Programming Language Concepts
Assignment 2

Arthur Nunes-Harwitt

Each function should have a comment before it indicating the types.
Following each function there should be commented out expressions to test the function.

Part A

Recall the definitions of the first and follow sets, which are defined with respect to a grammar $G = \langle \mathcal{V}, \Sigma, \Pi, S \rangle$.

$$\mathsf{FIRST}_G(\alpha) = \{\sigma \in \Sigma \mid \alpha \Rightarrow^* \sigma\beta\} \cup (\text{if } \alpha \Rightarrow^* \epsilon \text{ then } \{\epsilon\} \text{ else } \{\})$$

$$\mathsf{FOLLOW}_G(A) = \{\sigma \in \Sigma \mid S \Rightarrow^+ \alpha A \sigma\beta\} \cup (\text{if } S \Rightarrow^+ \alpha A \text{ then } \{\epsilon\} \text{ else } \{\})$$

Note that the intuition behind the definition of $\mathsf{FIRST}_G(\alpha)$ is that it is the set of all terminals (or epsilon) that α can start with. The intuition behind the definition of $\mathsf{FOLLOW}_G(A)$ is that it is the set of all terminals that can come after the variable A.

In this section, we will assume that the grammar has already been augmented with the new start rule S ::= S_O eof, where S_O is the original start variable. We represent a grammar in Scheme as a list of rules. A rule A ::= $X_1 \cdots X_n$ is expressed as the list `(A (`$X_1 \cdots X_n$`))`, where if X_i is a variable it is expressed as the Scheme symbol `Xi` and if X_i is a terminal it is expressed as the quoted symbol `'Xi`. For example, the rule S ::= E eof is expressed as `(S (E 'eof))`. Note that the empty string ϵ is expressed as the empty list.

1. (10 points) We will represent the sets as Scheme lists. Write a function `union` that has inputs `set1` and `set2`, where `set1` and `set2` are lists. It should return a list that represents the union of the two sets. In particular, there should be no repeats in the list returned. You will find the Scheme function `member` useful.
   Example:
   `(union '(1 2 3 4) '(5 2 6 3))` $\to^*$ `(1 4 5 2 6 3)`

2. (10 points) Write a function `first-alpha` that has inputs `grammar` and `alpha`, where `grammar` is a grammar and `alpha` is a list of variables or terminals such as the right-hand-side of a rule. It should return a list of (unquoted) terminals which may also include the empty list that represents FIRST_G(`alpha`). To implement `first-alpha` you will write two mutually recursive helper functions: `first3` and `first-var3`. The function `first3` sweeps across α, and `first-var3` expands variables. The function `first3` has inputs `grammar`, `alpha`, and `seen`, where `seen` is a list of rules that have been used already. The function `first-var3` has inputs `grammar`, `rules`, and `seen`, where `rules` is a subset of the rules in the grammar all with the same left-hand-side. The following facts will help you write these functions.

   (a) `(first3 grammar '() seen)` $= \{\epsilon\}$

(b) `(first3 grammar '(`σ X_2 $\cdots$ X_n`) seen)` $= \{\sigma\}$

(c) `(first-var3 grammar` $\hat{\Pi}$ `seen)`$-\{\epsilon\} \subseteq$ `(first3 grammar '(`A X_2 $\cdots$ X_n`) seen)` where $\hat{\Pi}$ is the set of rules that have A on the left-hand-side

(d) `(first3 grammar '(`X_2 $\cdots$ X_n`) seen)` $\subseteq$ `(first3 grammar '(`A X_2 $\cdots$ X_n`) seen)` if $\epsilon \in$ `(first-var3 grammar` $\hat{\Pi}$ `seen)` and $\hat{\Pi}$ is the set of rules that have A on the left-hand-side

(e) `(first-var3 grammar '() seen)` $= \{\}$

(f) `(first-var3 grammar '(`R_1 R_2 $\cdots$ R_n`) seen)` $=$ `(first-var3 grammar '(`R_2 $\cdots$ R_n`) seen)` if $R_1 \in$ `seen`

