CS 320: Stack Language Compiler

Part 3 Due: December 10, 2021, 11:59pm

1 Overview

The goal of part 3 is to compile a high level functional language (similar to OCaml) to the stack language of part 2. The following diagram depicts a program $Prog_f$ of the functional language being compiled to a program $Prog_s$ of the stack language. The interpreting $Prog_f$ and $Prog_s$ using their respective interpreters should result in the same output. Your task is to implement such a compiler.

$$\begin{tikzcd} Prog_f \arrow[r, "compile"] \arrow[dr, "interp_f"'] & Prog_s \arrow[d, "interp_s"] \\ & Output \end{tikzcd}$$

2 Syntax

2.1 Concrete Syntax

The following grammar is the concrete syntax of the functional language. This is mainly for reference purposes as the parser for generating abstract syntax will be provided as template code.

⟨*digit*⟩ ::= 0 ... 9

⟨*nat*⟩ ::= ⟨*digit*⟩ { ⟨*digit*⟩ }

⟨*letter*⟩ ::= a ... z | A ... Z

⟨*initial*⟩ ::= ⟨*letter*⟩ | _

⟨*name*⟩ ::= ⟨*initial*⟩ { ⟨*letter*⟩ | ⟨*digit*⟩ | _ | ' }

⟨*const*⟩ ::= ⟨*nat*⟩ | ()

⟨*opr*⟩ ::= + | - | * | /

⟨*term*⟩ ::= ⟨*name*⟩
  | fun ⟨*name*⟩ → ⟨*term*⟩
  | ⟨*term*⟩ ⟨*term*⟩
  | if ⟨*term*⟩ then ⟨*term*⟩ else ⟨*term*⟩
  | let ⟨*name*⟩ = ⟨*term*⟩ in ⟨*term*⟩
  | let rec ⟨*name*⟩ ⟨*name*⟩ { ⟨*name*⟩ } = ⟨*term*⟩ in ⟨*term*⟩
  | ⟨*const*⟩
  | ⟨*term*⟩ ⟨*opr*⟩ ⟨*term*⟩
  | trace ⟨*term*⟩
  | (⟨*term*⟩)

2.2 Abstract Syntax

The following grammar is the abstract syntax of the functional language. The provided parser will read a program written in the concrete syntax and produce its AST representation.

```
type name = string

type term =
| Name  of name
| Fun   of name * name * term
| App   of term * term
| Ifgz  of term * term * term
| LetIn of name * term * term
| Unit
| Int   of int
| Add   of term * term
| Sub   of term * term
| Mul   of term * term
| Div   of term * term
| Trace of term

type value =
| IntVal of int
| FunVal of name * name * term * env
| UnitVal
```

2.3 Parser

The parser of the functional language has the following type signature.

```
parse_prog : string -> (term * char list) option
```

When `parse_prog` is given a string, it will attempt to parse a functional program and generate its abstract syntax representation. If the parse is successful, then the generated AST and remaining input will be returned within a `Some` constructor. If the parse fails, then `None` is returned instead.

2.4 Description of AST

- `Name of name`
  A variable of the language. Can be used to bind values in the environment.
- `Fun of` $\texttt{name}_1$ `*` $\texttt{name}_2$ `* term`
  A possibly recursive function. Here, $\texttt{name}_1$ is the name of the function and $\texttt{name}_2$ is its argument. $\texttt{name}_1$ may be called recursively from within the function body `term`.
- `App of` $\texttt{term}_1$ `*` $\texttt{term}_2$
  An application form. Intuitively, $\texttt{term}_1$ is expected to be a function and $\texttt{term}_2$ is expected to be its argument.
- `Ifgz of` $\texttt{term}_1$ `*` $\texttt{term}_2$ `*` $\texttt{term}_3$
  The branching if-then-else expression. If $\texttt{term}_1$ is evaluated to be greater than 0, the value of the overall expression is determined by evaluating $\texttt{term}_2$, otherwise it is determined by evaluating $\texttt{term}_3$.
- `LetIn of name *` $\texttt{term}_1$ `*` $\texttt{term}_2$
  A let-binding expression. The value of $\texttt{term}_1$ is locally bound to `name` within the scope of $\texttt{term}_2$.

