Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Compiler Construction

WWW: `http://www.cs.uu.nl/wiki/Cco`

Edition 2011/2012

Agenda

Simple arithmetic expressions

- Syntax
- Semantics
- Implementation

3. Simple arithmetic expressions

Introduction

§3

To develop tools and techniques for reasoning about the syntax and semantics of high-level programming languages, we will now consider a small “programming language” for simple arithmetic expressions.

3.1 Syntax

Syntactic forms

Our language will consist of just a few syntactic forms: natural numbers, addition, and multipliation.

For example:

12

$7 + 21$

$18 * 2$

$3 * (17 + 5)$

Styles of syntax

- Inductive definitions.
- Inference rules.
- Generation procedures.
- BNF-notation.

Inductive definition

One way of formally defining the syntax of a language is by means of an inductive definition.

Let $\mathbb{N}$ be the set of natural numbers $\{0, 1, 2, \cdots\}$.

Definition: The set of *terms* is the smallest set $\mathbf{Tm}$ such that:

1. If $n \in \mathbb{N}$, then $n \in \mathbf{Tm}$.
2. If $t_1 \in \mathbf{Tm}$ and $t_2 \in \mathbf{Tm}$, then $t_1 + t_2 \in \mathbf{Tm}$.
3. If $t_1 \in \mathbf{Tm}$ and $t_2 \in \mathbf{Tm}$, then $t_1 * t_2 \in \mathbf{Tm}$.

☞ Note the use of the word “smallest”: $\mathbf{Tm}$ has no elements other than the ones required by the three clauses of the definition.

Concrete vs. abstract syntax

The given inductive definition does not say anything about the use of parentheses and precedence rules to disambiguate phrases, such as $3 * (17 + 5)$.

Formally, we are defining *trees* and the structure of a term is immediate from the shape of the tree. The rules that prescribe which trees form terms define the abstract syntax of our language.

However, if we write down terms in programs etc., we use a string representation, i.e., a concrete syntax, in which we do use parentheses and precedence rules.

In an implementation, the translation from concrete into abstract syntax is carried out by a parser.

Inference rules

An alternative approach to defining abstract syntax, is in natural deduction style, i.e., by means of so-called inference rules:

$$\frac{\textit{premise}_1 \quad \cdots \quad \textit{premise}_n}{\textit{conclusion}}\ [\textit{name}]$$

Each inference rule consists of zero or more premises, a conclusion, and, optionally, a name: if we have established all premises, then we may derive the conclusion.

An inference rule without premises, is called an axiom.

Inference rules (cont'd)

To define our abstract syntax, we employ the following rules:

$$\frac{n \in \mathbb{N}}{n \in \mathbf{Tm}}\ [\textit{num}]$$

$$\frac{t_1 \in \mathbf{Tm} \quad t_2 \in \mathbf{Tm}}{t_1 + t_2 \in \mathbf{Tm}}\ [\textit{add}]$$

$$\frac{t_1 \in \mathbf{Tm} \quad t_2 \in \mathbf{Tm}}{t_1 * t_2 \in \mathbf{Tm}}\ [\textit{mul}]$$

Generation procedure

Yet another means to defining abstract syntax is by giving an explicit generation procedure:

$$\begin{aligned} \mathbf{Tm}_0 &= \emptyset \\ \mathbf{Tm}_{i+1} &= \mathbb{N} \cup \{\, t_1 + t_2, t_1 * t_2 \mid t_1, t_2 \in \mathbf{Tm}_i \,\} \end{aligned}$$

And then let

$$\mathbf{Tm} = \bigcup_i \mathbf{Tm}_i$$

BNF-notation

During the course, we will mostly use Backus-Naur Form (BNF) to define abstract syntax.

First we introduce metavariables:

n	$\in$	$\mathbf{Num} = \mathbb{N}$	numerals
t	$\in$	$\mathbf{Tm}$	terms

The metavariable n ranges over the set $\mathbf{Num}$ of numerals, while the metavariable t (or variations such as t_1 and t_2) ranges over the set $\mathbf{Tm}$ of terms.