(g) `(first-var3 grammar '(`R_1 R_2 $\cdots$ R_n`) seen)` $=$ `(first3 grammar` α `seen)` $\cup$ `(first-var3 grammar '(`R_2 $\cdots$ R_n`) (cons` R_1 `seen))` if $R_1 = (A ::= \alpha)$

The two functions `first3` and `first-var3` play different roles in computing $\mathsf{FIRST}_G(\cdot)$. The function `first3` sweeps across the variables. The function `first-var3` is about going deeper, expanding a variable, looking at all the rules for a given variable, and seeing what is the first of the right-hand-sides of those rules. You may find the Scheme function `filter` useful for finding all the rules for a given variable.

The facts (a) and (b) are base cases for `first3`. The fact (e) is a base case for `first-var3`. When there is a subset relation that means that the subset is part of the answer. Piecing together the subsets gives the entire result (assuming the conditions are the same). You will use `union` to put the subsets together. For example, fact (d) is a recursive call. It says that if the beginning variable of the sequence can vanish, then the first set contains the first set of the sequence without the beginning variable. (Observe that this recursive fact yields a terminating computation. Eventually, we will run out of variables.) Fact (c) shows what else is in the first set. Although there might be rules that make the variable vanish, there might be other rules that make the variable expand into a sequence that starts with a terminal. The fact (f) is about avoiding using the same rule to expand the same variable over and over again; it prevents infinite loops. The fact (g) concerns iterating through the relevant rules and finding the first of each of those rule's right-hand-side.

3. (10 points) Write a function `follow-var` that has inputs `grammar` and `var`, where `grammar` is a grammar and `var` is a symbol that represents a variable. It should return a list of (unquoted) terminals that represents $\mathsf{FOLLOW}_G(\mathtt{var})$. To implement `follow-var` you will write a recursive helper function: `follow-rules4`. The function `follow-rules4` has inputs `grammar`, `var`, `rules`, and `seen`, where `rules` is a subset of `grammar`. The following facts will help you write this function.

   (a) `(follow-rules4 grammar var '() seen)` $= \{\}$

   (b) `(follow-rules4 grammar var '(`R_1 R_2 $\cdots$ R_n`) seen)` $=$ `(follow-rules4 grammar var '(`R_2 $\cdots$ R_n`) seen)` if $R_1 \in$ `seen`

   (c) `(follow-rules4 grammar var '(`R_2 $\cdots$ R_n`) seen)` $\subseteq$ `(follow-rules4 grammar` A `'(`R_1 R_2 $\cdots$ R_n`) seen)`

   (d) $\mathsf{FIRST}_G(\beta)-\{\epsilon\} \subseteq$ `(follow-rules4 grammar` A `'(`R_1 R_2 $\cdots$ R_n`) seen)` if $R_1 = (B ::= \alpha\, A\, \beta)$

   (e) `(follow-rules4 grammar` B `grammar (cons` R_1 `seen))` $\subseteq$ `(follow-rules4 grammar` A `'(`R_1 R_2 $\cdots$ R_n`) seen)` if $R_1 = (B ::= \alpha\ A\ \beta)$ and $\epsilon \in \mathsf{FIRST}_G(\beta)$

   When implementing $\mathsf{FOLLOW}_G(\cdot)$, we need only one essential helper function: `follow-rules4`. The original definition of $\mathsf{FOLLOW}_G(\cdot)$ involves sequences that we can derive from the start variable. The function `follow-rules4` inverts this, and assumes that all variables can be derived from the start variable. It then looks through all the grammar rules to see if the variable of interest is in the right-hand-side of a rule. (See facts (a) and (c).) When it finds the variable of interest in a right-hand-side, the follow set includes everything in the first set just after the variable. (See fact (d).) If what's after the variable can vanish, then what follows the variable of interest follows the variable that is the left-hand-side of the rule in which the variable was found. (See fact (e).) Again, fact (b) ensures that the computation does not go into an infinite loop by looking at the same rule over and over again. Note that since the grammar is assumed to have been augmented, the empty string can never be in the follow set.

