structure of the abstract syntax (this can also be done with a parse tree – in fact, the location information in Rascal is an annotation on the parse tree node).

Important abstract syntax design considerations are:

- Simplicity. Generally, your compiler tools will do a lot of work on AST, and the fewer different cases you have to worry about, the better. For example, if the processing of overloaded functions and operators is basically the same (which it is to some degree in C++), you may want to have only one AST node type to cover both. Having a lot of unnecessary nodes in the tree can be annoying as well, and may make processing slower.

- Good correspondence with the constructs of the language.

- Availability of information during processing. Some information that can be computed from the tree (such as type information) might be encoded directly in the tree (at least at later stages) for easy processing.

- Being end-user friendly or familiar to most programmers *isn't* an important consideration – the abstract syntax may be radically different from the surface concrete syntax if that helps the compiler writer.

Abstract Syntax in Rascal

An *alias type* creates a new name for an existing type:

```
1 data Var = str;
```

A *data declaration* declares a new *algebraic data type*, or adds new constructors to an already defined type:

```
1 data Val = Int(int i) | Fun(str arg, Expr body);
```

To make new values of the `Val` type, simply do: `Int(4)` or `Fun("x", expr1)`, etc. `Int` and `Fun` are known as *constructors*; they construct new objects of the type `Val`. It's fine for a type and a constructor to share the same name.

To extract the field values of a type, do `v.i` or `f.body`. You can also pattern match on algebraic data types, using the `:=` operator:

```
1 if(Fun(arg, body) := v) { ... }
```

Each field in a type should have the same type across all the constructors of the type (otherwise, we'd be confused when doing `x.f`, since the type would depend on the value of `x`). Rascal will complain loudly if you try to do this.

If you want to annotate your AST nodes, you must first declare the annotation:

```
1 anno Type Expr@typ;
```

This declares the annotation `typ` ("`type`" is a reserved keyword) on `Expr` nodes, with the type `Type`. You can create annotations that are valid on all data type constructors like this:

```
1 anno str node@doc;
```

This adds the string annotation `doc` as a valid annotation for the `node` type, which is the super type of all parse tree types and user-defined algebraic types.

You can add an annotation to an existing value `v` like this:

```
1 v = v[@doc="some string"];
```

You can read back the annotation like this:

```
1 rascal> v@doc str: "some string"
```

Parsing

Pretty Printing

Editing

Chapter 2

Evaluators & Dynamic Semantics

Semantics is the study of *meaning*, just as syntax is the study of structure. If you know the grammar of a language, you can tell whether a sentence is syntactically correct, and what its structure is. To understand its meaning, if there is any, you must know the semantics of the language.

'Meaning' can take many forms. In simple cases, we may be interested only in knowing what output we will get if we run a program with some particular input – exactly how we arrive at that output is uninteresting. In other cases, we may be interested in knowing exactly what steps are executed, and in what order. Or, we may want to know what arm movements a robot will make when we run its program. A good semantics description gives us exact, unambiguous answers to such questions, without overspecifying uninteresting details.

For example, the specification of C will tell you details of when you can be sure that values are written to memory, but leaves out other details, such as the evaluation order of expressions and the size and exact semantics of integers. The definition of a functional language will give rules for computing the value of expressions, and maybe also for how and when expressions are evaluated.

Dynamic and Static Semantics

For programming languages, we typically distinguish between *static* and *dynamic* semantics.

What is classified as static or dynamic depends a bit on the language, but in general, static semantics (Chapter 3) deals with determining whether a program makes enough sense to be executed at all, while dynamic semantics deals with what happens when you execute a program.

Static semantics may include some rules, such as typing rules, that could have been part of the grammar, but was left out to keep the grammar context-free. For example, that the types of the operands must match the declaration of an operator, or that a variable must be declared before it is used. Static semantics may also specify how the uses of names are resolved, and how overloaded names are dealt with.

```
module syn::SimplTrivial
extend syn::SimplLexical;

start syntax Program = Expr expr;

syntax Expr
  = INT num
  | bracket "(" Expr e ")"
  > left (Expr e1 "*" Expr e2 | Expr e1 "/" Expr e2)
  > left (Expr e1 "+" Expr e2 | Expr e1 "-" Expr e2)
  > left (Expr e1 "==" Expr e2 | Expr e1 "\<" Expr e2)
  | "if" Expr cond "then" e1 Expr "else" Expr e2 "end"
  ;
```

Listing 2.1: Evaluator/src/syn/SimplTrivial.rsc: Syntax for *Trivl* – a trivial expression language without variables or functions.

Dynamic semantics specifies what a program *does* – what its results are or what actions it performs. This may also entail some of what is listed under static semantics above, particularly in dynamic languages.

In this chapter, we will discuss dynamic semantics. We start by defining the semantics of simple programs that just return a value. Later we will also consider programs that work on state.

The *Simpl-Exp* Expression Language

We can specify semantics formally, using a specification formalism, or informally, using a natural language such as English. Another approach is to just give an implementation of a language, and say that the implementation is also the specification. This is often the case for languages that are under development, or for which there is only a single implementation. If the implementation is clear and readable, this can make a lot of sense, and gives a fairly decent specification – with the additional benefit that we can test our assumptions against an executable implementation.

For a widely adopted language with multiple implementations, there's a greater need for a proper specification, so that developers and language implementers can agree on the exact meaning of programs.

For us, simplicity and understanding the fundamental concepts of languages is the most important thing, so we'll start with the implementation-as-specification approach, and develop an evaluator in Rascal for a series of increasingly complex languages.

Evaluating Trivial Expressions

Let's start by considering the trivial expression language in Listing 2.1, and develop an evaluator for it.

The syntax only provides for numbers, arithmetic expressions and conditions. Valid programs include expressions such as `1 + 2 * 3`, `(2 * 3) + 7`,

Concrete Syntax and Pattern Matching in Rascal

Concrete Syntax

Rascal allows you to write parse trees and patterns using the concrete syntax of the object language (the language we're defining). Concrete syntax fragments are enclosed in back-quotes (`‘...‘`), and preceded by the name of the appropriate non-terminal in parenthesis:

```
(Expr)‘a + 2‘
```

Pattern variables in concrete syntax fragments are enclosed in angle brackets (`<...>`). The variables should be typed with the name of the corresponding non-terminal, in order to help out the Rascal parser:

```
(Expr)‘<Expr b> + 2‘
```

Variables such as `b` above are also called *meta variables* – they are variables in the host language, rather than the object language (like the variable `a` in the former code fragment).

Pattern Matching

Concrete syntax can be used in pattern matching. For example:

```
if((Expr)‘<Expr e1> + 0‘ := e) {
  e = e1;
}
```

On a successful match, the pattern variables are bound to the corresponding subtrees of the parse tree matched against.

You can also use concrete syntax patterns a the function parameter list:

```
str term((Expr)‘<Expr a>+<Expr b>‘) {
  // <> can be used inside strings to
  // ' interpolate ' values into the string.
  // They can contain any expression, incl
  // function calls
  return "Add(<term(a)>, <term(b)>)";
}
```

If you use pattern matching in parameter lists, it's a good idea to provide a default case that catches calls that don't match any of the patterns:

```
default str term(Expr e) {
  throw "unknown expression <e>";
}
```

To obtain the string corresponding to a parse tree (i.e., the input string before parsing), use the `unparse` function:

```
rascal> (Expr)‘a + b‘
  sort("Expr"): ‘a + b‘
  Tree: appl(prod(sort("Expr"),...
rascal> unparse((Expr)‘a + b‘)
  str: "a + b"
```

and `if 2 > 3 then 4 else 5 end`. We're of course free to assign any kind of meaning to our language, but to avoid surprising our users, we'll stick to a fairly conventional interpretation: A program yields a *value*, computed by applying standard integer arithmetic. For example, the value of the expression `(2 * 3) + 7` is 13.

First, we need to define what a value is. We'll make a `Value` type, which we'll use in our evaluator. Our language only supports integers, so our value type has only a single `Int` case. In Rascal, this gives us the data type:

```
8 data Value = Int(int intValue);
```
Evaluator/src/eval/EvalTrivial.rsc

To assign meaning to expressions, we define an `eval` function, with one case for each kind of expression. The function should accept an expression and return its value, giving us the signature:

```
1 public Value eval(Expr e);
```

The Trivial Evaluator

We now define evaluation functions for each language construct.

Integers: The value of an integer literal `i` (matched by the concrete syntax fragment `‘<INT i>‘`), is its integer value:

```
11 public Value eval((Expr)‘<INT i>‘) {
12   return Int(toInt(unparse(i)));
}
```
Evaluator/src/eval/EvalTrivial.rsc

We use `unparse` to make a string from a parse tree, then `toInt` to convert that string into an integer. The actual integer is wrapped in an `Int` constructor, in accordance with our `Value` data type.

Arithmetic: In the case of arithmetic expressions, we must first obtain the value of the operands, by evaluating them recursively. We can then apply the operation and wrap up the result. For example, for addition:

```
25 public Value eval((Expr)‘<Expr e1>+<Expr e2>‘) {
26   return Int(eval(e1).intValue + eval(e2).intValue);
}
```
Evaluator/src/eval/EvalTrivial.rsc

Parentheses: The value of a parenthesis expression is simply the value of the contained expression:

```
15 public Value eval((Expr)‘(<Expr e>)‘) {
16   return eval(e);
}
```
Evaluator/src/eval/EvalTrivial.rsc

Comparison: When it comes to comparison operators and conditionals, we must decide how to handle Boolean values, as our language so far only deals with integers. A convenient approach is that of C, where `0` corresponds to false, and any non-zero value is true. So, to compare two values, we first evaluate them, then do the comparison, and then select either `0` or `1` (any non-zero value would work) as the return value:

```
40 public Value eval((Expr)`<Expr e1>==<Expr e2>`) {
     return Int(eval(e1).intValue == eval(e2).intValue ? 1 : 0);
42 }
```
Evaluator/src/eval/EvalTrivial.rsc

Conditionals: For `if` expressions, we have to choose between two expressions, based on the value of the condition. We must be careful and only evaluate one of the branches. At best, evaluating both branches will result in somewhat slower execution time; worst case, one of the branches may contain code that fails in some way, or doesn't terminate. The code for `if` is:

```
51 public Value eval((Expr)
52       `if <Expr c> then <Expr e1> else <Expr e2> end`) {
     if(eval(c).intValue != 0)
54     return eval(e1);
     else
56     return eval(e2);
   }
```
Evaluator/src/eval/EvalTrivial.rsc

Programs: Finally, we provide an evaluator for programs – which is simple since a program is just an expression:

```
59 public Value eval((Program)`<Expr e>`) {
60   return eval(e);
   }
```
Evaluator/src/eval/EvalTrivial.rsc

We can now try the evaluator on a few simple programs:

```
1 rascal>eval((Program)`1+2*3`)
  Value: Int(7)
3
  rascal>eval((Program)`(1+2)*3`)
5 Value: Int(9)

7 rascal>eval((Program)`if 1 then 2 else 1/0 end`)
  Value: Int(2)
9
  rascal>eval((Program)`if 0 then 2 else 1/0 end`)
11 |rascal://eval::EvalTrivial|(1041,2,<36,37>,<36,39>):
     ArithmeticException("/ by zero")
```

Questions

1. Can you think of any drawbacks to specifying a language in this way, compared to using plain English or a mathematical formalism? What about advantages?

2. Let's say we implemented the evaluator in Java instead; using objects with an `eval` method as nodes in an abstract syntax tree representation of programs. For example, the tree node for addition would have this `eval` method:

```
module syn::SimplVars
extend syn::SimplLexical;

start syntax Program = Expr expr;

syntax Expr
  = ID var
  | INT num
  | bracket "(" Expr e ")"
  > left (Expr e1 "*" Expr e2 | Expr e1 "/" Expr e2)
  > left (Expr e1 "+" Expr e2 | Expr e1 "-" Expr e2)
  > left (Expr e1 "==" Expr e2 | Expr e1 "\<" Expr e2)
  | "if" Expr cond "then" e1 Expr "else" Expr e2 "end"
  | "let" ID var "=" Expr e1 "in" Expr e2 "end"
  ;
```

Listing 2.2: Evaluator/src/syn/SimplVars.rsc: Syntax for *SimplVars* – a trivial expression language with variables.

```
public Value eval() {
  return new IntValue(e1.eval().intValue()
                      + e2.eval().intValue());
}
```

Would this change the semantics in any way? (Hint: is there a difference between integers in Java and Rascal?)

Variables and Environments

The language in the previous section has the power of a very simple calculator, and is not very exciting. The next step is to add variables and variable declarations.

First, we will set some simple rules for variables:

- Variables are *bound* to values in a `let`-expression.
- The binding of a variable in a `let`-expression has no effect outside that expression.
- We track the currently bound variables in an *environment*.
- The value of a variable expression is the value of the variable in the environment.
- Accessing a variable that is not bound is illegal.

Next, we need to add productions to the grammar for defining and accessing variables. The extended grammar is shown in Listing 2.2.

Then, we must decide how to represent the environment. Fundamentally, its purpose is to keep track of mappings from variables to values, so we may represent it as a map from `str`ings (variable names) to `Value`s:

```
8 alias Env = map[str, Value];
```
Evaluator/src/eval/EvalVars.rsc

The environment is only used when defining and using variables. However, since other expressions may have variables uses as sub-expressions, we need to track the environment when processing other kinds of expressions as well. We modify the `eval` function to accept the current environment as a parameter:

```
1 public Value eval(Expr e, Env env);
```

We must then modify all the `eval` cases to accept the extra argument, and pass it on to recursive calls. For example, for addition:

```
33 public Value eval((Expr)`<Expr e1>+<Expr e2>`, Env env) {
34   return Int(eval(e1, env).intValue + eval(e2, env).intValue);
}
```
Evaluator/src/eval/EvalVars.rsc

We are now ready to tackle the variables themselves. To obtain the value of a variable, we look it up in the environment:

```
19 public Value eval((Expr)`<ID x>`, Env env) {
20   return env[unparse(x)];
}
```
Evaluator/src/eval/EvalVars.rsc

Again, we use `unparse` to obtain a string from the parse tree matched in the concrete syntax pattern `` `<ID x>` ``. We should also include error handling code, but we can postpone this, and rely on Rascal to throw an exception if the variable is not defined in the environment map.

The `let`-expression is slightly more complicated, but not much. It is used like this:

```
1 let x = 5+2 in x * x end
```

In the expression above, the variable `x` should be bound to the value of `5+2` while evaluating `x * x`. We implement this as follows:

```
70 public Value eval((Expr)
       `let <ID x> = <Expr e1> in <Expr e2> end`, Env env) {
72   v = eval(e1, env);
   return eval(e2, env + (unparse(x) : v));
74 }
```
Evaluator/src/eval/EvalVars.rsc

First, we evaluate the first expression (`e1`), obtaining a value `v`. Then we evaluate and return the value of the second expression (`e2`), in an environment to which we have added the binding of the variable identified by `x` to `v`.

Questions

1. What happens if we mention the same variable name we're defining in the initializer for that variable? For example:
   ```
   1 ... let x = 5+x in x * x end ...
   ```
2. What happens if we redefine a variable which is already defined? For example:

```
let x = 1 in let x = 5 in x * x end end
```

(To answer this, you need to know the semantics of Rascal maps.)

Environmental Concerns

Our language is still to simple to do useful computation, but we can use it to illustrate one important concept: the *environment*.

With the possible exception of constant expressions, the evaluation of any part of a program will depend on information from the surrounding environment. For example, an expression such as `x + 5` is meaningless on its own, as we don't know what `x` is. If we see the wider context of the expression, for example,

```
let x = 7 in
  x + 5
end
```

we can construct an environment $x \mapsto 7$ and find the value of `x + 5`.

Environments are used in both language specification and implementation. Compiler literature typically uses the term *symbol table*.

The environment is used to keep track of contextual information such as the bindings of variables, and possibly other information, like the name of the current function or class, information about language options, and so on, depending on the language. In its simplest form, as in the previous section, the environment is simply a map from variables to values.

Environment Interface

As we proceed with more advanced languages, it is useful to abstract away the details of how the environment is implemented, and focus just on how it is used. Our use of the environment in Section 2.2.3 can be summed up as the following operations:

- `env' = declare(x, v, env)` – binding a new variable `x` to the value `v` in `env`, overriding any previous binding in `env`. The result is a new environment `env'`.
- `v = lookup(x, env)` – look up a variable `x` in an environment `env`, yielding its binding `v`. The result is undefined if `x` is not bound in `env`.