The abstract syntax of terms is then given by:

$$t \quad ::= \quad n \mid t_1 + t_2 \mid t_1 * t_2$$

3.2 Semantics

The meaning of terms

Now we have established the precise syntax of terms, we need to rigourously define their meaning, i.e., we need to formally define the semantics of our language.

For example:

- The meaning of the program 12 is the natural number 12.
- The meaning of the program $7 + 21$ is the natural number 28.
- The program $18 * 2$ has the same meaning as the program $4 * 9$.
- The program $3 * (17 + 5)$ has the same meaning as the program $3 * 22$.

Styles of semantics

- Axiomatic semantics.
- Denotational semantics.
- Operational semantics.
  - Small-step (structural) operational semantics.
  - Big-step (natural) operational semantics.

Object language vs. metalanguage

Before we can formalise the semantics of our language, we need some notation.

For $n_1, n_2 \in \mathbb{N}$, we write $n_1 \underline{+} n_2$ to denote the "normal" addition of n_1 and n_2.

Hence, we distinguish between addition in the object language, i.e., $t_1 + t_2$, for which we have yet to provide a meaning, and addition in the metalanguage, of which the meaning is assumed to be well-understood.

Similarly, we write $n_1 \underline{*} n_2$ for the metalanguage multiplication of two natural numbers.

☞ $\underline{+}$ and $\underline{*}$ only work on natural numbers, while $+$ and $*$ can be used on arbitrary terms to form new syntactic terms.

Axiomatic semantics

An axiomatic semantics for a language is given by supplying a set of equalities between terms:

$$
\begin{aligned}
n_1 + n_2 &=_{\text{ax}} n_1 \underline{+} n_2 \\
n_2 * n_2 &=_{\text{ax}} n_1 \underline{*} n_2
\end{aligned}
$$

Example:

$$
\begin{array}{ll}
3 * (17 + 5) & \\
\quad = & \{17 + 5 =_{\text{ax}} 22\} \\
3 * 22 & \\
\quad = & \{3 * 22 =_{\text{ax}} 66\ \} \\
66 &
\end{array}
$$

Axiomatic semantics (cont'd)

An axiomatic semantics is undirected, rendering it less suitable for a direct mapping to an implementation:

$$
\begin{array}{ll}
3 * (17 + 5) & \\
\quad = & \{1 + 2 =_{\text{ax}} 3\} \\
(1 + 2) * (17 + 5) & \\
\quad = & \{11 + 6 =_{\text{ax}} 17\} \\
(1 + 2) * (11 + 6 + 5) &
\end{array}
$$

Intuitively, a semantics should provide a means to "simplify" a term.

Denotational semantics

A denotational semantics provides a mapping from the set of terms to another set:

$$
\begin{aligned}
&[\![\cdot]\!] : \mathbf{Tm} \to \mathbb{N} \\
&[\![n]\!] = n \\
&[\![t_1 + t_2]\!] = [\![t_1]\!] \underline{+} [\![t_2]\!] \\
&[\![t_1 * t_2]\!] = [\![t_1]\!] \underline{*} [\![t_2]\!]
\end{aligned}
$$

Denotational semantics (cont'd)

For example:

$$
\begin{aligned}
[\![3 * (17 + 5)]\!] &= [\![3]\!] \mathbin{\underline{*}} [\![17 + 5]\!] \\
&= [\![3]\!] \mathbin{\underline{*}} ([\![17]\!] \mathbin{\underline{+}} [\![5]\!]) \\
&= [\![3]\!] \mathbin{\underline{*}} ([\![17]\!] \mathbin{\underline{+}} 5) \\
&= 3 \mathbin{\underline{*}} ([\![17]\!] \mathbin{\underline{+}} 5) \\
&= 3 \mathbin{\underline{*}} (17 \mathbin{\underline{+}} 5) \\
&= 3 \mathbin{\underline{*}} 22 \\
&= 66
\end{aligned}
$$

☞ A denotational semantics does not necessarily prescribe an “order of evaluation.

