COMPSCI 3MI3 - Principles of Programming Languages

Topic 3 - Syntax & Untyped Arithmetic Expressions

NCC Moore

McMaster University

Fall 2021

Adapted from "Types and Programming Languages" by Benjamin C. Pierce

Introduction

Extended Backus Naur Form

Untyped Arithmetic Expression Syntax

Several Descriptions of Syntax

Induction on Terms

Introduction

In order to talk more rigorously about programming languages, we need to learn the languages of formal language description.

- To do so, we will discuss a very small (almost trivial) language of numbers and booleans, **Untyped Arithmetic Expressions (UAE)**
- We will use formal mathematical tools to reason about:
  - Abstract syntax
  - Evaluation
  - Modelling of run-time errors

Basic Nomenclature

The following nomenclature will be essential going forward. We will describe these informally now, and more formally as they come up throughout the course.

- **Literal** $\rightarrow$ The base of expressions. Examples include `4`, `9`, `true`, and `82.4`
- **Identifier** $\rightarrow$ Things we assign names to in our code, like `foo`, `bar`, `myFunc` and `myObject`
- **Expression** $\rightarrow$ Literals and Identifiers, plus an assortment of operators. For example, `3 + 5`, `True || False`
- **Statement** $\rightarrow$ Imperative commands like if-statements, loops, assignment statements, return statements, and the like.

All of these things are **terms**, but terms can be other things as well.

Extended Backus Naur Form

In order to define a language, we must have a way to define the grammar of that language. **EBNF** is a standardized notation that computer scientists use to define languages.

- The concept is somewhat similar to regular expressions.
- In EBNF, we create **rules**, which define valid grammatical constructions.
- These rules may be nested inside each other, and may be recursive.

EBNF Examples

The following grammar describes how we write the integers.

$\langle Integer \rangle ::= [\text{-}]\langle digit \rangle\{\langle digit \rangle\}$

$\langle digit \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- Here, we have two rules. The top level rule, <Integer>, and a sub-rule, <digit>.
- "|" denotes a set of options. For example, <digit> can be any of the numbers indicated, but no others.
- "[]" denote something which is optional. For example, the negative sign denoting a negative integer may be absent.
- "{}" denote zero or more repetitions of the contents. For example, zero or more digits may follow the first.

EBNF Examples

The following grammar is for writing hexadecimal integers.

$\langle \textit{Hex Integer} \rangle ::= [\text{-}]\langle \textit{hex digit} \rangle \{\langle \textit{hex digit} \rangle\}$

$\langle \textit{hex digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid A \mid B \mid C \mid D \mid E \mid F$

Note that the grammar describing decimal integers is a **subset of** the grammar for hexadecimal integers.

- Although every `Integer` is also a `Hex Integer`, this does not imply anything about how either group would be interpretted.
- EBNF describes *syntax*, not *semantics*!

Death and Syntaxes!

Here are a few more example grammars:

- A Python list. Items in quotes are taken literally.

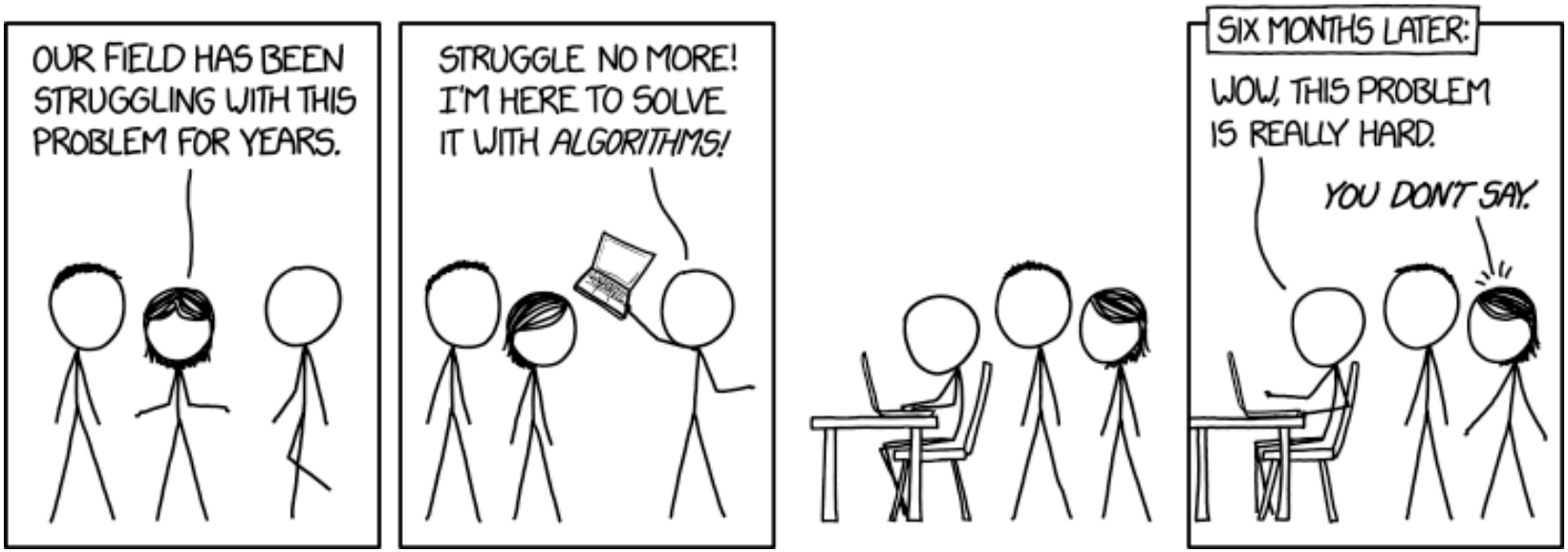
$\langle List \rangle$ ::= '[' $\langle Object\ List \rangle$ ']'