We also need a way to construct new environments, and check whether a variable is bound:

- `env = newEnv()` – create a fresh, empty environment `env`
- `isDefined(x, env)` – true if the variable `x` is bound in the environmentenv

For now, we'll store data values in the environment; later, we'll also use it bind variables to function definitions, types and storage locations.

We assume that values are immutable in our implementation language (as they are in Rascal), which is why `declare` returns a new environment.

The Rascal declarations for the above operations are shown in Fig. 2.1. Note that we've parameterized the `Env` type with the type of values we'll store in the

```
public Env[&T] declare(Tree name, &T val, Env[&T] env)

public &T lookup(Tree name, Env[&T] env)

public Env[&T] newEnv(type[&T] _)

public bool isDefined(Tree name, Env[&T] env)
```

Figure 2.1: Rascal declarations for the environment operations. `&T` indicates a type parameter (can be any type). `Tree` is the type of parse trees.

environment. We also use the parse trees of variable identifiers as keys in the environment, instead of strings. This will allow us to more easily use structured names later on, and also makes it easy to provide good error messages, since parse trees carry location information.

Semantics of the Environment

Since we're dealing with semantics of languages, we should keep all our ducks in a row, and not leave the semantics of fundamental concepts like the environment unspecified. Below, we provide equational axioms that define the important characteristics of the environment operations. In the axiom, x will be any variable, v any value, and e any environment.

New Environment. Nothing is defined in an empty environment, so `isDefined` always returns false. The *lookup* operation is undefined on empty environments (we may implement it in practice using exceptions, for example):

$$\text{isDefined}(x, \text{newEnv}()) \iff \textit{false} \tag{2.1}$$
$$\text{lookup}(x, \text{newEnv}()) \iff \textit{undefined} \tag{2.2}$$

Binding a Variable. Adding a variable to the environment, and then looking up the same variable, yields the same value. Similarly for *isDefined*.

$$\text{lookup}(x, \text{declare}(x, v, e)) \iff v \tag{2.3}$$
$$\text{isDefined}(x, \text{declare}(x, v, e)) \iff \textit{true} \tag{2.4}$$

This also implies that new bindings override old ones with the same name.

Binding Unrelated Variables. Adding a variable x_2 doesn't affect the binding of the variable x_1, as long as $x_1 \neq x_2$ – i.e., we get the same result from looking up x_1 whether or not we have declared some other variable x_2.

$$\text{lookup}(x_1, \text{declare}(x_2, v, e)) \iff \text{lookup}(x_1, e), \quad \text{for } x_1 \neq x_2 \tag{2.5}$$
$$\text{defined}(x_1, \text{declare}(x_2, v, e)) \iff \text{defined}(x_1, e), \quad \text{for } x_1 \neq x_2 \tag{2.6}$$

Namespaces

We may also want to have separate namespaces for different kinds of names, such type names, variable names, module names or function names. For example, if we allow user-defined types, we could allow types and variables to

share the same name, but have separate definitions. In this case, we could use one environment for each namespace, or add an extra parameter to the operations, selecting what kind of name we're working with.

Languages like C++ and Java support named scopes (e.g., `java.lang.io`) – this complicates things quite a lot, and requires a more advanced environment structure.

Functions

Let us now turn our attention back to our little language, and add one of the features it'll need to make it in the real world: *functions*.

Functions have two key features that make them important: they *abstract*, so that we don't need to know the details of how something is done in order to use the results; and they provide *parametrization*, so that we can use the same piece of code to do many different useful computations. Additionally, having recursive functions will allow us to do arbitraty computation, even without have loop constructs in the language.

First-class values can be passed as arguments, returned, assigned to a variables and generally be used the same way other values.

In some languages functions are special entities that are declared and named, but can't be passed as arguments, stored in variables or returned from functions. This is true for Java methods, for example. Other languages, such as Lisp and Haskell, treat functions as first-class values, which can be passed around like any other value.

Lambda expressions are functions that can be defined and used without being bound to an identifier.

In our language, which is (so far) expression-oriented, the latter approach makes sense: we will treat functions as values. Functions-as-values is normally combined with allowing anonymous functions, also known as *lambda expressions*. First class functions and lambda expressions are very popular on the interwebs, so these features should help our new language on its way to success.

With functions being values, we can reuse our existing `let` construct to define functions. For example:

```
1 let f = fun x => x * x in
    f(2)
3 end
```

A function expression is introduced by the '`fun`' keyword, followed by the list of parameters, a '`=>`' sign, and the body of the function (an expression).

The updated syntax is shown in Listing 2.3. In addition to declaring functions with `let`, we can also use them directly in a call – though this is usually less useful:

```
1 (fun x => x * x)(5)
```

Functions in the Evaluator

Having extended the syntax, we now need to define the semantics of function declaration and calls by extending the evaluator.

Function Values. First of all, we need to extend our idea of what a value is to include functions:

```
1  module syn::SimplFuns
   extend syn::SimplLexical;
3
   start syntax Program = Expr expr;
5
   syntax Expr
7    = ID var
     | INT num
9    | bracket "(" Expr e ")"
     | Expr fun "(" {Expr ","}* args ")"
11   > left (Expr e1 "*" Expr e2 | Expr e1 "/" Expr e2)
     > left (Expr e1 "+" Expr e2 | Expr e1 "-" Expr e2)
13   > left (Expr e1 "==" Expr e2 | Expr e1 "\<" Expr e2)
     | "if" Expr cond "then" e1 Expr "else" Expr e2 "end"
15   | "let" ID var "=" Expr e1 "in" Expr e2 "end"
     > "fun" {ID ","}* params "=\>" Expr body
17   ;
```

Listing 2.3: Evaluator/src/syn/SimplFuns.rsc: Syntax for *SimplFuns* – a simple expression language with functions. The new syntax rules are on lines 10 (function calls) and 16 (functions).

```
10 data Value
     = Int(int intValue)
12   | Fun(list[ID] params, Expr body)
     ;
```

Evaluator/src/eval/EvalFuns.rsc

The new case for functions include a parameter list, as a list of identifiers, and a body – an expression. Both `ID` and `Expr` are parse tree types, referring to non-terminals in our grammar.

We also need to tell the evaluator how to handle this kind of value:

```
38 public Value eval((Expr)`fun <{ID ","}* params> =\> <Expr body>`,
     Env[Value] env) {
40   return Fun([p | p <- params], body);
   }
```

Evaluator/src/eval/EvalFuns.rsc

The concrete syntax match expression is slightly more complicated than we've seen before. Essentially, `<{ID ","}* params>` means "match a comma-separated list of `ID`s, and bind to the variable `params`". In the returned expression, we use `[p | p <- params]` (build a list of `p`'s, with one `p` for each `p` in `params`) – this is just a simple way of converting a comma-separated list of identifiers into a plain list of identifiers.

For example, we can evaluate a function and obtain a function value like this (we've defined an extra function `printlnValue` to print values prettily, otherwise we'll get huge parsetrees in the output):

```
1  rascal> printlnValue(eval((Expr)`fun x, y, z =\> x * y + z`,
2                            newEnv()))
```

```
Fun([(ID)`x`, (ID)`y`, (ID)`z`], (Expr)`x * y + z`)
ok
```

Function Declarations. We have no separate function declaration construct (yet); we just use the same `let` as before. Given our above code, the old code for `let` should still work. Here's the code again, updated to use the new environment operations:

```
public Value eval((Expr)
    `let <ID x> = <Expr e1> in <Expr e2> end`, Env[Value] env) {
  v = eval(e1, env);
  return eval(e2, declare(x, v, env));
}
```

Evaluator/src/eval/EvalFuns.rsc

Notice how the evaluation of `let` starts by evaluating the expression `e1`. This would be the function in the case of a function declaration. This is why we need to have an evaluation function for function values, even though it hardly does any work.

Function Calls. Now for the interesting stuff. The evaluation of a function call consists of four steps:

1. Evaluate the function expression, to get the function value
2. Evaluate the actual arguments
3. Bind the actual arguments to the formal parameters of the function
4. Evaluate the function body in the resulting environment

The Rascal code for this is:

```
public Value eval((Expr)`<Expr f>(<{Expr ","}* args>)`,
       Env[Value] env) {
  fVal = eval(f, env); // evaluate function expression
  if(Fun(params, body) := fVal) {
    fEnv = env; // set up environment

    for(<arg, param> <- zip([a | a <- args], params)) {
      argVal = eval(arg, env); // evaluate argument
      fEnv = declare(param, argVal, fEnv); // bind parameter
    }

    return eval(body, fEnv); // evaluate body in fEnv
  }
  else throw "Calling a non-function: <unparse(f)>, at <f@\loc>";
}
```

Evaluator/src/eval/EvalFuns.rsc

We have combined step 2 and 3 into one pass over the argument and parameter lists (`zip`ped together into a list of pairs). We've also added a little bit of error checking by complaining if the thing we're trying to call is not a function (i.e., evaluates to something other than a function value), but further error checking (such as checking whether the arguments match the parameter list) has been omitted. Rascal will however throw its own exceptions in the case of errors.

Example 1: Step-by-Step Function Evaluation

Consider the following code:

```
let f = fun x, y, z => x * y + z in
  f(1, 2, 3)
end
```

Its evaluation will proceed as follows, starting with evaluating the outer `let` expression:

1. First, the function expression is evaluated, yielding a function value `Fun([(ID)'x', (ID)'y', (ID)'z'], (Expr)'x * y + z')`
2. Then, the variable `f` is bound to the function value.
3. Finally, the body of the `let`, containing the function call is evaluated:
   a) First, we evalute the function expression – `f`, which yields the function value above.
   b) Then, we evaluate each argument, yielding a series of values `Int(1)`, `Int(2)`, `Int(3)`, and bind them to the parameters; `x` $\mapsto$ `Int(1)`, `y` $\mapsto$ `Int(2)`, `z` $\mapsto$ `Int(3)`.
   c) Finally, we evaluate the function body in the environment produced by the previous step. This will produce the final result, `Int(5)` ($1 * 2 + 3 = 5$).

Exercises

1. Evaluate the following programs by hand:

   a)
   ```
   let f = fun x, y => x * y in
     f(f(2, 2), 3)
   end
   ```

   b)
   ```
   let fac = fun n => if n < 1 then 1 else n*fac(n-1) end in
     fac(3)
   end
   ```

   c)
   ```
   let x = 5 in
     let f = fun a => a * x in
       f(10)
     end
   end
   ```

   d)
   ```
   let x = 5 in
     let f = fun a => a * x in
       let x = 7 in
         f(10)
       end
     end
   end
   ```

   Study the last two programs carefully, consulting our definition of function calls. Is the answer in *d)* different from that in *c)*? Which value is the `x` inside the function bound to in each case?

Scoping

A *free variable* is one that is not bound (by `let` or as a parameter) in a certain context. E.g., `x` is free in `x * 2`, but bound in `let x = 4 in x * 2 end`.

If you paid careful attention to the exercise at the end of the previous section, you will have noticed that the binding of free variables in function bodies is determined by the caller's environment, rather than by the environment that was in place at the point where the function was defined. The example program below makes this apparent:

```
1 let f = fun => x * x in
2   let x = 2 in
      f()
4 end end // result is 4
```

There is no variable `x` at the point where the function `f` is defined. Instead, `x` refers to whatever variable `x` is active when the function is called. This is called *dynamic scoping*.

Another approach is to say that identifiers refer to whatever names are accessible at the point where the function is defined. This would make the above program invalid – since `x` is undefined when `f` is being defined. Instead, we need to add an outer definition of `x`. For example:

```
1 let x = 4 in
2   let f = fun => x * x in
      let x = 2 in
4       f()
end end end // result is 16
```

Here, the inner definition of `x` has no effect, as the `x` in `f` refers to the outermost `x = 4`. This is called *lexical* or *static scoping*.

Scope Terminology

A *scope* is a collection of identifier bindings – i.e., what is captured by the environment at some point in the code or in time.

The *scope* of a declaration includes all the points in the code where the declared identifier is bound according to the declaration. In lexical scoping, the scope of a declaration is typically either textually included in the declaration construct, or extends to the nearest scoping container. This is illustrated in Fig. 2.2, where the scopes of each variable declaration is marked in the code margin. In dynamic scoping, the scope of a declaration is determined at runtime, and lasts until the program exits from the `let` or other scoping construct.

Bindings can also be *shadowed* by declaring a new variable with the same name as an in-scope variable. This results in the outer variable being temporarily out of scope while the inner variable is in scope (though, it may still be possible to access the shadowed variable, e.g., through qualified names). The shadowed variable still exists and is active – it is only inaccessible due to its name being out of scope. This forbidden by some languages, such as Java, as it may sometimes lead to confusion.

An identifier is said to be *in scope* at a given point in the code if it occurs within the scope of its declaration.

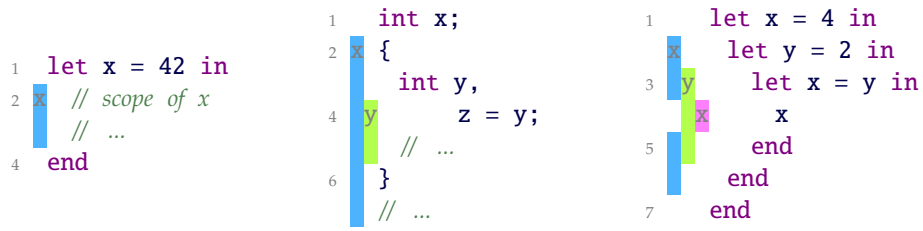


Figure 2.2: Examples of the scopes of declarations. The scope of a `let` (left) is the body of the `let`. The scope of Java variable declarations (middle) extend from immediately after the new variable is mentioned, to the end of the current block. Inner declarations may shadow outer declarations, putting some identifiers temporarily out of scope (right).

Dynamic Scoping

Dynamic scoping is seldom used nowadays, as it tends to lead to confusing programs. With dynamic scoping, you can't fully understand what a piece of code does without also looking at context in which it is used. This is particularly problematic if functions can be defined and rebound on the fly.

In an evaluator-based approach such as ours, dynamic scoping is very easy to implement – it's what you end up with if you don't do anything in particular to deal with scoping.

- For dynamic scoping, we only need to know the environment that is in place at the time a function is called. There is no need to know anything about the environment as it was when the function was called.
- Thus, our representation of functions from Section 2.4.1 is sufficient:

```
12    | Fun(list[ID] params, Expr body)
```
Evaluator/src/eval/EvalFuns.rsc

- Emacs Lisp, and some older versions of Lisp use dynamic scoping.

Lexical Scoping

With lexical scoping, the binding of an identifier follows from the surrounding program text – i.e., the lexical context. This is typically clearer for the programmer, and it also allows us to statically check whether a program uses variables that haven't been declared.

Lexical scoping complicates the dynamic semantics somewhat. We can no longer evaluate a function call without knowing what was in scope at the time the function was declared. For this, we need to store the *definition environment* of the function together with its body, so we can later restore the environment. This means extending the definition of function values:

```
10 data Value
     = Int(int intValue)
12   | Fun(list[ID] params, Expr body, Env[Value] env)
     ;
```
Evaluator/src/eval/EvalLex.rsc

Function declarations. Next, we must extend the evaluation of function expressions to save the environment in the function value:

```
38 Value eval((Expr)`fun <{ID ","}* params> =\> <Expr body>`,
     Env[Value] env) {
40   return Fun([p | p <- params], body, env);
   }
```

Evaluator/src/eval/EvalLex.rsc

Function calls. Finally, the evaluation of function calls must be changed, so that the function body gets evaluated in the function's own environment:

```
43 Value eval((Expr)`<Expr f>(<{Expr ","}* args>)`, Env[Value] env) {
44   fVal = eval(f, env); // evaluate function expression
     if(fVal is Fun) {
46     fEnv = fVal.env; // function environment
       for(<arg, param> <- zip([a | a <- args], fVal.params)) {
48       argVal = eval(arg, env); // evaluate argument
         fEnv = declare(param, argVal, fEnv); // bind parameter
50     }

52     return eval(fVal.body, fEnv); // evaluate body in fEnv
     }
54   else throw "Calling a non-function: <unparse(f)>, at <f@\loc>";
   }
```

Evaluator/src/eval/EvalLex.rsc

The main change compared to the dynamic scoping version of function calls, is that we now use a separate `fEnv` for evaluating the body. This environment is initialised with the saved declaration environment of the function (`fVal.env`). As before, the arguments are evaluated in the caller's environment (`env`) and bound in the callee environment (`fEnv`).

