CS235 Languages and Automata **Handout # 13**
Prof. Lyn Turbak **December 12, 2007**
Wellesley College

Problem Set 6 (Optional)
Due: 5pm on Thursday, December 20

All problems on this problem are completely **optional**. Any points you score on submitted problems can only help your grade, but not doing any problems cannot hurt your grade.

You can use these problems to study the material from the final segment of the course for the final exam. If you prefer to see the solutions to certain problems without submitting them, that's OK; just let me know, and I will put them in your `ps6` drop folder (as soon as they're ready, that is.)

Problems that are particularly worthwhile to study for the final are: Problem 2 (Closure Properties), Problem 4 (Decidable Languages), Problem 5 (Undecidable Languages), and Problem 6 (Parsing). In terms of studying for the exam, you should focus on the non-programming parts of Problem 6 and only do the programming parts later if/when you have time.

Notes:

- There are actually more parts to the parsing problem (Problem 6) than I have written up, but they are less important in terms of exam studying than the parts I have written up. So I'm posting the problem now rather than holding onto to it. I may flesh out the remaining parts later if I have time.
- Also in the "if I have time" category: I may post an additional optional problem on an ML-Yacc parser for the Kitty language along with some ML-Yacc notes. This is *not* material that is tested on the final exam.

Reading: Sipser, Chapters 3, 4, 5.1, 5.3; Appel Chapter 3; Stoughton Chapter 5.

Submission: Submit a hardcopy submission packet of all problems that you decide to submit.

If you do any of the programming problems, you should also submit a softcopy (consisting of your final `ps6` directory) to the drop directory `~cs235/drop/ps6/`*username*, where *username* is your username. To do this, execute the following commands in Linux:

```
cd /students/username/cs235
cp -R ps6 ~cs235/drop/ps6/username/
```

Problem 1 [45]: Turing Machines

a [25] Using the graphical transition notation presented in Lecture #36, give the definition of a Turing Machine that decides the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in Nat\}$. You need not show an explicit reject state and may assume that any transition not listed goes to the reject state.

b [20] Give high-level English description of Turing machines that decide the following languages:

- $L_1 = \{w \mid w \in \{\mathtt{a},\mathtt{b},\mathtt{c},\mathtt{d}\}^*$ and w has an equal number of as and bs and w has an equal number of cs and ds$\}$
- $L_2 = \{\mathtt{a}^n\mathtt{b}^{2n}\mathtt{a}^n \mid n \in Nat\}$
- $L_3 = \{w \mid w \in \{\mathtt{a}\}^*$ and $|w| = n^2$ for some $n \in Nat.\}$

Problem 2 [10]: Closure Properties

a [5] Show that **Dec** (the set of recursive = Turing-decidable languages) is closed under intersection.

b [5] Show that **RE** (the set of recursively enumerable = Turing-accceptable languages) is closed under intersection.

Problem 3 [15]: Diagonalization

Recall the following languages from lecture:

$$\begin{aligned} HALT_{TM} &= \{\langle\langle M\rangle, w\rangle \mid M \text{ halts on } w\} \\ ACCEPT_{TM} &= \{\langle\langle M\rangle, w\rangle \mid M \text{ accepts } w\} \end{aligned}$$

In lecture, we used diagonalization to prove that $HALT_{TM}$ is undecidable, and then used reduction of $HALT_{TM}$ to $ACCEPT_{TM}$ to prove that $ACCEPT_{TM}$ is undecidable.

In this problem, use diagonalization to directly prove that $ACCEPT_{TM}$ is undecidable.

Problem 4 [75]: Decidable Languages

Show that the following languages are decidable by describing high-level algorithms for deciding the languages. Recall that if X is a specification of a language (regular expression, finite automata, context-free grammar, etc.), then $L(X)$ is the language described by that specification.

