Developing the AST incrementally

Small steps: build a little, test a little

Objective

We need to develop a class hierarchy for the abstract syntax tree. We also need to add action routines to build the abstract syntax tree in parser.

This is too much to do in one step.

If we were developing the whole compiler incrementally, starting with a tiny core language, we'd build AST and parsing routines together in small bits. Instead, we have a full parser and no AST.

So we need to add bits of AST to a complete parser. How?

Order of development

We want small cycles of building and testing, e.g.,

add one new node type to the AST, add corresponding actions to the parser, test it.

But what does “test it” mean?

I want to print the structure so far, in human readable form.

Bottom up or top down?

Do I start by defining leaves of the tree, or the root of the tree? What do I do with parts that aren’t implemented yet?

Order of development: My choices

My tactics. Your mileage may vary.

- Let the AST get out a little ahead of the parser.
  - Sometimes I need to look ahead a couple steps to see if I'm on a reasonable path.
- Add to parser top-down.
  - I need a way to print results. The root of the AST needs to be transmitted back to the driver anyway, so it's easier to print an incomplete tree from the root than fragments below.
- Printable format: I chose json
  - Human readable (ish)
  - Easy to translate to other forms, e.g. for drawing
  - Might be useful for switching languages after this stage

Stub nodes

I want to build part of a tree, starting from root.

I need a way to stub out the missing parts.

```
class ASTNode {
public:
   virtual void json(std::ostream& out, AST_print_context& ctx) = 0;
   std::string str() { ... }
};

class Stub : public ASTNode {
   std::string name_;
public:
   explicit Stub(std::string name) : na   {name} {}
   void json(std::ostream& out, AST_print_c   & ctx) override;
};
```

Each concrete class will provide a method that emits a json representation

str() method is a simple wrapper to print the json

Parser can create Stub nodes at leaves of incomplete tree

Partial tree from the root

```
{ "kind" : "Program",
   "classes_" :
   { "kind" : "Classes","elements_" : [
      { "kind" : "Class",
         "name" :
         { "kind" : "Ident","text_" : "Pt"},
         "super" :
         { "kind" : "Ident","text_" : "Obj"},},
         ... ]},
   "statements_" :
   { "kind" : "Block","elements_" : [
      { "kind" : "Stub",
         "rule": "statement"},
      { "kind" : "Stub",
         "rule": "statement"},
      { "kind" : "Stub",
         "rule": "statement"}]},}
```

You may want to write some tiny, tiny programs to test your AST building, and/or write scripts to translate JSON into something more compact.

Factor out the json printing (as much as possible)

```
class ASTNode {
public:
   virtual void json(std::ostream& out, AST_print_context& ctx) = 0;
   std::string str() {
      std::stringstream ss;
      AST_print_context ctx;
      json(ss, ctx);
      return ss.str();
   }
protected:
   void json_indent(std::ostream& out, AST_print_context& ctx);
   void json_head(std::string node_kind, std::ostream& out, AST_print_context& ctx);
   void json_close(std::ostream& out, AST_print_context& ctx);
   void json_child(std::string field, ASTNode& child, std::ostream& out,
                                        AST_print_context& ctx, char sep=',');
};
```

Context object makes it easier to indent in a reasonable way. (Maybe the stream should be part of the context?)

Each concrete class needs its own json method, but they mostly just call these basic parts.

Example json printing method

```
void Assign::json(std::ostream& out, AST_print_context& ctx) {
   json_head("Assign", out, ctx);
   json_child("lexpr_", lexpr_, out, ctx);
   json_child("rexpr_", rexpr_, out, ctx, ' ');
   json_close(out, ctx);
 }
```

Additional json support in abstract base classes

```
void BinOp::json(std::ostream& out, AST_print_context& ctx) {
   json_head(opsym, out, ctx);
   json_child("left_", left_, out, ctx);
   json_child("right_", right_, out, ctx, ' ');
   json_close(out, ctx);
}

void Seq::json(std::ostream& out, AST_print_context& ctx) {
   json_head(kind_, out, ctx);
   out << "\"elements_\" : [";
   auto sep = "";
   for (ASTNode *el: elements_) {
      out << sep;
      el->json(out, ctx);
      sep = ", ";
   }
   out << "]";
   json_close(out, ctx);
```

In the parser:

Declare the possible types of tokens and non-terminals

```
%union {
   /* Tokens */
   int   num;
   char*  str;
   /* Abstract syntax tree values */
   AST::ASTNode* node;  // Most general class, for most nodes
   AST::Seq* seq;       // Where we need the 'append' method
   AST::Class* clazz;
   AST::Ident* ident;   // Identifiers are used in many places
}
```

This is bison code for C++. CUP and other parser generators do something similar, e.g., the Symbol type in CUP.

How many different node types? Fewer is simpler, but we need access to some methods while building.

In the parser:

Associate non-terminals with node types

```
%union {
   /* Tokens */
   int   num;
   char*  str;
   /* Abstract syntax tree values */
   AST::ASTNode* node;  // Most general class, for most nodes
   AST::Seq* seq;            // Where we need the 'append' method
   AST::Class* clazz;
   AST::Ident* ident;       // Identifiers are used in many places
}

%type <node> pgm
%type <seq> classes statements
%type <clazz> class class_sig
%type <ident> ident optExtends
```

My final version had many more alternatives in the %union, and many more %type clauses, because I needed to access class-specific fields and methods in static semantic checking.