Recursion. If you study the rules for function values and declarations carefully, you will see that the `env` that gets added to the function value doesn't contain the binding of the function itself. For example, in the following piece of code, we'll evalute `fun a => f(a)` first, resulting in a function value capturing the environment as it is before we bind the `f`:

```
1 let f = fun a => f(a) in ... end
```

This means that `f` is not bound within its own body.

An easy solution to this problem is to just add the binding of the function to the environment while evaluating the call:

```
58 Value eval((Expr)`<Expr f>(<{Expr ","}* args>)`, Env[Value] env) {
     fVal = eval(f, env); // evaluate function expression
60   if(fVal is Fun) {
       fEnv = fVal.env; // function environment
62     if((Expr)`<ID fName>` := f) { // check if f is a name
         fEnv = declare(fName, fVal, fEnv); // for recursive calls
64     }
       for(<arg, param> <- zip([a | a <- args], fVal.params)) {
66       argVal = eval(arg, env); // evaluate argument
```

```
      fEnv = declare(param, argVal, fEnv); // bind parameter
68   }

70   return eval(fVal.body, fEnv); // evaluate body in fEnv
  }
72  else throw "Calling a non-function: <unparse(f)>, at <f@\loc>";
}
```

Evaluator/src/eval/EvalLex.rsc

This works nicely for simple recursion, but fails for mutual recursive, e.g., if `f` calls `g` which calls `f` again. We will deal with this problem later.

- We might also mix scope rules; for example, we could make functions lexically scoped, so that any function call in the body of a function is resolved at definition time, while still having dynamically scoped variables.

Imperative Languages

Store and References

See code from Lecture 11:
https://bitbucket.org/anyahelene/inf225public/src/master/inf225l11?at=master
See exercise 4A,C,D:
https://bitbucket.org/anyahelene/inf225public/wiki/Exercise%204

Advanced Scoping

Closures

See exercise 4B:
https://bitbucket.org/anyahelene/inf225public/wiki/Exercise%204

Nested Scopes

See code from Lecture 10:
https://bitbucket.org/anyahelene/inf225public/src/master/inf225l10?at=master

The Power of the Simpl-Exp Language

You may think that what we've made so far is just a toy, but it is in fact a fairly powerful programming language – at least if we're primarily interested in manipulating integers.

Algorithms

Simple algorithms like computing the Fibonacci numbers are easy to implement:

```
let fib = fun n => if n < 2
          then 1
          else fib(n-1)+fib(n-2)
          end
in
  fib(10)
end // result is 89
```

Encoding Data Structures

We can create simple data structures by encoding them as functions that return fields based on their arguments. For example, a pair data structure would be a function returning the first field if the argument is `0` and the second field otherwise. We can then make getters and setters (the setters construct new pairs – we can't make mutable data structures) on top of this crude interface:

```
let pair = fun x, y =>
  fun i => if i then y else x end
in
let getX = fun p => p(0) in
let getY = fun p => p(1) in
let setX = fun p, x =>
             pair(x, getY(p)) in
let setY = fun p, y =>
             pair(getX(p), y) in
  getY(setY(pair(17,19),21))
end end end end end // result is 21
```

This only works with lexical scoping, since the function returned by `pair` needs access to the `x` and `y` at the time it was created.

With functions being values, we could of course also store functions in data structures. This brings us within reach of object-orientation and dynamic dispatch (with very ugly syntax).

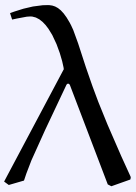
This approach is similar to *Church encoding*, which can be used to encode data structures, booleans, conditionals and even integers as functions in lambda calculus.

The Let Construct

The `let` construct is strictly speaking not necessary, since functions provide the same power of binding identifiers. Any `let` declaration `let x = e1 in e2 end` is exactly equivalent to `(fun x => e2)(e1)` – i.e., making a function with the variable as a parameter and the `let`-body as the function body, and then immediately calling it with the initialiser expression as the argument.

Lambda Calculus

The language we've created is closely related to lambda calculus, which is also the basis of Lisp, Scheme, and functional programming languages. The combination of scoping rules and the ability to create anonymous functions gives us far more power than one might expect.



Chapter 3

Typecheckers & Static Semantics

Static versus Dynamic Typing

In a statically typed language, typechecking happens at compile time. Statically typed languages typically have these properties:

- Variables and data structure fields must be declared before use.
- Variables and data structure fields can only hold values of the declared type.
- Operations (functions, procedures, methods) and types must be declared. In some languages, such as C and Pascal, declarations must be placed textually before uses of the names (this makes it easier to process a source file in a single pass). Other languages, such as Java, allow such declarations to occur in any order.
- Declaring the exact types of variables and operations may or may not be needed (some languages to type inference).

By contrast, a dynamically typed language does no checking before a program is run – though it may still do strict checking while the program is running. For a dynamic language, typing rules are part of the dynamic semantics, which we have covered in the previous chapter. Dynamic languages typically have these properties:

- A variable need not be declared before it is used.
- A variable can hold values of any type, and assigning to the variable may change its type.
- Fields in structures and classes are likewise liberal when it comes to using them with values of different types.
- Fields may or may not need to be declared before use.
- Operations must be declared before they are called.

```
module syn::SimplVarsTyped
extend syn::SimplLexical;

start syntax Program = Expr expr;

syntax Expr
  = Name var
  | INT intVal
  | Bool boolVal
  | bracket "(" Expr e ")"
  > left (Expr e1 "*" Expr e2 | Expr e1 "/" Expr e2)
  > left (Expr e1 "+" Expr e2 | Expr e1 "-" Expr e2)
  > left (Expr e1 "==" Expr e2 | Expr e1 "\<" Expr e2)
  | "if" Expr cond "then" e1 Expr "else" Expr e2 "end"
  | "let" TypeExpr Name var "=" Expr e1 "in" Expr e2 "end"
  ;

syntax Name = ID \ Bool;

syntax Bool = "true" | "false";

syntax TypeExpr
  = "int"
  | "bool"
  ;
```

Listing 3.1: Evaluator/src/syn/SimplVarsTyped.rsc: Syntax for *SimplVarsTyped* – a simple expression language with types.

Some languages are *weakly typed*, meaning that most operations accept values of any type which are then either adapted to the type the operation needs (*coercing*), or the operation fails unceremoniously. Javascript is an example of this – you can use a string where an integer is needed and vice versa. Most statically typed languages and many dynamic languages are *strictly typed*. In this case, operations require exactly matching types (modulo certain well-defined *promotions* and *conversions*). For example, you cannot call the add function with a string, even if that string happens to contain the text representation of a number.

Simple Typechecking

Let's illustrate typechecking on a simple language similar to the one we worked on in Chapter 2. The overall scheme is similar to the one we used when we implemented the evaluator – in fact, we can think of typechecking as a form of simplified evaluation where the values are types.

Listing 3.1 shows the syntax of the language we'll start with. It's similar to what we had in Section 2.2.3, but with added syntax for types and typed variable declarations, and for boolean literals.

First of all, we need to define what a type is. In order to make the language slightly more interesting, we've added booleans to the the language, so we have two types, `int` and `bool`:

```
12 data Type = Int() | Bool();
```
Evaluator/src/check/CheckVarsTyped.rsc

Next, we define a series of recursive `checkExpr` functions, one for each kind of expression. Just like in the evaluator, we'll need an environment `Env[Type]`, to keep track of the types of variables.

Base case: In the default case, we just throw an exception. This will help us in debugging.

```
14 public default Type checkExpr(Expr expr, Env[Type] env) {
     throw "Unknown expr: <[expr]>";
16 }
```
Evaluator/src/check/CheckVarsTyped.rsc

Integer literals: The type of an integer is always `int`:

```
19 public Type checkExpr((Expr)`<INT i>`, Env[Type] env) {
20   return Int();
   }
```
Evaluator/src/check/CheckVarsTyped.rsc

Note that we don't care about the actual value of the integer at all, only about the type.

Boolean literals: The type of a boolean is always `bool`:

```
24 public Type checkExpr((Expr)`<Bool b>`, Env[Type] env) {
     return Bool();
26 }
```
Evaluator/src/check/CheckVarsTyped.rsc

Arithmetic: In the case of arithmetic expressions, we must first obtain the type of the operands, by checking them recursively. Arithmetic operators require integer arguments and provide an integer result. For example, for addition:

```
52 public Type checkExpr((Expr)`<Expr e1>+<Expr e2>`, Env[Type] env) {
     t1 = checkExpr(e1, env);
54   t2 = checkExpr(e2, env);

56   if(t1 == Int() && t2 == Int())
       return Int();
58   else
       throw "Type error: expected (int, int), got (<t1>, <t2>)";
60 }
```
Evaluator/src/check/CheckVarsTyped.rsc

Parentheses: The type of a parenthesis expression is simply the type of the contained expression:

```
36 public Type checkExpr((Expr)`(<Expr e>)`, Env[Type] env) {
     return checkExpr(e, env);
38 }
```
Evaluator/src/check/CheckVarsTyped.rsc

Comparison: Unlike the language we evaluated in Chapter 2, this language has Boolean values, which comes into play when we consider comparisons and conditionals. Comparison requires integer arguments and gives a Boolean result. For example, for the less-than operator:

```
96 public Type checkExpr((Expr)‘<Expr e1>\<<Expr e2>‘, Env[Type] env) {
     t1 = checkExpr(e1, env);
98   t2 = checkExpr(e2, env);

100  if(t1 == Int() && t2 == Int())
       return Bool();
102  else
       throw "Type error: expected (int, int), got (<t1>, <t2>)";
104 }
```

Evaluator/src/check/CheckVarsTyped.rsc

Equivalence: The equals operator also gives a Boolean result. But it makes sense to be able to apply it to both Booleans and integers, so we will say that the operands must have the same type:

```
85 public Type checkExpr((Expr)‘<Expr e1>==<Expr e2>‘, Env[Type] env) {
86   t1 = checkExpr(e1, env);
     t2 = checkExpr(e2, env);
88
     if(t1 == t2)
90     return Bool();
     else
92     throw "Type error: operands have different types (<t1>, <t2>)";
   }
```

Evaluator/src/check/CheckVarsTyped.rsc

Conditionals: The `if` expression selects one of two branch values depending on the condition. The condition should be a Boolean value.

The types of the branches don't matter when we consider the `if` expression in isolation, but they are important when we consider that the result may be used in a larger expression. Somehow, the branches must have 'compatible' types, so that the program is type correct regardless of which branch is chosen. For our simple language, this means that the `if` branches should both have the same type.

```
108 public Type checkExpr((Expr)
        ‘if <Expr c> then <Expr e1> else <Expr e2> end‘, Env[Type] env) {
110   if(checkExpr(c, env) != Bool())
        throw "Condition must be bool";
112
    t1 = checkExpr(e1, env);
114 t2 = checkExpr(e2, env);

116 if(t1 == t2)
      return t1;
118 else
      throw "Type error: branches have different types (<t1>, <t2>)";
```

```
120 }
```
Evaluator/src/check/CheckVarsTyped.rsc

Note that, unlike the evaluator which would only evaluate one of the branches, we typecheck both branches.

Variables: The type of a variable is its type in the environment, if any. If the variable is undefined (not in the environment), the program is incorrect and we should give an error message.

```
29 public Type checkExpr((Expr)‘<ID x>‘, Env[Type] env) {
30   if(isDefined(x, env))
       return lookup(x, env);
32   else
       throw "Undefined variable: <x>";
34 }
```
Evaluator/src/check/CheckVarsTyped.rsc

Variable declarations: A `let`-expression has a type name, a variable name, an initialiser expression and a body expression. We must first check that the type of the initialiser expression corresponds to the declared type. Then, we can add the variable to the environment in which we typecheck the body:

```
125 public Type checkExpr((Expr)
126       ‘let <TypeExpr te> <ID x> = <Expr e1> in <Expr e2> end‘,
          Env[Type] env) {
128   typ = checkType(te, env);
      t1 = checkExpr(e1, env);
130   if(typ != t1)
        throw "Type mismatch declaring <x>: expected <typ>, got <t1>";
132
      return checkExpr(e2, declare(x, typ, env));
134 }
```
Evaluator/src/check/CheckVarsTyped.rsc

This code makes use of an auxiliary function `checkType` which checks the `TypeExpr` and returns the corresponding type. This is needed because the `TypeExpr` is just a parse tree fragment – we need to convert it to our abstract representation (`Int()` or `Bool()`). Additionally, if our language was a little bit more complicated, we might have to check the if the type expressions are well formed, for example, whether a user-defined type has been properly declared.

The code for `checkType` is:

```
136 public Type checkType((TypeExpr)‘int‘, Env[Type] env) {
      return Int();
138 }

140 public Type checkType((TypeExpr)‘bool‘, Env[Type] env) {
      return Bool();
142 }
```
Evaluator/src/check/CheckVarsTyped.rsc

```
module syn::SimplFunsTyped
extend syn::SimplLexical;

start syntax Program = Expr expr;

syntax Expr
  = Name var
  | INT intVal
  | Bool boolVal
  | bracket "(" Expr e ")"
  | Expr fun "(" {Expr ","}* args ")"
  > left (Expr e1 "*" Expr e2 | Expr e1 "/" Expr e2)
  > left (Expr e1 "+" Expr e2 | Expr e1 "-" Expr e2)
  > left (Expr e1 "==" Expr e2 | Expr e1 "\<" Expr e2)
  | "if" Expr cond "then" e1 Expr "else" Expr e2 "end"
  | "let" TypeExpr Name var "=" Expr e1 "in" Expr e2 "end"
  > "fun" {Param ","}* params "=\>" Expr body
  ;

syntax Param
  = TypeExpr paramType ID paramName
  ;

syntax Name = ID \ Bool;

syntax Bool = "true" | "false";

syntax TypeExpr
  = "int"
  | "bool"
  | TypeExpr "(" {TypeExpr ","}* ")"
  ;
```

Listing 3.2: Evaluator/src/syn/SimplFunsTyped.rsc: Syntax for *SimplFunsTyped* – a simple expression language with functions and types. Function parameter declarations must include types, but to reduce syntacic clutter, we automatically infer the return type of functions.

Functions

Let us now consider typechecking of a language with functions. Listing 3.2 shows the extended syntax, similar to the language in Section 2.4.

There are two new kinds of expressions to typecheck: functions and function calls. Since functions are values in our language, we need to provide a type for them, hence we should extend our idea of types to include function types:

```
data Type
  = Int()
```

```
14    | Bool()
      | Fun(list[Type] params, Type retType)
16    ;
```
Evaluator/src/check/CheckFunsTyped.rsc

The type of a function depends on the type of its parameters and its return type. For example, `int(int,bool)` – represented abstractly as `Fun([Int(),Bool()], Int())` – would be the type of a function that accepts an `int` and a `bool` as arguments and returns an `int`.

Functions: A function expression has a list of parameter declarations and a body. We need to check the parameter types, then bind the parameters to their respective types and use the resulting environment to check the body. The return type of the function is the type of the body expression.

```
44 public Type checkExpr((Expr)`fun <{Param ","}* params> =\> <Expr body>`,
       Env[Type] env) {
46     list[Type] paramTypes = [];
       for((Param)`<TypeExpr te> <ID p>` <- params) {
48       t = checkType(te, env);
         env = declare(p, t, env); // for use in checking body
50       paramTypes += t; // accumulate list of parameter types
       }
52     retType = checkExpr(body, env);

54   return Fun(paramTypes, retType);
   }
```
Evaluator/src/check/CheckFunsTyped.rsc

In the evaluator, we just stored the function body, since there was nothing we could do with it until we had values for the parameters. Here, we can check that the body is valid based on the parameter types, without knowing the argument values.

We might want to have an additional wellness constraint of function declarations: that names in the parameter list are unique. We could do this, for example, by maintaining a list of the names seen so far, and then checking if the next parameter we examine has already been used:

```
66       if(p in paramNames)
           throw "Duplicate parameter name <p>";
68       else
           paramNames += p;
```
Evaluator/src/check/CheckFunsTyped.rsc

Function calls: A function calls consists of a function expression and a list of argument expressions. To typecheck, we need to type check the subexpressions, then match the types of the argument expressions with the parameter types of the function.

Since functions are just expressions, we need to handle the case where we try to call something which is not a function. We must also deal with the possibility of a call being made with the wrong number or wrong type of arguments.

