Abstract syntax trees

A compiler pass is a traversal of the program. A compiler phase is a group of related passes.

A one-pass compiler scans the program only once. It is naturally single-phase. The following all happen at the same time:

- scanning
- parsing
- weeding
- symbol table creation
- type checking
- resource allocation
- code generation
- optimization
- emitting

This is a terrible methodology:

- it ignores natural modularity;
- it gives unnatural scope rules; and
- it limits optimizations.

However, it used to be popular:

- it’s fast (if your machine is slow); and
- it’s space efficient (if you only have 4K).

A modern multi-pass compiler uses 5–15 phases, some of which may have many individual passes: you should skim through the optimization section of ‘man gcc’ some time!

A multi-pass compiler needs an intermediate representation of the program between passes.

We could use a parse tree, or concrete syntax tree (CST):

or we could use a more convenient abstract syntax tree (AST), which is essentially a parse tree/CST but for a more abstract grammar:
Instead of constructing the tree:

```
+   *
 id  id
```

a compiler can generate code for an internal compiler-specific grammar, also known as an *intermediate language*.

Early multi-pass compilers wrote their IL to disk between passes. For the above tree, the string `+(id,*(id,id))` would be written to a file and read back in for the next pass.

It may also be useful to write an IL out for debugging purposes.

Examples of modern intermediate languages:
- Java bytecode
- C, for certain high-level language compilers
- Jimple, a 3-address representation of Java bytecode specific to Soot that you learn about in COMP 621
- Simple, the precursor to Jimple that Laurie Hendren created for McCAT
- Gimple, the IL based on Simple that gcc uses

In this course, you will generally use an AST as your IR without the need for an explicit IL.

Note: somewhat confusingly, both industry and academia use the terms IR and IL interchangeably.

```
$ cat tree.h tree.c # AST construction for Tiny language

[...]
typedef struct EXP {
    enum {idK,intconstK,timesK,divK,plusK,minusK} kind;
    union {
        char *idE;
        int intconstE;
        struct {struct EXP *left; struct EXP *right;} timesE;
        struct {struct EXP *left; struct EXP *right;} divE;
        struct {struct EXP *left; struct EXP *right;} plusE;
        struct {struct EXP *left; struct EXP *right;} minusE;
    } val;
} EXP;

EXP *makeEXPid(char *id)
{
    EXP *e;
    e = NEW(EXP);
    e->kind = idK;
    e->val.idE = id;
    return e;
}
[...]

EXP *makeEXPminus(EXP *left, EXP *right)
{
    EXP *e;
    e = NEW(EXP);
    e->kind = minusK;
    e->val.minusE.left = left;
    e->val.minusE.right = right;
    return e;
}
[...]
```

```
$ cat tiny.y # Tiny parser that creates EXP *theexpression

[...]
%union {
    int intconst;
    char *stringconst;
    struct EXP *exp;
}
%token <intconst> tINTCONST
%token <stringconst> tIDENTIFIER
%type <exp> program exp

[...]
```

```
$ cat tiny.y # Tiny parser that creates EXP *theexpression

[...]
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"
extern char *yytext;
extern EXP *theexpression;
void yyerror() {
    printf("syntax error before \%s\n", yytext);
}
%
%union {
    int intconst;
    char *stringconst;
    struct EXP *exp;
}
%token <intconst> tINTCONST
%token <stringconst> tIDENTIFIER
%type <exp> program exp
[...]```
Abstract syntax trees (9)

%start program
%left '+' '-'
%left '*' '/'
%%
program: exp
  { theexpression = $1; }
;
exp : tIDENTIFIER
  ( $$ = makeEXPid ($1); )
tINTCONST
  ( $$ = makeEXPintconst ($1); )
| exp '*' exp
  ( $$ = makeEXPmult ($1, $3); )
| exp '/' exp
  ( $$ = makeEXPdiv ($1, $3); )
| exp '+' exp
  ( $$ = makeEXPplus ($1, $3); )
| exp '-' exp
  ( $$ = makeEXPminus ($1, $3); )
| '(' exp ')'
  ( $$ = $2; )

%%

Abstract syntax trees (10)

Constructing an AST with flex/bison:

- AST node kinds go in tree.h
  
  ```
  enum {idK, intconstK, timesK, divK, plusK, minusK} kind;
  ```

- AST node semantic values go in tree.h
  
  ```
  struct {struct EXP *left; struct EXP *right;} minusE;
  ```

- Constructors for node kinds go in tree.c
  
  ```
  EXP *makeEXPminus(EXP *left, EXP *right)
  {
    EXP *e;
    e = NEW(EXP);
    e->kind = minusK;
    e->val.minusE.left = left;
    e->val.minusE.right = right;
    return e;
  }
  ```

- Semantic value type declarations go in tiny.y
  
  ```%union {
    | int intconst;
    | char *stringconst;
    | struct EXP *exp;
  }%token <intconst> tINTCONST
  ```

- (Non-)terminal types go in tiny.y
  
  ```%token <stringconst> tIDENTIFIER
  %type <exp> program exp
  ```

- Grammar rule actions go in tiny.y
  
  ```
  exp : exp '-' exp { $$ = makeEXPminus ($1, $3); }
  ```

Abstract syntax trees (11)

A "pretty"-printer:

