Copyright (C) 2021 Timothy Fossum

Permission is granted to copy, distribute and/or modify this docu
terms of the GNU Free Documentation License, Version 1.1 or an
published by the Free Software Foundation. A copy of the license is
file "COPYING", entitled "GNU Free Documentation License".

PLCC

PLCC is the name of a tool set that takes a language specification file and gene
source files for an interpreter for the language.

The specification file is in three sections:

1. lexical specification
2. syntax
3. semantics

A line containing a single '%' is used to separate the lexical specification sectio
section and to separate the syntax section from the semantics secton.

A specification file can consist of just the lexical specification, in which case t
creates only a token scanner (`Scan`) for the language.

A specification file can consist of just the lexical specification and syntax, in whi
tool set creates only the scanner and parser (`Parse`) for the language.

The format of a language specification file is shown here:

```
# lexical specification
...
%
# syntax
...
%
# semantics
...
```

PLCC – Tokens

Comments may appear in a language specification file beginning with '#' and cor
of the line, except inside a token specification regular expression or in Java cod
section.

In the lexical specification section of a language specification file, a line is either
tion or a `token` specification. These specifications have the following form:

`{skip|token}` *NAME* *'re'*

where *NAME* is a string of uppercase letters, decimal digits, and underscores, st
percase letter, and *re* is a *regular expression* as defined by the Java `Pattern` clas

Here are some regular expression examples:

re	matches
`d`	the letter `d`
`\d`	a single decimal digit
`d+`	one or more `d`s
`\d+`	one or more decimal digits
`.`	any character
`\.`	the dot character (`.`)
`.*`	zero or more characters
`%.*`	the character `%` followed zero or more characters
`[a-z]`	any lowercase letter in the range `a` to `z`
`\w`	any letter (lowercase or uppercase), digit, or underscore
`[A-Za-z0-9_]`	the same as `\w`

PLCC regular expressions do not match anything that crosses a line boundary, so t
sion `'.*'` matches everything up to the end of the current line, including the end

PLCC – Tokens

A skip specification is used to identify things in a program that are otherwise
the syntax structure of the program. We generally skip whitespace (spaces, tabs,
language-defined comments. The format of a comment is language-dependent, bu
a comment starts with a '%' character and continues to the end of the line. Here
skip specifications for whitespace and comments:

```
skip WHITESPACE '\s+'
skip COMMENT '%.*'
```

A token specification is used to identify things in a program that are meaningful t
ture of the program. Examples are a language reserved word (such as `if`), a
sequence (such as a left bracket '`[`'), a numeric literal (such as `346`), or a variabl
`xyz`). Here are specifications that match these tokens.

```
token IF 'if'
token LBRACK '\['
token LIT '\d+'
token VAR '[A-Za-z]\w*'
```

For token specifications, the initial '`token`' can be omitted. Notice that regular
characters such as '`[`' must be quoted with a backslash '`\`' if they are to be t
characters.

PLCC – Tokens

The PLCC tool set uses the skip and token specifications to crea
`Token.java` and `Scan.java`.

The `Token.java` file defines the `Token.Match enum` class –
each skip and token specification name. It also defines the structu
object consisting of a `match` field of type `Token.Match`, a `st`
`String`, a `lno` field of type `int`, and a `line` field of type `Str`
field consists of the characters in the input stream that match the toke
– the token's *lexeme*, the `lno` field contains the line number on th
where the token appears, and the `line` field contains the input stre
the token appears. The `str` field always contains at least one chara

The `Scan.java` file defines an object that is constructed from an
`BufferedReader`). The `cur()` method delivers the current `Tok`
client (skipping over strings that match skip specifications), and the
advances to the next token. Multiple calls to `cur` without any inter
`adv` returns the same token repeatedly.