The type of a function call is the return type of the function called.

```
76 public Type checkExpr((Expr)`<Expr f>(<{Expr ","}* args>)`,
       Env[Type] env) {
```

```
78   fType = checkExpr(f, env); // get function type

80   if(Fun(paramTypes, retType) := fType) {
       if(size([a | a <- args]) != size(paramTypes))
82       throw "Wrong number of arguments for <f>";
       // match each argument against the corresponding parameter type
84     for(<arg, paramType> <- zip([a | a <- args], paramTypes)) {
         argType = checkExpr(arg, env);
86       if(argType != paramType)
           throw "Type mismatch in call to <f>: expected <paramType>, got <argType>";
88     }

90     return retType; // type of call is return type of function
     }
92   else throw "Attempt to call a non-function: <f>";
}
```

Evaluator/src/check/CheckFunsTyped.rsc

Design Issues

Our little language has a few design issues. First of all, the `let` syntax is a little cumbersome, particularly when it comes to functions:

```
let int(int,int) f =
  fun int x, int y => x + y
in
  f(2,3)
end
```

There isn't really any need to declare the type of the variable, since we can always obtain it from the type of the initialiser expression:

```
let f = fun int x, int y => x + y in f(2,3) end
```

This kind of simple inference of variable types is common in languages where type expressions can get very complicated (e.g., newer editions of C++ and Java).

Another approach is to introduce a new syntax for function declarations, such as, for example:

```
let int f(int x, int y) => x + y in f(2,3) end
```

This second form is useful when we try to tackle the other problem our language design has: there's no way to make recursive functions, since the variable in a `let`-expression is only bound inside the `let` body, not inside the initialiser (which is where the function body is).

A common way to fix this recursion problem is to introduce a new construct, `letrec`, which changes the binding rules so that the new variable is also bound inside the initialiser expression. Typically, `letrec` also allows multiple variable bindings, allowing for mutually recursive functions:

```
letrec int f(int x, int y) => x + g(y)
       int g(int x) => f(x, x)
```

```
3 in f(2,3) end
```

More examples

See code from Lecture 12 and 13:
https://bitbucket.org/anyahelene/inf225public/src/master/inf225l12?at=master
https://bitbucket.org/anyahelene/inf225public/src/master/inf225l13?at=master

Appendix A

Glossary

Entires marked with a star* is not part of the curriculum. Entries **in bold** are particularly important for the exam.

Term Definitions

Abstract data type*
TYPES

A type which is defined only through an interface of operations used to manipulate its value, and where the data representation is hidden.

[Wikipedia]

Abstract syntax tree
ABSTRACTION, PARSING, SYNTAX

A tree representation of the syntactic structure of a sentence; similar to a parse tree, but usually ignoring literal nonterminals and nodes corresponding to productions that don't directly contribute to the structure of the language (e.g., parenteses, productions used to encode operator priority and so on). May represent a slightly simpler language that the parse tree, for example, with operator calls desugared to function calls, and variations of a construct folded into one. Can be represented using as trees or terms, and described by an algebraic data type or a regular tree grammar.

[Wikipedia]

Abstract value*
ABSTRACTION, SEMANTICS

A value (of an abstract data type) known only through the operations used to create it. The user of an abstract data type operates on abstract values; only the implementation of the abstract data type sees the concrete value. A single abstract value may have many different concrete representations (either because there are several implementations of the data type, or because several representations mean the same – for example, $1/2 = 2/4$)

Abstraction*
ABSTRACTION

Focusing on relevant details while leaving irrelevant details unspecified or hidden away. *Data abstraction* (e.g., classes and interfaces) hides the details of how a data structure is represented, instead giving an interface to manipulate the data. *Control abstraction* (e.g., functions, methods) hides *how* something is done (the algorithm), focusing instead on *what* is done.

[Wikipedia]

Algebraic data type
SEMANTICS, TYPES

A composite data type defined inductively from other types. Typically,

each type has a number of cases or alternatives, which each case having a *constructor* with zero or more arguments.

For example, `data Expr = Int(int i) | Plus(Expr a, Expr b)`. The data type can be seen as an algebraic *signature*, with the expressions written using the constructors being *terms*. For our purposes, values may be interpreted as trees, with the constructor name being node labels, arguments being children, and atomic values and nullary constructors being leaves.

[Wikipedia]

Ambiguous grammar
AMBIGUITY, SYNTAX

A grammar for which there is a string which has more than one leftmost derivation.

– *Parsing* Requires a generalised parser, produces a parse forest.

[Wikipedia]

Analytic grammar*
PARSING, SYNTAX

A grammar which corresponds directly to the structure and semantics of a parser. For example, parser combinators and parsing expression grammars (PEGs).

[Wikipedia]

Anonymous function
SEMANTICS

A function occuring as a value, without being bound (directly) to a name. C.f. closure.

Application binary interface*
COMPILATION

Specifies how software modules or components interact with each other at the machine code level. Typically includes such things function calling conventions (whether arguments are passed on the stack or in registers, and so on), the binary layout of data structures and how system calls are done.

[Wikipedia]

Application programming interface*
ABSTRACTION

Specifies how software modules or components interact with each other.

[Wikipedia]

Associativity
AMBIGUITY, SYNTAX

A property of binary operators in parsing, indicating whether expressions such as $a+b+c$ should be interpreted as $(a+b)+b$ (left associative), $a+(b+c)$ (right associative) or as illegal (non-associative).

– *Left* Operations are grouped on the left, giving a tree which is "heavy" on the left side; typically used for arithmetic operators. Without associativity rules, grammar productions usually look like `PlusExpr ::= PlusExpr "+" MultExpr | ...`, forcing any expression containing the operator to be on the left side.

– *Right* Operations are grouped on the right, giving a tree which is "heavy" on the right side; typically used for assignment and exponentiation operators.

[Wikipedia]

Associativity rule
AMBIGUITY, SYNTAX

A disambiguation rule stating that an operator is either left-, right- or non-associative. E.g., in Rascal: `syntax Expr = left (Expr "*" Expr | Expr "/" Expr );`

Attribute grammar
PARSING, SYNTAX

A grammar where each production rule has attached attributes that are evaluated whenever the production rule is used in parsing. Can be used, for example, to build an intermediate representation directly from the

parser, or to do typechecking while parsing.

[Wikipedia]

Backend
COMPILATION

The final stage of a compiler or language processor, often tasked with low-level optimisation and code generation targeted at a particular machine architecture.

Backus-Naur form
SYNTAX

A formal notation for grammars, where productions are written `<symbol> ::= <symbol1> "literal" ...`. Often extended with support for repetition (`*`, `+`), optionality (`?` or `[]`) and alternatives (`|`).

[Wikipedia]

Bottom-up parser
PARSING

A parser that works by identifying the lowest-level details first, rather than working top-down from the start symbol. For example, an LR parser.

[Wikipedia]

Chomsky normal form*
SYNTAX

A simplified form of grammars where all the production rules are of the form $A \to BC$ or $A \to a$ or $S \to \epsilon$, where A, B and C are nonterminals (with neither B nor C being the start symbol), a is a terminal, ϵ is the empty string, and S is the start symbol. The third rule is only applicable if the empty string is in the language. Any context-free grammar can be converted to this form, and any grammar in this form is context free.

[Wikipedia]

Closure
SEMANTICS

A function (or other operation) packaged together with all the variables it can access from the surrounding scope in which it was defined. See Section 2.7.1. A related term is anonymous function, which does not necessarily imply access to variables from the surrounding scope.

[Wikipedia]

Composite data type
TYPES

A data type constructed from other types (or itself, in the case of a recursive data type), e.g., a structure or an algebraic data type.

Context-free grammar
SYNTAX

A formal grammar in which every production rule has a form of $A \to w$, where A is a single nonterminal symbol and w is a sequence of terminals and nonterminals.

[Wikipedia]

Continuation*
SEMANTICS

An abstract representation of the control state of a program. Often used in the sense of first-class continuations which let the execution state of a program be saved, passed around and then later restored.

[Wikipedia]

Cross-cutting concern*
ABSTRACTION

A programming concern, such as logging or security, which impacts many parts of a program and is difficult to decompose (cleanly separate into a library) from the rest of the system.

[Wikipedia]

Dangling else problem
AMBIGUITY, SYNTAX

A common ambiguity in programming languages (particularly those with C-like syntax) in which an optional `else` clause may be interpreted as belonging to more than one `if` sentence. Usually resolved in favour of

the closest `if`, often by an implicit disambiguation rule (at least in non-generalised parsing).

[Wikipedia]

Declarative programming*

A programming paradigm where programs are built by expressing the logic of a computation rather than the control flow; i.e., *what* should be done, rather than *how*.

[Wikipedia]

Definite clause grammar*
Syntax

A way of expressing grammars in logic programming languages such as Prolog.

[Wikipedia]

Derivation
Parsing

A sequence of production rule applications that rewrites the start symbol into the input string (i.e., by replacing a nonterminal symbol by its expansion at each step). This can be seen as a trace of a parser's actions or as a proof that the string belongs to the language.

– *Leftmost* A derivation where the leftmost nonterminal symbol is selected at every rewrite step.

– *Rightmost* A derivation where the rightmost nonterminal symbol is selected at every rewrite step.

Desugaring
Syntax, Transformation

Removal of syntactic sugar. Sometimes used in a frontend to translate convenient language constructs used by the programmer into more fundamental constructs. For example, translating Java's enhanced `for` into a more basic iterator use.

[Wikipedia]

Deterministic context-free grammar*
Ambiguity, Syntax

A context-free grammar that can be derived from a deterministic pushdown automaton (DPDA). Always unambiguous.

[Wikipedia]

Disambiguation rule
Ambiguity, Parsing, Syntax

Used to resolve ambiguities in a grammar, so that the parser yields a single unambiguous parse tree. Includes techniques such as follow restrictions, precede restrictions, priority rules, associativity rules, keyword reservation and implicit rules.

Domain-specific language
Abstraction, Languages

A language (i.e., not just a library) with abstractions targeted at a specific problem domain.

– *Benefits* Easier programming, more efficient or secure, possibly better error reports

– *Drawbacks* Lots of implementation work, language fragmentation, learning/training issues, less tooling, troublesome interoperability, possibly worse error reports

– *External DSL* A DSL defined as a separate programming language.

– *Internal or embedded DSL* A DSL defined as language-like interface to library.

[Wikipedia]

Duck typing
Types

A typing style where the exact type of an object is not important, rather, any object is usable in any situation as long as it supports whatever methods are called on it. Used in many dynamic languages, such as Python, and in C++ templates. C.f. structural typing.

Named after the *duck test* (attributed to James Whitcomb Riley): "When I see a bird that walks like a duck and swims like a duck and

quacks like a duck, I call that bird a duck."

[Wikipedia]

Dynamic dispatch
Compilation, Languages, Semantics

The process of selecting, at runtime, which implementation of a method to call at runtime; typically based on the the actual class of the object on which the method is called (as opposed to the static type of the variable). With *multiple dispatch*, the selection is done based on some or all arguments, making it a kind of runtime overload resolution.

[Wikipedia]

Dynamic language
Languages

A language where most or all of the language semantics is processed at runtime, including aspects such as name binding and typing. May have features such as duck typing, dynamic typing, runtime reflection and introspection, and often allows code to be replaced and objects to be extended at runtime.

[Wikipedia]

Dynamic scoping
Languages, Semantics

When names are resolved by finding the closest binding in the runtime environment (i.e., the execution stack), rather than in the local lexical environment (i.e., the containing scopes at the use site). C.f. lexical scoping.

[Wikipedia]

Dynamic semantics
Semantics

Gives the meaning of a program at execution time; either in terms of values being computed, actions being performed and so on.

[Wikipedia]

Dynamic typing
Types

When type safety is enforced at runtime. Values are associated with type information, which can also be used for other purposes, such as runtime reflection. Used in languages such as Python, Ruby, Lisp, Perl, etc.

– *Benefits* Compiler may run faster; easy to load code dynamically at runtime; allows some things that are type safe but are still excluded by a static type system; easy to use duck typing to get naturally generic code with little overhead for the programmer; reflection, introspection and metaprogramming becomes easier.

– *Drawbacks* Type errors cannot be detected at compile time; rigorous testing is needed to avoid type errors; some optimisations may be difficult to perform (less of a problem with just-in-time compilation).

Dynamic typing does not imply weak typing. C.f. static typing, duck typing.

[Wikipedia]

Environment
Semantics

A mapping of names to values or types. Used in evaluation and type-checking to carry name bindings. The environment is passed around (propagated) according to the scoping rules of the language, and may use a more complicated data structure to accomodate nested and/or named scopes.

[Wikipedia]

Epsilon
Syntax

In a grammar, the empty string.

Evaluator (also Interpreter)
Compilation, Languages, Semantics

A program that executes another program.

[Wikipedia]

Extended Backus-Naur form*
Syntax

A syntax notation introduced by Niklaus Wirth, which includes notation for optionality, repetition, alternation, grouping and so on.

[Wikipedia]

Field
Types

A data member of a data structure.

Follow restriction
Ambiguity, Syntax

A disambiguation technique where a symbol is forbidden from or forced to be immediately followed by a certain terminal.

Formation rule
Semantics, Syntax

A grammar; rules for describing which strings are valid in a language. This term is used mainly in logic.

Frontend
Compilation

The first stage of a compiler or language processor, typically including a parser (possibly with a tokeniser), and a typechecker (semantic analyser). Sometimes also includes desugaring. Is typically responsible for giving the programmer feedback on errors, and translating to the internal AST or representation used by the rest of the system.

Function
Semantics

An abstraction over expressions (or more generally, over expressions, statements and algorithms).

Function type
Semantics, Types

The representation of a function in the type system. Typically includes parameter types and return type, written $t_1, ..., t_k \rightarrow t$.

Function value
Semantics

The representation of a function in an evaluator or in a dynamic semantics specification. Usually includes the parameter names and the function body. Forms a closure together with an environment giving the function's declaration scope.

Functional programming
Languages

A programming paradigm based on mathematical functions, usually without state and mutable variables. Pure functional languages have referential transparency, and typically allows Higher-order functions.

[Wikipedia]

Generalised parser
Ambiguity, Parsing

A parser that can handle the full range of context-free grammars, including nondeterministic and ambiguous grammars. For example, a GLL parser or a GLR parser.

Generative grammar*
Syntax

An approach to specifying the syntax of the language, using a set of rules that produce all strings in the language. Formalised by Noam Chomsky in the late 1950s, but the idea goes back to Pāṇini's Sanskrit grammar, 4th century BCE.

[Wikipedia]

Generative programming*
Abstraction, Languages, Transformation

Programming aided by automatic generation of code, including techniques such as generic programming, templates, aspects, code generation, etc.

[Wikipedia]

Generic programming*
ABSTRACTION, LANGUAGES

A programming style which allows the same piece of code to deal with many different types, for example through polymorphism.

[Wikipedia]

GLL parser*
PARSING

An LL parsing algorithm extended to handle nondeterministic and ambiguous grammars, making it capable of parsing any context-free grammar. Unlike normal LL or recursive descent parsers, it can also handle left recursion.

[WWW]

GLR parser*
PARSING

An LR parsing algorithm extended to handle nondeterministic and ambiguous grammars, making it capable of parsing any context-free grammar.

[Wikipedia]

Grammar
SYNTAX

A formal set of rules defining the syntax of a language. Formally, a tuple $\langle N, T, P, S \rangle$ of nonterminal symbols N, terminal symbols T, production rules P, and a start symbol $S \in N$. In software languages, the most frequently used kinds are context-free and regular grammars.

[Wikipedia]

Grammar in a broad sense*
SYNTAX

A structural description in software systems, and a description of structures used in software systems: a parser specification is an enriched grammar, a type definition is a grammar, an attribute grammar comprises a grammar, a class diagram can be considered a grammar, a metamodel must contain a grammar, an algebraic signature is a grammar, an algebraic data type is a grammar, a generalized algebraic data type is a very powerful grammar, a graph grammar and a tree grammar are grammars for visual concrete notation, an object grammar contains two grammars and a mapping between them.

[Paper: Toward an Engineering Discipline for Grammarware]

Higher-order function
LANGUAGES

A function which takes takes functions as arguments or returns function values.

[Wikipedia]

Imperative programming
LANGUAGES

A programming paradigm based on statements that change program state; as opposed to declarative programming. May be combined with object-oriented programming.