```$ cat pretty.h
#include <stdio.h>
#include "pretty.h"

void prettyEXP(EXP *e)
{
  switch (e->kind) {
  case idK:
    printf("%s", e->val.idE);
    break;
  case intconstK:
    printf("%i", e->val.intconstE);
    break;
  case timesK:
    printf("*", e->val.timesE.left);
    prettyEXP(e->val.timesE.right);
    break;
  case minusK:
    printf("-", e->val.minusE.left);
    prettyEXP(e->val.minusE.right);
    break;
  ...
  }
}
```

The following pretty printer program:

```$ cat main.c
#include "tree.h"
#include "pretty.h"

void yyparse();

EXP *theexpression;

void main()
{
yyparse();
  prettyEXP(theexpression);
}
```

will on input:

```a*(b-17) + 5/c
```

produce the output:

```((a*(b-17))+(5/c))
```
As mentioned before, a modern compiler uses 5–15 phases. Each phase contributes extra information to the IR (AST in our case):

- scanner: line numbers;
- symbol tables: meaning of identifiers;
- type checking: types of expressions; and
- code generation: assembler code.

**Example: adding line number support.**

First, introduce a global `lineno` variable:

```
$ cat main.c
[...]
int lineno;
void main()
{ lineno = 1; /* input starts at line 1 */
yyparse();
prettyEXP(theexpression);
}
```

Second, increment `lineno` in the scanner:

```
$ cat tiny.l # modified version of previous exp.l
%
#include "y.tab.h"
#include <string.h>
#include <stdlib.h>
extern int lineno; /* declared in main.c */
%
[/t]+ /* ignore */; /* no longer ignore \n */
\n lineno++; /* increment for every \n */
[...]
```

Third, add a `lineno` field to the AST nodes:

```
typedef struct EXP {
  int lineno;
  enum {idK,intconstK,timesK,divK,plusK,minusK} kind;
  union {
    char *idE;
    int intconstE;
    struct {struct EXP *left; struct EXP *right;} timesE;
    struct {struct EXP *left; struct EXP *right;} divE;
    struct {struct EXP *left; struct EXP *right;} plusE;
    struct {struct EXP *left; struct EXP *right;} minusE;
  } val;
} EXP;
```

Fourth, set `lineno` in the node constructors:

```
extern int lineno; /* declared in main.c */
EXP *makeEXPid(char *id)
{ EXP *e;
  e = NEW(EXP);
  e->lineno = lineno;
  e->kind = idK;
  e->val.idE = id;
  return e;
}
EXP *makeEXPintconst(int intconst)
{ EXP *e;
  e = NEW(EXP);
  e->lineno = lineno;
  e->kind = intconstK;
  e->val.intconstE = intconst;
  return e;
}
[...]
EXP *makeEXPminus(EXP *left, EXP *right)
{ EXP *e;
  e = NEW(EXP);
  e->lineno = lineno;
  e->kind = minusK;
  e->val.minusE.left = left;
  e->val.minusE.right = right;
  return e;
}
```

The SableCC 2 grammar for our Tiny language:

```
Package tiny;