- `Unit`
  A unit constant. Corresponds to `()` of the stack language.

- `Int of int`
  A constant integer. You may assume the integer values here are always non-negative.

- `Add of` $\texttt{term}_1$ `*` $\texttt{term}_2$
  Addition of the values of $\texttt{term}_1$ and $\texttt{term}_2$.

- `Sub of` $\texttt{term}_1$ `*` $\texttt{term}_2$
  Subtraction of the values of $\texttt{term}_1$ and $\texttt{term}_2$.

- `Mul of` $\texttt{term}_1$ `*` $\texttt{term}_2$
  Multiplication of the values of $\texttt{term}_1$ and $\texttt{term}_2$.

- `Div of` $\texttt{term}_1$ `*` $\texttt{term}_2$
  Division of the values of $\texttt{term}_1$ and $\texttt{term}_2$.

- `Trace of term`
  Conversion the value of `term` to a string and logging it.

3 Semantics

The behavioral specification of the functional language will be described using big-step operational semantics. The functional language interpreter used to test your compiler is implemented based on these rules. The evaluation judgment is of the form $(prog/env/log) \Downarrow (value/log')$, where *prog* is a term, *env* is an environment, *value* is the result of evaluation and *log* is the trace.

When compiling *prog*, the generated stack program must produce *value* on top of the stack with the log' when evaluated by the stack language interpreter of Part 2.

3.1 Names

Evaluation of a name x is accomplished by looking up its associated value within the environment. If the name cannot be found within the environment, an error is raised.

$$\frac{x \in env \qquad lookup(env, x) = v}{(\texttt{Name}(x)\ /\ env\ /\ log) \Downarrow (v\ /\ log)}\ \text{NAME} \qquad\qquad \frac{x \notin env}{(\texttt{Name}(x)\ /\ env\ /\ log) \Downarrow \texttt{Error}}\ \text{NAME-ERR}$$

3.2 Functions and Applications

Functions evaluate immediately to a function closure, capturing the variable bindings of the current environment the function was defined in.

During an application, both sides of the application are evaluated with the log threaded through. If the left-hand-side evaluates to a closure, then the closure's body is evaluated using its local environment extended with the value of the right-hand-side and the closure itself. If the left-hand-side does not evaluate to a closure, then an error is raised.

$$\frac{}{(\texttt{Fun}(f, x, t)\ /\ env\ /\ log) \Downarrow (\texttt{FunVal}(f, x, t, env)\ /\ log)}\ \text{FUNCTION}$$

$$\frac{\begin{array}{c}(t_1\ /\ env\ /\ log_1) \Downarrow (\texttt{FunVal}(f, x, t, env')\ /\ log_2) \qquad (t_2\ /\ env\ /\ log_2) \Downarrow (v_1\ /\ log_3) \\ (t\ /\ (x, v_1) :: (f, \texttt{FunVal}(f, x, t, env')) :: env'\ /\ log_3) \Downarrow (v_2\ /\ log_4)\end{array}}{(\texttt{App}(t_1, t_2)\ /\ env\ /\ log_1) \Downarrow (v_2\ /\ log_4)}\ \text{APP}$$

$$\frac{(t_1\ /\ env\ /\ log) \Downarrow (\texttt{IntVal}(i)\ /\ log')}{(\texttt{App}(t_1, t_2)\ /\ env\ /\ log) \Downarrow \texttt{Error}}\ \text{APP-ERR1} \qquad\qquad \frac{(t_1\ /\ env\ /\ log) \Downarrow (\texttt{UnitVal}\ /\ log')}{(\texttt{App}(t_1, t_2)\ /\ env\ /\ log) \Downarrow \texttt{Error}}\ \text{APP-ERR2}$$