[Wikipedia]

Implicit disambiguation rule
AMBIGUITY, PARSING, SYNTAX

A disambiguation rule built into the parser, such as longest match for regular expressions, or resolving the dangling else problem by preferring shift over reduce in an LR parser.

Inheritance
ABSTRACTION, LANGUAGES

A technique in object-oriented programming which combines automatic code reuse with subtyping.

[Wikipedia]

Inlining
ABSTRACTION, COMPILATION, TRANSFORMATION

A technique in language processing where a call to a function or procedure is replaced by the code being called. Often used as part of code optimisation; removes abstraction introduced by the programmer.

[Wikipedia]

Island grammar*
Syntax

A grammar which describes only small parts of a language, skipping over the rest. Used, for example, to recover documentation from a program.

[Program Transformation Wiki]

Just-in-time compilation*
Compilation

A technique used in interpreted or bytecode-compiled languages where program code is compiled at runtime, during evaluation. This gives the speed advantages of compilation, while retaining the dynamic flexibility and architecture-independence of a interpreted or bytecode-based language. Can sometimes give even better performance than static compilation, since more information may be available at runtime, leading to better optimisation. Heavily used in modern implementations of JVM and .NET.

[Wikipedia]

Kleene closure
Syntax

A metasyntactic sugar for repetition: `x*` means that `x` can be repeated zero or more times. The language that the Kleene star generates, is a monoid with concatenation as the binary operation and epsilon as the identity element.

[Wikipedia]

Language
Languages

A system of communication, with structure (syntax) and meaning (semantics), and abstractions that allow you to communicate usefully at different levels (i.e., more than just pointing at concrete things or showing a picture of something).

Left factoring*
Syntax

A technique used to avoid backtracking in top-down parsing, by ensuring that the productions for a nonterminal don't have alternatives that start with the same terminals.. Used to massage a grammar to LL form.

[Wikipedia]

Left recursion*
Syntax

When production rules have the form $A \rightarrow A\alpha|b$, with the Nonterminal occurring directly or indirectly to the left of all terminals on the right-hand side. Must be eliminated in order to use an LL parser.

[Wikipedia]

Lexeme
Syntax

A string of characters that is significant as a group; a word or token.

Lexical analysis (also scanner, lexer or tokeniser)
Compilation, Parsing

Converting a sequence of characters (letters) to a sequence of tokens (lexemes or words).

Lexical scoping (also Static scoping)
Languages, Semantics

When names are resolved (possibly statically) by finding the closest binding in the lexical environment (i.e., by looking at the scopes that lexically contains the name). C.f. dynamic scoping.

[Wikipedia]

Lexical syntax
Syntax

Describes (often using a Regular grammar) the syntax of tokens; e.g., what constitutes an identifier, a number, different operators and the whitespace that separates them.

[Wikipedia]

Literate programming*
LANGUAGES, SYNTAX

A programming style where programs can be read as documents that explain the implementation, with explanations in a natural language. Tools allow programs to be compiled as either code or documents.

[Wikipedia]

LL parser
PARSING

A table-driven top-down parser, similar to a recursive descent parser. Has trouble dealing with left recursion in production rules, so the grammar must typically be left factored prior to use. The LL parser reads its input in one direction (left-to-right) and produces a leftmost derivation, hence the name LL. Often referred to as LL(k), where the k indicates the number of tokens of lookahead the parser uses to avoid backtracking.

[Wikipedia]

LL grammar
SYNTAX

A grammar that can be parsed by a LL parser.

[Wikipedia]

Logic programming

A declarative programming paradigm based on formal logic, inference and reasoning. Useful for many purposes, including formal specification of language semantics. Prolog is the most well-known logic language.

[Wikipedia]

LR parser
PARSING

A bottom-up parser that can handle deterministic context-free languages in linear time. Common variantes are LALR parsers and SLR parsers. It reads its input in one direction (left-to-right) and produces a rightmost derivation, hence the name LR.

[Wikipedia]

LR grammar
SYNTAX

A grammar that can be parsed by a LR parser.

[Wikipedia]

Massaging*
PARSING, SYNTAX, TRANSFORMATION

The act of modifying a grammar to make it fit a particular technology or purpose.

Megamodel*

A result of megamodelling — a model which elements are software languages, models, metamodels, transformations, etc

[Paper: On the Need for Megamodels]

Member (also Field)
TYPES

An element of a structure or class; a field, method or inner class/-type.

Method (also Member function)
LANGUAGES, SEMANTICS

A function which is a member of a class. Typically receives a self-reference to an object as an implicit argument.

[Wikipedia]

Mixin*

A partial class (data fields and methods) that can be used to plug functionality into another class using inheritance.

[Wikipedia]

Multi-paradigm programming*
LANGUAGES

Programming which combines several paradigms, such as functional, imperative, object-oriented or logic programming. Languages that support multiple paradigms include, for example, C++, Scala, Oz, Lisp, and many others.

[Wikipedia]

Name binding
COMPILATION, SEMANTICS

A part of language processing where names are associated with their declarations, according to scoping and namespace rules. A name's binding is typically determined by checking the environment at the use site.

– *Static* When done statically (or *early*), name binding is often combined with typechecking.

– *Dynamic* Names are bound at runtime; also applies to dynamic dispatch (where it is sometimes called *late* or *virtual* binding), where certain properties (such as types) may be known statically, but the exact operation called is determined at runtime.

[Wikipedia]

Named tuple
TYPES

A tuple where the elements are named, like in a structure. Often exhibits structural type equivalence, even in languages that normally use nominative type equivalence

Namespace
SEMANTICS

Some kind name grouping that makes it possible to distinguish different uses of the same name. For example, having variable names be distinct from type names; or treating names in one module as distinct from the same names in another module (In this sense, namespaces are related to scope).

[Wikipedia]

Nominative type equivalence* (also Nominal/Nominative type system)
TYPES

A system where type equivalence or compatibility is determined based on the type names (or, more strictly, which declaration the names refer to) and not the structure of the type. C.f. structural type equivalence.

[Wikipedia]

Nonterminal footprint*
SYNTAX

A non-recursive measure of nonterminal symbol usage in a grammatical expression: a multiset of presence indicators (1 for the nonterminal itself, ? for its optional use, * for its Kleene closure, etc). A usefulness of a footprint for grammar matching depends on how rich the metalanguage is.

Nonterminal symbol
SYNTAX

A symbol in a grammar which is defined by a production. Can be replaced by terminal symbols by applying the production rules of the grammar. In a context-free grammar, the left-hand side of a production rule consists of a single nonterminal symbol.

Object-oriented programming*
LANGUAGES

A programming paradigm based on modelling interactions between objects. Objects have fields and methods and encapsulate state. Provides data abstraction, and usually supports inheritance, dynamic dispatch and subtype subtype polymorphism.

[Wikipedia]

Optimisation
COMPILATION, TRANSFORMATION

The process of transforming program code to make it more efficient, in terms of time or space or both.

[Wikipedia]

Overloading
SEMANTICS

When the same name is used for multiple things (of the same kind). For example, several functions with the same name, distinguished based on the parameter types. C.f. namespace, where the same name can have different meanings in different context (e.g.,

type names are distinct from variable names).

[Wikipedia]

Overload resolution
COMPILATION, SEMANTICS

A compilation step, usually combined with typechecking, where the name of an overloaded function is resolved based on the types of the actual arguments. C.f. dynamic dispatch, which does something similar at runtime.

[Wikipedia]

Parse forest
AMBIGUITY, PARSING, SYNTAX

The parse trees that are the result of parsing an ambiguous grammar using a generalised parser.

Parse tree
PARSING, SYNTAX

A tree that shows the structure of a string according to a grammar. The tree contains both the tokens of the original string, and a trace of the derivation steps of the parse, thus showing how the string is a valid parse according to the grammar. Typically, each leaf corresponds to a token, each interior node corresponds to a production rule, and the root node to a production rule of the start symbol.

Parser
PARSING

A program that recognises input according to some grammar, checking that it conforms to the syntax and builds a structured representation of the input.

Parser combinator*
PARSING

A way of expression a grammar and a parser using higher-order functions. Each combinator accepts parsers as arguments and returns a parser.

[Wikipedia]

Parsing expression grammar*
PARSING, SYNTAX

A form of analytic grammar, giving rules that can be directly applied to parse a string top-down. Similar to a context-free grammar, but the rules are unambiguously interpreted; for example, alternatives are tried in order. Related to parse combinators. PEGs are a useful and straightforward technique for parsing software languages.

It is suspected that there are context-free languages that cannot be parsed by a PEG, but this has not been proved.

[Wikipedia]

Parsing
PARSING

Recovering the grammatical structure of a string. The task done by a parser.

Pattern matching
LANGUAGES, TRANSFORMATION

A technique for comparing (typically algebraic) data structures, where one or both structures may contain variables (sometimes refered to as meta-variables). Upon successful match, variables are bound to the corresponding substructure from the other side. Related to unification in Prolog, but often more restricted.

[Wikipedia]

Polymorphism*
LANGUAGES, TYPES

– *Ad hoc* Another name for function or operator overloading.

– *Parametric* When a function or data type is generic and handle values of different types in the same; for instance, `List<T>` – a list with elements of an arbitrary type. The specific type in question is often given as a parameter, hence the name.

– *Subtype* When an object belonging to a subtype can be used in a place

where the supertype is expected (as with classes and inheritance in object-oriented programming).

[Wikipedia]

Precede restriction
AMBIGUITY, SYNTAX

A disambiguation technique where a symbol is forbidden from or forced to be immediately preceded by a certain terminal.

Predictive parser
PARSING

A recursive descent parser which does not require backtracking. Instead, it looks ahead a finite number of tokens and decides which parsing function should be called next. The grammar must be LL(k) for this to work, where k is the maximum lookahead.

Priority rule
AMBIGUITY, PARSING, SYNTAX

A disambiguation rule declaring an operator's priority/precedence. E.g., in Rascal: `syntax Expr = Expr "*" Expr > Expr "+" Expr;`

Procedural programming

A programming paradigm based around procedure calls. Sometimes considered the same as imperative programming and typically based on structured programming.

[Wikipedia]

Production rule
SYNTAX

A rule describing which symbols may replace other symbols in a grammar. In a context-free grammar, the left-hand side consists of a single nonterminal symbol, while the right-hand side may be any sequence of terminals and nonterminals. For example, `Expr ::= Expr "+" Expr` says that anywhere you may have an expression, you can have an expression plus another expression. The rules may be used to generate syntactically correct strings, by applying them as rewrite rules starting with the start symbol, or be used to parse strings, e.g., in a top-down parser or bottom-up parser.

Program slicing
TRANSFORMATION

A program transformation technique where all code that is irrelevant to a certain set of inputs or outputs is removed. Applied *forwards*, any code not directly or indirectly using a selection of inputs is discarded; applied *backwards*, all code that does not contribute to the computation of the selected outputs is discarded. Mainly used in debugging (e.g., to find the code that might have contributed to an error), but sometimes also as an optimisation technique. Originally formalised by Mark Weiser in the early 1980s.

[Wikipedia]

Recogniser
PARSING

A program that recognises input according to some grammar, giving an error if it does not conform to the grammar, but does not build a data structure.

Record, Record type
TYPES

See Structure, Structure type.

[Wikipedia]

Recursive data type*
TYPES

A composite data type, such as an algebraic data type which may contain itself. Used, for example, to define data structures such as lists and trees.

Recursive descent parser
PARSING

A top-down parser built from mutually recursive functions, where each function typically implements

one production rule of the grammar.

[Wikipedia]

Referential transparency
LANGUAGES, SEMANTICS

When an expression can be replaced by its value without changing the meaning of the program; i.e., it will evaluate to the same value every time and not cause side effects. Usually a property of functional programming languages.

[Wikipedia]

Regular expression
SYNTAX

A formalism for describing a regular grammar, using the normal alphabet mixed with special *metasyntactic* symbols, such as the Kleene star. Commonly used to specify the lexical syntax of a language, and also for searching and string matching in many different applications.

[Wikipedia]

Regular grammar
SYNTAX

A formal grammar where every production rule has the form $A \rightarrow aB$ (for a *right regular grammar*), or $A \rightarrow a$ or $A \rightarrow \epsilon$, where A and B are nonterminal symbols and a is a terminal symbol, and ϵ is the empty string. Alternatively, the first production form may be $A \rightarrow Ba$, for a *left regular grammar*.

– *Limitations* Can't express arbitrary nesting, such as nested parentheses or block structure in a language.

[Wikipedia]

Reserve rule
AMBIGUITY, SYNTAX

A disambiguation rule which states that a grammar symbol cannot match some constraint. For example, identifiers could be defined as any word matching `[a-zA-Z]+` *except* `if`, `while`, ...

Scannerful parsing
PARSING

Parsing is divided into two parts; a tokeniser that deals with the lexical syntax and a parser that deals with the sentence syntax.

– *Benefits* Faster than scannerless parsing, because the lexical syntax is specified with a regular grammar which can be parsed very efficiently.

– *Drawbacks* Cannot deal with arbitrary composition of languages.

Scannerless parsing
AMBIGUITY, PARSING, SYNTAX

When scanning and parsing is unified into one process that deals with with the input characters directly.

– *Benefits* Can parse combinations of languages that have different lexical syntax. Lexical syntax can be context-free, not just regular.

– *Drawbacks* Slower than scannerful parsing. Can lead to hard to find lexical ambiguities.

Scope
SEMANTICS

A collection of identifier bindings – i.e., what is captured by the environment at some point in the code or in time.

– *Nested* With nested scopes, variables in inner scopes may *shadow* those in outer scopes, and variables are removed as control flows out of the scope. Variable shadowing may be forbidden in some languages.

– *Named* With named scopes, we can refer to names in scopes that are not ancestors of the current scope. For example, with C++ classes and namespaces and Java packages and (static) classes.

[Wikipedia]

Semantic analysis
COMPILATION, SEMANTICS

A phase of language processing that enforces the static semantics

of a language. Includes typechecking, name binding, overload resolution and checking other static constraints.

Software Language
LANGUAGES

An artificial language used in software development.For example, Java (programming language), HTML (markup language), XML (data language), CSS (domain-specific language).

Start symbol
SYNTAX

The nonterminal symbol in a grammar that generates all valid strings in a language.

Static semantics
SEMANTICS, TYPES

The part of language semantics which is processed at compile time (statically). Often includes constraints that might be part of the syntax, but which is done separately in order to keep the grammar context-free. Includes concepts like name binding and typechecking, and is used to eliminate a large class of invalid or erroneous programs. See static typing.

Static typing
TYPES

When type safety is enforced at compile type (though some tests, such as for typecasting, may be done at runtime).

– *Benefits* Detects a large an important class of errors (type errors) at compile time; enables advanced optimisations and efficient memory use.

– *Drawbacks* Type system may become either overly complicated or overly restrictive; doesn't help with non-type errors; makes dynamic loading of code somewhat more complicated; type declarations may be cumbersome if the language lacks type inference.

Strong typing
TYPES

When a language (to some degree) enforces type safety.

[Wikipedia]

Structural type equivalence* (also Structural type system)
TYPES

A system where two types are equal or compatible if they have the same structure; e.g., have the same fields with the same types in the same order. C.f. nominative type equivalence.

[Wikipedia]

Structure, Structure type (also Record, Record type)
TYPES

A composite data type with named fields members; such as `struct` in C or `record` in Pascal. Similar to (or same as, with structural type equivalence) a named tuple.

[Wikipedia]

Structured programming*
LANGUAGES

A programming paradigm where the clarity of programs is improved by nestable language constructs like `if`, `while`, as opposed to conditional jumps.

[Wikipedia]

Syntactic sugar
SYNTAX

See Desugaring.

[Wikipedia]

Terminal symbol
SYNTAX

An elementary symbol in the language defined by a grammar, which cannot be changed/matched by the production rules in the grammar (i.e., the symbol doesn't occur (alone) on the left-hand side of a production). Corresponds to a token or an element of the alphabet of a language.

Token
PARSING

A lexeme or group of characters that forms a basic unit of parsing, categorised according to type, e.g., identifier, number, addition operator, etc. Forms the alphabet of the parser in scannerful parsing.

[Wikipedia]

Tokeniser (also lexer or scanner)
PARSING

A program that performs lexical analysis, grouping and classifying input into tokens.

Top-down parser
PARSING

A parser using a strategy where the top-level constructs are recognised first, starting with the start symbol. The parser starts at the root of the parse tree, and builds it top-down, according to the rules of the grammar. Includes LL parsing and Recursive descent parsing.