$\langle Object\ List \rangle$ ::= $\langle Object \rangle$, $\langle Object\ List \rangle$ | $\langle Object \rangle$

- A Python function.

$\langle Function \rangle$ ::= def $\langle Identfier \rangle$ ($\langle Argument\ List \rangle$) : '\n\t'
$\langle Statement\ List \rangle$

Untyped Aritmetic Expressions!



UAE Syntax

We are going to use a number of conventions to describe UAE, starting with EBNF.

$$
\begin{aligned}
\langle t \rangle ::= \ & \texttt{true} \\
| \ & \texttt{false} \\
| \ & \texttt{if } \langle t \rangle \texttt{ then } \langle t \rangle \texttt{ else } \langle t \rangle \\
| \ & \texttt{0} \\
| \ & \texttt{succ } \langle t \rangle \\
| \ & \texttt{pred } \langle t \rangle \\
| \ & \texttt{iszero } \langle t \rangle
\end{aligned}
$$

Here, `t` is a **metavariable**. EBNF is a **metalanguage** (a language which describes languages), and `t` is a variable of that language.

Semantic Observations

You may have noticed that we are defining the natural numbers in terms of zero and a successor function, similar to Peano Arithmetic.

- For convenience, we will sometimes abbreviate successive applications of `succ` using arabic numerals. For example,

$$succ(succ(succ(0))) = 3 \tag{1}$$

- We will use a number of metavariable names, such as `s`, `t`, `u` and `v` throughout this course, in the same way that we might use `n` and `m` for natural numbers.
- You may notice as well that we have a very minimal number of operations on our three literals, `true`, `false`, and `0`.
- We only have a conditional, a predicate testing if a term is eqivalent to zero, a successor and a predecessor.

Writing Expressions

We compose terms in our language using our recursive grammar.

```
1 if false then 0 else 1
2  >> 1
3 iszero pred succ 0
4  >> true
```

We will use >> to denote the **values** resulting from executing small programs in our small languages. For ease of reading, we will also often use:

- Round braces for grouping and precedence.
- Semicolons to terminate lines.

```
1 if false then 0 else 1;
2 iszero (pred (succ 0));
```

Invalid Forms

Note that, according to our grammar, we can construct a number invalid programs, such as:

```
1 if 0 then 0 else 0;
2 succ true;
3 iszero false;
```

Our grammar alone is not sufficient to preclude this type of error. Remember, grammars do not describe semantics, only syntax! The above statements are equivalent to an English sentence like "Plastic orbits acrue Dakota portably." Grammatically correct, but nonsense.

- In general, these are examples of **type errors**.
- This type of error is precisely the sort of thing we want our type system to exclude.

Define Your Terms!

In the previous section, we defined the language of a small grammar using EBNF. Let's examine some equivalent descriptions using the following:

- Defining Terms Inductively
- Defining Terms Using Inference Rules
- Defining Terms Concretely

Inductive Definition

The set of *terms* is the smallest set $\mathcal{T}$ such that:

$$\{\texttt{true}, \texttt{false}, 0\} \subseteq \mathcal{T} \tag{2}$$

$$t_1 \in \mathcal{T}, \rightarrow \{\texttt{succ}\ t_1, \texttt{pred}\ t_1, \texttt{iszero}\ t_1\} \subseteq \mathcal{T} \tag{3}$$

$$t_1, t_2, t_3 \in \mathcal{T} \rightarrow \{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3\} \subseteq \mathcal{T} \tag{4}$$

Induction! And Destruction!

