PLCC

PLCC is the name of a tool set that takes a language specification file and generates a set of Java 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 section from the syntax section and to separate the syntax section from the semantics secton.

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

A specification file can consist of just the lexical specification and syntax, in which case the PLCC toolset 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 continuing to the end of the line, except inside a token specification regular expression or in Java code in the semantics section.

In the lexical specification section of a language specification file, a line is either a `skip` specification 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, starting with an uppercase letter, and *re* is a *regular expression* as defined by the Java `Pattern` class.

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 the regular expression `'.*'` matches everything up to the end of the current line, including the end-of-line marker.

PLCC – Tokens

A skip specification is used to identify things in a program that are otherwise not meaningful to the syntax structure of the program. We generally skip whitespace (spaces, tabs, and newlines) and language-defined comments. The format of a comment is language-dependent, but in our languages, a comment starts with a ‘%’ character and continues to the end of the line. Here is an example of 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 to the syntax structure of the program. Examples are a language reserved word (such as `if`), a special character sequence (such as a left bracket ‘`[`’), a numeric literal (such as `346`), or a variable symbol (such as `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 expression metacharacters such as ‘`[`’ must be quoted with a backslash ‘`\`’ if they are to be treated as ordinary characters.

PLCC – Tokens

The PLCC toolset uses the skip and token specifications to create source files `Token.java` and `Scan.java`.

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

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

Upon encountering the end of the input stream, `cur` returns a special `Token` object whose `match` field is `$EOF` and whose `toString()` representation is `!EOF`. The `cur` method is *lazy*, meaning that the `Scan` class does not read any characters from the input stream until necessary to satisfy an explicit `cur` method call.

PLCC – Tokens

If there are characters remaining in the input stream, the `cur()` method checks the token and skip regular expressions one-by-one, in the order in which they appear in the lexical specification. Each regular expression is matched against the current unmatched input, up to (and including) the end of the current line. If no characters match the regular expression, the next specification is tried.

If the regular expression is a skip specification that matches at least one input character and if no prior token specification has found a matching token candidate, the matched part is skipped and processing continues on the remainder of the input, *starting over from the first lexical specification*. If a prior token specification has found a matching token candidate, the skip specification is ignored.

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

PLCC – Tokens

If – after iterating through all of the lexical specifications – a token candidate has been identified (first longest match), the token candidate is used to create an instance of the `Token` class. The `cur()` method advances the input stream beyond the characters matched, saves the `Token` object for possible future calls to `cur()`, and returns the `Token` object. If no token candidate has been identified, the `cur()` method returns a special `$ERROR` Token object. The `str` field of this object is a string of the form '`!ERROR(...)`', where the '`...`' part is (a representation of) the current input stream character where the match failed.

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

The `adv()` method replaces the saved `Token` object with `null`, so that a subsequent call to `cur()` will be forced to get the next token from the input stream. (If it first detects that the saved token object is already null, it calls `cur()` to consume the next input token.)

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

PLCC – Syntax

The second section of a PLCC language specification file is the syntax section consisting of grammar rules in the style of Backus-Naur Form (BNF), as described in 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 (the *Right Hand Side*) is a sequence of nonterminals and terminals. The individual parts of a PLCC grammar rule, including the '`::=`' part, are separated by whitespace.

Nonterminals in PLCC are identifiers enclosed between angle brackets '<' and '>'. The identifier must begin with a lowercase letter and can consist of zero or more additional letters, digits, or underscores. Identifiers that match Java reserved words should be avoided.

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

PLCC associates every grammar rule with a unique Java class with a class name derived from the LHS of the grammar rule by converting its first character to uppercase. Class names that match standard Java class names should be avoided.

PLCC – Syntax (continued)

If the LHS is a simple nonterminal, the Java class name associated with the BNF rule is the nonterminal name with its first letter converted to uppercase. In this 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 Java class name, then the class name associated with the BNF rule is the annotated Java class name. In this case, an abstract base class is also created whose Java class name is the nonterminal name with its first letter converted to uppercase (as described above), 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 this class extends the base class `Exp`.

PLCC – Syntax (continued)

PLCC requires that there are no duplicates of Java class names associated with the given grammar rules. In particular, if two grammar rules have the same LHS nonterminal name, then their left-hand sides must have annotations giving distinct 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 this:

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

When LHS rules are annotated in this way, the nonterminal class name (`Nums`, in this example) becomes an abstract Java class, and the annotated class names are used to generate classes that extend the abstract class. In this example, both the `NumsNode` and `NumsNull` classes are declared to extend the abstract `Nums` class.