As of Oct 29, my %union has expanded ...

```
%union {
   /* Tokens */
   int   num;
   char*  str;
   /* Abstract syntax tree values */
   AST::ASTNode* node;  // Most general class
   AST::Class* clazz;
   AST::Ident* ident;   // Identifiers are used in many places
   AST::LExpr* l_expr;
   AST::Load*  load;
   AST::Formal* formal;
   AST::Method* method;
   AST::Statement* statement;
   AST::Expr* expr;
   AST::Methods* methods;
   AST::Formals* formals;
   AST::Actuals* actuals;
   AST::Block* block;
   AST::Classes* classes;
   AST::Type_Alternatives* type_alternatives;
}
```

Probably still not the final version

Sometimes an action builds a node in the obvious way

```
ident: IDENT { $$ = new AST::Ident($1); } ;

class_sig: CLASS ident '(' args ')' optExtends
    { $$ = new AST::Class(*$2, *$6); }
    ;
```

Work in progress ... I didn't have the part of the AST for arguments yet. Final version will also need *$4 here and a corresponding field in the constructor for AST::Class

But this is an abstract syntax tree … parts don't follow the syntax exactly

```
statements: statements statement
                    { $$ = $1;
                      $$->append(new AST::Stub("statement"));
                     }
  | /* empty */     { $$ = new AST::Block(); }
  ;
```

This production is the base case. It will always be reduced before the production that adds a statement to the sequence.

New users of LR parsers are often surprised by this, but if you work it through you'll see it.

The grammar may require adjustment

```
class: class_sig class_body     ;

class_sig: CLASS ident '(' args ')' optExtends    ;

class_body:  '{' statements methods '}' ;
```

Oops ... I want to treat the first statements as a constructor method. It will be easier if I rearrange the grammar to match the class signature and statements together.

Refactored to make AST building easier ...

```
class: CLASS ident '(' formals ')' optExtends
     '{' statements methods '}'
   {  AST::ASTNode* constructor = new AST::Method(*$2, *$4, *$2, *$8);
      $$ = new AST::Class(*$2, *$6, *constructor, *$9);
   }
   ;

optExtends: EXTENDS ident { $$ = $2; } ;
optExtends: /* empty */ { $$ = new AST::Ident("Obj"); } ;
/* omitted 'extends' is equivalent to 'extends Obj' */
```

Removed ‘class_sig’ because it made building the constructor clumsy.

More syntactic sugar: Omitting the ‘extends’ is like declaring ‘extends Obj’.

Big picture, small steps

Have an overall plan in mind.

Work out some details where you're not sure.

Proceed in small steps:

- Add the AST classes you need (sometimes none)
- Add just enough parser code to test your addition
- If needed, write a small Quack program to test your addition
- Test and fix before continuing

Don't be afraid to refactor and improve to avoid very redundant code.

Alternatives

We could create a completely generic AST

- Just one node type ... with a list of children, and an attached table of potential operations, or with operations in functions rather than methods.
- Very simple to print. Less red tape in the parser. Less error checking by the C++/Java/etc type system, but maybe ok?

We could write (or generate!) an AST with different classes for each non-terminal, following the grammar exactly.

- Which might be ok if we had powerful tools for tree transformation ... we could write tools to transform parse tree to AST, and to do other work.

No clear “best” way to build an AST.

What do others do?

There are a lot of open source interpreters and compilers. It can be useful to look at them. Here's a bit of CPython's Grammar source ...

```
stmt: simple_stmt | compound_stmt
simple_stmt: small_stmt (';' small_stmt)* [';'] NEWLINE
small_stmt: (expr_stmt | del_stmt | pass_stmt | flow_stmt |
             import_stmt | global_stmt | nonlocal_stmt | assert_stmt)
expr_stmt: testlist_star_expr (annassign | augassign (yield_expr|testlist) |
                    ('=' (yield_expr|testlist_star_expr))*)
annassign: ':' test ['=' test]
testlist_star_expr: (test|star_expr) (',' (test|star_expr))* [',']
augassign: ('+=' | '-=' | '*=' | '@=' | '/=' | '%=' | '&=' | '|=' | '^=' |
            '<<=' | '>>=' | '**=' | '//=')
```

CPython uses an LL(1) parser

... and the grammar has been tweaked to work with it.

```
# The reason that keywords are test nodes instead of NAME is that using NAME
# results in an ambiguity. ast.c makes sure it's a NAME.
# "test '=' test" is really "keyword '=' test", but we have no such token.
# These need to be in a single rule to avoid grammar that is ambiguous
# to our LL(1) parser. Even though 'test' includes '*expr' in star_expr,
# we explicitly match '*' here, too, to give it proper precedence.
# Illegal combinations and orderings are blocked in ast.c:
# multiple (test comp_for) arguments are blocked; keyword unpackings
# that precede iterable unpackings are blocked; etc.
argument: ( test [comp_for] |
        test '=' test |
        '**' test |
        '*' test )
```

from https://github.com/python/cpython/blob/master/Grammar/Grammar

CPython's AST is generated from ASDL

```
module Python
{
  ...
 stmt = FunctionDef(identifier name, arguments args,
                    stmt* body, expr* decorator_list, expr? returns)
          ...
          | ClassDef(identifier name,
            expr* bases,
            keyword* keywords,
            stmt* body,
            expr* decorator_list)
          | Return(expr? value)

          | Delete(expr* targets)
          | Assign(expr* targets, expr value)
          | AugAssign(expr target, operator op, expr value)
          -- 'simple' indicates that we annotate simple name without parens
          | AnnAssign(expr target, expr annotation, expr? value, int simple)
```

Looking forward

Every AST class will need a type-checking method

(or cases in your type-checking code)

So simpler is better!

Every AST class will need a code generation method

or cases in code generation code ... again simpler is better

But differentiating AST node classes has some benefits

in error checking (building a malformed tree)

in factoring out parts that really do need different treatment

Consider your AST for now as “best guess at what I’ll need”

You may make adjustments later