3.3 Branching

Ifgz corresponds to if-then-else branching expressions. If the conditional expression evaluates to a number greater than 0, then true branch is evaluated, otherwise the false branch is evaluated. If the conditional does not evaluate to a number, then an error is raised.

$$\frac{(t \;/\; env \;/\; log_1) \Downarrow (\texttt{IntVal}(i) \;/\; log_2) \qquad i > 0 \qquad (t_1 \;/\; env \;/\; log_2) \Downarrow (v \;/\; log_3)}{(\texttt{Ifgz}(t, t_1, t_2) \;/\; env \;/\; log_1) \Downarrow (v \;/\; log_3)} \textsc{ ifgz-true}$$

$$\frac{(t \;/\; env \;/\; log_1) \Downarrow (\texttt{IntVal}(i) \;/\; log_2) \qquad i \leq 0 \qquad (t_2 \;/\; env \;/\; log_2) \Downarrow (v \;/\; log_3)}{(\texttt{Ifgz}(t, t_1, t_2) \;/\; env \;/\; log_1) \Downarrow (v \;/\; log_3)} \textsc{ ifgz-false}$$

$$\frac{(t \;/\; env \;/\; log) \Downarrow (\texttt{FunVal}(f, x, t', env') \;/\; log')}{(\texttt{Ifgz}(t, t_1, t_2) \;/\; env \;/\; log) \Downarrow \texttt{Error}} \textsc{ ifgz-err1} \qquad \frac{(t \;/\; env \;/\; log) \Downarrow (\texttt{UnitVal} \;/\; log')}{(\texttt{Ifgz}(t, t_1, t_2) \;/\; env \;/\; log) \Downarrow \texttt{Error}} \textsc{ ifgz-err2}$$

3.4 Local Definitions

For a LetIn expression, the local term t_1 is evaluated to its value v_1. The body t_2 is evaluated in the environment extended with x bound to v_1. It is important to note that the variable x is locally scoped to t_2.

$$\frac{(t_1 \;/\; env \;/\; log_1) \Downarrow (v_1 \;/\; log_2) \qquad (t_2 \;/\; (x, v_1) :: env \;/\; log_2) \Downarrow (v_2 \;/\; log_3)}{(\texttt{LetIn}(x, t_1, t_2) \;/\; env \;/\; log_1) \Downarrow (v_2 \;/\; log_3)} \textsc{ letin}$$

3.5 Constant Values

Constant numbers and units evaluate immediately to their value counterparts.

$$\frac{}{(\texttt{Unit} \;/\; env \;/\; log) \Downarrow (\texttt{UnitVal} \;/\; log)} \textsc{ unit} \qquad \frac{}{(\texttt{Int}(i) \;/\; env \;/\; log) \Downarrow (\texttt{IntVal}(i) \;/\; log)} \textsc{ int}$$

3.6 Addition

Addition may be performed if its left and right arguments evaluate to numbers, resulting in the sum of these numbers. If the left or right cannot be evaluated to numbers, then an error is raised.

$$\frac{(t_1 \;/\; env \;/\; log_1) \Downarrow (\texttt{IntVal}(i_1) \;/\; log_2) \qquad (t_2 \;/\; env \;/\; log_2) \Downarrow (\texttt{IntVal}(i_2) \;/\; log_3)}{(\texttt{Add}(t_1, t_2) \;/\; env \;/\; log_1) \Downarrow (\texttt{IntVal}(i_1 + i_2) \;/\; log_3)} \textsc{ add}$$

$$\frac{(t_1 \;/\; env \;/\; log) \Downarrow (\texttt{FunVal}(f, x, t, env') \;/\; log')}{(\texttt{Add}(t_1, t_2) \;/\; env \;/\; log) \Downarrow \texttt{Error}} \textsc{ add-err1} \qquad \frac{(t_1 \;/\; env \;/\; log) \Downarrow (\texttt{UnitVal} \;/\; log')}{(\texttt{Add}(t_1, t_2) \;/\; env \;/\; log) \Downarrow \texttt{Error}} \textsc{ add-err2}$$