a $BALANCED\_PALINDROME = \{w \mid w \in \{a,b\}*,\ w = w^R,$ and the number of `a`s in w = the number of `b`s in $w\}$

b $ACCEPT\_AB_{Reg} = \{\langle Reg\rangle \mid Reg$ is a regular expression and `ab` $\in L(Reg)\}$

c $ACCEPT\_AB_{NPDA} = \{\langle NPDA\rangle \mid NPDA$ is a nondeterministic pushdown automaton and `ab` $\in L(NPDA)\}$

d $EITHER_{CFG} = \{\langle\langle CFG_1\rangle, \langle CFG_2\rangle, w\rangle \mid CFG_1$ and CFG_2 are context-free grammars and $w \in L(CFG_1)$ or $w \in L(CFG_2)\}$

e $SUBSET_{DFA} = \{\langle\langle DFA_1\rangle, \langle DFA_2\rangle\rangle \mid DFA_1$ and DFA_2 are deterministic finite automata and $L(DFA_1) \subseteq L(DFA_2)\}$

f $INFINITE_{DFA} = \{\langle DFA\rangle \mid DFA$ is a deterministic finite automaton and $L(DFA)$ is infinite$\}$

g $PALINDROMIC_{DFA} = \{\langle DFA\rangle \mid DFA$ is a deterministic finite automaton and $L(DFA) = (L(DFA))^R\}$

h $EMPTY_{CFG} = \{\langle CFG\rangle \mid CFG$ is a context-free grammar $L(CFG) = \emptyset\}$

i $SOME\_BS_{CFG} = \{\langle CFG\rangle \mid CFG$ is a context-free grammar over $\{a,b\}*$ and $L($`b*`$) \cap L(CFG) \neq \emptyset\}$

j $ALL\_BS_{CFG} = \{\langle CFG\rangle \mid CFG$ is a context-free grammar over $\{a,b\}*$ and $L($`b*`$) \subseteq L(CFG)\}$

k $EMPTY\_TAKES\_235_{TM} = \{\langle M\rangle \mid M$ is a Turing Machine and M takes more than 235 steps on input `%`$\}$

l $SOME\_TAKE\_235_{TM} = \{\langle M\rangle \mid M$ is a Turing Machine and M takes more than 235 steps on some input$\}$

m $ALL\_TAKE\_235_{TM} = \{\langle M\rangle \mid M$ is a Turing Machine and M takes more than 235 steps on all inputs$\}$

Problem 5 [75]: Undecidable Languages

a Use reducibility to show that the following languages are undecidable. Although many of these results are a consequence of Rice's theorem, you should not invoke Rice's theorem in any of the parts. In your reducibility arguments You may assume that the following languages discussed in lecture are undecidable:

$HALT_{TM} = \{\langle\langle M\rangle, w\rangle \mid M \text{ is a Turing Machine and } M \text{ halts on } w\}$

$ACCEPT_{TM} = \{\langle\langle M\rangle, w\rangle \mid M \text{ is a Turing Machine and } w \in L(M)\}$

$EMPTY_{TM} = \{\langle M\rangle \mid M \text{ is a Turing Machine and } L(M) = \emptyset\}$

$EQ_{TM} = \{\langle\langle M_1\rangle, \langle M_2\rangle\rangle \mid M_1 \text{ and } M_2 \text{ are Turing Machines and } L(M_1) = L(M_2)\}$

i. $CFL_{TM} = \{\langle M\rangle \mid M$ is a Turing Machine and $L(M)$ is a context-free language$\}$

ii. $DEC_{TM} = \{\langle M\rangle \mid M$ is a Turing Machine and $L(M)$ is a recursive (Turing-decidable) language$\}$

iii. $INCLUDES\_A_{TM} = \{\langle M\rangle \mid M$ is a Turing Machine over $\{\texttt{a}, \texttt{b}\}^*$ and $\texttt{a} \in L(M)\}$

iv. $ALL_{TM} = \{\langle M\rangle \mid M$ is a Turing Machine over $\{\texttt{a}, \texttt{b}\}^*$ and $L(M) = L(\texttt{(a+b)*})\}$

v. $INFINITE_{TM} = \{\langle M\rangle \mid M$ is a Turing Machine and $L(M)$ is infinite.$\}$

vi. $SOME\_EVEN_{TM} = \{\langle M\rangle \mid M$ is a Turing Machine over $\{\texttt{a}, \texttt{b}\}^*$ and $L(\texttt{((a+b)(a+b))*}) \cap L(M) \neq \emptyset\}$

vii. $ALL\_EVEN_{TM} = \{\langle M\rangle \mid M$ is a Turing Machine over $\{\texttt{a}, \texttt{b}\}^*$ and $L(\texttt{((a+b)(a+b))*}) \subseteq L(M)\}$

viii. $SUBSET_{TM} = \{\langle\langle M_1\rangle, \langle M_2\rangle\rangle \mid M_1$ and M_2 are Turing Machines and and $L(M_1) \subseteq L(M_2)\}$

b For each of the following languages mentioned above, indicate whether it is semi-decidable+ (i.e., in **RE** − **Dec**), semi-decidable− (i.e., in **co-RE** − **Dec**), or not-even-semidecidable (i.e., in **Lan** − (**RE** ∪ **co-RE**)).