Upon encountering the end of the input stream, `cur` returns a spec
ject whose `match` field is `$EOF` and whose `toString()` represen
The `cur` method is *lazy*, meaning that the `Scan` class does not read
from the input stream until necessary to satisfy an explicit `cur` met

PLCC – Tokens

If there are characters remaining in the input stream, the `cur()` met
token and skip regular expressions one-by-one, in the order in whi
in the lexical specification. Each regular expression is matched aga
unmatched input, up to (and including) the end of the current line. I
match the regular expression, the next specification is tried.

If the regular expression is a skip specification that matches at l
character and if no prior token specification has found a matching to
the matched part is skipped and processing continues on the remaind
starting over from the first lexical specification. If a prior token sp
found a matching token candidate, the skip specification is ignored.

If a token specification match of length at least one occurs, and i
longer than the match length of the previous token candidate (if a
specification is chosen as the current token candidate. If not, the
candidate is retained.

PLCC – Tokens

If – after iterating through all of the lexical specifications – a toker been identified (first longest match), the token candidate is used t stance of the `Token` class. The `cur()` method advances the input the characters matched, saves the `Token` object for possible future c and returns the `Token` object. If no token candidate has been identif method returns a special `$ERROR` Token object. The `str` field of string of the form '`!ERROR(...)`', where the '`...`' part is (a rep the current input stream character where the match failed.

If the `cur()` method finds that there is a non-`null` saved `Toker` prior call to `cur()`, it simply returns that `Token` object without a cessing. Otherwise, processing occurs as described above.

The `adv()` method replaces the saved `Token` object with `null`, s quent call to `cur()` will be forced to get the next token from the ir it first detects that the saved token object is already null, it calls `cur` the next input token.)

As described earlier, the `cur` method returns a special `$EOF` token the end of the `BufferedReader` input.

PLCC – Syntax

The second section of a PLCC language specification file is the synt sisting of grammar rules in the style of Backus-Naur Form (BNF), Slide Set 1. Recall that a BNF grammar rule has the following form

```
LHS ::= RHS
```

where LHS (the *Left Hand Side*) is a nonterminal and the RHS (t *Side*) is a sequence of nonterminals and terminals. The individual p grammar rule, including the '`::=`' part, are separated by whitespace

Nonterminals in PLCC are identifiers enclosed between angle brack The identifier must begin with a lowercase letter and can consist o additional letters, digits, or underscores. Identifiers that match Java should be avoided.

Terminals in PLCC must begin with an uppercase letter and can co more additional uppercase letters, digits, or underscores. A termin as the name of a token in the lexical specification section.

PLCC associates every grammar rule with a unique Java clas name derived from the LHS of the grammar rule by converting acter to uppercase. Class names that match standard Java class na avoided.

PLCC – Syntax (continued)

If the LHS is a simple nonterminal, the Java class name associated rule is the nonterminal name with its first letter converted to upp example

```
<proc> ::= PROC LPAREN <formals> RPAREN <exp>
```

the Java class name is `Proc`.

If the LHS is a nonterminal annotated by adding a colon and a Ja then the class name associated with the BNF rule is the annotated Ja In this case, an abstract base class is also created whose Java cla nonterminal name with its first letter converted to uppercase (as de with the annotated Java class name as a subclass. In this example

```
<exp>:AppExp ::= DOT <exp> LPAREN <rands> RPAREN
```

the Java class name associated with this BNF rule is `AppExp`, and th the base class `Exp`.

PLCC – Syntax (continued)

PLCC requires that there are no duplicates of Java class names a the given grammar rules. In particular, if two grammar rules have nonterminal name, then their left-hand sides must have annotations Java class names. For example, the grammar rules on Slide 1.3

```
<nums> ::= <NUM> <nums>
<nums> ::=
```

would not be acceptable to PLCC. The following annotations fix thi

```
<nums>:NumsNode ::= <NUM> <nums>
<nums>:NumsNull ::=
```

When LHS rules are annotated in this way, the nonterminal class in this example) becomes an abstract Java class, and the annotate are used to generate classes that extend the abstract class. In this ex `NumsNode` and `NumsNull` classes are declared to extend the abstr

PLCC – Syntax (continued)

The RHS entries in a grammar rule are used to declare public fiel abstract) class associated with the grammar rule. Only those RHS e in angle brackets ‘`<...>`’ correspond to fields in the class.