Undergraduate Extra Credit Part A

1. (5 points) Write a function `predict` that has inputs `grammar` and `rule`, where `grammar` is a grammar and `rule` is a rule from the grammar. It should return a list of unquoted terminals that represents the predict set of the rule. The augmented start rule is not in the domain.
2. (5 points) Write a function `make-oracle` that has input `grammar`, where `grammar` is a grammar. It should return a function that takes a variable and a terminal and returns a right-hand-side of a rule or generates an error. Note that the augmented start symbol is not among the variables that can be an argument.
3. (5 points) Write a function `make-recognizer` that has input `grammar`, where `grammar` is a grammar. It should return a function that takes a list of (unquoted) terminals and returns a Boolean indicating whether or not the sequence of terminals is in the language. (It will be useful to write a recursive helper function that has additional arguments such as a stack.)

Parts B & C

Consider the following lexical and syntactic specifications, and its DrRacket implementation.

Lexical Specification

lower	=	a | $\cdots$ | z
upper	=	A | $\cdots$ | Z
letter	=	lower | upper
digit	=	0 | $\cdots$ | 9
ident	=	letter$^+$
number	=	digit$^+$

Syntactic Specification

$\langle expr \rangle$	$::=$	let $\langle letdefs \rangle$ in $\langle expr \rangle$	[let $2 $4]
$\langle expr \rangle$	$::=$	$\langle mathexpr \rangle$	[$1]
$\langle letdefs \rangle$	$::=$	$\langle letdef \rangle$	[list $1]
$\langle letdefs \rangle$	$::=$	$\langle letdef \rangle$, $\langle letdefs \rangle$	[cons $1 $3]
$\langle letdef \rangle$	$::=$	ident $=$ $\langle expr \rangle$	[list $1 $3]
$\langle mathexpr \rangle$	$::=$	$\langle mathexpr \rangle + \langle term \rangle$	[sum $1 $3]
$\langle mathexpr \rangle$	$::=$	$\langle mathexpr \rangle - \langle term \rangle$	[diff $1 $3]
$\langle mathexpr \rangle$	$::=$	$\langle term \rangle$	[$1]
$\langle term \rangle$	$::=$	$\langle term \rangle * \langle factor \rangle$	[prod $1 $3]
$\langle term \rangle$	$::=$	$\langle term \rangle$ / $\langle factor \rangle$	[quotient $1 $3]
$\langle term \rangle$	$::=$	$\langle factor \rangle$	[$1]
$\langle factor \rangle$	$::=$	ident	[$1]
$\langle factor \rangle$	$::=$	number	[$1]
$\langle factor \rangle$	$::=$	$- \langle factor \rangle$	[negate $2]
$\langle factor \rangle$	$::=$	($\langle expr \rangle$)	[$2]