$$\frac{(t_1 \;/\; env \;/\; log_1) \Downarrow (\texttt{IntVal}(i_1) \;/\; log_2) \qquad (t_2 \;/\; env \;/\; log_2) \Downarrow (\texttt{FunVal}(f, x, t, env') \;/\; log_3)}{(\texttt{Add}(t_1, t_2) \;/\; env \;/\; log_1) \Downarrow \texttt{Error}} \textsc{ add-err3}$$

$$\frac{(t_1 \;/\; env \;/\; log_1) \Downarrow (\texttt{IntVal}(i_1) \;/\; log_2) \qquad (t_2 \;/\; env \;/\; log_2) \Downarrow (\texttt{UnitVal} \;/\; log_3)}{(\texttt{Add}(t_1, t_2) \;/\; env \;/\; log_1) \Downarrow \texttt{Error}} \textsc{ add-err4}$$

3.7 Subtraction

Subtraction may be performed if its left and right arguments evaluate to numbers. If the numeric value of the left argument is greater than or equal to the numeric value of the right argument, then the entire expression evaluates to the difference of both values. Otherwise an error is raised.

$$\frac{(t_1 \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1) \ / \ log_2) \qquad (t_2 \ / \ env \ / \ log_2) \Downarrow (\texttt{IntVal}(i_2) \ / \ log_3) \qquad i_1 \geq i_2}{(\texttt{Sub}(t_1, t_2) \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1 - i_2) \ / \ log_3)} \textsc{ sub}$$

$$\frac{(t_1 \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1) \ / \ log_2) \qquad (t_2 \ / \ env \ / \ log_2) \Downarrow (\texttt{IntVal}(i_2) \ / \ log_3) \qquad i_1 < i_2}{(\texttt{Sub}(t_1, t_2) \ / \ env \ / \ log_1) \Downarrow \texttt{Error}} \textsc{ sub-err1}$$

$$\frac{(t_1 \ / \ env \ / \ log) \Downarrow (\texttt{FunVal}(f, x, t, env') \ / \ log')}{(\texttt{Sub}(t_1, t_2) \ / \ env \ / \ log) \Downarrow \texttt{Error}} \textsc{ sub-err2} \qquad \frac{(t_1 \ / \ env \ / \ log) \Downarrow (\texttt{UnitVal} \ / \ log')}{(\texttt{Sub}(t_1, t_2) \ / \ env \ / \ log) \Downarrow \texttt{Error}} \textsc{ sub-err3}$$

$$\frac{(t_1 \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1) \ / \ log_2) \qquad (t_2 \ / \ env \ / \ log_2) \Downarrow (\texttt{FunVal}(f, x, t, env') \ / \ log_3)}{(\texttt{Sub}(t_1, t_2) \ / \ env \ / \ log_1) \Downarrow \texttt{Error}} \textsc{ sub-err4}$$

$$\frac{(t_1 \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1) \ / \ log_2) \qquad (t_2 \ / \ env \ / \ log_2) \Downarrow (\texttt{UnitVal} \ / \ log_3)}{(\texttt{Sub}(t_1, t_2) \ / \ env \ / \ log_1) \Downarrow \texttt{Error}} \textsc{ sub-err5}$$

3.8 Multiplication

Multiplication may be performed if its left and right arguments evaluate to numbers, resulting in the sum of these numbers. If the left or right cannot be evaluated to numbers, then an error is raised.