Inductive definitions are ubiquitous in the study of programming languages.

- The first clause gives us three simple expressions that are in $\mathcal{T}$.
- The second and third clauses give us four compound expressions that are in $\mathcal{T}$.
- The word "smallest" tells us that $\mathcal{T}$ has no elements except those required by the above clauses.

Fundamentally, this defines our set of terms as a set of *trees*. Although this grammar does not mention parentheses, we will use them for clarification.

Terms By Rules of Inference

We can also use rules of inference to represent our grammar, similarly to rules of *natural deduction* used in the presentation of logical systems.

$\mathcal{T}$ shall be the set of terms defined by the following rules:

$$\texttt{true} \in \mathcal{T} \qquad \texttt{false} \in \mathcal{T} \qquad 0 \in \mathcal{T}$$

$$\frac{t \in \mathcal{T}}{\texttt{succ}\ t \in \mathcal{T} \qquad \texttt{pred}\ t \in \mathcal{T} \qquad \texttt{iszero}\ t \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T} \qquad t_2 \in \mathcal{T} \qquad t_3 \in \mathcal{T}}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \in \mathcal{T}}$$

Powers of Inference

Rules of inference are read, "if we have established the premise(s) above the line, we may derive the conclusion(s) below the line."

- These are also referred to as **antecedents** and **consequents**.
- In this type of formulation, the fact that $\mathcal{T}$ the smallest set satisfying these rules is often not stated explicitly.
- Rules with no premises (such as $\texttt{true} \in \mathcal{T}$) are often called **axioms**.
- Formally, the above "inference rules" are properly termed "rule schemas".
  - They may represent infinite sets of concrete rules, via the inclusion of **metavariables**.

Terms, Concretely

For each natural number i, define the set S_i as follows:

$$
\begin{aligned}
S_0 &= \varnothing \\
S_{i+1} &= \quad \{\mathit{true}, \mathit{false}, 0\} \\
&\quad \cup \{t \in S_i \mid \texttt{succ}\ t, \texttt{pred}\ t, \texttt{iszero}\ t\} \\
&\quad \cup \{t_1, t_2, t_3 \in S_i \mid \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3\}
\end{aligned}
$$

And let

$$S = \bigcup_i S_i$$

Concrete and Steel

The previous description provides not only a definition for the language, but a mechanism for generating the terms of the language as well.

- S_0 is empty.
- S_1 contains only our constants
- S_2 contains all terms built from our constants
- S_3 contains all terms built from the terms in S_2
- And so on...

Does $\mathcal{T} = S$?

While it may be obvious to casual inspection that the inductive forumation of UAE is equivalent to the constructive formulation, a rigorous mathematician doesn't jump to such conclusions!

- Proving that the two formulations are equivalent will expose any inequivalencies, should they exist.
- Such inequivalencies, if carried forward, may have disastrous consequences!

As a demonstration of mathematical reasoning over these formulations, we will prove that the set of terms $\mathcal{T}$, inductively defined, is equivalent to the set S, which has been constructively defined.

Let's Prove $\mathcal{T} = S$

We defined $\mathcal{T}$ as the smallest set satisfying the following conditions:

$$\{\texttt{true}, \texttt{false}, 0\} \subseteq \mathcal{T} \tag{5}$$

$$t_1 \in \mathcal{T}, \rightarrow \{\texttt{succ}\ t_1, \texttt{pred}\ t_1, \texttt{iszero}\ t_1\} \subseteq \mathcal{T} \tag{6}$$

$$t_1, t_2, t_3 \in \mathcal{T} \rightarrow \{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3\} \subseteq \mathcal{T} \tag{7}$$

To prove $S = \mathcal{T}$, it is sufficient to prove:

- That S also satisfies the above conditions.
- For S', which represents any set satisfying the above conditions, S must be a subset of that set. That is, S is always *smaller than or equal to* a generic set satisfying the conditions.

Does S Satisfy the Inductive Definition?

Let's consider our conditions one at a time.

