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 parts:

1. lexical specification
2. grammar
3. semantics

A line containing a single '`%`' is used to separate the lexical specification part from the grammar part and to separate the grammar part from the semantics part.

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

A specification file can consist of just the lexical specification and grammar, in which case the PLCC toolset creates only the scanner and parser for the language.

The format of a language specification file is shown here:

```
# lexical specification
...
%
# grammar rules
...
%
# 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 part 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 behavioral structure of the program. We generally skip whitespace (spaces, tabs, and newlines) and 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 behavioral structure of the program. Examples are a language reserved word (such as `if`), a punctuation mark (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 meta-characters 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.Val` `enum` class – one `enum` for each skip and token specification name. It also defines the structure of a `Token` object consisting of a `val` field of type `Token.Val`, a `str` field of type `String`, and a `lno` field of type `int`. The `str` field consists of the characters in the input stream that match the token specification – the token's *lexeme* – and the `lno` field contains the line number on the input stream 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 end of input, `cur` returns a special `Token` object whose `val` field is `$EOF` and whose `str` field 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

In the `cur()` method, the token and skip specifications are processed 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 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 itself, and returns the `Token` object. If – after processing any relevant skip specifications – no token candidate has been identified, the `cur()` method throws an exception.

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 end of input.

PLCC – Grammar

The second section of a PLCC language specification consists 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.

Every grammar rule is associated 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 – Grammar

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 – Grammar

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 1.12

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

would not be acceptable to PLCC. The following annotations would fix this:

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

When LHS rules are annotated in this way, the nonterminal class name (`Tree`, 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 `Leaf` and `Interior` classes are declared to extend the abstract `Tree` class.

PLCC – Grammar

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 that are enclosed in angle brackets ‘`<...>`’ correspond to fields in the class.

An RHS field can correspond to a token (*e.g.* `<NUMBER>`, shown on the previous slide) or a nonterminal (*e.g.* `<formals>`, shown on slide 1a.6).

If the RHS field corresponds to a token, its field *name* defaults to the token name with all of its letters in lowercase. For example, the RHS field name corresponding to `<NUMBER>` would be `number`, and its field *type* is `Token`. If the RHS field corresponds to a nonterminal, its field *name* defaults to the nonterminal name (without change). For example, the RHS field name coressponding to `<formals>` would be `formals`, and its field *type* is the underlying type of the nonterminal – in this case, `Formals` – obtained by converting the first character of the nonterminal name to uppercase.

PLCC – Grammar

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
```

PLCC requies that there are no duplicates of class field names associated with the RHS of a given grammar rule. The above grammar rule would therefore not be acceptable to PLCC. We can annotate the duplicate field entries by appending an alternate field name (any name 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 – those of the form `<TOKEN>`. 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 – Grammar

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>
```

would coresspond 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>
```

would correspond 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.

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

Here's a simple example grammar:

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

The abstract `Tree` class looks like this:

```
public abstract class Tree {

    public Tree parse(Scan scn) {
        Token t = scan.cur();
        switch (t.val) {
        case NUMBER:
            return Leaf.parse(scn);
        case LPAREN:
            return Interior.parse(scn);
        default:
            throw new RuntimeException("parse error");
        }
    }
}
```

PLCC – Parsing

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.Val.LPAREN);
        Token symbol = scn.match(Token.Val.SYMBOL);
        Tree left = Tree.parse(scn);
        Tree right = Tree.parse(scn);
        scn.match(Token.Val.RPAREN);
        return new Interior(symbol, left, right);
    }
}
```

PLCC – Parsing

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

```
// <nums> **= <NUM>
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();
            switch(t.val) {
            case NUM:
                break;
            default:
                return new Nums(numList);
            }
            numList.add(scn.match(Token.Val.NUM));
        }
    }
}
```

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

PLCC – Semantics

For a given "program" in the language defined by the grammar, the `parse` method of the start symbol class (the first nonterminal in the language grammar rules) returns an instance of this class. This instance is the root of a parse tree for the program. For example, given the grammar

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

the `parse` method returns an instance of a `Tree` object.

The default semantics of a PLCC program is to return the `toString` value of this instance. This means that, in this example, a `toString` method must be defined in the `Tree` class. Since this class appears more than once on the LHS of the grammar rules, each derived class `Leaf` and `Interior` must 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

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.

To define these semantic actions, we add these Java methods to the grammar specifications after the grammar rules, separated by a single line containing a ‘`%`’

Semantic actions for a class defined by the grammar rules are generated by entries having the form

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

where `ClassName` is the name of the PLCC generated class name, such as `Interior`.

PLCC – Semantics

For example, the `toString` semantics of an `Interior` class can be given as follows:

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

Similar remarks apply to the semantics of a `Leaf` class.

PLCC initially generates a *stub* for each of the classes – both abstract and non-abstract – that are specified by the grammar rules. The code for PLCC semantic actions are simply added to these Java class files. In the above example, the code for the `toString` method becomes part of the `Interior.java` file whose stub is generated automatically by the grammar rules.

PLCC – Semantics

If PLCC encounters an entry of the form

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

where `ClassName` is not one of the automatically generated classes, then PLCC generates a file `ClassName.java` containing the enclosing code. 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 stub file, so the Java code between the '`%%%`' lines must be a complete Java program, not just a method.

PLCC – Semantics

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 – Miscellanea

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 a grammar rule such as

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

will create a Java class named `String`, resulting in a compile-time error.

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 that a grammar rule such as

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

will result in a compile-time error.

PLCC itself does not attempt to identify these errors, so the errors will only show up when compiling the Java source files.

PLCC – Miscellanea

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) debug value
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 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 – Miscellanea

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 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 specification file:

```
!LL1=
```

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

See the `plcc.py` program for details.