i. $INCLUDES\_A_{TM}$

ii. ALL_{TM}

iii. $INFINITE_{TM}$

iv. $SOME\_EVEN_{TM}$

v. ALL_EVEN_{TM}

vi. $SUBSET_{TM}$

Problem 6 [65]: Parsing Types

This problem involves parsing types in a type language that is simple subset of Standard ML types. Here is a context-free grammar for these types:

```
T → BASE(t) | T list | T * T | T -> T | (T)
```

In this grammar,

- `BASE(t)` stands for a token that is one of the **base types** of the type system:

  ```
  unit, bool, int, real, char, string
  ```

- the type T `list` stands for a type whose values are lists with elements of type T.
- the type T_1 `*` T_2 stands for a type whose values are pairs whose first component has type T_1 and whose second component has type T_2. (Unlike SML, this simple language does *not* have a notion of tuple types with more than two components.)
- the type T_1 `->` T_2 stands for a type whose values are functions with argument type T_1 and result type T_2.

Although not explicit in the grammar, it is intended that:

- `list` has a higher precdence than `*` and `*` has a higher precedence than `->`. For example,

  ```
  int * char list -> bool list
  ```

  is parsed as if it were written

  ```
  (int * (char list)) -> (bool list)
  ```

- `*` is left-associative and `->` is right-associative. For example,

  ```
  int * real * char -> string -> bool
  ```

  is parsed as if it were written

  ```
  ((int * real) * char) -> (string -> bool)
  ```

Explicit parentheses can be used to override the default precedences and associativities. E.g.,

```
int * real * char list -> string list -> bool list
```

would normally be parsed as

```
((int * real) * (char list)) -> ((string list) -> (bool list))
```

but we could explicilty parenthesize it instead as

```
((int * ((((real * char) list) -> string) list)) -> bool) list
```

We will call the above grammar the **infix grammar** for types because the binary operators for products (`*`) and arrows (`->`) are placed between the two operaands.

In this problem, we explore analyzing and constructing parsers for several variants of this type language. All files mentioned can be found in the `ps6` directory.

a [10]: Ambiguity The above grammar is ambiguous. Demonstrate this by constructing all the possible parse for the type `int * char -> bool list`.

b [20]: A Prefix Syntax for Types One way to remove ambiguity is to change the syntax for types. In this part we consider a **prefix syntax** for our types, in which all type operators come before their operands:

```
E → BASE(t) | list E | * E E | -> E E
```

i. **[5]** Explain why the prefix grammar LL(1) – i.e., it can be used for predictive parsing by a recursive descent parser using only 1 token of lookahead.

ii. **[15]** In the file `TypesParserPrefix.sml`, flesh out the definition of the function `eatTyp : unit -> AST.typ` so that it performs recursive descent parsing on types written in prefix syntax.

Your function should consume tokens from the implicit token stream in an imperative way using the `nextToken()` function, which removes the first token from the token stream and returns it. Tokens are elements of the `token` data type from the `Token` module, which is presented in Fig. 1. The scanner for these tokens and all other supporting code has already been written for you; all you need to write is the `eatTyp` function.

Your function should return a parse tree expressed using the `AST.typ` data type from the `AST` module (Fig. 2). For example, parsing the string

```
"-> * int char list bool"
```

should yield the AST

```
Arrow (Prod (Base Int,Base Char),List (Base Bool))
```

Notes:

- Load all the parser files for this problem using

  ```
  use "load-types-parser.sml";
  ```

- You indicate an error in SML by raising an exception. The simplest exception to raise is a **failure exception**, which can be accomplished as follows:

  ```
  raise Fail error-message-string-goes-here
  ```

- You can test your prefix parser with the functions illustrated in the following example:

  ```
  TypesParserPrefix.stringToTyp "-> * int char list bool";
  val it = Arrow (Prod (Base Int,Base Char),List (Base Bool)) : AST.typ

  - TypesParserPrefix.stringToTypString "-> * int char list bool";
  val it = "((int * char) -> (bool list))" : string
  ```