PLCC – Syntax (continued)

The RHS entries in a grammar rule are used to declare public fields in the (non-abstract) class associated with the grammar rule. Only those RHS entries enclosed in angle brackets '`<...>`' correspond to fields in the class.

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

If the field corresponds to an RHS *token*, its field *name* defaults to the token name with all of its letters in lowercase. For example, the field name corresponding to the RHS token `<NUM>` is `num`, and its field *type* is `Token`. If the field corresponds to an RHS *nonterminal*, its field *name* defaults to the nonterminal name (without change). For example, the field name coressponding to the RHS nonterminal `<nums>` would be `nums`, and its field *type* is the underlying type of the non-terminal – in this case, `Nums` – obtained by converting the first character of the nonterminal name to uppercase.

PLCC – Syntax (continued)

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

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

(see Slide 1.2). PLCC requies that there are no duplicates of class field names associated with the RHS of a given grammar rule, so the above grammar rule would not be acceptable to PLCC. To solve the problem of duplicate field names, we annotate the duplicate field entries by appending an alternate field name (any Java identifier will do, but the convention is to have it start with a lowercase letter) which becomes the name of the corresponding field. The following annotations fix the above example:

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

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

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

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

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

PLCC – Syntax (continued)

Repeating grammar rules – ones that have ‘**=’ instead of ‘::=’ – result in RHS fields similar to non-repeating grammar rules, except that their RHS fields are `Lists` of the appropriate type, and their field names have the string `List` appended.

The following grammar rule (see Slide 1.36)

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

coressponds to the Java class `Nums`, having a single field `numList` of type `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 `varList` of type `List<Token>` and another field `expList` of type `List<Exp>`.

PLCC – Parsing

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

Similar remarks apply to repetition rules, where the `parse` method uses a loop to populate the `List` fields in the class. The members of the `List` fields appear in the same order that their corresponding syntax entities appear in the input file.

An abstract Java class generated by a nonterminal that appears on the LHS of more than one grammar rules also defines a static `parse` method. This method looks at the current token (delivered by the `Scan` object) and determines which of the RHS grammar rules corresponds to that token. It then returns the value obtained by calling the `parse` method on the derived class corresponding to the selected grammar rule. The result is an instance of the derived class, which is also an instance of the given abstract class.

PLCC – Parsing (continued)

Here’s a simple example grammar:

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

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$.errString()
            );
        }
    }
}
```

PLCC – Parsing (continued)

PLCC generates the `Interior` class as follows:

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

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

    public Interior (Token symbol, Tree left, Tree right) {
        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 the grammar rule `<nums> **= <NUM>`, PLCC generates the following Java class `Nums`:

```
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$.match(Token.Match.NUM, trace$));
                continue;
            default:
                return new Nums(numList);
            }
        }
    }
}
```

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

PLCC – Parsing (continued)

The code samples shown above omit a `Trace` parameter named `trace$` that plays a role in calling the `parse` methods. If this parameter is not `null` – when the '`-t`' command-line option is given to the `Parse` or `Rep` programs – parsing the program will display a visual "parse trace" to standard error (by default). In Language `LON`, an example parse trace for the program '`(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 nonterminal, this shows that the parser calls the `parse` method for that nonterminal (possibly in the given subclass). If it displays a token, then the scanner also displays the token's lexeme. The decimal number at the beginning of each line is the line number in the source file where the parse found the nonterminal or 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 specification, the `parse` method in the *start symbol class* – the class determined by the first nonterminal in the language BNF grammar rules – returns an instance of this class. This instance is the root of the parse tree for the program. For example, given the BNF rules

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

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

The runtime semantics of any PLCC program is the behavior obtained by calling the `void $run()` method on the parse tree. For any PLCC language, the start symbol class extends a special _`Start` class generated automatically by `PLCC`, so the parse tree is also an instance of _`Start`. Because the _`Start` class defines a `$run()` method whose behavior is to display the `toString` representation of this object to standard output, and because the parse tree is an instance of the _`Start` class, the `$run()` behavior defined on the parse tree defaults to displaying this `toString` representation. Here is the code for `$run()` in the _`Start` class:

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

PLCC – Semantics (continued)

To get a behavior different from this default behavior, the `$run()` method can be redefined in the start symbol class or any of its subclasses. For example, the `$run()` method may be redefined in the `Tree` class so that it displays a more human-readable `toString` representation of the object than the somewhat uninformative default representation. Here is code for the `$run()` method in the `Tree` 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 grammar rules, each derived class `Leaf` and `Interior` can define its respective `toString` method.

For example, the `Leaf` class has a field `num` of type `Token`. The `toString` 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` method can be:

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

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

Recall that PLCC generates a Java class for each of the BNF grammar rules given in the syntax section of the language specification. We can define *semantic actions* for these classes by adding entries to the semantics section of the language specification file having the form

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

where `ClassName` stands for the name of a PLCC-generated class name, such as `Tree` in Language TREE. PLCC inserts the lines of code bracketed by the '`%%%`' lines verbatim into the `ClassName.java` file. Most often, these lines define one 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 Interior class
        return "("+symbol.toString()+" "+left+" "+right+")";
    }
%%%
```