Typechecker
COMPILATION, TYPES, SEMANTICS

A program that detects type errors, ensuring type safety in a statically typed language. Often combined with other static semantic checks and processing, such as overload resolution, name binding, and checking access restrictions on names. Can be seen as a form of abstract interpretation, where the abstract values are the types, and all operations are defined according to their static semantics (i.e., type signature) rather than their dynamic semantics. C.f. semantic analysis.

[Wikipedia]

Type inference
TYPES

Automatic deduction of the types in a language. Used with static typing to avoid having to declare types for variables and functions. Particularly useful in generic programming and type polymorphism where type expressions can become quite complicated. Used, e.g., in Haskell and Standard ML.

[Wikipedia]

Type safety
LANGUAGES, TYPES

Whether a language protects against type errors, such as when a value of one data type is interpreted as another type (e.g., an `int` as a `float` or as a pointer to a string).

[Wikipedia]

Typecasting* (also type conversion)
TYPES

Forced conversion from one type to another. In a languages with weaker type systems, may lead to data being misinterpreted.

[Wikipedia]

Unification*
LANGUAGES

A pattern matching-like operation, where the goal is to find an assignment of variables so that two terms become equal. Used heavily in Prolog.

[Wikipedia]

Weak typing
TYPES

When a language (to some degree) does not enforces type safety.

[Wikipedia]

Yield*
PARSING, TRANSFORMATION

The *yield* of a parsetree is the unparsed input text.

Tag Index

Abstraction

Ambiguity

Compilation

Languages

Parsing

Semantics

Appendix B

Overview of the Course, Fall 2013

What is a language?

A language is a form of communication, with structure (syntax) and meaning (semantics).

For our purposes:

- A formal, human-made language
- Has a grammar
- Can be parsed
- Can be processed by a computer

Formal language definition

- It's described by a grammar: a tuple
  - Non-terminals – kinds of phrases
  - Terminals – tokens, words
  - Start symbol
  - Productions – depending on kind of grammar, for example
  - Context-free N := (N | T)*
  - Regular: N := T N or N := N T
- It's all the strings over the terminals which are valid according to the grammar

Syntax

Languages and Grammars

Discussed in: Lession 1, Lession 2, Exercise 1

- Context-free and regular grammars – used in defining the syntax of formal languages

- Grammars describe the *concrete syntax* of a language
- Parsing determines if a text is legal with respect to the grammar and builds a tree/structure for further processing
- Context-free grammars are typically written in (some variant of) BNF – Backus-Naur Form; regular grammars are typically written as regular expressions.

Classes of languages

- Regular – can be parsed by a finite-state machine; extremely efficient
  - Can't specify that things should be nested. E.g., count parentheses – this would require a stack.
- Context-Free – can be parsed by a push-down automaton (FSM + a stack); very efficient
  - Can't specify that something should be declared before use (incl. type checking), or support user-define delimiters (e.g., as in MIME multipart email messages).
  - Solution: do it anyway, and check after parsing.
- Context-Sensitive – Linear-bounded non-deterministic Turing machine; complicated
- Recursively enumerable – turing machine; very complicated

Context-Free Languages

- Used in programming languages – most programming langs more or less context-free
  - C++ is slighty outside
  - Many languages are in subclass LALR
  - Grammars in standards often don't correspond to those in the implementations, and might be internally inconsistent
  - In practice, parsers for CFGs in realistic languages are often hand-implemented. This is due to the need for good error reporting.
- A context-free language has a context-free grammar. In practice, the language is context-sensitive, but this is dealt with after syntax processing.
- The syntax of a CFG defines the structure; says nothing about the semantics.

Regular Languages

- Can't defined languages with nested constructs.

Parsing

Discussed in: Exercise 1

- Scanner/lexer – for regular grammars. Can be implemented using a finite-state machine (FSM)
- Parser – for context-free grammars. Can be implemented using a push-down automaton (PDA – an FSM with a stack)
- Scannerless parsing – used in Rascal and SDF2/SGLR. Useful in language composition and language extension, and for some language like Markdown, TeX, various Wikis etc.
- Parser generators – create an executable parser automatically from a grammar
- Generalised parsers – ([GLR][], GLL) can parse any context-free grammar. Rascal, SDF2/SGLR and Bison are systems with generalised parsing support. Useful in language composition and language extension.
- Classical parsers ([LALR][], [LL][]) can only parse a subset of context-free grammars – and combining or extending grammars may require extensive rewrites. Some example systems are Yacc and Bison (both LALR) and ANTLR (LL(k) / LL(*)).

Parse Trees

Discussed in: Lession 2

- A structured representation of a text, with respect to a grammar (and the parsing process)
- Contains details of the concrete syntax

Abstract Syntax Trees

- A structured representation of a program, without
  - details related to concrete syntax (which is needed to unambiguously encode the program as text),
  - details related to the parsing scheme or the parsing process.
- May be derived from the grammar (concrete syntax);
- Or, may be designed as a program representation unrelated to the concrete syntax.

Generalised parsing

- Parses the entire class of context-free grammars
- Produces a *parse forest*
- Can be used for context-sensitive languages:
  - Make a context-free grammar with ambiguities
  - Resolve ambiguities later
- Reasons why you'd use it:
  - Grammars can be a lot nicer, useful in a standard or a textbook
  - You can combine different languages, for example in a meta-programming system
  - Tradeoffs: can be slower (n^3), parse forest generation can take time
  - Grammars are prettier, might be easier to write and easier to modify
  - Non-generalised grammars might need a lot of changes to do a small modification
  - CFG are closed under composition. If A, B are CFGs, then A + B is a CFG. Not true for restricted classes like LALR.
  - But: parse results may be ambiguous, which is normally undesirable. Might be hard to debug

Ambiguities

Discussed in: Exercise 2

- An ambiguous grammar has more than one valid parse tree for a valid input. For example, `a + b + c` – should this be `(a + b) + c` or `a + (b + c)`?
- If a grammar is ambiguous, generalised parsing produces a parse forest instead of a tree – all the possible trees that can be obtained.
- Simpler parsing schemes typically don't allow ambiguous grammars.
- The union of two unambiguous grammars can be an ambiguous one.
- Ambiguous grammars are difficult to detect statically (undecidable in general).
- In Rascal and with SDF2/SGLR, you'll find `amb` nodes in the parse tree if there is an ambiguity.
- Expression ambiguities, like with `a + b + c` are typically resolved with operator precedence rules and associativity rules.
- Other cases may be more tricky, such as with the dangling else problem.

Precedence / Priorities

Discussed in: Exercise 2

Scanners vs. Scannerless

Discussed in: Lesson 6

Spaces and Layout

Discussed in: Lesson 6, Exercise 2

Domain-Specific Languages (DSLs)

Discussed in: Lesson 5

- What – targeted at and restricted to domain, has a language interface.
- Why – easier programming, more efficient or secure, possibly better error reports.
- Why not – lots of implementation work, language fragmentation, learning/training issues, less tooling, troublesome interoperability.
- How – external DSL (separate programming language), internal DSL (language-like interface as a library).

Semantics

- Syntax gives structure, semantics gives meaning
- Can be specified formally, for example using (big-step) operational semantics. Can also be done informally, e.g., using English text, or via a reference implementation.
- The nicer, more structured your implementation is, the better it specifies the semantics.

Types

- Used to validate programs in static semantics
- Used to specify data layout (and for safety checks) in dynamic semantics
- We've looked at
  - Plain data types; ints, etc.
  - Function types
  - Structure types (records) – a little bit
- We can have various degrees of strong or weak typing (strong being usual nowadays), and static and dynamic typing (both are popular nowadays).

Dynamic semantics

- Dynamic semantics gives the meaning of the program at execution time.
- We've specified it using evaluators
- Used to define code generation in a compiler
- Dynamic semantic rules tell you:
  - How things are computed
  - How environment and store are propagated throughout the computation

Static semantics

Includes:

- Additional constraints on what a legal or meaningful program is
- Rules for which names refer to what; names are linked to declarations – this is used to work around the context-freeness of the syntax.

...

- Semantics "executed" at compile time, using types instead of values.
- Reject meaningless programs
- We've dealt with static semantics in typecheckers; it "evaluates" the static semantics, gives you yes/no answer to wellformedness / legality, and possibly a annotated program.
- Only in *static(ally typed) languages*, not in dynamic languages like Python or Lisp.
- Important to code generation – used to figure out the memory layout and size of variables, and how they should be allocated.

Environment and Store

- Environment contains bound variables – variables that correspond to a declaration
- Variables may be bound to values or locations (if we have a store).
- If every step in the program execution produces a new environment, we can change the value of variables without having a store. But we'd need a store to have *references*.
- Store is an abstraction of the machine memory.
  - Store has locations
  - Each location contains a value

  - Each program step may update the store, resulting in a new store
  - (In a language with a store) Variables are typically bound to store locations.
  - Get value from store on variable use, put value in store when assigning to variable
  - This is used for *references*. When the value of a variable is a location, you have references in your language.
  - In real life: store is heap memory and stack. Locations are addresses, either relative to stack pointer, or on the heap (main memory). Stack is used for temporary / local variables. Heap is used in dynamic memory allocation (e.g., `new` in C++ and Java, `malloc` in C).
  - You could use an array or a map to implement it. (Array would most machine-like)
  - Static vs. dynamic scoping – what is the environment used when evaluating a function?
  - Compilers typically implement the enivornment as a symbol table – either a simple hash table, or a more complex structure with support for nested scopes.

- *Garbage collection* is going through the heap (store) in order to find values that can't be accessed anymore (by any available location). Often done by a *mark and sweep* algorithm.
- Reasoning about semantics is very difficult if you allow multiple variables to point to the same location. This is called *aliasing*. Compilers will try to do *alias analysis* in order to do optimisations. This is not a problem in languages with immutable values (e.g., Haskell), or without aliasing (e.g., Magnolia).

Appendix C

Formal Semantics

This chapter gives a quick introduction to how to do formal semantics, using the running example from INF225, Fall 2011. The notation is a variant of big-step structural operational semantics.

A good reference if you want to learn more is Hüttel [1].

This is not officially part of the syllabus, but we have discussed it in the lectures, so you should know a little bit about it.

Specification of MyLang – the Simple Version

Syntax of MyLang

```
lexical INT = [0-9]+;
lexical VAR = [a-zA-Z_][a-zA-Z0-9_]*;
lexical LAYOUT = [\t-\n \r \ ];
layout LAYOUTLIST
    = LAYOUT* !>> [\t-\n \r \ ];

syntax Program = Expr;

syntax Expr
    = VAR
    | INT
    | "(" Expr ")"
    | VAR "(" Expr ")"
    > left (
        Expr "*" Expr
      | Expr "/" Expr
    )
    > left (
        Expr "+" Expr
      | Expr "-" Expr
    )
    > "let" VAR "=" Expr "in" Expr
```

```
23      | "let" VAR "(" VAR ")" "=" Expr "in" Expr
        | "if" Expr "then" Expr "else" Expr
25      ;
```

Operational Semantics of MyLang

Used symbols and variables (possibly with subscripts):

- X : `VAR` – variables
- I : `INT` – integers
- E : `Expr` – expressions
- v : Val – values
- e : Env$v$ – environments
- $\Rightarrow$: Expr, Env$v$ $\rightarrow$ Val – evaluation

Each *inference rule* has a *conclusion* of the form $\texttt{A} \overset{e}{\Longrightarrow} a$, meaning "the expression `A` evaluates to the value a in the environment e". The *premises*, above the horizontal line, must hold (i.e., be inferable from the other rules) in order for the conclusion to hold.

See Section 2.3 for a definition of the environment.

Integers The value of a literal integer is the integer's value:

$$\texttt{I} \overset{e}{\Longrightarrow} \texttt{I} \tag{C.1}$$

Variables The value of a variable is the value of that variable in the environment (or undefined if the variable is unbound in the environment):

$$\frac{v = \text{lookup}(\texttt{X}, e)}{\texttt{X} \overset{e}{\Longrightarrow} v} \tag{C.2}$$

Parenthesis The value of a parenthesis expression is the value of the expression between the parenthesis, evaluated in the current environment.

$$\frac{\texttt{E} \overset{e}{\Longrightarrow} v}{\texttt{(E)} \overset{e}{\Longrightarrow} v} \tag{C.3}$$

Addition The value of a plus expression is the sum of the values of the operand expressions evaluated in the current environment.

$$\frac{\texttt{E}_1 \overset{e}{\Longrightarrow} v_1 \qquad \texttt{E}_2 \overset{e}{\Longrightarrow} v_2}{\texttt{E}_1 \texttt{ + } \texttt{E}_2 \overset{e}{\Longrightarrow} v_1 + v_2} \tag{C.4}$$

Subtraction The value of a minus expression is the difference of the values of the operand expressions evaluated in the current environment.

$$\frac{\texttt{E}_1 \overset{e}{\Longrightarrow} v_1 \qquad \texttt{E}_2 \overset{e}{\Longrightarrow} v_2}{\texttt{E}_1 \texttt{ - } \texttt{E}_2 \overset{e}{\Longrightarrow} v_1 - v_2} \tag{C.5}$$

Multiplication The value of a multiplication expression is the product of the values of the operand expressions evaluated in the current environment.

$$\frac{\mathtt{E}_1 \overset{e}{\Longrightarrow} v_1 \qquad \mathtt{E}_2 \overset{e}{\Longrightarrow} v_2}{\mathtt{E}_1 \ \mathtt{*} \ \mathtt{E}_2 \overset{e}{\Longrightarrow} v_1 \times v_2} \tag{C.6}$$

Division The value of a division expression is the quotient of the values of the operand expressions evaluated in the current environment.

$$\frac{\mathtt{E}_1 \overset{e}{\Longrightarrow} v_1 \qquad \mathtt{E}_2 \overset{e}{\Longrightarrow} v_2}{\mathtt{E}_1 \mathtt{/} \ \mathtt{E}_2 \overset{e}{\Longrightarrow} \frac{v_1}{v_2}} \tag{C.7}$$

Conditionals The value of a conditional expression on the value of the condition $\mathtt{E}_c$ in the current environment. If the condition is different from 0, the value is the value of the then-branch in the current environment, otherwise the value of the else-branch.

$$\frac{\mathtt{E}_c \overset{e}{\Longrightarrow} v_c \qquad v_c \neq 0 \qquad \mathtt{E}_t \overset{e}{\Longrightarrow} v_t}{\mathtt{if}\ \mathtt{E}_c\ \mathtt{then}\ \mathtt{E}_t\ \mathtt{else}\ \mathtt{E}_e \overset{e}{\Longrightarrow} v_t} \tag{C.8}$$

$$\frac{\mathtt{E}_c \overset{e}{\Longrightarrow} v_c \qquad v_c = 0 \qquad \mathtt{E}_e \overset{e}{\Longrightarrow} v_e}{\mathtt{if}\ \mathtt{E}_c\ \mathtt{then}\ \mathtt{E}_t\ \mathtt{else}\ \mathtt{E}_e \overset{e}{\Longrightarrow} v_e} \tag{C.9}$$

Let Bindings The value of a let expression is the value of the sub-expression $\mathtt{E}_2$ evaluated in an environment where the variable X is bound to the value of $\mathtt{E}_1$ evaluated in the original environment.

$$\frac{\mathtt{E}_1 \overset{e}{\Longrightarrow} v_1 \qquad e' = \mathrm{add}(\mathtt{X}, v_1, e) \qquad \mathtt{E}_2 \overset{e'}{\Longrightarrow} v_2}{\mathtt{let}\ \mathtt{X}\ \mathtt{=}\ \mathtt{E}_1\ \mathtt{in}\ \mathtt{E}_2 \overset{e}{\Longrightarrow} v_2} \tag{C.10}$$

Function Bindings The value of a function let expression is the value of the sub-expression $\mathtt{E}_2$ evaluated in an environment where the variable X is bound to a pair of the function's formal parameter name and the function's body.[1]

$$\frac{e' = \mathrm{add}(\mathtt{X}_f, \langle \mathtt{X}_p, \mathtt{E}_{body} \rangle, e) \qquad \mathtt{E}_2 \overset{e'}{\Longrightarrow} v_2}{\mathtt{let}\ \mathtt{X}_f\ \mathtt{(X}_p\mathtt{)}\ \mathtt{=}\ \mathtt{E}_{body}\ \mathtt{in}\ \mathtt{E}_2 \overset{e}{\Longrightarrow} v_2} \tag{C.11}$$

Function Application First, the actual argument is evaluated in the current environment. Then, we look up the function ($\mathtt{X}_f$) in the environment, giving us the name of its formal parameter ($\mathtt{X}_p$) and the function body ($\mathtt{E}_{body}$). Then we create an environment e' where the formal parameter is bound to the actual argument value, and where the function name is bound as well (to allow recursion). The result is the function body evaluated in the environment e'.

$$\frac{\begin{array}{c}\mathtt{E}_a \overset{e}{\Longrightarrow} v_a \qquad \langle \mathtt{X}_p, \mathtt{E}_{body} \rangle = \mathrm{lookup}(\mathtt{X}_f, e) \\ e' = \mathrm{add}(\mathtt{X}_p, v_a, \mathrm{add}(\mathtt{X}_f, \langle \mathtt{X}_p, \mathtt{E}_{body} \rangle, \mathrm{empty}())) \\ \mathtt{E}_{body} \overset{e'}{\Longrightarrow} v\end{array}}{\mathtt{X}_f\ \mathtt{(E}_a\mathtt{)} \overset{e}{\Longrightarrow} v} \tag{C.12}$$

[1] We may have to tweak this a bit, depending on the semantics of function calls (C.12).