DrRacket Implementation

Make sure to use Pretty Big in the Legacy Language section.

```
;; Import the parser and lexer generators.

(require (lib "yacc.ss" "parser-tools")
         (lib "lex.ss" "parser-tools")
         (prefix : (lib "lex-sre.ss" "parser-tools")))

(require (lib "pretty.ss"))

(define-tokens value-tokens (NUM ID))

(define-empty-tokens op-tokens
  (OP
   CP
   COMMA
   EQ1
   LET
   IN
   +
   -
   *
   /
   EOF))

(define-lex-abbrevs
 (lower-letter (:/ "a" "z"))
 (upper-letter (:/ "A" "Z"))
 (letter (:or lower-letter upper-letter))
 (digit (:/ "0" "9"))
 (ident (:+ letter))
 (number (:+ digit)))

;get-token: inputPort -> token
(define get-token
  (lexer
   ((eof) 'EOF)
   ("let" 'LET)
   ("in" 'IN)
   ("(" 'OP)
   (")" 'CP)
   ("," 'COMMA)
   ("=" 'EQ1)
   ("+" '+)
   ("-" '-)
   ("*" '*)
   ("/" '/)
   (number (token-NUM (string->number lexeme)))
   (ident (token-ID (string->symbol lexeme)))
   (whitespace (get-token input-port))))

;;; data definitions

;; A small language expression (SmallLangExp) is one of the following.
;; a number n
;; an identifier x
;; a sum with parts e1 and e2,
;;   where e1 and e2 are small language expressions
;; a difference with parts e1 and e2,
;;   where e1 and e2 are small language expressions
;; a product with parts e1 and e2,
;;   where e1 and e2 are small language expressions
;; a quotient with parts e1 and e2,
;;   where e1 and e2 are small language expressions
;; a negation with part e,
;;   where e is an small language expression
;; a bindings with parts defs and e,
;;   where defs is a list of identifiers * SmallLangExp
;;   and e is an small language expression

;; functions for associated with each part: predicate, constructor, selectors.

;; Number is a Scheme number

;; Identifier is a Scheme symbol

; make-sum: SmallLangExp * SmallLangExp -> SumExp
(define (make-sum exp1 exp2)
  (list 'sum exp1 exp2))

; (equal? (make-sum 2 3) '(sum 2 3))


; make-diff: SmallLangExp * SmallLangExp -> DiffExp
(define (make-diff exp1 exp2)
  (list 'diff exp1 exp2))

; (equal? (make-diff 2 3) '(diff 2 3))


; make-prod: SmallLangExp * SmallLangExp -> ProdExp
```

```
(define (make-prod exp1 exp2)
  (list 'prod exp1 exp2))

; (equal? (make-prod 2 3) '(prod 2 3))

; make-quo: SmallLangExp * SmallLangExp -> QuoExp
(define (make-quo exp1 exp2)
  (list 'quo exp1 exp2))

; (equal? (make-quo 2 3) '(quo 2 3))

; make-neg: SmallLangExp -> NegExp
(define (make-neg exp)
  (list 'neg exp))

; (equal? (make-neg 2) '(neg 2))

; make-let: Listof(Identifier*SmallLangExp) * SmallLangExp -> BindingExp
; Identifier*SmallLangExp is represented as a two element list
(define (make-let defs exp)
  (list 'with-bindings defs exp))

; (equal? (make-let (list (list 'x 1) (list 'y 2)) 3) '(with-bindings ((x 1) (y 2)) 3))

; parse-small-lang: (() -> token) -> SmallLangExp
(define parse-small-lang
  (parser
   (start exp)
   (end EOF)
   (tokens value-tokens op-tokens)
   (error (lambda (a b c) (error 'parse-small-lang "error occurred, ~v ~v ~v" a b c)))
   (grammar
    (exp ((LET let-defs IN exp) (make-let $2 $4))
         ((math-exp) $1))
    (let-def ((ID EQ1 exp) (list $1 $3)))
    (let-defs ((let-def) (list $1))
              ((let-def COMMA let-defs) (cons $1 $3)))
    (math-exp ((math-exp + term) (make-sum $1 $3))
              ((math-exp - term) (make-diff $1 $3))
              ((term) $1))
    (term ((term * factor) (make-prod $1 $3))
          ((term / factor) (make-quo $1 $3))
          ((factor) $1))
    (factor ((ID) $1)
            ((NUM) $1)
            ((- factor) (make-neg $2))
            ((OP exp CP) $2)))))

; lexer/parser test
(let* ((example "let x = -2 + 3 * 4, y = 0 in -2+5*x+y")
       (i (open-input-string example))) ; convert string to inputPort
  (equal? (parse-small-lang (lambda () (get-token i)))
          '(with-bindings ((x (sum (neg 2) (prod 3 4)))
                           (y 0))
             (sum (sum (neg 2) (prod 5 x)) y))))
```

A Larger Language

Consider the following lexical and syntactic extensions to the previous language. These extensions make the language defined previously look more like Java.