$$\frac{(t_1 \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1) \ / \ log_2) \qquad (t_2 \ / \ env \ / \ log_2) \Downarrow (\texttt{IntVal}(i_2) \ / \ log_3)}{(\texttt{Mul}(t_1, t_2) \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1 \times i_2) \ / \ log_3)} \textsc{ mul}$$

$$\frac{(t_1 \ / \ env \ / \ log) \Downarrow (\texttt{FunVal}(f, x, t, env') \ / \ log')}{(\texttt{Mul}(t_1, t_2) \ / \ env \ / \ log) \Downarrow \texttt{Error}} \textsc{ mul-err1} \qquad \frac{(t_1 \ / \ env \ / \ log) \Downarrow (\texttt{UnitVal} \ / \ log')}{(\texttt{Mul}(t_1, t_2) \ / \ env \ / \ log) \Downarrow \texttt{Error}} \textsc{ mul-err2}$$

$$\frac{(t_1 \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1) \ / \ log_2) \qquad (t_2 \ / \ env \ / \ log_2) \Downarrow (\texttt{FunVal}(f, x, t, env') \ / \ log_3)}{(\texttt{Mul}(t_1, t_2) \ / \ env \ / \ log_1) \Downarrow \texttt{Error}} \textsc{ mul-err3}$$

$$\frac{(t_1 \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1) \ / \ log_2) \qquad (t_2 \ / \ env \ / \ log_2) \Downarrow (\texttt{UnitVal} \ / \ log_3)}{(\texttt{Mul}(t_1, t_2) \ / \ env \ / \ log_1) \Downarrow \texttt{Error}} \textsc{ mul-err4}$$

3.9 Division

Subtraction may be performed if its left and right arguments evaluate to numbers. If the numeric value of the right argument is not equal to 0, then the entire expression evaluates to the quotient of both values. Otherwise an error is raised.

$$\frac{(t_1 \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1) \ / \ log_2) \qquad (t_2 \ / \ env \ / \ log_2) \Downarrow (\texttt{IntVal}(i_2) \ / \ log_3) \qquad i_2 \neq 0}{(\texttt{Div}(t_1, t_2) \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1 \div i_2) \ / \ log_3)} \text{ DIV}$$

$$\frac{(t_1 \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1) \ / \ log_2) \qquad (t_2 \ / \ env \ / \ log_2) \Downarrow (\texttt{IntVal}(0) \ / \ log_3)}{(\texttt{Div}(t_1, t_2) \ / \ env \ / \ log_1) \Downarrow \texttt{Error}} \text{ DIV-ERR1}$$

$$\frac{(t_1 \ / \ env \ / \ log) \Downarrow (\texttt{FunVal}(f, x, t, env') \ / \ log')}{(\texttt{Div}(t_1, t_2) \ / \ env \ / \ log) \Downarrow \texttt{Error}} \text{ DIV-ERR2} \qquad \frac{(t_1 \ / \ env \ / \ log) \Downarrow (\texttt{UnitVal} \ / \ log')}{(\texttt{Div}(t_1, t_2) \ / \ env \ / \ log) \Downarrow \texttt{Error}} \text{ DIV-ERR3}$$

$$\frac{(t_1 \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1) \ / \ log_2) \qquad (t_2 \ / \ env \ / \ log_2) \Downarrow (\texttt{FunVal}(f, x, t, env') \ / \ log_3)}{(\texttt{Div}(t_1, t_2) \ / \ env \ / \ log_1) \Downarrow \texttt{Error}} \text{ DIV-ERR4}$$

$$\frac{(t_1 \ / \ env \ / \ log_1) \Downarrow (\texttt{IntVal}(i_1) \ / \ log_2) \qquad (t_2 \ / \ env \ / \ log_2) \Downarrow (\texttt{UnitVal} \ / \ log_3)}{(\texttt{Div}(t_1, t_2) \ / \ env \ / \ log_1) \Downarrow \texttt{Error}} \text{ DIV-ERR5}$$

3.10 Trace Logging

To trace a term t, it is first evaluated to value v. A *tostring* function converts v to its string representation and appended to the log.

$$\frac{(t \ / \ env \ / \ log) \Downarrow (v \ / \ log')}{(\texttt{Trace}(t) \ / \ env \ / \ log) \Downarrow (\texttt{UnitVal} \ / \ tostring(v) :: log')} \text{ TRACE}$$

4 Compilation

Each term of the functional language is compiled to a list of stack language commands. The trick to generating correct stack commands for a particular term is to ensure that after executing these stack commands, the expected value of the term is on top of the stack.

