CS 301 Homework 3 – due 3:30pm, Wednesday, February 24, 1999

Parsing

Write a parser for the complete **PCAT** language. The defining grammar for **PCAT** is in Section 12 of the Language Reference Manual; a copy is also available online in the file `concrete.txt`. Use this grammar as a guideline for writing your parser.

Your executable, called `parser`, must read a stream of lexical tokens and attributes from standard input. The lexical token stream is represented by a special-purpose byte-code, defined in detail below; an executable that converts a `.pcat` source file to such a token stream is provided as `/u/cs301acc/3/lex2bc`. If the input represents a legal program, your parser must write a readable representation of the program's abstract syntax to standard output, using the format given in Section 13 of the Language Reference Manual (also available in the online file `ast.txt`). More precisely, your parser should produce exactly the same output as the reference parser executable `/u/cs301acc/3/parser`. If the input is invalid, your parser must write a suitable error message to standard error and halt; it should not attempt error recovery. Error message text need not match the reference parser exactly.

The usual mode of operation is to glue the output of `lex2bc` to the parser via a Unix pipe, e.g.,

```
lex2bc < test.pcat | parser > test.ast
```

Alternatively, the byte stream can be written to an intermediate file.

```
lex2bc < test.pcat > test.bc
parser < test.bc > test.ast
```

This rather unnatural set-up is intended to permit the parser to be written and tested in complete isolation from the lexer and subsequent compiler stages, and without requiring a commitment to any particular internal organization. (Of course, in a "real" compiler, the parser would call the lexer directly, without going through the token byte code, and would typically build an internal data structure representing the program's abstract syntax.)

The "correct" behavior of the parser, i.e., the correct mapping from concrete to abstract syntax, is defined by the behavior of `/u/cs301acc/3/parser`. In most cases, this behavior should be obvious; here are a few notable points:

1. The AST is capable of describing programs that are not type-correct; type-checking will be done in a later assignment.

2. To help make error messages from such a type-checker meaningful, most AST constructs contain a `line` field; this should be the source line number associated with the construct. For constructs spanning several lines, the line number containing the *last* token should be used.

3. The concrete syntax allows several variables to be declared with the same type and initializing expression; the type and initializer must be replicated into each AST `VarDec`.

4. The AST format allows arbitrary `type`s in `TypeDec`, `VarDec`, `ProcDec`, `ArrayTyp`, `Comp`, and `Param` constructs; in fact, `NamedTyp`s and `NoTyp`s should not appear in `TypeDec`s and *only* `NamedTyp`s or `NoTyp`s should appear in the other contexts. `NoTyp`s are used only to represent omitted optional types.

5. If the `BY` clause in a `FOR` statement is omitted, supply 1 in the AST.

6. If the count expression is omitted in an array initializer, supply 1 in the AST.

7. Expand `ELSIF` clauses into nested `IfSt` structures in the AST. If the `THEN` branch is missing from an `IF` or `ELSIF`, use an empty `SeqSt` in the AST.

8. The correct precedence and associativity for operators is specified in the Language Reference Manual, Section 10.8.

Your error messages need not match the working version exactly, but at a minimum they should indicate the nature of the error and reflect the approximate source line number at which the error occurred.

Implementation and Program Submission

The internal details of your parser are up to you, but it is *strongly* recommended that you implement it using the `yacc` or `bison` parser generator, and that you build and subsequently print out an actual abstract syntax tree data structure, using the constructor and printing routines provided in the files `ast.h` and `ast.c`. In particular, using the AST printing routines will make it trivial to produce output that is formatted identically to the reference parser; it will also make future assignments easier.

`bison` is the GNU version of `yacc`; it produces somewhat better diagnostic output than original `yacc`, and it has proper on-line documentation. To access the documentation you must run the GNU `info` program; assuming you have done `addpkg gnu`, type `info bison`. Once inside `info`, type `h` for a tutorial on how to use the `info` reader. To avoid confusing (non-GNU) `make`, you should run `bison` with the `-y` flag.

In addition, file `lexin.c` contains routines to process the token byte-code stream transparently, in the form of definitions for `yyparse`, `yylval`, and `yylineno`. By using this file, you can write your `yacc`-based parser just as if it were linked to a `lex`-generated lexical analyzer in the usual fashion.

As noted above, an executable `lex2bc` is provided. However, if you wish to integrate your lexer from assignment 2 (encouraged if it works!), all you need do is replace its main driver program with code that writes appropriate byte-codes to standard output instead. Model code that performs this task is available in the file `lexout.c`. `lexin.c` and `lexout.c` share a header file `token_bytecodes.h` which defines the token byte codes as an enumeration. Should it be necessary to inspect the contents of byte-code files, you may find the unix command `od` useful.

To submit the assignment, prepare a `makefile` that builds your parser and driver given your source files and leaves the executable program in `parser`. If you want to use your own lexer code, the makefile should build it as well, under the name `lex2bc`; if the makefile lacks a `lex2bc` target, the executable in `/u/cs301acc/3/lex2bc` will be used when testing your submission.

Submit your program by `mail`ing a `shar` "bundle" containing the relevant files to `cs301acc`, as described in the "Handing In Assignments" handout.

Token Byte Code

Each token, and the pseudo-token `LINE`, corresponds to a single byte code, as listed below and repeated in the file `byte_codes.txt`. Certain tokens are followed in the byte stream by a sequence of bytes representing an attribute, as follows: `LINE` and `INTEGER` tokens are followed by an integer attribute, and `ID`, `REAL`, and `STRING` tokens are followed by a string attribute. Integer attributes are encoded as a sequence of four bytes, least-significant-byte first. String attributes are encoded as a one-byte length count $i \geq 0$, followed by i bytes containing the string's ASCII characters in order; there is *no* terminating `'\0'` character.

Note that the byte-code is completely machine independent: for example, it should be possible to generate byte code files on Sparc and consume them on Intel machines, or vice-versa.

Token	Code	Token	Code	Token	Code	Token	Code
LINE	0						
ID	1	FOR	15	TO	29	(	43
STRING	2	IF	16	TYPE	30	)	44
REAL	3	IS	17	VAR	31	<	45
INTEGER	4	LOOP	18	WHILE	32	>	46
AND	5	MOD	19	WRITE	33	=	47
ARRAY	6	NOT	20	:=	34	+	48
BEGIN	7	OF	21	<=	35	-	49
BY	8	OR	22	>=	36	*	50
DIV	9	PROCEDURE	23	<>	37	/	51
DO	10	PROGRAM	24	[<	38	{	52
ELSE	11	READ	25	>]	39	}	53
ELSIF	12	RECORD	26	;	40	[	54
END	13	RETURN	27	,	41	]	55
EXIT	14	THEN	28	:	42	.	56