- $\{\texttt{true}, \texttt{false}, 0\} \subseteq \mathcal{T}$
  - In our constructive definition, S_1 contains only these constant terms.

$$\therefore \{\texttt{true}, \texttt{false}, 0\} = S_1 \subset S \tag{8}$$

- $t \in \mathcal{T}, \rightarrow \{\texttt{succ}\ t, \texttt{pred}\ t, \texttt{iszero}\ t\} \subseteq \mathcal{T}$
  - If $t \in S$, then it follows that t must be an element of some S_i, since there is no other possible origin for t.
  - Given that one of sets used to construct S_{i+1} is $\{t \in S_i \mid \texttt{succ}\ t, \texttt{pred}\ t, \texttt{iszero}\ t\}$, and since $S_{i+1} \subset S$, it follows that:

$$\texttt{succ}\ t, \texttt{pred}\ t, \text{and } \texttt{iszero}\ t \in S \tag{9}$$

- The same argument as above may be applied to $t_1, t_2, t_3 \in \mathcal{T} \rightarrow \{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3\} \subseteq \mathcal{T}$

Is S the Smallest?

Suppose some set S' satisfies the conditions set out in our inductive definition of $\mathcal{T}$. We will start by proving, via complete induction, that $S_i \subseteq S'$. It will be then trivial to show that $S \subseteq S'$

- Premise: $\forall j < i$, assume $S_j \subseteq S$.
  - Case: $i = 0$
    - If $i = 0$, then $S_0 = \varnothing$, which is trivially $\in S'$
  - Otherwise, $i = j + 1$ for some $j \in \mathbb{N}$
    - Let $t \in S_{j+1}$. S_{j+1} is constructed from three smaller sets, so t has three possible origins.

$$t \in \{\texttt{true}, \texttt{false}, 0\} \tag{10}$$

$$t \in \{\texttt{succ}\ t_1, \texttt{pred}\ t_1, \texttt{iszero}\ t_1\} \tag{11}$$

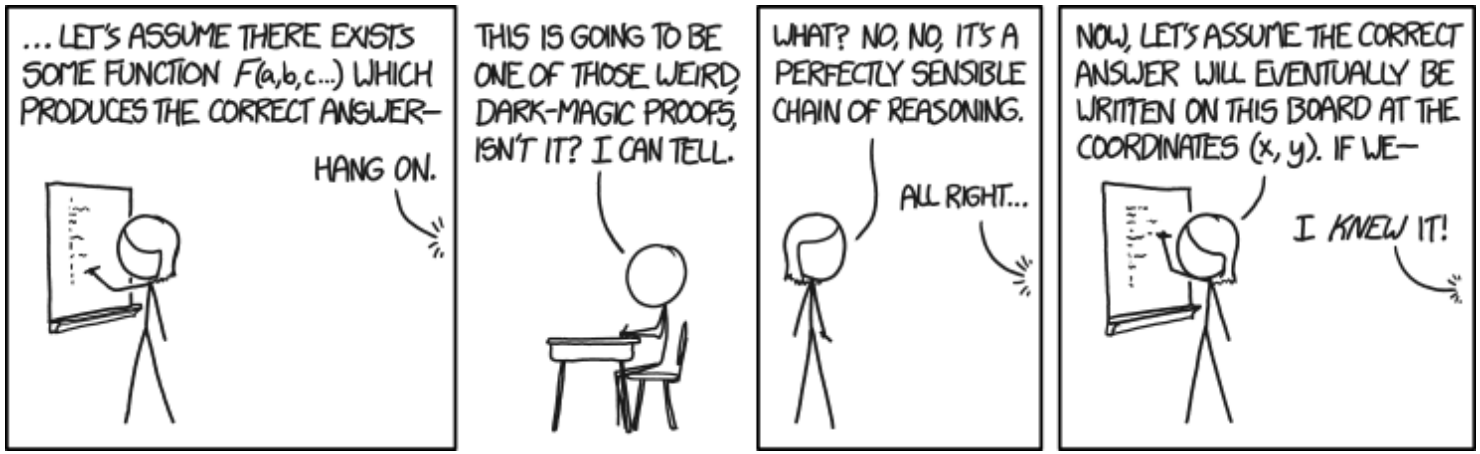
$$t = \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \tag{12}$$

Where $t_1, t_2, t_3 \in S_j$

Is S the Smallest? (cont.)

- Following from this:
  - For equation 10, we know $t \in S'$ via equation 5 above.
  - For equation 11, we know $t \in S'$ via equation 6 above, and the fact that $t_1 \in S'$ via the induction hypthosesis.
  - For equation 12, we know $t \in S'$ via equation 7 above, and the induction hypothesis over t_1, t_2 and t_3.
- Since every possible t is an element of S', it follows that S_i, which is the union of every t we just discussed, is also a subset of S', thus completing our inductive sub-proof.
- By a similar argument, since each $S_i \subseteq S'$, and S is the union of all S_i, it follows that $S \subseteq S'$.
- Therefore, for any arbitrarily constructed set S' satisfying the conditions of $\mathcal{T}$, the S obtained by our constructive definition is at most as big as S'. Therefore, S is the *smallest set for which the conditions hold*.

