Homework 7 Supplement: Another view of Mini-Cecil's Syntax

It seems useful to present an abstract syntax of mini-Cecil in a way that more closely resembles the λProlog abstract syntax; which I consider to be "the truth" for this homework. To that end I offer the following alternative to the handout you got earlier. To clarify the distinction between abstract and concrete syntax, I'll present both the abstract syntax and a concrete syntax.

1 Abstract Syntax of Mini-Cecil

There many ways to present abstract syntax. The handout you received earlier resembles the use of abstract syntax that is traditional in denotational semantics (for example in Schmidt's book), which introduces abbreviations for each syntactic category. In the abstract syntax below I'll follow Watt's book, *Programming Language Syntax and Semantics*, which does not use such abbreviations.

1.1 Macrosyntax

program ::= (`programDecl` class* method* procedure* main)
class* ::= `nil` | class :: class*
class ::= (`classDecl` classId decl*)
method* ::= `nil` | method :: method*
method ::= (`methodDecl` msgName methodFormal* typeExpr procBody)
methodFormal* ::= `nil` | methodFormal :: methodFormal*
methodFormal ::= (`methodFormalDecl` varName typeExpr)
procedure* ::= `nil` | procedure :: procedure*
procedure ::= (`procedureDecl` procName decl* typeExpr procBody)
procBody ::= (`body` statement resultExpr)
resultExpr ::= (`return` expression)
main ::= (`mainDecl` decl* statement decl* statement)
decl* ::= `nil` | decl :: decl*
decl ::= (`varDecl` varName typeExpr)
statement* ::= `nil` | statement :: statement*
statement ::= (`exprStmt` expression)
  | (`assignStmt` varName expression)
  | (`setFieldStmt` varName fieldName expression)
  | (`ifStmt` expression statement statement)
  | (`seqStmt` statement*)
expresion* ::= `nil` | expression :: expression*
expresion ::= (`varExpr` varName)
  | (`intLiteral` int)
  | `true`
  | `false`
  | (`msgExpr` msgName expression*)
  | (`procCallExpr` procName expression*)
  | (`newExpr` classId expression*)
  | (`getExpr` expression fieldName)
typeExpr* ::= `nil` | typeExpr :: typeExpr*
typeExpr ::= (`ty` typeName)
  | (`procType` typeExpr* typeExpr)

1.2 Microsyntax

The nonterminals varName, msgName, classId, typeName, procName, and fieldName are all strings. The nonterminal int is an integer literal.

```
varName ::= string
msgName ::= string
classId ::= string
typeName ::= string
procName ::= string
fieldName ::= string
string ::= " ( string-char | ^ ^ | ^ " )* "
string-char ::= “any character except ^ or "”
int ::= digit digit*
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

2 Concrete Syntax

For the concrete syntax, I have tried to follow the flavor of Cecil, as described Craig Chamber's technical report *The Cecil Language Specification and Rationale* (University of Washington, Department of Computer Science and Engineering, TR 93-03-05, March 1993), whenever reasonable.

2.1 Macrosyntax

```
program ::= class* method* procedure* main
class* ::= | class class*
class ::= class classId { decl* }
method* ::= | method method*
method ::= method msgName ( ) : typeExpr { procBody }
    | method msgName ( methodFormal-list ) : typeExpr { procBody }
methodFormal-list ::= methodFormal
    | methodFormal , methodFormal-list
methodFormal ::= varName : typeExpr
procedure* ::= | procedure procedure*
procedure ::= procedure procName ( ) : typeExpr { procBody }
    | procedure procName ( decl-list ) : typeExpr { procBody }
procBody ::= statement resultExpr
resultExpr ::= ^ expression
main ::= procedure main () : Void { decl-list ; statement { decl-list ; statement } }
    | procedure main () : Void { statement { decl-list ; statement } }
    | procedure main () : Void { decl-list ; statement { statement } }
    | procedure main () : Void { statement { statement } }
decl-list ::= decl | decl-list , decl
decl ::= varName : typeExpr
statement* ::= | statement statement*
statement ::= expression ;
    | varName := expression ;
    | varName . fieldName := expression ;
    | if expression then statement else statement ;
    | statement*
expresion* ::= | expression-list
```

expression-list ::= expression | expression-list , expression
expresion ::= varName
| int
| **true**
| **false**
| **send** msgName (expression*)
| **call** procName (expression*)
| **new** classId (expression*)
| expression . fieldName
| (expression)
typeExpr-list ::= typeExpr | typeExpr-list , typeExpr-list
typeExpr ::= (**ty** typeName)
| **&** ( ) : typeExpr
| **&** (typeExpr-list) : typeExpr

2.2 Microsyntax

varName ::= name
msgName ::= name
classId ::= name
typeName ::= name
procName ::= name
fieldName ::= name
name ::= letter letter-or-digit*
letter ::= a | b | ... | z | A | B | ... | Z | _
letter-or-digit ::= letter | digit
int ::= digit digit*
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9