A field can correspond to an RHS token (*e.g.* `<NUM>`) or an RHS nc `<nums>`).

If the field corresponds to an RHS *token*, its field *name* defaults to t with all of its letters in lowercase. For example, the field name cc the RHS token `<NUM>` is `num`, and its field *type* is `Token`. If the fie to an RHS *nonterminal*, its field *name* defaults to the nonterminal change). For example, the field name coressponding to the RH `<nums>` would be `nums`, and its field *type* is the underlying ty terminal – in this case, `Nums` – obtained by converting the first c nonterminal name to uppercase.

PLCC – Syntax (continued)

A BNF grammar rule may have the same nonterminal name appea once on its RHS, as in

```
<tree>:Interior ::= LPAREN <SYMBOL> <tree> <tree> RI
```

(see Slide 1.2). PLCC requies that there are no duplicates of cla associated with the RHS of a given grammar rule, so the above gram not be acceptable to PLCC. To solve the problem of duplicate fi annotate the duplicate field entries by appending an alternate field r identifier will do, but the convention is to have it start with a lowercas becomes the name of the corresponding field. The following anno above example:

```
<tree>:Interior ::= LPAREN <SYMBOL> <tree>left <tree
```

The same requirement that there be no duplicate field entries applies ciated with tokens instead of nonterminals. So for a BNF rule such a

```
<hhmmss> ::= <TWOD> COLON <TWOD> COLON <TWOD> NL
```

we would annotate the three `<TWOD>` token fields with different nan

```
<hhmmss> ::= <TWOD>hh COLON <TWOD>mm COLON <TWOD>ss
```

PLCC – Syntax (continued)

Repeating grammar rules – ones that have '**=' instead of '::=' - fields similar to non-repeating grammar rules, except that their l `Lists` of the appropriate type, and their field names have the st pended.

The following grammar rule (see Slide 1.36)

```
<nums> **= <NUM>
```

coressponds to the Java class `Nums`, having a single field `num` `List<Token>`.

Similarly, the following grammar rule (we will encounter this later)

```
<letDecls> **= LET <VAR> EQUALS <exp>
```

corresponds to the Java class `LetDecls` having one field `var` `List<Token>` and another field `expList` of type `List<Exp>`.

A repeating rule cannot be the first rule in a grammar file. A notated) class cannot define a repeating rule. A repeating rule c empty RHS.

PLCC – Parsing

PLCC generates a unique Java class for every grammar rule. Each s static `parse` method that is called with a `Scan` object parameter a an instance of the class with the class fields (defined by the gramm described above) populated with appropriate values. For an RHS fie ing to a token, the field value – a `Token` – comes directly from a input file being parsed. For an RHS field corresponding to a nonter value comes from calling the `parse` method on the nonterminal cla

Similar remarks apply to repeating rules, where the `parse` method populate the `List` fields in the class. The members of the `List` f the same order that their corresponding syntax entities appear in the

An abstract Java class generated by a nonterminal that appears on th than one grammar rules also defines a static `parse` method. This m the current token (delivered by the `Scan` object) and determines wh grammar rules corresponds to that token. It then returns the value ob ing the `parse` method on the derived class corresponding to the sel rule. The result is an instance of the derived class, which is also an given abstract class.

PLCC – Parsing (continued)

Here's a simple example grammar:

```
<tree>:Leaf     ::= <NUM>
<tree>:Interior ::= LPAREN <SYMBOL> <tree>left <tree
```

The abstract `Tree` class looks similar to this:

```
public abstract class Tree {

    public static Tree parse(Scan scn$) {
        Token t$ = scn$.cur();
        Token.Match match$ = t$.match;
        switch(match$) {
        case NUM:
            return Leaf.parse(scn$);
        case LPAREN:
            return Interior.parse(scn$);
        default:
            throw new PLCCException(
                "Parse error",
                "Tree cannot begin with " + t$.errSt
            );
        }
    }
}
```