Induction on Terms



"Induction is the glory of science and the scandal of philosophy."
– C.D. Broad

Depth and Size

To facilitate reasoning about current and future languages, it will be useful to define the concepts **depth** and **size**.

- The depth of a term is analogous to the depth of a tree, i.e., the longest path from our starting term to a terminal constant.
- Continuing the tree analogy, the size of a term is the number of nodes in the tree. Here, a node is any of the terms described by our grammar.

In addition, it will be useful to be able to extract the set of constants used in any particular term.

Set of Constants

The set of constants appearing in a term t, written $Consts(t)$ is written as follows:

$$
\begin{array}{lcl}
Consts(\texttt{true}) & = & \{\texttt{true}\} \\
Consts(\texttt{false}) & = & \{\texttt{false}\} \\
Consts(\texttt{0}) & = & \{\texttt{0}\} \\
Consts(\texttt{succ}\ t_1) & = & Consts(t_1) \\
Consts(\texttt{pred}\ t_1) & = & Consts(t_1) \\
Consts(\texttt{iszero}\ t_1) & = & Consts(t_1) \\
Consts(\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) & = & Consts(t_1) \cup Consts(t_2) \\
 & & \quad \cup\ Consts(t_3)
\end{array}
$$

Size of a Term

The size of a term t, written $size(t)$ is written as follows:

$$
\begin{array}{lcl}
size(\texttt{true}) & = & 1 \\
size(\texttt{false}) & = & 1 \\
size(\texttt{0}) & = & 1 \\
size(\texttt{succ}\ t_1) & = & size(t_1) + 1 \\
size(\texttt{pred}\ t_1) & = & size(t_1) + 1 \\
size(\texttt{iszero}\ t_1) & = & size(t_1) + 1 \\
size(\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) & = & size(t_1) + size(t_2) + size(t_3) + 1
\end{array}
$$

Depth of a Term

The depth of a term t, written $depth(t)$ is written as follows:

$$
\begin{array}{lcl}
depth(\texttt{true}) & = & 1 \\
depth(\texttt{false}) & = & 1 \\
depth(\texttt{0}) & = & 1 \\
depth(\texttt{succ}\ t_1) & = & depth(t_1) + 1 \\
depth(\texttt{pred}\ t_1) & = & depth(t_1) + 1 \\
depth(\texttt{iszero}\ t_1) & = & depth(t_1) + 1 \\
depth(\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3) & = & max(depth(t_1), depth(t_2), \\
 & & \quad depth(t_3)) + 1
\end{array}
$$

Induction on Size and Depth

With these new definitions, we can now introduce three exciting new forms of induction!

[Induction on Size]

- If, for $s \in \mathcal{T}$
  - Given $P(r)$ for all r such that $size(r) < size(s)$
  - we can show $P(s)$
- We may conclude $\forall s \in \mathcal{T} \mid P(s)$

[Induction on depth]

- If, for $s \in \mathcal{T}$
  - Given $P(r)$ for all r such that $depth(r) < depth(s)$
  - we can show $P(s)$
- We may conclude $\forall s \in \mathcal{T} \mid P(s)$

These two forms of induction are derived from Complete Induction over $\mathbb{N}$

Structural Induction over Terms

[Structural Induction Over Terms]

- If, for $s \in \mathcal{T}$
- We can show $P(c)$ for the language constants, and
  - Given $P(r)$ for all immediate subterms of r of s
  - we can show $P(s)$
- We may conclude $\forall s \in \mathcal{T} \mid P(s)$

These methods of induction are equivalent to each other, but using one or the other can *simplify our proofs*.

- Formally, these three forms of induction are interderivable.
- As a matter of style, we will often use structural induction:
  1. Because it is a bit more intuitive.
  2. To avoid having to detour into numbers.

Structural Induction over Terms (cont.)

In general, proofs in the above style will have the following structure:

- We are given a term t and a property P.
- We assume P holds for all subterms of t.
- We then break the proof into cases. Our goal is to demonstrate $P(t)$, for each possible subterm.
- Since languages can have a *lot* of terms, we will often only create proofs explicitly for the interesting cases.
  - What constitutes an "uninteresting" case will become more obvious as we progress through the course.
- This will be sufficient to complete our proof.

Last Slide Comic