CSCI 3366 Programming Languages

Prof. R. Muller

The Programming Language Mercury

Mercury is the simplest "programming" language imaginable — it has only basic integer expressions. We develop it here for the purposes of showing how to formally specify both the syntax and semantics of a programming language, for showing how to left-factor a grammar to ease the process of writing a recursive-descent parser and to establish something of a baseline when it comes to possible errors that can occur during the execution of a program.

An example Mercury program is

```
Mercury> 2 + 3 * 4
ast = +(2, *(3, 4))
value = 14

Mercury> 4 - 1 - 1
ast = -(-(4, 1), 1)
value = 2
```

The first example shows that the parser for Mercury follows the familiar rules assigning higher precedence to multiplication (and division and mod) than addition (or subtraction). The second example shows that the parser follows the familiar rules for *left associativity* of operators.

As far as semantics goes, the first example can be reduced to its *value* in two steps:

```
2 + 3 * 4 ->
  2 + 12 ->
  14
```

As in basic arithmetic and alegbra, computation in Mercury can be specified by simplification rules that drive expressions toward their values, if they have one.

Possible Errors

Mercury is so simple that most of the errors that are found in more realistic programming languages cannot occur in Mercury. The only two errors that can occur in a well-formed Mercury-program, are *division by zero*

```
mercury> 1 / 0
```

and *integer overflow*. These are examples of a large class of errors that in real languages are generally impossible to detect by inspecting the code — they can

only be detected by actually running the program. Division by zero and integer overflow are both examples of *trapped errors* — they are detected at run-time (the former in software and the latter in hardware) and allow for the possibility of recovery. The more problematic *untrapped errors* cannot occur in Mercury.

Syntax

Let `i` denote an integer.

Concrete Syntax

The concrete syntax for Mercury is given by the following context-free grammar where `E` is the start symbol.

$$\begin{array}{lcl} \texttt{V} & ::= & \texttt{i} \\ \texttt{E} & ::= & \texttt{E} + \texttt{T} \mid \texttt{E} - \texttt{T} \mid \texttt{T} \\ \texttt{T} & ::= & \texttt{T} * \texttt{F} \mid \texttt{T} \,/\, \texttt{F} \mid \texttt{T} \,\%\, \texttt{F} \mid \texttt{F} \\ \texttt{F} & ::= & \texttt{V} \mid (\ \texttt{E}\ ) \end{array}$$

The grammar gives the appropriate precedence and associativity but is incompatible with recursive-descent parsing because it implements left-associativity using left-recursion. The grammar can be factored to form an equivalent right-recursive grammar by *left factoring*. Replace left-recursive rules of the form:

$$A ::= A\alpha \mid \beta$$

with pairs of rules:

$$\begin{array}{lcl} A & ::= & \beta A' \\ A' & ::= & \alpha A' \mid \epsilon \end{array}$$

Abstract Syntax

The abstract syntax of Mercury-programs is given by the `E` production of the following grammar:

$$\begin{array}{lcl} \texttt{o} & ::= & + \mid - \mid * \mid / \mid \% \\ \texttt{V} & ::= & \texttt{i} \\ \texttt{E} & ::= & \texttt{V} \mid \texttt{E}\ \texttt{o}\ \texttt{E} \end{array}$$

Semantics

The semantics of Mercury-programs is defined by a *reduction relation* Eval $\subseteq$ $(\texttt{E} \times \texttt{V})$. Reduction relations will be used to define the semantics for all of the simple programming languages considered in this course.

Reduction relations are usually defined using an *axiomatic system* with axioms and inference rules as shown in Figure 1. The system is set up to allow for

$$\frac{}{\text{axiomatic judgement}}\,(\texttt{axiom}) \qquad \frac{\text{... antecedents ...}}{\text{conclusion judgement}}\,(\texttt{ inference rule})$$

Figure 1: Axioms and Inference Rules

$$\frac{}{\vdash \mathtt{V} \Downarrow \mathtt{V}}\,(\texttt{int})$$

$$\frac{\vdash \mathtt{E}_1 \Downarrow \mathtt{V}_1;\ \vdash \mathtt{E}_2 \Downarrow \mathtt{V}_2;\ \mathrm{Apply}_2(\mathtt{o}, \mathtt{V}_1, \mathtt{V}_2) = \mathtt{V}}{\vdash \mathtt{E}_1\ \mathtt{o}\ \mathtt{E}_2 \Downarrow \mathtt{V}}\,(\texttt{BinOp})$$

Figure 2: $\vdash \mathtt{E} \Downarrow V$

the logical deduction of *judgement* forms of one form or another.

The simple axiomatic system specifying the semantics of Mercury is defined with judgements relating Mercury-programs to their values.

$$\mathrm{Eval} = \{(\mathtt{E}, \mathtt{V}) \mid \vdash \mathtt{E} \Downarrow \mathtt{V}\}$$

We want to define the judgements, axioms and rules in such a way that the pair $(+(2, *(3, 4)), 14) \in \mathrm{Eval}$ but the pair $(+(2, 3), 4)$ is not and neither, for that matter, is $(+(2, *(3, 4)), +(2, 12))$ (among other reasons, $+(2, 12) \notin \mathtt{V}$).

The axiomatic system defining the semantics of Mercury is shown in Figure 2. It has one axiom relating constants to themselves and one inference rule for operators. It uses an auxiliary function Apply_2 to define the semantics of the primitive operations.