Operational semantics

An operational semantics describes an abstract “machine” that operates on terms, simplifying them to values.

For our language, values are just numerals:

$v \quad \in \quad \mathbf{Val}$ values

$v \quad ::= \quad n$

Often, but not always, we have that $\mathbf{Val} \subset \mathbf{Tm}$.

Small-step operational semantics

The first type of operational semantics that we shall consider is a so-called small-step or structured operational semantics.

A small-step operational semantics defines a machine that takes a term and performs a single step of computation on it, yielding a simpler term. Every step is referred to as a reduction.

Computation halts as soon as the term is transformed into a value.

We define the semanics as a set of inference rules with conclusions of the form

$t \longrightarrow t'$

Small-step operational semantics (cont'd)

$$\frac{}{n_1 + n_2 \longrightarrow n_1 \mathbin{\underline{+}} n_2}\ [\textit{r-add}]$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 + t_2 \longrightarrow t_1' + t_2}\ [\textit{r-add}_1]$$

$$\frac{t_2 \longrightarrow t_2'}{v_1 + t_2 \longrightarrow v_1 + t_2'}\ [\textit{r-add}_2]$$

Rules [*r-add*$_1$] and [*r-add*$_2$] are examples of so-called congruence rules: as opposed to the axiom [*r-add*] they do not denote a “real” computation step, but rather guide the reduction of a compound term, effectively dictating an order of evaluation.

Small-step operational semantics (cont'd)

For multiplication, we have a similar triple of rules:

$$\frac{}{n_1 * n_2 \longrightarrow n_1 \underline{*} n_2}\ [r\text{-}mul]$$

$$\frac{t_1 \longrightarrow t_1'}{t_1 * t_2 \longrightarrow t_1' * t_2}\ [r\text{-}mul_1]$$

$$\frac{t_2 \longrightarrow t_2'}{v_1 * t_2 \longrightarrow v_1 * t_2'}\ [r\text{-}mul_2]$$

Small-step operational semantics (cont'd)

Example:

$$3 * (17 + 5) \longrightarrow 3 * 22 \longrightarrow 66$$

As a pair of deriviation trees, one for each step:

$$\frac{\overline{17 + 5 \longrightarrow 22}}{3 * (17 + 5) \longrightarrow 3 * 22}$$

$$\overline{3 * 22 \longrightarrow 66}$$

Small-step operational semantics (cont'd)

A sequence of reduction steps.

$$t_1 \longrightarrow t_2 \longrightarrow \cdots \longrightarrow t_n$$

such that $t_i \longrightarrow t_{i+1}$ for $1 \leqslant i \leqslant n$ is sometimes also written as $t_1 \longrightarrow^* t_n$.

If t_n is a value, $t_n = v$, we say that t_1 evaluates to v in $(n-1)$ steps.

For instance, $3 * (17 + 5)$ evaluates to 66 in two steps:

$$3 * (17 + 5) \longrightarrow^* 66$$

Big-step operational semantics

An alternative style of operational semantics, big-step or natural operational semantics, formalises the concept of evaluation directly.

A big-step operational semantics for our language of arithmetic expressions is defined as an natural deduction system with rules for deriving judgements of the form

$$t \Downarrow v$$

where t is term and v the value that it evaluates to.

Big-step operational semantics (cont'd)

$$\frac{}{n \Downarrow n}\ [\textit{e-num}]$$

$$\frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{t_1 + t_2 \Downarrow n_1 \underline{+} n_2}\ [\textit{e-add}]$$

$$\frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{t_1 * t_2 \Downarrow n_1 \underline{*} n_2}\ [\textit{e-mul}]$$

- ☞ The big-step operational semantics is quite similar to the denotational semantics for our language.
- ☞ The big-step operational semantics is not as precise as the small-step operational semantics with respect to the order of evaluation.