4.1 Command Generation

For simple terms that do not recursively contain sub-terms, they can be directly compiled to singleton commands that put the correct values onto the stack.

Original Term	Generated Commands
`Name "x"`	`[ Push (N "x") ]`
`Unit`	`[ Push U ]`
`Int 7`	`[ Push (I 7) ]`

For terms that contain sub-terms, the sub-terms must be recursively compiled. The following example for addition attempts to demonstrate this.

Original Term	Generated Commands
`Int 1`	`[ Push (I 1) ]`
`Int 2`	`[ Push (I 2) ]`

`Add (Int 1, Int 2)`	`[ Push (I 1) ] @`
	`[ Push (I 2) ] @`
	`[ Add ]`

`Int 3`	`[ Push (I 3) ]`

`Add (Add (Int 1, Int 2), Int 3)`	`[ Push (I 1) ] @`
	`[ Push (I 2) ] @`
	`[ Add ]`
	`[ Push (I 3) ] @`
	`[ Add ]`

1. To compile the term `Add (Add (Int 1, Int 2), Int 3)`, its sub-terms `Add (Int 1, Int 2)` and `Int 3` must be compiled first.
2. To compile the term `Add (Int 1, Int 2)`, its sub-terms `Int 1` and `Int 2` must be compiled first.
3. `Int 1` compiles to `[ Push (I 1) ]` which we shall refer to as c_1.
4. `Int 2` compiles to `[ Push (I 2) ]` which we shall refer to as c_2.
5. Now we can backtrack to step 2. We know that executing c_1 and c_2 in sequence will push `I 1` and `I 2` onto the stack, so performing `Add` immediately after will push the correct sum onto the stack. So `Add (Int 1, Int 2)` compiles to `[ Push (I 1)] @ [ Push (I 2)] @ [ Add ]` which we shall denote as c_3.
6. `Int 3` compiles to `[ Push (I 3) ]` which we shall refer to as c_4.
7. Now we can backtrack to step 1. We know that executing c_3 and c_4 will push the correct values onto the first and second positions of the stack, so performing `Add` immediately after will push the correct sum onto the stack. So `Add (Add (Int 1, Int 2), Int 3)` compiles to `[ Push (I 1)] @ [ Push (I 2)] @ [ Add ] @ [ Push (I 3)] @ [ Add ]`.

4.2 Command Execution

Let's execute our generated stack commands to see it in action.

Commands	Stack

`Push (I 1)`
`Push (I 2)`
`Add`
`Push (I 3)`
`Add`
	-

`Push (I 2)`
`Add`
`Push (I 3)`
`Add`
	`IVal 1`

`Add`
`Push (I 3)`
`Add`
	`IVal 2`
	`IVal 1`

`Push (I 3)` `Add`	`IVal 3`
`Add`	`IVal 3` `IVal 3`
-	`IVal 6`

After interpreting the generated commands, the value on top of the stack `(IVal 6)` is exactly the same value we would obtain by directly evaluating the term `Add (Add (Int 1, Int 2), Int 3)`.

The general procedure for compiling expressions of the form `Add` ($\texttt{t}_1$, $\texttt{t}_2$) is to recursively compile term $\texttt{t}_1$ into commands c_1, then recursively compile term $\texttt{t}_2$ into commands c_2. Append them together as c_1 `@` c_2 and append a singleton `Add` command, culminating in c_1 `@` c_2 `@ [ Add ]`. This procedure is applicable to all expressions in general.

5 Examples

The zip file containing examples for Part 2 also contain the source files written in the functional language they were compiled from.

- `examples`
  - `out`
    Stack language programs generated by compiling the corresponding source program in the `src` folder.
  - `res`
    Expected results for running each program.
  - `src`
    Source programs written in the functional language.