PLCC – Semantics (continued)

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

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

where `ClassName` stands for a class that is *not* one of the automatically generated classes, then PLCC generates a new file `ClassName.java` containing the code bracketed by the '`%%%`' lines. This makes it possible for PLCC to generate Java source files that can be used to augment the semantics of the language. For example, we will use this to implement *environments*, which we describe later.

In this situation, there is no automatically generated source file, so the Java code bracketed by the '`%%%`' lines must be a complete Java source file, not just a method.

PLCC – Semantics (continued)

An include feature allows a PLCC language specification file to include the contents of other files, making them part of a single specification. In this way, separately created files can be combined together to form a single language specification. The names of include files must be given in the semantics section of the specification file, and generally appear at the end of the specification. Here is an 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 considered as if appended to the grammar file, then the env file, and so forth. The names of the include files will normally be representative of their purposes, but these names 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 grammar section of a PLCC file must not conflict with standard Java class names. This means that with a grammar rule such as

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

PLCC creates a Java class named `String`, which results in a Java compile-time error since ‘`String`’ is a reserved class name.

Similarly, the names of fields in the RHS of rules in the grammar section must not conflict with Java reserved words or predefined identifiers. This means with a grammar rule such as

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

PLCC creates a field named `if` in the `Foo` class, which results in a Java compile-time error since `if` is a reserved word in Java. PLCC itself does not attempt to identify these errors, so these errors will only show up during compilation.

PLCC – plcc/plccmk command line arguments

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

PLCC – Flags

PLCC has several internal name/value bindings that control its behavior. For example, the value of `LL1` defaults to `True`; if it is changed to `False`, then PLCC will not check the grammar for being LL1. Here are some of the default bindings:

name	value	meaning
Token	True	(boolean) create a Token.java file
debug	0	(integer) larger values give more verbose output
destdir	’Java’	(string) Java source file destination directory
pattern	True	(boolean) create a scanner that uses re. patterns
LL1	True	(boolean) check for LL(1)
parser	True	(boolean) create a parser
semantics	True	(boolean) create semantics routines
nowrite	False	(boolean) when True, produce *no* file output
PP	’’	(string) cmd to filter (preprocess) generated files

Names bound to integer or string values can be changed using command line arguments:

```
... --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 command line as in the 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 command line as in the following example:

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

or at the top of the language specification file:

```
!LL1=
```

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

PLCC – Comments in the lexical specification section

In the lexical specification section, PLCC comments appearing in a token or skip specification line are defined as matching the Java regular expression `"\s+#.*"`. Such comments are removed before attemtping to process the remaining characters in the token or skip specification line. This means that a skip or token regular expression containing a substring like `"  #"` will mistakenly be considered as the start of a PLCC comment. In this case, use `"[  ]#"` instead.

PLCC – Hooks

PLCC creates Java source files from the BNF grammar rules in a language specification file. For a grammar rule whose LHS is a nonterminal such as `<Formals>` PLCC creates a Java file named `Formals.java`. The initial contents of this file appears as follows:

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

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

    public static final String $className = "Formals";
    public static final String $ruleString = "<formals> **= <VAR> +COMMA";

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

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

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

//Formals//
}
```

PLCC – Hooks (continued)

The following lines in this file are called “hooks”:

```
//Formals:top//
//Formals:import//
//Formals:init//
//Formals//
```

In the semantics section of the language specification file, you can arrange to replace these hooks with arbitrary text as needed.

Use the `:init` hook to add additional lines of Java code at the beginning of the class constructor (which is called when parsing a program). Consider an instance of the `Formals` class shown on the previous slide. If you want to check the `varList` field for duplicate identifiers, include the following lines in your `grammar` file (or in your `code` file, if there is an `include code` line in your `grammar` file):

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

This code will then replace the `//Formals:init//` line appearing at the beginning of the `Formals` class constructor. (Since the “hook” appears as a comment in the original Java source code, it will not affect the compiled code if left unchanged.)

PLCC – Hooks (continued)

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

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

Similar comments apply to the `:top` hook.

PLCC – Hooks (continued)

You use the final hook in an automatically-generated Java source file to add method definitions (and sometimes field declarations) that will be appended to the class definition. We have already seen how this is used, for example, in Language LON 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(")");
    }
%%%
```