Big-step operational semantics (cont'd)

For example:

$$3 * (17 + 5) \Downarrow 66$$

Derivation tree:

$$\frac{\frac{}{3 \Downarrow 3} \quad \frac{\frac{}{17 \Downarrow 17} \quad \frac{}{5 \Downarrow 5}}{17 + 5 \Downarrow 22}}{3 * (17 + 5) \Downarrow 66}$$

Language = syntax + semantics

Syntax:

n	$\in$	$\mathbf{Num}$	numerals
t	$\in$	$\mathbf{Tm}$	terms
v	$\in$	$\mathbf{Val}$	values

$$t ::= n \mid t_1 + t_2 \mid t_1 * t_2$$
$$v ::= n$$

Semantics:

$t \longrightarrow t'$ reduction

or

$t \Downarrow v$ evaluation

3.3 Implementation

Syntax: numerals


```
type Num_ = Integer
```

- *Integer* is the Haskell type of arbitrary sized integers. In contrast to the the set **Num** it also contains negative numbers, so some carefulness is in order.
- The name *Num_* (with the underscore postfix) is used to avoid a name conflict with the type class *Num* of numeric types.

Syntax: terms and values


```
data Tm = Num Num_ | Add Tm Tm | Mul Tm Tm
data Val = VNum Num_
```

```
instance Tree Tm where
  fromTree (Num n)     = App "Num" [fromTree n    ]
  fromTree (Add t1 t2) = App "Add" [fromTree t1 t2]
  fromTree (Mul t1 t2) = App "Mul" [fromTree t1 t2]

  toTree = parseTree [app "Num" (Num <$> arg         )
                     ,app "Add" (Add <$> arg <*> arg)
                     ,app "Mul" (Mul <$> arg <*> arg)
                     ]
```

```
instance Tree Val where
  fromTree (VNum n) = App "VNum" [fromTree n]
  toTree = parseTree [app "VNum" (VNum <$> arg)]
```


Syntax: terms and values (cont'd)

Some helper functions related to the observation that $\mathbf{Val} \subseteq \mathbf{Tm}$:

```
isValue :: Tm -> Bool

isValue (Num _) = True
isValue _       = False
```

```
fromTm :: Tm -> Val

fromTm (Num n)   = VNum n
fromTm (Add _ _) = error "fromTm: Add"
fromTm (Mul _ _) = error "fromTm: Mul"
```


Small-step operational semantics


```
reduce :: Tm -> Maybe Tm    -- produces Nothing for irreducible
                            -- terms
```

(exercise)

```
eval :: Tm -> Val

eval t = case reduce t of
           Nothing | isValue t -> fromTm t
                   | otherwise -> ⊥        -- impossible
           Just t'             -> eval t'
```

Big-step operational semantics

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Tm} \to \mathit{Val}\\
&\mathit{eval}\ (\mathit{Num}\ n) \quad = \mathit{VNum}\ n\\
&\mathit{eval}\ (\mathit{Add}\ t_1\ t_2) = \textbf{let}\ \mathit{VNum}\ n_1 = \mathit{eval}\ t_1\\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{VNum}\ n_2 = \mathit{eval}\ t_2\\
&\qquad\qquad\qquad\quad\ \ \textbf{in}\ \ \mathit{VNum}\ (n_1 + n_2)\\
&\mathit{eval}\ (\mathit{Mul}\ t_1\ t_2) = \textbf{let}\ \mathit{VNum}\ n_1 = \mathit{eval}\ t_1\\
&\qquad\qquad\qquad\qquad\quad\ \ \mathit{VNum}\ n_2 = \mathit{eval}\ t_2\\
&\qquad\qquad\qquad\quad\ \ \textbf{in}\ \ \mathit{VNum}\ (n_1 * n_2)
\end{aligned}
$$

☞ Big-step semantics is implemented more directly.
☞ Evaluation order is enforced by Haskell's $+$ and $*$.