Lexical Specification

lower	=	a | ⋯ | z
upper	=	A | ⋯ | Z
letter	=	lower | upper
digit	=	0 | ⋯ | 9
idfirst	=	letter | _ | $
idrest	=	idfirst | digit
ident	=	idfirst idrest*
digits	=	digit$^+$
number	=	digits(. digits)$^?$((E | e)(+ | −)$^?$digits)$^?$

Syntactic Specification

⟨*program*⟩	::=	⟨*classdecls*⟩⟨*expr*⟩	[program $1 $2]
⟨*classdecls*⟩	::=	ϵ	[null]
⟨*classdecls*⟩	::=	⟨*classdecl*⟩ ⟨*classdecls*⟩	[cons $1 $2]
⟨*classdecl*⟩	::=	class ident extends ident{ ⟨*fielddecls*⟩⟨*methdecls*⟩}	[class $2 $4 $6 $7]
⟨*fielddecls*⟩	::=	ϵ	[null]
⟨*fielddecls*⟩	::=	field ident ⟨*fielddecls*⟩	[cons $2 $3]
⟨*methdecls*⟩	::=	ϵ	[null]
⟨*methdecls*⟩	::=	method ident (⟨*formals*⟩)⟨*expr*⟩ ⟨*methdecls*⟩	[cons (method $2 $4 $6) $7]
⟨*expr*⟩	::=	let ⟨*letdefs*⟩ in ⟨*expr*⟩	[let $2 $4]
⟨*expr*⟩	::=	procedures ⟨*procdefs*⟩ in ⟨*expr*⟩	[procs $2 $4]
⟨*expr*⟩	::=	{ ⟨*exprs*⟩ }	[sequence $2]
⟨*expr*⟩	::=	if ⟨*expr*⟩ then ⟨*expr*⟩ else ⟨*expr*⟩	[if $2 $4 $6]
⟨*expr*⟩	::=	\ ⟨*formals*⟩ \ −> ⟨*expr*⟩	[proc $2 $5]
⟨*expr*⟩	::=	new ident (⟨*actuals*⟩)	[new $2 $4]

$\langle expr \rangle$	::=	super ident ($\langle actuals \rangle$)	[super $2 $4]
$\langle expr \rangle$	::=	ident = $\langle expr \rangle$	[assign $1 $3]
$\langle expr \rangle$	::=	$\langle compexpr \rangle$	[$1]
$\langle letdefs \rangle$	::=	$\langle letdef \rangle$	[list $1]
$\langle letdefs \rangle$	::=	$\langle letdef \rangle$, $\langle letdefs \rangle$	[cons $1 $3]
$\langle letdef \rangle$	::=	ident = $\langle expr \rangle$	[list $1 $3]
$\langle procdefs \rangle$	::=	$\langle procdef \rangle$	[list $1]
$\langle procdefs \rangle$	::=	$\langle procdef \rangle$, $\langle letdefs \rangle$	[cons $1 $3]
$\langle procdef \rangle$	::=	ident ($\langle formals \rangle$) = $\langle expr \rangle$	[list $1 (proc $3 $6)]
$\langle exprs \rangle$	::=	$\langle expr \rangle$	[list $1]
$\langle exprs \rangle$	::=	$\langle expr \rangle$; $\langle exprs \rangle$	[cons $1 $3]
$\langle compexpr \rangle$	::=	$\langle mathexpr \rangle == \langle mathexpr \rangle$	[equal $1 $3]
$\langle compexpr \rangle$	::=	$\langle mathexpr \rangle$	[$1]
$\langle mathexpr \rangle$	::=	$\langle mathexpr \rangle + \langle term \rangle$	[sum $1 $3]
$\langle mathexpr \rangle$	::=	$\langle mathexpr \rangle - \langle term \rangle$	[diff $1 $3]
$\langle mathexpr \rangle$	::=	$\langle term \rangle$	[$1]
$\langle term \rangle$	::=	$\langle term \rangle * \langle factor \rangle$	[prod $1 $3]
$\langle term \rangle$	::=	$\langle term \rangle$ / $\langle factor \rangle$	[quotient $1 $3]
$\langle term \rangle$	::=	$\langle factor \rangle$	[$1]
$\langle factor \rangle$	::=	$\langle simple \rangle$	[$1]
$\langle factor \rangle$	::=	number	[$1]
$\langle factor \rangle$	::=	$- \langle factor \rangle$	[negate $2]
$\langle simple \rangle$	::=	ident	[$1]
$\langle simple \rangle$	::=	$\langle simple \rangle$. ident	[access $1 $3]
$\langle simple \rangle$	::=	$\langle simple \rangle$ ($\langle actuals \rangle$)	[funcall $1 $3]
$\langle simple \rangle$	::=	($\langle expr \rangle$)	[$2]
$\langle actuals \rangle$	::=	ϵ	[null]
$\langle actuals \rangle$	::=	$\langle nonemptyactuals \rangle$	[$1]
$\langle nonemptyactuals \rangle$	::=	$\langle expr \rangle$	[list $1]
$\langle nonemptyactuals \rangle$	::=	$\langle expr \rangle$, $\langle nonemptyactuals \rangle$	[cons $1 $3]
$\langle formals \rangle$	::=	ϵ	[null]
$\langle formals \rangle$	::=	$\langle nonemptyformals \rangle$	[$1]
$\langle nonemptyformals \rangle$	::=	ident	[list $1]
$\langle nonemptyformals \rangle$	::=	ident , $\langle nonemptyformals \rangle$	[cons $1 $3]