Helpers
tab = 9;
cr = 13;
lf = 10;
digit = ['0'..'9'];
lowercase = ['a'..'z'];
uppercase = ['A'..'Z'];
letter = lowercase | uppercase;
idletter = letter | '_';
idchar = letter | '_' | digit;

Tokens
eol = cr | lf | cr lf;
blank = '' | tab;
star = '*';
slash = '/';
plus = '+';
minus = '-';
l_par = '(';
r_par = ')';
number = '0'| [digit-'0'] digit*;
id = idletter idchar*;

Ignored Tokens
blank, eol;
```
Productions

\[
\begin{align*}
\text{exp} & = \\
& \{\text{plus}\} \text{ exp} \text{ plus} \text{ factor} \mid \\
& \{\text{minus}\} \text{ exp} \text{ minus} \text{ factor} \mid \\
& \{\text{factor}\} \text{ factor}; \\
\text{factor} & = \\
& \{\text{mult}\} \text{ factor} \text{ star} \text{ term} \mid \\
& \{\text{divd}\} \text{ factor} \text{ slash} \text{ term} \mid \\
& \{\text{term}\} \text{ term}; \\
\text{term} & = \\
& \{\text{paren}\} \text{ l_par} \text{ exp} \text{ r_par} \mid \\
& \{\text{id}\} \text{ id} \mid \\
& \{\text{number}\} \text{ number};
\end{align*}
\]

SableCC generates subclasses of the 'Node' class for terminals, non-terminals and production alternatives:

- Node classes for terminals: 'T' followed by (capitalized) terminal name:
  TEOl, TBlank, ..., TNumber, TId
- Node classes for non-terminals: 'P' followed by (capitalized) non-terminal name:
  PExp, PFactor, PTerm
- Node classes for alternatives: 'A' followed by (capitalized) alternative name and (capitalized) non-terminal name:
  APlusExp (extends PExp), ..., ANumberTerm (extends PTerm)

SableCC populates an entire directory structure:

<table>
<thead>
<tr>
<th>tiny/</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Given some grammar, SableCC generates a parser that in turn builds a concrete syntax tree (CST) for an input program.

A parser built from the Tiny grammar creates the following CST for the program `a+b*c':

\[
\begin{align*}
\text{Start} & = \\
& \{\text{APlusExp}\} \\
& \{\text{AFactorExp} \text{ AMultFactor} \mid \} \\
& \{\text{ATermFactor} \text{ ATermFactor} \text{ AIdTerm} \mid \} \\
& \{\text{AIdTerm} \text{ AIdTerm} \text{ c} \mid \} \\
& \{\text{a} \text{ b} \}
\end{align*}
\]

This CST has many unnecessary intermediate nodes. Can you identify them?
We only need an abstract syntax tree (AST) to operate on:

```
APlusExp
/  \
AIdExp AMultExp
|   |
a  AIdExp AIdExp
|   |
    b  c
```

Recall that *bison* relies on user-written actions after grammar rules to construct an AST.

As an alternative, SableCC 3 actually allows the user to define an AST and the CST→AST transformations formally, and can then translate CSTs to ASTs automatically.

### AST for the Tiny expression language:

**Abstract Syntax Tree**

```
exp =
{plus} [l]:exp [r]:exp |
{minus} [l]:exp [r]:exp |
{mult} [l]:exp [r]:exp |
{divd} [l]:exp [r]:exp |
{id} id |
{number} number;
```

AST rules have the same syntax as rules in the **Production** section except for CST→AST transformations (obviously).

### Extending Tiny productions with CST→AST transformations:

**Productions**

```
cst_exp (-> exp) =
  (cst_plus)  cst_exp plus factor
            (-> New exp.plus(cst_exp.exp,factor.exp)) |
  (cst_minus) cst_exp minus factor
            (-> New exp.minus(cst_exp.exp,factor.exp)) |
  (factor)   factor (-> factor.exp);

factor (-> exp) =
  (cst_mult)  factor star term
            (-> New exp.mult(factor.exp,term.exp)) |
  (cst_divd)  factor slash term
            (-> New exp.divd(factor.exp,term.exp)) |
  (term)     term (-> term.exp);

term (-> exp) =
  (paren)    l_par cst_exp r_par (-> cst_exp) |
  (cst_id)   id (-> New exp.id(id)) |
  (cst_number) number (-> New exp.number(number));
```

**A CST production alternative for a plus node:**

