CSCI 3366 Programming Languages

Spring 2022

R. Muller, minor revisions by **J. Tassarotti**

Lecture Notes: Week 5

1. Extra information about parsing
2. Semantics and the mini-PL Mercury

1. More on Parsing

For "small" languages, i.e., languages without too much complexity in their syntax, recursive-descent parsing is often a reasonable strategy. With a little practice, it's often possible to type in a recursive-descent parser program, reading the grammar as a script. But in many cases, it's more convenient to make use of one of the many parser generator utilities.

Parser Generators

A *parser generator* is a program that accepts a context-free grammar (of one kind or another) and which generates a parser function in the compiler implementation language. The generated parser is integrated into the front-end of a compiler, situated between a tokenizer component and whatever compiler component follows parsing, usually elaboration or analysis. Parser generation has been one of the great success stories in computing.

In this course we're working with 4 little programming languages:

1. Mercury – students write a recursive descent parser;
2. Venus – a recursive descent parser is provided as part of the harness code;
3. Earth – a recursive descent parser is provided as part of the harness code;
4. Mars/MiniC – a parser produced by the ML-Yacc parser generator is provided as part of the harness code.

Semantics

Let P be a program. Two of the major foundational concerns of computer science are: A. how much work does P do when executed? and B. what does P mean? There are several approaches to specifying the semantics of a programming language ranging from reference compilers ("P means what compiler C says it means.") to informal text descriptions to formal approaches using mathematical models. Most approaches ascribe meanings to *abstract syntax*, that is, the essential structure of the program after all of the details of the concrete syntax have been stripped away.

Heads up! From here on, we're primarily going to be working with abstract syntax rather than concrete syntax. Also, let A be a non-terminal symbol in a CFG grammar such that $A \rightarrow^* w$. In this case, A defines the language $\mathcal{L}(A)$. From here on, we're going to feel free to use A ambiguously as a non-terminal symbol, as the set $\mathcal{L}(A)$ and also as a typical element of $\mathcal{L}(A)$.

Denotational Semantics

The denotational idea is simple on its face: the meaning of a program is an element of mathematical set, a so-called *semantic domain*. This approach was pioneered by Christopher Strachey and Dana Scott at the University of Oxford in the early 1970s. The Scott-Strachey approach generally works very well though the sets serving as semantic domains require special structure (they are complete partial-orders) and the mathematical functions serving as meanings of programs are required to preserve the order. Further details on this can be found in the lecture notes in `hutr0692.pdf`.

One of the attractive aspects of the Scott-Strachey approach is that the same semantic domains can be used for meanings of programs in different languages. This can be useful in verifying the correctness of translation – one would have to show that the source program and the translated program denote the same semantic value.

Operational Semantics

As its name suggests, the operational approach is generally concerned with machine states and the transitions between states that arise during the step-by-step execution of a program. The semantics relates initial states to final states. The operational approach was developed over many years and with contributions from many people, including Peter Landin. But operational semantics was fully developed in its modern from by Gordon Plotkin. Plotkin is known in particular for the widely-used Structural Approach to Operational Semantics (or SOS). This kind of semantics is sometimes calls "small-step" because each state transition step is explicitly specified.

The computation steps in an operational semantics are often specified using an *axiomatic system* with inference rules of the form

```
  antecendent1; ... ; antecedentk
-----------------------------------
             consequent
```

The items occurring in an inference rule will usually be *judgments*. In the simple examples here they are of the form `E -+-> E'`, pronounced "`E` steps to `E'`". We'll use the symbol `-+->` both as this syntactic item but also for the relation the system defines:

$$-+->\ = \{(E, E') \mid E-+->E' \text{ is derivable from the rules}\}$$

We'll use the symbol `-+->*` for the relation defined by zero or more steps of `-+->`. The inference rules define an evaluation relation

$$Eval \subseteq E \times V = \{(E, V) \mid E -+->^* V\}$$

Example

Abstract syntax

```
V ::= i           i an integer
E ::= V | E + E
```

Operational Semantics

We have 3 rules.

```
        E1 -+-> E1'                E -+-> E'              add(V1, V2) = V
  -------------------------   --------------------   ----------------------
    E1 + E2 -+-> E1' + E2       V + E -+-> V + E'          V1 + V2 -+-> V
```

Example

```
                        add(2, 3) = 5
                      --------------------
                          2 + 3 -+-> 5          add(5, 4) = 9
                    -------------------------   -----------------
                      (2 + 3) + 4 -+-> 5 + 4;     5 + 4 -+-> 9
                    ---------------------------------------------(Transitivity)
                                (2 + 3) + 4 -+->* 9
```

Evaluation Semantics

In this course, we're going to define the evaluation relation Eval using a less precise relation which abstracts away the individual computation steps. This is sometime called *Natural Semantics* or *Big Step* semantics. It is due to Giles Kahn.

```
                                    E1 | V1;  E2 | V2;  add(V1, V2) = V
  --------------------      ----------------------------------------
        V | V                            E1 + E2 | V
```

Then the evaluation relation doesn't require a transitive and reflexive closure

$$Eval = \{(E, V) \mid E \mid V \text{ is derivable from the rules}\}$$