Today we talk about ASTs (abstract syntax tree and what we do with parser)
What are the parsers actually doing?
Parser takes a stream of tokens as input, figures out what the grammatical structure it has
Derivation: figure out if a string is a part of the language.
Top down parsing one strategy to do this job.
There is an algorithm to create parse trees: we construct parse tree, where each node is a token of the language.
But our parser constructs an AST

**Parse Trees vs ASTs:**

Parse tree: we take every production we derive our string
AST advantages:

- Abstracts away the unnecessary details (just a design decision.). All the punctuation, keywords, etc goes away. The useful data stays.
- It uses data structures such as lists, unions, struct, etc.
- It uses simpler expression tree (no need to keep order of operations)

ast.yml : specification language (included in your project repository)
Just like our PL0 grammar defines a tree structure,

Remember EBNF notation, which is used while defining PL0 grammar (grammar.md):
Take a look at pl0 context free grammars. The terminals are all in capital letters and they are tokens in the language, and nonterminals are in uppercase letters. Like any other CFGs you got productions.
“::=” production
“{ }” : zero or more
“[ ]” : zero or one

See ast.md
It takes each one of the productions of the language, tells what abstract syntax tree node gets created.

**Example of parse tree and AST:**

Let’s say we have string “1 + 0 * v” which corresponds to the following series of tokens:
“NUMBER PLUS NUMBER MULT IDENT”

**Part of our grammar for PL0**

```
expr         ::= simpleexpr [ relop simpleexpr ]
simpleexpr   ::= term [ termop term ]
term         ::= factor [ factorop factor ]
factor       ::= IDENT [ LPAREN exprlist RPAREN ] | NUMBER | TRUE |
FALSE | LPAREN expr RPAREN | NOT factor | MINUS factor
```

**The parse tree for “1 + 0 * v”:**

```
                        expr
                         |
                     simplexpr
                    /    |    \
                term   termop  term
                 /       |      |      \        \
             factor    PLUS    factor factorop factor
               /         |       |       |       |
           NUMBER        |      NUMBER   MULT   IDENT
             /           |       |       |       |
            1            +       0       *       v
```

**AST definition for the string “1 + 0 * v”:**

```
           BinaryExpression
          /      |         \
NumberFactor   PLUS        BinaryExpression
        |        |        /        |       \
        |        |  NumberFactor  MULT    VariableFactor
        |        |        |        |        |
        1        +        1        *        v
```

**Notes about ASTs:**

- AST is cleaner in design
    - Removes punctuation and keywords. With AST, you remove things that you know that you are not going to need. It is much more easier to deal with less number of nodes and as you can see in the AST for “1 + 1 * v”, we have much less nodes in the AST when compared to the parse tree.
    - Much more memory efficient
    - Instead of tree structures, we can use other data structures also, such as unions, lists
- Semantic actions: for each production, we attach a piece of code
- Each nonterminal has a value
- Each production action updates values
- For variables, instead of having an entire parse tree, we can represent them as a “list” of typed identifiers
- In AST, we don’t need a different type for each of our productions in the language
- All expressions are shown as binary expressions or unary expressions
- It is also the practice in industry to use AST rather than parse tree
- You will see why AST is better when you get into the harder phases of the compiler

**Tree** and **recursion** are closely related.
Tree traversals are simply recursive functions.

For example, say we have a very simple function for binary tree traversal:
Every recursion function has two recursive calls, we also need base case. The recursive calls are describing the right and left child, and the base case is describing the leaves of the tree.

```
rec(node):
                              ← preorder traversal processing func call comes here
  rec(node.left)
                              ← inorder traversal processing func call comes here
  rec(node.right)
                              ← postorder traversal processing func call comes here
```

Pre means before. Hence, you put the processing function before the recursive calls if you are doing pre-order traversal. Post is after, hence, after recursive calls. In is in between, hence, in between recursive calls if inorder traversal.

**TODOs:**

- Go over the bonus homework to better understand ASTs and parse trees.
- See syllabus repository for parser outputs: syllabus/project/tests/*.ast
  Parser will output abstract syntax tree (files of format .ast)
  The idea is the same with parse trees but less nodes to deal with.
  You don't see VAR, you don't see keywords, or punctuations, you only see the things you will need in the future.
  When you build large systems, this kind of designs will be very helpful.

**Parser code:**

- parser.c is the only file you need to complete.
- You can compile parser code using `./compiler --parse file.pl0 > file.ast`
- If your lexer does not work properly, your parser will not work because of the lexer.
  However, you still have the working version of lexer: lexer.o
  To do that, download binaries.tar, extract files, copy lexer.o to your project folder.
  You can use **cp** "lexer.o in binaries directory" "your project directory" to copy the directory via terminal. Then, your compiler will be compiled with lexer.o instead of your lexer.c file. Run "make" to compile your project.
  Remember recursive descent parsers. In our parser, we will again have functions for each non-terminal. See them at the beginning of parser.c
- So you need to check the tokens. block() function calls next token. You can match the token to see if the token is the one which you are looking for.
- See the given part of the file to understand how to complete the rest. See the pattern in the examples. You need to follow the similar pattern.

- We are not going to **only** call functions and consume terminals, we are also going to construct the AST in parser.c. This is the difference when compared to recursive descent parser.
- Helper functions to create AST nodes:
  newNAME_OF_THE_AST_NODE()
  For example, you have newTranslationUnit() and newBlock() functions
- Each one of the parsing functions are going to return the corresponding AST node.
- Each function is going to use the return values from parsing functions to fill its children
  When we call the functions for children node, we basically recognize it.
  When we create an AST node and fill its children by using the return values of the functions called, we construct our AST tree.
  For the implementation of block:

```
struct Block * node = newBlock(); ← here we create an empty block node
node->vardecls = vardecls();      ← here we use the return value of vardecls()
                                    to fill vardecls child of block
node->funcdecls = funcdecls();
        ...
        …
return node;
```

- Using the grammar, you need to implement the bodies of the functions given you as the template
- Always first advance to the next token: the invariant for our functions
- We used fgetc() in lexer to advance to the next character. In parser, we will use next() function to advance to the next token. Also, we can go back in parse to recover.
- ensure_token(TOKEN_NAME): this function does the error handling for you. By using that, you can “ensure” the type of the current token. Do not assume the type of the token. Always use ensure_token() function.
- token() gets you the current token
- The non-terminals are responsible for advancing the tokens for themselves.
- Read the documentation to further understand your helper functions and the strategy suggested you to use.
- First, use a function assuming that it is implemented correctly, even when you did not implement the function. Then, you will implement the functions that you assumed to be implemented.
- We use previous() to back up the token. If we go too far, we can go backwards in stream of tokens by using previous().

**Coding part for today:**

```
// block          ::= vardecls funcdecls statement
static struct  Block *block() {
  // given
  struct Block *node = newBlock();

  node->vardecls = vardecls();
  node->funcdecls = funcdecls();
  node->statement = statement();

  return node;
}

// vardecls       ::= { VAR IDENT COLON type }
static struct  TypedIdentList *vardecls() {
  // given
  struct TypedIdentList *list = newTypedIdentList();

  next();
  while (is_token(VAR)) {
          struct TypedIdent *node = newTypedIdent();

          next();
          ensure_token(IDENT);
          node->identifier = token()->identifier;

          next();
          ensure_token(COLON);

          node->type = type();

          next();

          addTypedIdent(list, node);
  }

  previous();

  return list;
}
```