Lab 1
Sax

15-417/817: HOT Compilation
Frank Pfenning

Due Thu Jan 23 (tests), Thu Jan 30 (compilers)
150 points

In this first lab we study our intermediate language Sax with a linear type system. We further restrict ourselves to the first-order fragment where all types are positive, augmented with metavariables that internalize the typing judgment. Extensions will continue to be the core of our compiler throughout the semester.

1 Submissions

Your submissions should be handed in directly to Gradescope from Github or Bitbucket. You may hand in as often as you like.

1.1 Test Cases (30 points)

Your handin should have a directory `tests/` that contains 10 distinct files with a variety of Sax source files `<file>.sax`. Your files should have a mix of negative tests (which are required to fail) and positive tests. Among the positive tests should be procedures with parameters and those with only a destination. Parameterless procedures are executed, each of the resulting values will be printed to a file `<file>.sax.val`. You may validate your test files using the reference implementation of Sax available at `~/bin/sax` and `~/bin/sax-test`. Some of your test files should implement some interesting algorithms. Efficiency, whether of type-checking or execution, is not yet an issue so the test cases should not aim to excessively stress the implementation.

All test cases should lex and parse correctly, and pass static checks that cannot be negated.

1.2 Compiler (120 points)

Your handin should contain a `Makefile` at the top level that compiles your sources to create the executable `./sax`. This executable should take a single `<file>.sax` as an argument and write a file `<file>.sax.val` (which may be empty if there are no paramaterless procedures in `<file>.sax`). These are so-called *large values* rather than the contents of memory at the end of execution, which we don't consider directly observable.

There is starter code available in Standard ML, OCaml, and Rust. You may use another implementation language, but you should contact the course staff to make sure it is available in the correct configuration in the autograder on Gradescope.

2 Grammars

2.1 Lexical Analysis

```
// ... \n or // ... <eof> for single-line comment
/* ... */ for multi-line (nested) comment

<whitespace> ::= [ \t\r\n]

<idstart> ::= [a-zA-Z_]
<idchar> ::= [a-zA-Z_0-9$]
<id> ::= <idstart> <idchar>*
<label> ::= ' <idchar>+

<keywords> ::= 'type' | 'proc' | 'fail'
             | 'read' | 'write' | 'cut' | 'id' | 'call'
             | 'value'
```

Keywords cannot be used as identifiers `<id>`.

2.2 Sax Grammar (files `*.sax`)

```
<prog> ::= <defn>*

<defn> ::= 'type' <id> '=' <tp>
         |  'proc' <id> <parm> <parm>* '=' <cmd>
         |  'fail' <defn>

<cmd> ::= 'read' <id> <pat> <cmd>
        |  'read' <id> '{' <branch>+ '}'
        |  'write' <id> <pat>
        |  'cut' <id> ':' <tp> <cmd> <cmd>
        |  'id' <id> <id>
        |  'call' <id> <id> <id>*
        |  '{' <cmd> '}'

<branch> ::= '|' <pat> '=>' <cmd>

<pat> ::= <label> '(' <id> ')'
        |  '(' <id> ',' <id> ')'
        |  '(' ')'

<parm> ::= '(' <id> ':' <tp> ')'

<tp> ::= '+' '{' <alts> '}'
       |  <tp> '*' <tp>
       | '1'
       | <id>
       | '(' <tp> ')'
```

```
<alts> ::= <alt>
         |  <alt> ',' <alts>

<alt> ::= <label> ':' <tp>
```

`'*'` is right associative, so `A * B * C == A * (B * C)`

2.3 Statics

Note that all types and procedures may be mutually recursive, respectively. In addition to linear typing as detailed in the lecture notes, we have some additional static requirements. The resulting errors are classified into *recoverable* and *fatal*. Definition with recoverable static errors can be preceded by **`fail`** so that the negated definitions succeeds. Fatal errors will cause an error in processing a programs, whether they are preceded by **`fail`** or not. Valid test files should not have fatal errors.

Static requirements with fatal errors

1. Adherence to lexer and grammar specifications
2. Sums must be nonempty (enforced by the grammar)
3. Branches must be nonempty empty (enforced by the grammar)
4. Type names may be defined at most once
5. Procedures may be defined at most once
6. **`fail`** `<defn>` succeeds if `<defn>` fails a recoverable static requirement, assuming all other **`fail`** definitions do not contribute to the program. Because of this latter proviso, there may be definitions `<defn>` that succeed such that **`fail`** `<defn>` also succeeds.
7. **`fail`** `<defn>` fails if `<defn>` succeeds.
8. **`fail`** cannot be nested

Static requirements with recoverable errors We classify variables into *sources* (those in the antecedent that must be read) and *destinations* (those in the succedent that must be written to).

1. Type names that are used must be defined
2. Procedures that are called must be defined
3. Sums may not contain any duplicate labels
4. Type definitions must be contractive, that is, their right-hand side cannot be a type name
5. Destinations and sources in procedure definitions must be pairwise distinct
6. Sources can shadow each other, but must always be distinct from the destination. This is determined lexically, rather then based on which sources may occur in a command. In other words, it should be considered with respect to Γ (sources) and d (destination) in the $\Gamma \vdash P :: (d : A) \,/\, \Xi$ judgment.

2.4 Value Grammar (files `*.sax.val`)

```
<valenv> ::= <valdefn>*

<valdefn> ::= 'value' <id> '=' <value>

<value> ::= '(' <value> ',' <value> ')'
          |  '(' ')'
          |  <label> <value>
```

The value file `*.sax.val` may not have any missing or extraneous definitions, compared to the parameterless procedures in `*.sax`

2.5 Typing

Typing rules can be found in Section 9.1 of the notes for Lecture 1 as augmented with subtyping as in Section 5 of notes for Lecture 2. In these rules, the order of antecedents is seen as irrelevant, and the comma operator conjoins contexts with disjoints sets of variables. Some shadowing is allowed (see subsection 2.3) which could be implemented via silent renaming of bound variables, or via keeping the context ordered in your implementation.

In addition the restrictions from subsection 2.3 are to be respected.

Section 3 of Lecture 2 provides a definition of so-called *additive algorithmic typing*. This represents one possible way type checking might be implemented, but it is in no way required. The typing rules themselves provide the specification.

2.6 Subtyping

The coinductive rules for subtyping $A \leq B$ are given in Section 5 of Lecture 2.

2.7 Dynamics

The multiset rewriting rules for the dynamics can be found in Section 9.2 of Lecture 1. Again this is a specification which can be implemented any number of ways. In particular, we recommend a sequential implementation to keep matters simple. The implementation should execute parameterless procedures in the input file `<file>.sax` and write the corresponding values to the output file `<file>.sax.val`.