- If the full types are not displaying, don't forget to set `Control.Print.printDepth` to a large value. E.g.

  ```
  Control.Print.printDepth := 1000;
  ```

```
structure Token =

struct

  datatype baseTy = Unit | Bool | Int | Real | Char | String

  datatype token = EOF
                 |   BASE of baseTy
                 |   PROD
                 |   ARROW
                 |   LIST
                 |   LPAREN
                 |   RPAREN

  fun eof() = EOF

  fun isEof(EOF) = true
    | isEof(_) = false

  fun baseTyToString(Unit) = "unit"
    | baseTyToString(Bool) = "bool"
    | baseTyToString(Int) = "int"
    | baseTyToString(Real) = "real"
    | baseTyToString(Char) = "char"
    | baseTyToString(String) = "string"

  fun toString(EOF) = "[EOF]"
    | toString(BASE(b)) = "[" ^ (baseTyToString(b)) ^ "]"
    | toString(PROD) = "*"
    | toString(ARROW) = "[->]"
    | toString(LIST) = "[list]"
    | toString(LPAREN) = "[(]"
    | toString(RPAREN) = "[)]"

end
```

Figure 1: The `Token` module for the type language.

```
structure AST = struct

  datatype typ = Base of Token.baseTy
    | List of typ
    | Prod of typ * typ
    | Arrow of typ * typ

  (* Construct a fully-parenthesized infix type string *)
  fun typToString (Base(b)) = Token.baseTyToString b
    | typToString (List(t)) = "(" ^ (typToString t) ^ " list)"
    | typToString (Prod(t1,t2)) = "(" ^ (typToString t1) ^ " * " ^ (typToString t2) ^ ")"
    | typToString (Arrow(t1,t2)) = "(" ^ (typToString t1) ^ " -> " ^ (typToString t2) ^ ")"

  (* Construct a string for a list of types *)
  fun typListToString t = "[" ^ (typEltsToString t) ^ "]"

  and typEltsToString [] = ""
    | typEltsToString [t] = typToString t
    | typEltsToString (t::ts) = (typToString t) ^ "," ^ (typEltsToString ts)

end
```

Figure 2: The `AST` module for the type language.

c [35]: A Postfix Syntax for Types Another way to remove ambiguity is to change the syntax for types to a **prefix syntax**, in which all type operators come after their operands:

```
O → BASE(t) | O list | O O * | O O ->
```

i. **[5]** Explain why this postfix grammar is not predictive – i.e., it is not LL(k) for any k.

ii. **[15]** The postfix grammar is LR(0) – i.e., it can be implemented by a shift-reduce parser in which the decision to shift or reduce depends only on the parser state, not on any tokens of lookahead.

Give an informal explanation of how a shift-reduce parser can process a stream of tokens for the postfix grammar. (You do *not* have to construct parser states or parser tables for this part!) In which situations should the parser shift tokens to the stack? In which situations should the parser reduce elements on the stack? As part of your explanation, explain the steps of parsing the following token stream:

```
int char * bool list -> EOF
```

iii. **[15]** In the file `TypesParserPostfix.sml`, flesh out the definition of the function `processToken : (Token.token * AST.typ list) -> AST.typ list` so that it performs shift-reduce parsing on types written in postfix syntax.

As in `TypesParserPrefix`, you should consume tokens an implicit token stream in an imperative fashion via the `nextToken()` function.

In this file, a stack is represented as a list of type ASTs, with the top of the stack at the front of the list. In a more general shift-reduce parser, it would be necessary for the stack to hold both ASTs and tokens, but this parser is so simple that it combines each shift with the subsequent reduce into one step and can get by with just having a stack of type ASTs.

The `processToken` function takes the first token from the token stream and the current stack of type ASTs and returns the stack of type ASTs that results from processing the token (i.e., effectively pushing it on the stack and then immediately performing the appropriate reduction).

You can test your postfix parser with the functions illustrated in the following example:

```
- TypesParserPostfix.stringToTyp "int char * bool list ->";
val it = Arrow (Prod (Base Int,Base Char),List (Base Bool)) : AST.typ

- TypesParserPostfix.stringToTypString "int char * bool list ->";
val it = "((int * char) -> (bool list))" : string
```