PLCC – Parsing (continued)

PLCC generates the `Interior` class as follows:

```
// <tree>:Interior ::= LPAREN <SYMBOL> <tree>left <t
public class Interior extends Tree {

    public Token symbol;
    public Tree left;
    public Tree right;

    public Interior (Token symbol, Tree left, Tree r
        this.symbol = symbol;
        this.left = left;
        this.right = right;
    }

    public static Interior parse(Scan scn$) {
        scn$.match(Token.Match.LPAREN);
        Token symbol = scn$.match(Token.Match.SYMBOL
        Tree left = Tree.parse(scn$);
        Tree right = Tree.parse(scn$);
        scn$.match(Token.Match.RPAREN);
        return new Interior(symbol, left, right);
    }
}
```

PLCC – Parsing (continued)

The `parse` method for a repeating grammar rule uses a loop. For th
`<nums> **= <NUM>`, PLCC generates the following Java class N

```
public class Nums {

    public List<Token> numList;

    public Nums (List<Token> numList) {
        this.numList = numList;
    }

    public static Nums parse(Scan scn$) {
        List<Token> numList = new ArrayList<Token>();
        while (true) {
            Token t$ = scn$.cur();
            Token.Match match$ = t$.match;
            switch(match$) {
            case NUM:
                numList.add(scn$.match(Token.Match.NUM));
                continue;
            default:
                return new Nums(numList);
            }
        }
    }
}
```

Similar code is generated by repeating rules with a token separator.

PLCC – Parsing (continued)

While not shown in the examples above, PLCC adds a `Trace` parameter name
signature of all `parse` methods. If this parameter is not `null` – when the '–
option is given to the `Parse` or `Rep` programs – parsing the program will displ
trace" to standard error (by default). In Language `LON`, an example parse trac
'`(1 3 5)`' looks like this:

```
1: <lon>
1: | LPAREN "("
1: | <nums>:NumsNode
1: | | NUM "1"
1: | | <nums>:NumsNode
1: | | | NUM "3"
1: | | | <nums>:NumsNode
1: | | | | NUM "5"
1: | | | | <nums>:NumsNull
1: | RPAREN ")"
```

Each line in the parse trace displays either a nonterminal or a token. If it displays a
shows that the parser calls the `parse` method for that nonterminal (possibly in th
If it displays a token, then the scanner also displays the token's lexeme. The deci
beginning of each line is the line number in the source file where the parse found t
token; the number of vertical bars indicates the recursive depth of the parse.

PLCC – Semantics

For a given “program” in the language defined by its specificatic method in the *start symbol class* – the class determined by the first the language BNF grammar rules – returns an instance of this class is the root of the parse tree for the program. For example, given the

```
<tree>:Leaf     ::= <NUMBER>
<tree>:Interior ::= LPAREN <SYMBOL> <tree>left <tree
```

the `parse` method defined in the `Tree` class returns an instance of which is the root of the parse tree. (In what follows, we use the term refer to the root of the parse tree).

The runtime semantics of any PLCC program is the behavio calling the `void $run()` method on the parse tree. For any P the start symbol class extends a special `_Start` class generated au `PLCC`, so the parse tree is also an instance of `_Start`. Because the defines a `$run()` method whose behavior is to display the `toStri` tion of this object to standard output, and because the parse tree is an `_Start` class, the `$run()` behavior defined on the parse tree defa ing this `toString` representation. Here is the code for `$run()` i class:

```
public void $run() {
    System.out.println(this.toString());
}
```

PLCC – Semantics (continued)

To get a behavior different from this default behavior, the `$run` be redefined in the start symbol class or any of its subclasses. Fo `$run()` method may be redefined in the `Tree` class so that it d human-readable `toString` representation of the object than the s formative default representation. Here is code for the `$run()` meth class that redefines this behavior:

```
public void $run() {
   System.out.println("Tree: " + this.toString(
}
```

Since the `Tree` class appears more than once on the LHS of the each derived class `Leaf` and `Interior` can define its respecti method.