Translation Rules

$\overline{X}$ is the translation from lexical mathematical notation to code.

$\overline{\mathbf{a}}$	=	`"a"`
$\overline{\mathbf{abc}}$	=	`"abc"`
$\overline{XY}$	=	`(::` $\overline{X}$ $\overline{Y}$`)`
$\overline{X^*}$	=	`(:*` $\overline{X}$`)`
$\overline{X^+}$	=	`(:+` $\overline{X}$`)`
$\overline{X^?}$	=	`(:?` $\overline{X}$`)`
$\overline{X \mid Y}$	=	`(:or` $\overline{X}$ $\overline{Y}$`)`

$\overline{\overline{P}}$ is the translation from syntactic mathematical notation to code.

$$\overline{\overline{X ::= X_1 \cdots X_n \; [f \; \$1 \; \cdots \; \$n] \cdots}} \quad = \quad \texttt{(X ((X1 $\cdots$ Xn) (make-f \$1 $\cdots$ \$n)) $\cdots$)}$$
$$\overline{\overline{X ::= \epsilon \; [\text{null}]}} \quad = \quad \texttt{(X (() null))}$$

Part B

1. (10 points) While the definition of the small language expression structure is complete, the implementation is not. The constructors are provided, but the predicates and selectors are missing. Selectors for the binary arithmetic operators should be polymorphic, as in the derivative example. Write the following predicates and selectors.
   Predicates:

   - `sum?`
   - `difference?`
   - `product?`
   - `quotient?`
   - `negate?`
   - `let?`

   Selectors:

   - `arg1`, `arg2`
   - `neg-exp`
   - `let-defs`, `let-exp`

2. (10 points) First, write a definition for the large language in the comments. You should define the large language expression structure, the large language method structure, the large language class structure, and the large language program structure. Then, implement this definition by writing the following additional predicates, constructors, and selectors. Check your implementation with the example tests.
   Predicates:

   - `program?`
   - `class-decl?`
   - `method?`
   - `new?`
   - `supercall?`
   - `seq?`
   - `procs?`
   - `if?`
   - `assign?`
   - `equality?`
   - `proc?`

- `access?`
- `funcall?`

Constructors:

- `make-program` use the tag `program`
- `make-class` use the tag `class`
- `make-method` use the tag `method`
- `make-new` use the tag `new`
- `make-supercall` use the tag `super`
- `make-seq` use the tag `sequence`
- `make-procs` use the tag `procedures`
- `make-if` use the tag `if`
- `make-assign` use the tag `assign!`
- `make-equal` use the tag `equality?`
- `make-proc` use the tag `proc`
- `make-access` use the tag `send`
- `make-funcall` use the tag `funcall`