```
cst_exp = {cst_plus} cst_exp plus factor
```

needs extending to include a CST→AST transformation:

```
cst_exp (-> exp) =
  (cst_plus)  cst_exp plus factor
            (-> New exp.plus(cst_exp.exp,factor.exp))
```

**cst**

```
cst_exp (-> exp) on the LHS specifies that the CST node **cst_exp** should be transformed to the AST node **exp**.
```

```
{-> New exp.plus(cst_exp.exp, factor.exp)}
```

on the RHS specifies the action for constructing the AST node.

**exp.plus** is the kind of **exp** AST node to create.

**cst.exp** refers to the transformed AST node **exp** of **cst_exp**, the first term on the RHS.
5 types of explicit RHS transformation (action):

1. Getting an existing node:
   \( \{ \text{paren}\} \ l\_\text{par} \ \text{cst}\_\text{exp} \ r\_\text{par} \ \rightarrow \ \text{cst}\_\text{exp} \cdot \text{exp} \)

2. Creating a new AST node:
   \( \{ \text{cst}\_\text{id}\} \ \text{id} \ \rightarrow \ \text{New} \ \text{exp} \cdot \text{id(id)} \)

3. List creation:
   \( \{ \text{block}\} \ l\_\text{brace} \ \text{stm*} \ r\_\text{brace} \ \rightarrow \ \text{New} \ \text{stm} \cdot \text{block}([\text{stm}]) \)

4. Elimination (but more like nullification):
   \( \rightarrow \ \text{Null} \)
   \( \rightarrow \ \text{New} \ \text{exp} \cdot \text{id(Null)} \)

5. Empty (but more like deletion):
   \( \rightarrow \ \) 

Writing down straightforward, non-abstracting CST→AST transformations can be tedious.

\[ \text{prod} = \text{elm1} \ \text{elm2*} \ \text{elm3+} \ \text{elm4?}; \]

This is equivalent to:

\[ \text{prod}(\rightarrow \text{prod}) = \text{elm1} \ \text{elm2*} \ \text{elm3+} \ \text{elm4}? \]
\[ (\rightarrow \text{New} \ \text{prod} \cdot \text{prod} (\text{elm1.elm1}, \{\text{elm2.elm2}\}, [\text{elm3.elm3}], \text{elm4.elm4})) ; \]

More SableCC 3 documentation:

- [http://sablecc.sourceforge.net/documentation.html](http://sablecc.sourceforge.net/documentation.html)

The JOOS compiler has the AST node types:

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>CLASSFILE</th>
<th>CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIELD</td>
<td>TYPE</td>
<td>LOCAL</td>
</tr>
<tr>
<td>METHOD</td>
<td>FORMAL</td>
<td>LOCAL</td>
</tr>
<tr>
<td>STATEMENT</td>
<td>EXP</td>
<td>RECEIVER</td>
</tr>
<tr>
<td>ARGUMENT</td>
<td>LABEL</td>
<td>CODE</td>
</tr>
</tbody>
</table>

with many extra fields:

```c
typedef struct METHOD {
   int lineno;
   char *name;
   ModifierKind modifier;
   int localslimit; /* resource */
   int labelcount; /* resource */
   struct TYPE *returntype;
   struct FORMAL *formals;
   struct STATEMENT *statements;
   char *signature; /* code */
   struct LABEL *labels; /* code */
   struct CODE *opcodes; /* code */
} METHOD;
```

The JOOS constructors are as we expect:

```c
METHOD *makeMETHOD(char *name, ModifierKind modifier,
                   TYPE *returntype, FORMAL *formals,
                   STATEMENT *statements, METHOD *next)
{
    METHOD *m;
    m = NEW(METHOD);
    m->lineno = lineno;
    m->name = name;
    m->modifier = modifier;
    m->returntype = returntype;
    m->formals = formals;
    m->statements = statements;
    m->next = next;
    return m;
}
```