For example, the `Leaf` class has a field `num` of type `Token`. Tl method in this class can be easily written as follows:

```
public String toString() {
    return num.toString();
}
```

PLCC – Semantics (continued)

For an instance of class `Interior`, an appropriate `toString` me

```
public String toString() {
    return "("+symbol.toString()+" "+left+" "+right
}
```

This relies on the proper (recursive) `toString` behavior of the `le` fields, both of which are defined as instances of the `Tree` class.

Recall that PLCC generates a Java class for each of the BNF gramma the syntax section of the language specification. We can define *sema* these classes by adding entries to the semantics section of the langua file having the form

```
ClassName
%%%
...
%%%
```

where `ClassName` stands for the name of a PLCC-generated class `Tree` in Language TREE. PLCC inserts the lines of code brackete lines verbatim into the `ClassName.java` file. Most often, these l or more Java methods that can be applied to instances of the class.

PLCC – Semantics (continued)

The entire semantics section of the TREE language appears here:

```
Tree
%%%
    public void $run() {
        System.out.println("Tree: " + this.toString
    }
%%%

Leaf # extends the Tree class
%%%
    public String toString() {
        // 'num' is a Token field in the Leaf class
        return num.toString();
    }
%%%

Interior # extends the Tree class
%%%
    public String toString() {
        // 'left' and 'right' are Tree fields in the
        return "("+symbol.toString()+" "+left+" "+r
    }
%%%
```

PLCC – Semantics (continued)

As we observed above, PLCC automatically generates a Java sour
of the classes – both abstract and non-abstract – that are derived
grammar rules in the syntax section. In the semantics section of the
ification file, if PLCC encounters an entry of the form

```
ClassName
%%%
...
%%%
```

where `ClassName` stands for a class that is *not* one of the automati
classes, then PLCC generates a new file `ClassName.java` conta
bracketed by the '`%%%`' lines. This makes it possible for PLCC tc
source files that can be used to augment the semantics of the languag
we will use this to implement *environments*, which we describe late

In this situation, there is no automatically generated source file, sc
bracketed by the '`%%%`' lines must be a complete Java source file, no

PLCC – Semantics (continued)

An `#include` feature allows a PLCC language specification file
contents of other files, making them part of a single specification
separately created files can be combined together to form a single l
fication. The names of included files must be given in the semantic
specification file, and generally appear at the end of the specificati
example from a `grammar` file:

```
...
#include code
#include env
#include prim
#include val
```

In this example, the entire contents of the `code` file will be read
appended to the `grammar` file, then the `env` file, and so forth. Th
`#include` files will normally be representative of their purposes, b
do not otherwise play a role in the generated Java files.

PLCC – Java predefined/reserved name conflicts

The Java class names generated by the LHS of rules in the gramm
PLCC file must not conflict with standard Java class names. This me
grammar rule such as

```
<string> ::= ...
```

PLCC creates a Java class named `String`, which results in a Jav
error since '`String`' is a reserved class name.

Similarly, the names of fields in the RHS of rules in the gramma
not conflict with Java reserved words or predefined identifiers. Thi
grammar rule such as

```
<foo> ::= <IF> <blah>null
```

PLCC creates a field named `if` in the `Foo` class, which results in a
time error since `if` is a reserved word in Java. PLCC itself does
identify these errors, so these errors will only show up during comp

PLCC – plcc/plccmk command line arguments

The `plcc` script runs the `plcc.py` program with input from the
ments given on the command line. The `plccmk` script does the sa
filename defaults to `grammar`, and also compiles all of the Java fil
directory. If `plccmk` is given the '`-c`' command line argument, all
in the `Java` directory are removed before running `plcc.py`.