Selectors:

- `program-decls`, `program-exp`
- `class-name`, `class-parent`, `class-fields`, `class-methods`
- `method-name`, `method-formals`, `method-exp`
- `new-name`, `new-rands`
- `supercall-name`, `supercall-rands`
- `seq-exps`
- `procs-defs`, `procs-exp`,
- `if-exp1`, `if-exp2`, `if-exp3`
- `assign-var`, `assign-exp`
- `proc-formals`, `proc-exp`
- `access-exp`, `access-message`
- `funcall-rator`, `funcall-rands`

3. (10 points) Extend the lexical analyzer so that it uses the more detailed definitions of identifiers and numbers.

Part C

1. (20 points) Make sure that the lexical analyzer can scan all the new atomic tokens mentioned in the grammar. Note that == and $->$ should each be a single token.
2. (20 points) Extend the parser so that it can parse the Java-like language using all the rules above. Call the parser `parse-lang`. Check your implementation with the example tests.

Example Tests

```
(let* ((example "let x = -(1+1) + 3 * 4, y = 0 in {y = 14; x == y}")
       (i (open-input-string example)))
  (equal? (parse-lang (lambda () (get-token i)))
          '(program
            ()
            (with-bindings
             ((x (sum (neg (sum 1 1)) (prod 3 4))) (y 0))
             (sequence (assign! y 14) (equality? x y))))))
```

In Scheme, a double backslash must be used within a string to get one, since backslash is the string escape character.

```
(let* ((example
"let pred = \\k\\->k-1
  in procedures f(n) = if n == 0
                       then 1
                       else n * f(pred(n))
      in f(4+1)
")
       (i (open-input-string example)))
  (equal? (parse-lang (lambda () (get-token i)))
          '(program
            ()
            (with-bindings
             ((pred (proc (k) (diff k 1))))
             (procedures
              ((f
                (proc
                 (n)
                 (if (equality? n 0) 1 (prod n (funcall f (funcall pred n)))))))
              (funcall f (sum 4 1)))))))

(let* ((example
"class point extends object{
  field x
  field y
  method init(initx, inity){
   x = initx;
   y = inity
  }
  method move(dx, dy){
   x = x + dx;
   y = y + dy
  }
}
let ob = new point(2+3, 1+4*7) in
  ob.move(0.1,3)
")
       (i (open-input-string example)))
  (equal? (parse-lang (lambda () (get-token i)))
          '(program
            ((class point object
                    (x y)
                    ((method
                      init
                      (initx inity)
                      (sequence (assign! x initx) (assign! y inity)))
                     (method
                      move
                      (dx dy)
                      (sequence (assign! x (sum x dx)) (assign! y (sum y dy)))))))
            (with-bindings
             ((ob (new point (sum 2 3) (sum 1 (prod 4 7)))))
             (funcall (send ob move) 0.1 3)))))
```

Undergraduate Extra Credit Parts B & C

This problem involves adding the code necessary to parse strings and lists. A string is simply a sequence of characters (other than a double quote) inside double quotes. The empty-list is written [], and a non-empty-list is written $[e_1, e_2, \cdots, e_n]$ where e_i is an expression in the language. The equivalent of `cons` involves special syntax; it should be written as the infix operator :: which is right associative (e.g., $1 :: 2 :: [3, 4]$ = `(make-cons 1 (make-cons 2 (make-explicit-list '(3 4))))`). You will need to do the following.

- (5 points) Modify the lexical and syntactic specifications so that strings can be parsed.
- (5 points) Modify the lexical and syntactic specifications so that lists can be parsed. Write the constructor `make-explicit-list` which tags its list argument by consing on the symbol `elist`.
- (5 points) Modify the lexical and syntactic specifications so that the cons operator :: can be parsed. Write the constructor `make-cons` that builds abstract syntax with tag `cons`.