There are several ways to build the environment we'll use to evaluate the function body. The one chosen above isn't particularly good – we can call the function itself recursively, but we can't call any other functions, since they're not found in the environment (we start from an empty environment). These are the (main) alternatives:

- $e' = \text{add}(X_p, v_a, \text{empty}())$ – in which case only the argument is visible. We don't have global variables, but this then would not allow calls to other functions or recursion either. Surely we want to allow other surrounding lets to define functions and values that we can use.

- $e' = \text{add}(X_p, v_a, e)$ – in which case the argument is visible, as are all the variables and functions *at the call site* (those contained in the current environment e). This is *dynamic scoping*, which is used in some languages. It may make some sense in our case, since the call site is always nested within the scope of the definition site.

- $e' = \text{add}(X_p, v_a, e_X)$ – where e_X is the environment at the function definition site. In that case, we'd have to modify the rule for function definition (C.11) to make the definition site environment available. This is *lexical scoping* which is the most common case in modern languages. If local names/variables at the definition site is available, then we have a *closure* (this distinction doesn't make much sense in our case, since all the variables are local).

- We could also create an environment by adding everything from e_X to e; in this case all bound variables at the definition site are lexically scoped, but all the rest (the *free variables*) will be dynamically scoped (accessed through the call site environment).

Things to Think About

- We're already half way to making functions into values. Perhaps we should take this all the way, in which case we wouldn't need a separate `let` for function definition (instead, use something like `let f = x => x*x in ...`; and we could allow any expression in the function call construct (`syntax Expr = Expr "(" Expr ")"`). This is normal in functional programming languages – it would also allow us to emulate multi-argument functions through a mechanism known as *currying* (after *Haskell Curry*, American mathematician).

- We're lacking in comparison operators – we can compare with zero by using the minus operator, but we need at least a less-than operator as well if we want to compare our integers. Also, since we only allow single-argument functions, we can't define these operators within the language.

- Having just a single type (integers) is very boring...

Specification of MyLang
– with Type Checking and Stores

Syntax of MyLang

Quick reference for the grammar:

- "Name:" in front of a production associates a name or constructor with that production.
- "left", "right" or "non-assoc" in front of a production (or group of productions) makes that production left, right or non-associative.
- ">" is used for operator priority.
- Lower-case names are field names for the parse tree – e.g., with "Expr e1 "+" Expr e2", the left operand of a tree "t" can be accessed with "t.e1".

```
lexical INT = [0-9]+;
lexical BOOL = "true" | "false" ;
lexical VAR = [a-zA-Z_][a-zA-Z0-9_]*;
lexical TYPE = [a-zA-Z_][a-zA-Z0-9_]*;
lexical LAYOUT = [\t-\n \r \ ];
layout LAYOUTLIST
  = LAYOUT* !>> [\t-\n \r \ ];

syntax Program = Expr;

syntax Expr
  = Var:          VAR var
  | Int:          INT i
  | Bool:         BOOL b
  | Paren:        "(" Expr e ")"
  | Apply:        VAR fun "(" {Expr ","}* args ")"
  | Not:          "!" Expr
  > left (
      Times:      Expr e1 "*" Expr e2
    | Divide:     Expr e1 "/" Expr e2
  )
  > left (
      Plus:       Expr e1 "+" Expr e2
    | Minus:      Expr e1 "-" Expr e2
  )
  > non-assoc Less: Expr "\<" Expr
  > non-assoc Equal: Expr "==" Expr
  > left And:     Expr "&&" Expr
  > left Or:      Expr "||" Expr
  > Let:          "let" TYPE typ VAR var "=" Expr e1
                      "in" Expr e2 "end"
  | LetFun:       "let" TYPE typ VAR fun "(" {Param ","}* params ")"
                      "=" Expr e1 "in" Expr e2 "end"
```

```
  | If:            "if" Expr cond "then" Expr e1
35                    "else" Expr e2 "end"
  > right Assign: VAR var "=" Expr e
37  > left Seq:     Expr e1 ";" Expr e2
  ;
39
syntax Param
41  = TYPE typ VAR var
  ;
```

Static Semantics of MyLang

This is the formal specification of the MyLang type checker. Used symbols and variables (possibly with subscripts):

- $\mathtt{X} : \mathtt{VAR}$ – variables
- $\mathtt{I} : \mathtt{INT}$ – integers
- $\mathtt{E} : \mathtt{Expr}$ – expressions
- $\mathrm{T}, \tau : \mathrm{Type}$ – types
- $e : \mathrm{Env}$ – environments
- $\longrightarrow$: $\mathrm{Expr}, \mathrm{Env} \to \mathrm{Expr}, \mathrm{Type}$ – type checking

Types are coloured green, environments are coloured red, and code is coloured blue.

Each inference rule has a conclusion of the form $\mathtt{E} \xrightarrow{e} \mathtt{E}' : \tau$, meaning "the expression E type checks to the expression E′ with the type τ in the environment e". The premises, above the horizontal line, must hold (i.e., be inferable from the other rules) in order for the conclusion to hold.

The resulting type checked expression might be slightly different from the original expression – e.g., it might be annotated with type information, or with a unique name for overloaded operations. We won't do much changes to the expressions in this version of the static semantics, except give operator calls the same form as function calls.

See Section 2.3 for a definition of the environment.

Programs Programs don't have types, so we can just type check the expression in an empty environment, and discard the type.

$$\frac{\mathtt{E} \xrightarrow{\mathtt{empty}()} \mathtt{E}' : \tau}{\mathtt{E} \longrightarrow \mathtt{E}'} \tag{C.13}$$

Integers An integer has the type int:

$$\mathtt{I} \xrightarrow{e} \mathtt{I} : \mathtt{int} \tag{C.14}$$

Booleans Booleans have the type bool:

$$\mathtt{true} \xrightarrow{e} \mathtt{true} : \mathtt{bool} \tag{C.15}$$

$$\mathtt{false} \xrightarrow{e} \mathtt{false} : \mathtt{bool} \tag{C.16}$$

Variables The type of a variable is the type of that variable in the environment (or undefined if the variable is unbound in the environment):

$$\frac{\tau = \text{lookup}(X, e)}{\texttt{X} \xrightarrow{e} \texttt{X} : \tau} \tag{C.17}$$

Parenthesis The type of a parenthesis expression is the type of the expression between the parenthesis, checked in the current environment. We can drop the parentheses in the result.

$$\frac{\texttt{E} \xrightarrow{e} \texttt{E}' : \tau}{\texttt{(E)} \xrightarrow{e} \texttt{E}' : \tau} \tag{C.18}$$

Addition, Subtraction, Multiplication, Division Arithmetic operators are only valid on integers. The result of type checking an operator call is a call to a corresponding function, and the type int:

$$\frac{\texttt{E}_1 \xrightarrow{e} \texttt{E}_1' : \texttt{int} \quad \texttt{E}_2 \xrightarrow{e} \texttt{E}_2' : \texttt{int}}{\texttt{E}_1 \texttt{ + } \texttt{E}_2 \xrightarrow{e} \texttt{plus(E}_1\texttt{', E}_2\texttt{')} : \texttt{int}} \tag{C.19}$$

$$\frac{\texttt{E}_1 \xrightarrow{e} \texttt{E}_1' : \texttt{int} \quad \texttt{E}_2 \xrightarrow{e} \texttt{E}_2' : \texttt{int}}{\texttt{E}_1 \texttt{ - } \texttt{E}_2 \xrightarrow{e} \texttt{minus(E}_1\texttt{', E}_2\texttt{')} : \texttt{int}} \tag{C.20}$$

$$\frac{\texttt{E}_1 \xrightarrow{e} \texttt{E}_1' : \texttt{int} \quad \texttt{E}_2 \xrightarrow{e} \texttt{E}_2' : \texttt{int}}{\texttt{E}_1 \texttt{ * } \texttt{E}_2 \xrightarrow{e} \texttt{times(E}_1\texttt{', E}_2\texttt{')} : \texttt{int}} \tag{C.21}$$

$$\frac{\texttt{E}_1 \xrightarrow{e} \texttt{E}_1' : \texttt{int} \quad \texttt{E}_2 \xrightarrow{e} \texttt{E}_2' : \texttt{int}}{\texttt{E}_1 \texttt{ / } \texttt{E}_2 \xrightarrow{e} \texttt{divide(E}_1\texttt{', E}_2\texttt{')} : \texttt{int}} \tag{C.22}$$

Note: *If we were to support overloaded operators – for example, addition on strings – we could make separate inference rules for this, yielding a different form for the type checked expression:*

$$\frac{\texttt{E}_1 \xrightarrow{e} \texttt{E}_1' : \texttt{str} \quad \texttt{E}_2 \xrightarrow{e} \texttt{E}_2' : \texttt{str}}{\texttt{E}_1 \texttt{ + } \texttt{E}_2 \xrightarrow{e} \texttt{str\_append(E}_1\texttt{', E}_2\texttt{')} : \texttt{str}} \tag{C.23}$$

Comparison Operators Ordered comparison is only valid on integers. Equality can also work on bools. The type is always bool.

$$\frac{\texttt{E}_1 \xrightarrow{e} \texttt{E}_1' : \texttt{int} \quad \texttt{E}_2 \xrightarrow{e} \texttt{E}_2' : \texttt{int}}{\texttt{E}_1 \texttt{ < } \texttt{E}_2 \xrightarrow{e} \texttt{lessThan(E}_1\texttt{', E}_2\texttt{')} : \texttt{bool}} \tag{C.24}$$

$$\frac{\texttt{E}_1 \xrightarrow{e} \texttt{E}_1' : \texttt{int} \quad \texttt{E}_2 \xrightarrow{e} \texttt{E}_2' : \texttt{int}}{\texttt{E}_1 \texttt{ == } \texttt{E}_2 \xrightarrow{e} \texttt{int\_equal(E}_1\texttt{', E}_2\texttt{')} : \texttt{bool}} \tag{C.25}$$

$$\frac{\texttt{E}_1 \xrightarrow{e} \texttt{E}_1' : \texttt{bool} \quad \texttt{E}_2 \xrightarrow{e} \texttt{E}_2' : \texttt{bool}}{\texttt{E}_1 \texttt{ == } \texttt{E}_2 \xrightarrow{e} \texttt{bool\_equal(E}_1\texttt{', E}_2\texttt{')} : \texttt{bool}} \tag{C.26}$$

Boolean Operators These are only valid on bool operands, and always return bool.

$$\frac{\texttt{E}_1 \xrightarrow{e} \texttt{E}_1' : \texttt{bool} \qquad \texttt{E}_2 \xrightarrow{e} \texttt{E}_2' : \texttt{bool}}{\texttt{E}_1 \texttt{ \&\& } \texttt{E}_2 \xrightarrow{e} \texttt{and(E}_1\texttt{', E}_2\texttt{')} : \texttt{bool}} \tag{C.27}$$

$$\frac{\texttt{E}_1 \xrightarrow{e} \texttt{E}_1' : \texttt{bool} \qquad \texttt{E}_2 \xrightarrow{e} \texttt{E}_2' : \texttt{bool}}{\texttt{E}_1 \texttt{ || } \texttt{E}_2 \xrightarrow{e} \texttt{or(E}_1\texttt{', E}_2\texttt{')} : \texttt{bool}} \tag{C.28}$$

$$\frac{\texttt{E} \xrightarrow{e} \texttt{E}' : \texttt{bool}}{\texttt{!E}_1 \xrightarrow{e} \texttt{not(E')} : \texttt{bool}} \tag{C.29}$$

Let Bindings The type of a `let` expression is the type of the sub-expression $\texttt{E}_2$ type checked in an environment where the variable X is bound to the type T. The type of the bound expression $\texttt{E}_1$ must match the declared type T of the variable X.

$$\frac{\texttt{E}_1 \xrightarrow{e} \texttt{E}_1' : \texttt{T} \qquad e' = \text{add}(\texttt{X}, \texttt{T}, e) \qquad \texttt{E}_2 \xrightarrow{e'} \texttt{E}_2' : \tau}{\texttt{let T X = E}_1 \texttt{ in E}_2 \texttt{ end} \xrightarrow{e} \texttt{let T X = E}_1\texttt{' in E}_2\texttt{' end} : \tau} \tag{C.30}$$

Function Bindings The type of a function `let` expression is the type of the sub-expression $\texttt{E}_2$ type checked in an environment where the variable X is bound to a tuple of the function's formal parameter types and the function's return type.

The function's body $\texttt{E}_{\texttt{body}}$ must be type checked as well. Since we allow recursive functions, the function itself must be defined in the environment we use for this. Also, we need to add all the formal parameters to the environment we use when we type check the function body. In the rule below, e'_k (built by adding the parameters one by one) has both the function definition and all the parameters. The type of the function body must match the declared return type of the function.

The body of the `let` is checked in the environment e', which has the function definition, but not the parameters.

$$\frac{\begin{gathered} e' = \text{add}(\texttt{X}_\texttt{f}, \langle [\texttt{T}_0, \ldots, \texttt{T}_k], \texttt{T} \rangle, e) \\ e'_0 = \text{add}(\texttt{X}_0, \texttt{T}_0, e') \ \ldots \ e'_k = \text{add}(\texttt{X}_k, \texttt{T}_k, e'_{k-1}) \\ \texttt{E}_{\texttt{body}} \xrightarrow{e'_k} \texttt{E}_{\texttt{body}}' : \texttt{T} \qquad \texttt{E}_2 \xrightarrow{e'} \texttt{E}_2' : \tau \end{gathered}}{\begin{gathered} \texttt{let T X}_\texttt{f} \texttt{ (T}_0 \texttt{ X}_0\texttt{, ..., T}_k \texttt{ X}_k\texttt{) = E}_{\texttt{body}} \texttt{ in E}_2 \texttt{ end} \\ \xrightarrow{e} \texttt{let T X}_\texttt{f} \texttt{ (T}_0 \texttt{ X}_0\texttt{, ..., T}_k \texttt{ X}_k\texttt{) = E}_{\texttt{body}}\texttt{' in E}_2\texttt{' end} : \tau \end{gathered}} \tag{C.31}$$

Function Application In the case of function application, we must first type check all the arguments. Then we look up the function ($\texttt{X}_\texttt{f}$) in the current environment, getting its formal parameter list and return type. The formal parameter list must match the actual argument types. The result is a type checked function call, where the type is the return type of the function.

$$\frac{\begin{gathered} \texttt{E}_0 \xrightarrow{e} \texttt{E}_0' : \tau_0 \quad \ldots \quad \texttt{E}_k \xrightarrow{e} \texttt{E}_k' : \tau_k \\ \langle [\tau_0, \ldots, \tau_k], \tau \rangle = \text{lookup}(\texttt{X}_\texttt{f}, e) \end{gathered}}{\texttt{X}_\texttt{f} \texttt{ (E}_0\texttt{, ..., E}_k\texttt{)} \xrightarrow{e} \texttt{X}_\texttt{f} \texttt{ (E}_0\texttt{', ..., E}_k\texttt{')} : \tau} \tag{C.32}$$

Note: *In a language with support for overloading, the matching process becomes more complicated as we may have to chose between multiple overloaded candidates. If the language supports type conversions, type promotions or generics, the matching becomes even more complicated.*

Conditionals The type of a conditional expression is the type of the branches. The condition itself should be of type bool, and the type of the `then` branch should be the same as the type of the `else` branch.

$$\frac{\mathtt{E_c} \xrightarrow{e} \mathtt{E_c}' : \mathtt{bool} \qquad \mathtt{E_t} \xrightarrow{e} \mathtt{E_t}' : \tau_t \qquad \mathtt{E_e} \xrightarrow{e} \mathtt{E_e}' : \tau_e \qquad \tau_t = \tau_e}{\mathtt{if\ E_c\ then\ E_t\ else\ E_e\ end} \xrightarrow{e} \mathtt{if\ E_c}'\ \mathtt{then\ E_t}'\ \mathtt{else\ E_e}'\ \mathtt{end} : \tau_t} \tag{C.33}$$

Assignment In an assignment, the type of the expression must match the type of the variable – and this is also the return type of the expression.

$$\frac{\tau = \mathrm{lookup}(X, e) \qquad \mathtt{E} \xrightarrow{e} \mathtt{E}' : \tau}{\mathtt{X = E} \xrightarrow{e} \mathtt{X = E}' : \tau} \tag{C.34}$$

Note: *In a language with type conversions or promotions, we would have to loosen the matching criterion. Instead of the expression type being equal to the variable type, it would be sufficient that it is* assignable *to the variable type. For example, an* int *may be assignable to a* float*. For simplicity in later processing, we could insert an explicit conversion during type checking. Typically the matching criterion for function arguments is the same as for assignment.*

Sequencing Sequencing enforces an evaluation order, so that one expression is evaluated before another. We must type check both expressions, the return type is the type of the second expression.

$$\frac{\mathtt{E_1} \xrightarrow{e} \mathtt{E_1}' : \tau_1 \qquad \mathtt{E_2} \xrightarrow{e} \mathtt{E_2}' : \tau_2}{\mathtt{E_1;E_2} \xrightarrow{e} \mathtt{E_1}';\mathtt{E_2}' : \tau_2} \tag{C.35}$$

Dynamic Semantics of MyLang

Used symbols and variables (possibly with subscripts):

- `X`, X : VAR – variables
- `I`, I : INT – integers
- `E`, E : Expr – expressions
- v : Val – values
- e : Env – environments
- s : Store – stores
- l : Location – a storage location (or address)
- $\Longrightarrow$: Expr, Store, Env $\rightarrow$ Val, Store – evaluation

Stores and locations are coloured teal, and values are coloured magenta.