PLCC – Flags

PLCC has several internal name/value bindings that control its beha
ple, the value of `LL1` defaults to `True`; if it is changed to `False`, t
not check the grammar for being LL1. Here are some of the default

```
name       value   meaning
----       -----   -------
Token      True    (boolean) create a Token.java file
debug      0       (integer) larger values give more 
destdir    'Java'  (string)  Java source file destinat
pattern    True    (boolean) create a scanner that use
LL1        True    (boolean) check for LL(1)
parser     True    (boolean) create a parser
semantics  True    (boolean) create semantics routine
nowrite    False   (boolean) when True, produce *no* 
```

Names bound to integer or string values can be changed using comr
ments:

```
... --debug=3 --destdir=JJJ ...
```

or at the top of the specification file on lines beginning with '`!`':

```
!debug=3
!destdir=JJJ
```

PLCC – Flags (continued)

Names bound to a boolean value can be set to True on the comman
following example:

```
... --nowrite ...
```

or at the beginning of the language specification file:

```
!nowrite
```

Names bound to a boolean value can be set to False on the comman
following example:

```
... --LL1= ...
```

or at the top of the language specification file:

```
!LL1=
```

Bindings given in the language specification file always override bir
the command line.

PLCC – Comments in the lexical specification section

In the lexical specification section, PLCC comments appearing in a specification line are defined as matching the Java regular expressi Such comments are removed before attemtping to process the remai in the token or skip specification line. This means that a skip or expression containing a substring like ' #' will mistakenly be co start of a PLCC comment. In this case, use '[ ]#' instead.

PLCC – Hooks

PLCC creates Java source files from the BNF grammar rules in a la cation file. For a grammar rule whose LHS is a nonterminal such a PLCC creates a Java file named `Formals.java`. The initial cont appears as follows:

```
//Formals:top//
//Formals:import//
import java.util.*;

// <formals> **= <VAR> +COMMA // the original BNF rule
public class Formals /*Formals:class*/ {

    public static final String $className = "Formals";
    public static final String $ruleString = "<formals> **= <V

    public List<Token> varList; // the RHS field

    public Formals(List<Token> varList) { // the Formals const
//Formals:init//
        this.varList = varList;
    }

    public static Formals parse(Scan scn$, Trace trace$) {
        ... code to parse the RHS of the <formals> grammar rul
    }

//Formals//
}
```

PLCC – Hooks (continued)

The following lines in this file are called "hooks":

```
//Formals:top//
//Formals:import//
/*Formals:class*/
//Formals:init//
//Formals//
```

In the semantics section of the language specification file, you car place these hooks with arbitrary text as needed.

Use the `:init` hook to add additional lines of Java code at the be class constructor (called when parsing a program). For example `Formals` constructor shown on the previous slide. If you want to check the `varList` field for duplicate identifiers, include the fol your language specification file:

```
Formals:init
%%%
    Env.checkDuplicates(varList, " in proc formals")
%%%
```

This code will then replace the `//Formals:init//` line appearing a of the `Formals` class constructor. (Since the "hook" appears as a c original Java source code, it will not affect the compiled code if left

PLCC – Hooks (continued)

You can use the `:import` hook to add Java `import` lines to source that need to import Java packages other than the default `java.uti` see this used, for example, in Language ABC:

```
Program:import
%%%
import abcdatalog.engine.bottomup.sequential.*;
%%%
```

Similar comments apply to the `:top` hook, which can be used to a comment or such.

The `:class` hook can be used to declare any interfaces that should class definition.

PLCC – Hooks (continued)

You use the final hook in an automatically-generated Java source file definitions (and sometimes field declarations) that will be append definition. We have already seen how this is used, for example, in La on Slide 1.48, which we repeat here:

```
Lon
%%%
    public void $run() {
        System.out.print("( ");
        for (Token tok: nums.numList)
            System.out.print(tok.toString() + " ");
        System.out.println(")");
    }
%%%
```

For temporary testing purposes, you may disable adding the `$run(` nition into the `Lon.java` source file by replacing the `Lon` line with

```
Lon:ignore!
```

This is simpler (and easier to undo) than "commenting out" all of cluding the '`%%%`' lines) by turning them into PLCC '`#`' comment PLCC reads the lines in the `grammar` file bracketed by the '`%%%`' wise ignores them, making no changes to `Lon.java`.