```c
STATEMENT *makeSTATEMENTwhile(EXP *condition,
                                STATEMENT *body)
{
    STATEMENT *s;
    s = NEW(STATEMENT);
    s->lineno = lineno;
    s->kind = whileK;
    s->val.whileS.condition = condition;
    s->val.whileS.body = body;
    return s;
}
```
Highlights from the JOOS scanner:

```c
[ 	
]+ /* ignore */;
\n lineno++;
\n//[^\n\n]* /* ignore */;
abstract return tABSTRACT;
boolean return tBOOLEAN;
break return tBREAK;
byte return tBYTE;
-
-
"!=" return tNEQ;
"&&" return tAND;
"||" return tOR;
"+" return '+';
-0|([1-9]\[0-9\]*) {yylval.intconst = atoi(yytext);
true {yylval.boolconst = 1;
false {yylval.boolconst = 0;
\"([^\\"]\*)\" {yylval.stringconst =
(\char*malloc(strlen(yytext)-1);
yytext[strlen(yytext)-1] = '\0';
sprintf(yylval.stringconst,"%s",yytext+1);
return tSTRINGCONST;
```

Building LALR(1) lists:

```c
formals : /* empty */
{($$ = NULL;)
 | neformals
 $$ = $1;)
 ;

neformals : formal
{($$ = $1;)
 | neformals , formal
 $$ = $3; $$->next = $1;)
 ;

formal : type tIDENTIFIER
{($$ = makeFORMAL($2,$1,NULL);)
 ;
```

The lists are naturally backwards.

Highlights from the JOOS parser:

```c
method : tPUBLIC methodmods returntype
tIDENTIFIER '(' formals ')' '{' statements '}'
{$$ = makeMETHOD($4,$2,$3,$6,$9,NULL;)
| tPUBLIC returntype
tIDENTIFIER '(' formals ')' '{' statements '}'
{$$ = makeMETHOD($3,modNONE,$3,$5,$8,NULL;)
| tPUBLIC tABSTRACT returntype
tIDENTIFIER '(' formals ')' :
{$$ = makeMETHOD($4,modABSTRACT,$3,$6,$8,NULL;)
| tPUBLIC tSTATIC tVOID
tMAIN '(' mainargv ')' '{' statements '}'
{$$ = makeMETHOD("main",modSTATIC,
makeTYPEvoid(),NULL,$9,NULL;)

whilestatement : tWHILE '(' expression ')' statement
{$$ = makeSTATEMENTwhile($3,$5;)
```

Notice the conversion from concrete syntax to abstract syntax that involves dropping unnecessary tokens.

Using backwards lists:

```c
typedef struct FORMAL {
 int lineno;
 char *name;
 int offset; /* resource */
 struct TYPE *type;
 struct FORMAL *next;
} FORMAL;

typedef struct FORMAL {
 int lineno;
 char *name;
 int offset; /* resource */
 struct TYPE *type;
 struct FORMAL *next;
} FORMAL;

void prettyFORMAL(FORMAL *f)
{ if (f=NULL) {
 prettyFORMAL(f->next);
 if (f->next=NULL) printf(",
 prettyTYPE(f->type);
 printf(" %s",f->name);
 }
}
```

What effect would a call stack size limit have?
The JOOS grammar calls for:
\[
\text{castexpression : } \left(\text{identifier}\right)\text{ unaryexpressionnotminus}
\]
but that is not LALR(1).

However, the more general rule:
\[
\text{castexpression : } \left(\text{expression}\right)\text{ unaryexpressionnotminus}
\]
is LALR(1), so we can use a clever action:
\[
\text{castexpression : } \left(\text{expression}\right)\text{ unaryexpressionnotminus}
\]
\[
\text{if ($2->\text{kind}!=\text{idK}) \text{ yyerror("identifier expected")};}
\]
\[
\text{ $$ = \text{makeEXPcast($2->\text{val.idE.name},$4);}$$;}
\]

Hacks like this only work sometimes.

LALR(1) and Bison are not enough when:
- our language is not context-free;
- our language is not LALR(1) (for now let’s ignore the fact that Bison now also supports GLR); or
- an LALR(1) grammar is too big and complicated.

In these cases we can try using a more liberal grammar which accepts a slightly larger language. A separate phase can then weed out the bad parse trees.

Example: disallowing division by constant 0:
\[
\text{exp : tIDENTIFIER}
\mid \\text{tINTCONST}
\mid \text{exp ' * ' exp}
\mid \text{exp ' / ' pos}
\mid \text{exp ' + ' exp}
\mid \text{exp ' - ' exp}
\mid \left(\text{ exp }\right)
\]
\[
\text{pos : tIDENTIFIER}
\mid \text{tINTCONSTPOSITIVE}
\mid \text{exp ' * ' exp}
\mid \text{exp ' / ' pos}
\mid \text{exp ' + ' exp}
\mid \text{exp ' - ' exp}
\mid \left(\text{ pos }\right)
\]

We have doubled the size of our grammar.
This is not a very modular technique.

Instead, weed out division by constant 0:
\[
\text{int zerodivEXP(EXP *e)}
\{/\text{ switch (e->kind) }\}
\]
\[
\text{ case idK:}
\]
\[
\text{ case intconstK:}
\]
\[
\text{ case timesK:}
\]
\[
\text{ case divK:}
\]
\[
\text{ case plusK:}
\]
\[
\text{ case minusK:}
\]

A simple, modular traversal.
Requirements of JOOS programs:

- All local variable declarations must appear at the beginning of a statement sequence:

```c
int i;
int j;
i=17;
int b; /* illegal */
b=i;
```

- Every branch through the body of a non-void method must terminate with a return statement:

```c
boolean foo (Object x, Object y) {
    if (x.equals(y))
        return true;
} /* illegal */
```

Also may not return from within a while-loop etc.

These are hard or impossible to express through an LALR(1) grammar.

Weeding bad local declarations:

```c
int weedSTATEMENTlocals(STATEMENT *s, int localsallowed)
{
    int onlylocalsfirst, onlylocalssecond;
    if (s!=NULL) {
        switch (s->kind) {
        case skipK:
            return 0;
        case localK:
            if (!localsallowed) {
                reportError("illegally placed local declaration",s->lineno);
            }
            return 1;
        case expK:
            return 0;
        case returnK:
            return 0;
        case sequenceK:
            onlylocalsfirst =
            weedSTATEMENTlocals(s->val.sequenceS.first,localsallowed);
            onlylocalssecond =
            weedSTATEMENTlocals(s->val.sequenceS.second,onlylocalsfirst);
            return onlylocalsfirst && onlylocalssecond;
        case ifK:
            (void)weedSTATEMENTlocals(s->val.ifS.body,0);
            return 0;
        case ifelseK:
            (void)weedSTATEMENTlocals(s->val.ifelseS.thenpart,0);
            (void)weedSTATEMENTlocals(s->val.ifelseS.elsepart,0);
            return 0;
        case whileK:
            (void)weedSTATEMENTlocals(s->val.whileS.body,0);
            return 0;
        case blockK:
            (void)weedSTATEMENTlocals(s->val.blockS.body,1);
            return 0;
        case superconsK:
            return 1;
        }
    }
}
```

Weeding missing returns:

```c
int weedSTATEMENTreturns(STATEMENT *s)
{
    if (s!=NULL) {
        switch (s->kind) {
        case skipK:
            return 0;
        case localK:
            return 0;
        case expK:
            return 0;
        case returnK:
            return 1;
        case sequenceK:
            return weedSTATEMENTreturns(s->val.sequenceS.second);
        case ifK:
            return 0;
        case ifelseK:
            return weedSTATEMENTreturns(s->val.ifelseS.thenpart) &&
            weedSTATEMENTreturns(s->val.ifelseS.elsepart);
        case whileK:
            return 0;
        case blockK:
            return weedSTATEMENTreturns(s->val.blockS.body);
        case superconsK:
            return 0;
        }
    }
}
```

The testing strategy for a parser that constructs an abstract syntax tree $T$ from a program $P$ usually involves a pretty printer.

If $parse(P)$ constructs $T$ and $pretty(T)$ reconstructs the text of $P$, then:

$$\text{pretty}(parse(P)) \approx P$$

Even better, we have that:

$$\text{pretty}(parse(pretty(parse(P)))) \equiv \text{pretty}(parse(P))$$

Of course, this is a necessary but not sufficient condition for parser correctness.