In the dynamic semantics, each inference rule has a conclusion of the form $\langle \mathtt{E}, s \rangle \overset{e}{\Longrightarrow} \langle v, s' \rangle$, meaning "the expression `E` evaluates to the value v and the store s' in the environment e and the store s". The premises, above the horizontal line, must hold (i.e., be inferable from the other rules) in order for the conclusion to hold.

See Section 2.3 for a definition of the environment.

Order of Evaluation

With store semantics, we might end up with the evaluation order affecting the semantics. For example, in the program

```
x = 1; y = (x = 2) + x
```

the value of `y` will be 3 or 4, depending on the evaluation order of the arguments to `+`.

For simplicity, we'll assume that argument lists are evaluated left-to-right, so the value of `y` above will be 4.

Note: *Specifying the evaluation order may restrict the compiler's opportunity to make some optimisations – particularly in terms of evaluating things in parallel. The benefit is that the code will always mean the same across multiple compilers and platforms – something which isn't necessarily true for C code, for example (C has undefined evaluation order). Bugs related to undefined evaluation order can be hard to find. A way around the whole problem is to disallow updates in expressions.*

Specification

Programs The value of a program, is the value of its expression, evaluated in an empty environment and store:

$$\frac{\langle \mathtt{E}, \mathtt{blank}() \rangle \overset{\mathtt{empty}()}{\Longrightarrow} \langle v, s \rangle}{\mathtt{E} \Longrightarrow v} \tag{C.36}$$

Note: *Another possibility is to return both the value and the store – but as long as we start with an empty environment, all variables will have been deleted from the store before evaluation is complete (otherwise, we have a memory leak).*

Built-in operations can be provided by making an initial environment, sometimes called a prelude*. This is also an easy way to provide a standard library.*

Command-line arguments can also be provided through the initial environment and store. We could, for example, provide a count of the arguments in a variable `argc` *and then a function* `arg(i)` *to access the argument values (we don't have arrays or lists yet, but we can emulate them using functions).*

Integers The value of a literal integer is the integer's value:

$$\langle \mathtt{I}, s \rangle \overset{e}{\Longrightarrow} \langle \mathrm{I}, s \rangle \tag{C.37}$$

Booleans The value of a literal Boolean is the booleans value:

$$\langle \mathtt{true}, s \rangle \overset{e}{\Longrightarrow} \langle \mathrm{true}, s \rangle \tag{C.38}$$

$$\langle \mathtt{false}, s \rangle \overset{e}{\Longrightarrow} \langle \mathrm{false}, s \rangle \tag{C.39}$$

Variables The value of a variable is the value of the storage location of that variable in the environment (or undefined if the variable is unbound in the environment):

$$\frac{l = \text{lookup}(X, e)}{\langle \texttt{X}, s\rangle \overset{e}{\Longrightarrow} \langle s[l], s\rangle} \tag{C.40}$$

The store is unchanged.

Parenthesis The value of a parenthesis expression is the value of the expression between the parenthesis, evaluated in the current environment.

$$\frac{\langle \texttt{E}, s\rangle \overset{e}{\Longrightarrow} \langle v, s'\rangle}{\langle \texttt{(E)}, s\rangle \overset{e}{\Longrightarrow} \langle v, s'\rangle} \tag{C.41}$$

Arithmetic and Comparison Operators Operators may be handled either by built-in functions (in which case, see (C.50)), or they may be defined directly in the specification. For example, for the plus operator:

$$\frac{\langle \texttt{E}_1, s\rangle \overset{e}{\Longrightarrow} \langle v_1, s'\rangle \qquad \langle \texttt{E}_2, s'\rangle \overset{e}{\Longrightarrow} \langle v_2, s''\rangle}{\langle \texttt{plus(E}_1\texttt{, E}_2\texttt{)}, s\rangle \overset{e}{\Longrightarrow} \langle v_1 + v_2, s''\rangle} \tag{C.42}$$

Boolean Operators These are only valid on bool operands, and always return a bool.

$$\frac{\langle \texttt{E}_1, s\rangle \overset{e}{\Longrightarrow} \langle v_1, s'\rangle \qquad \langle \texttt{E}_2, s'\rangle \overset{e}{\Longrightarrow} \langle v_2, s''\rangle}{\langle \texttt{and(E}_1\texttt{, E}_2\texttt{)}, s\rangle \overset{e}{\Longrightarrow} \langle v_1 \wedge v_2, s''\rangle} \tag{C.43}$$

$$\frac{\langle \texttt{E}_1, s\rangle \overset{e}{\Longrightarrow} \langle v_1, s'\rangle \qquad \langle \texttt{E}_2, s'\rangle \overset{e}{\Longrightarrow} \langle v_2, s''\rangle}{\langle \texttt{or(E}_1\texttt{, E}_2\texttt{)}, s\rangle \overset{e}{\Longrightarrow} \langle v_1 \vee v_2, s''\rangle} \tag{C.44}$$

$$\frac{\langle \texttt{E}, s\rangle \overset{e}{\Longrightarrow} \langle v, s'\rangle}{\langle \texttt{not(E)}, s\rangle \overset{e}{\Longrightarrow} \langle \neg v, s'\rangle} \tag{C.45}$$

Note: *There's one sticky point with the Boolean operators – many languages (including C, Java and ML) have* short-circuiting *Boolean operators. This means that evaluation stops as soon as the result is known. For example,* `false && E` *is always false, so there is no need to evaluate* `E`*. This can be just an optimisation – or it can be required by the language definition. In the latter case, the* and *and* or *operators are more like a form of* if *expressions, and we can handle them this way in the semantics:*

$$\langle \texttt{and(}E_1\texttt{, }E_2\texttt{)}, s\rangle \overset{e}{\Longrightarrow} \langle \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else false end}, s\rangle \tag{C.46}$$

$$\langle \texttt{or(}E_1\texttt{, }E_2\texttt{)}, s\rangle \overset{e}{\Longrightarrow} \langle \texttt{if } E_1 \texttt{ then true else } E_2 \texttt{ end}, s\rangle \tag{C.47}$$

Let Bindings The `let` operation becomes a whole lot trickier when we have to deal with storage. First, we need get a new storage location `l` for the variable X. The location / size may depend on the variable type, so add this as a parameter to the `new` store operation.

We get the value of the new variable by evaluating E_1 (this may of course also result in an updated store). The final value of the `let` expression is then the value of the sub-expression E_2 evaluated in an environment where the variable X is bound to the new storage location `l` and a store where the location `l` is set to the value of E_1.

The resulting store is the output store of the evaluation, with the new variable location deleted – variables disappear when they go out of scope.

$$\frac{\langle E_1, s\rangle \overset{e}{\Longrightarrow} \langle v_1, s'\rangle \qquad \langle l, s''\rangle = \text{new}(T, v_1, s') \qquad e' = \text{add}(X, l, e) \qquad \langle E_2, s''\rangle \overset{e'}{\Longrightarrow} \langle v_2, s'''\rangle}{\langle \texttt{let T X = } E_1 \texttt{ in } E_2 \texttt{ end}, s\rangle \overset{e}{\Longrightarrow} \langle v_2, \text{del}(l, s''')\rangle} \tag{C.48}$$

Function Bindings The value of a function `let` expression is the value of the sub-expression E_2 evaluated in an environment where the variable X is bound to a triple of the function's formal parameter list, the function's body, and the definition environment. The definition environment is needed to support lexical scoping.

$$\frac{e' = \text{add}(X_f, \langle [T_0\ X_0, \ldots, T_k\ X_k], E_{body}, e\rangle, e) \qquad \langle E_2, s\rangle \overset{e'}{\Longrightarrow} \langle v_2, s'\rangle}{\langle \texttt{let T } X_f \texttt{ (}T_0\ X_0\texttt{, ..., }T_k\ X_k\texttt{) = } E_{body} \texttt{ in } E_2 \texttt{ end}, s\rangle \overset{e}{\Longrightarrow} \langle v_2, s'\rangle} \tag{C.49}$$

Note: *If the definition environment is not made available at the call site, we have to use dynamic scoping – variables in the function body refer to variables in the calling environment. Alternatively, we could pre-evaluate the function body and replace variable references with storage locations, in which case we would not longer need to pass along the environment. In our solution, the environment contains the mapping of variables to locations, so we'll need that at the call site.*

Note: *Above, we've made functions into a special kind of entity which is not assigned a location in the store. This makes the inference rule slightly simpler. But we could allow functions to be stored just as variables are – in that case, functions would be* first-class values *in the language; they can be assigned, returned and passed as parameters. Depending on your implementation of the first version of MyLang, functions may have been first-class values there.*

In the semantics above, we need the environment to map names to storage locations or *functions. It may be somewhat cleaner to always map to a storage location, so that functions are stored just the same way as values. Python, for instance, works this way – and you can actually update an object's methods as your program is running. The negative effect of this is that some optimisations are harder – for example, you'd only be able to do inlining of functions that are known constants at compile time (C and C++ struggle with inlining function pointers).*

Function Application Function application is somewhat complicated, now that we have to deal with both multiple arguments and variable storage. These are the steps involved:

- Evaluate each argument in turn, yielding a value and an updated store. The next argument is evaluated in the store of the previous evaluation – this enforces left-to-right evaluation.
- Allocate new storage locations for all the arguments. This corresponds to building a stack frame in a compiler, and is consistent with call-by-value semantics – changing the value of an argument only has an effect inside the function itself.
- Look up the function name, and get the parameter list (with types and parameter names), the body, and the definition environment.
- Start with the definition environment, and add each parameter to the environment, bound to the storage location of the corresponding argument.
- Add the function definition to the environment (for recursive calls).
- Evaluate the function body in the environment, and the store containing all the evaluated arguments.
- Evaluating the body yields a new store. Because the function body may have assigned to variables in outer scopes, we need to keep those changes to the store – but we also need to get rid of all the arguments we added to the store. In a compiler implementation, we can get this effect simply by adjusting / unwinding the stack pointer. Here, we explicitly delete each of the argument locations.

$$\frac{\begin{array}{c}\langle \mathtt{E}_0, s\rangle \overset{e}{\Longrightarrow} \langle v_0, s_0\rangle \ \ldots\ \langle \mathtt{E}_k, s_{k-1}\rangle \overset{e}{\Longrightarrow} \langle v_k, s_k\rangle \\ \langle l_0, s'_0\rangle = \mathrm{new}(T_0, v_0, s_0)\ \ldots\ \langle l_k, s'_k\rangle = \mathrm{new}(T_k, v_k, s_k) \\ \langle [T_0 X_0, \ldots, T_k X_k], \mathtt{E}_{body}, e_f\rangle = \mathrm{lookup}(X_f, e) \\ e'_0 = \mathrm{add}(X_0, l_0, e_f)\ \ldots\ e'_k = \mathrm{add}(X_k, l_k, e'_{k-1}) \\ e' = \mathrm{add}(X_f, \langle [T_0 X_0, \ldots, T_0 X_k], \mathtt{E}_{body}, e_f\rangle, e'_k) \\ \langle \mathtt{E}_{body}, s_k\rangle \overset{e'}{\Longrightarrow} \langle v, s'\rangle\end{array}}{\langle \mathtt{X_f\ (E_0,\ \ldots,\ E_k)}, s\rangle \overset{e}{\Longrightarrow} \langle v, \mathrm{del}([l_0, \ldots, l_k], s')\rangle} \tag{C.50}$$

Note: *Bonus points to the one who finds the bug(s) in the above definition.*

Conditionals The value of a conditional expression depends on the value of the condition $\mathtt{E}_c$ in the current environment. If the condition is true then the value is the value of the then-branch in the current environment, otherwise the value of the else-branch.

$$\frac{\langle \mathtt{E}_c, s\rangle \overset{e}{\Longrightarrow} \langle \mathrm{true}, s'\rangle \qquad \langle \mathtt{E}_t, s'\rangle \overset{e}{\Longrightarrow} \langle v_t, s''\rangle}{\langle \mathtt{if\ E_c\ then\ E_t\ else\ E_e\ end}, s\rangle \overset{e}{\Longrightarrow} \langle v_t, s''\rangle} \tag{C.51}$$

$$\frac{\langle \mathtt{E}_c, s\rangle \overset{e}{\Longrightarrow} \langle \mathrm{false}, s'\rangle \qquad \langle \mathtt{E}_e, s'\rangle \overset{e}{\Longrightarrow} \langle v_e, s''\rangle}{\langle \mathtt{if\ E_c\ then\ E_t\ else\ E_e\ end}, s\rangle \overset{e}{\Longrightarrow} \langle v_e, s''\rangle} \tag{C.52}$$

Note: *In an implementation, it is important not to evaluate the other branch of the if-statement, since that could have undesirable effects on the store. Without a store, this effect isn't noticeable – except in cases where the computation may not terminate (for example, in a recursive function). Some functional languages evaluate on demand (lazy evaluation), which is another way of avoiding the non-termination problem.*

Assignment Assignment is (fortunately) straight-forward. We evaluate the expression, getting a value v and a new store s'. We then lookup the variable to get its storage location l, and the result is the value v and the store s' updated with $l = v$.

$$\frac{\langle \mathtt{E}, s\rangle \overset{e}{\Longrightarrow} \langle v, s'\rangle \qquad l = \mathrm{lookup}(\mathrm{X}, e)}{\langle \mathtt{X\ =\ E}, s\rangle \overset{e}{\Longrightarrow} \langle v, s'[l = v]\rangle} \tag{C.53}$$

Sequencing We evaluate both expressions, the output store of the left expression is the input of the right expression. The value is the value of the right expression.

$$\frac{\langle \mathtt{E}_1, s\rangle \overset{e}{\Longrightarrow} \langle v_1, s'\rangle \qquad \langle \mathtt{E}_2, s'\rangle \overset{e}{\Longrightarrow} \langle v_2, s''\rangle}{\langle \mathtt{E}_1\,\mathtt{;E}_2, s\rangle \overset{e}{\Longrightarrow} \langle v_2, s''\rangle} \tag{C.54}$$

Bibliography

[1] Hans Hüttel. *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge University Press, Cambridge, UK, June 2010. ISBN 978-0521147095.