Project 1 Related; Static Semantics Overview[1]

[1]From material by R. Bodik and P. Hilfinger

Answer to Question from Last Lecture

Q: Are there languages where it's impossible to implement Bison-style `%left`/`%right` precedence directives in a pure context-free grammar (one that doesn't use these directives)? That is, are tt %left and `%right` a necessity or a convenience?

A: Yes and no. Given, for example, the grammar

```
%left '-'
%%
e : ID | e '-' e ;
```

it is **not** possible to get the same parse tree from a grammar without precedence declarations.

However, there is always a grammar that accepts the same language as one that uses such declarations. The nodified LR(1) machine that is created using these declarations is still a "*push-down automaton*", and there is an equivalence between push-down automata and (pure) context-free grammars, just as there is between FSAs and regular grammars.

Furthermore, one can use semantic actions to get the desired ASTs (or other output) from a "pure" BNF grammar.

Answer to Question from Last Year

Q: What is an example of an unambiguous, non-LR grammar?
A: There are many, but consider

```
A ::= /* empty */
    | 'x' A 'x'
    | 'y' A 'y'
    ;
```

- This is the language $\{ww^R \mid w \in \{\mathrm{x}, \mathrm{y}\}\}$, where w^R is the reverse of w.
- It is unambiguous, since there is only one derivation for any string in the language.
- But it is not LR(k) for any k. (How can you see this?)
- In fact, there is no alternative grammar for this language that is LR(k)!

Project 1 Related

CUP/JFlex interface

- Lexer communicates syntactic categories of tokens as integers.
- These may be defined in the CUP file as symbolic constants (in `terminal` declarations).
- They are converted to Java constants in the generated class

  ```
  chocopy.pa1.ChocoPyTokens
  ```

  which the lexer can then use.
- The lexer bundles syntactic values, semantic values, and source locations into objects of type `java_cup.runtime.Symbol`, which it returns to the parser.
- The `terminal` and `non terminal` declarations in the CUP file tell what types of semantic value the declared symbols have: both from lexical actions (for terminals) and parser actions (nonterminals).

Lexer Features

- In lexical actions, `yytext()` is a Java string containing the matched token, and `yylength()` is its length.
- Actions that execute **return** cause the lexer to deliver a token (a `Symbol`).
- Actions that don't return indicate tokens that are skipped.
- It's always the action of the longest match that gets chosen (or the first in case of ties). As a result,

```
"for"                  { return symbol(ChocoPyTokens.FOR); }
[A-Za-z][A-Za-z0-9]*  { return symbol(ChocoPyTokens.IDENTIFIER,
                                            yytext()); }
```

  will return `FOR` for the input "`for`" and `IDENTIFIER` for the input "`forage`," just as is usually intended.
- And in the case of "`forage`," the lexer will also include additional semantic information: the text of the identifier itself.

Lexer Features: Macros

- You can define abbreviations ("macros") above the first %% in the lexer file for use in patterns, as in

```
ALPHA = [a-zA-Z_]
ALNUM = [a-zA-Z_0-9]
```

  which allows you to write

```
{ALPHA}{ALNUM}* { rule for ID; }
```

- Use this to simplify and clarify your actions.

Lexer Features: Using Java Directly

- The converted JFlex program is a Java program. The actions are general Java statements. Use this for "special effects", such as keeping track of indentation levels.
- The Chocopy lexical structure has been considerably simplified from Python's, so that you don't have to worry about continuation lines.
- However, if you did want to follow full Python's rules, you'd need to keep track of when you are in the midst of a bracketed construct (`'(...'`, `'[...]'`, `'{...}'`), because in those cases, newlines behave like spaces.
- Expedient solution: keep a bracket count in a variable and test in the lexical action for `"\n"` to decide whether to return a NEWLINE token.
- For indentation, you'll presumably need some sort of stack to keep track of valid levels of indentation and deal with them at the beginnings of lines.

Lexer Start States

- The lexer is essentially a DFSA that starts over in some initial state whenever the lexer's `next_token` method is called. You can define alternative starting states in this DFSA with `%state` declarations above the first %%, as in

  ```
  %state SPECIAL
  ```

- This says that patterns or groups of patterns that start with `<SPECIAL>` match only when the lexer starts the machine in state `SPECIAL`, and in that state, other patterns do not match.
- In actions, one can change the start state for subsequent calls of the lexer with the call

  ```
  yybegin(SPECIAL);
  ```

  to make `SPECIAL` the start state. Initially, the starting state is `YYINITIAL`.

Example

One way to handle C-style comments might be this:

```
%state COMMENT

%%

<YYINITIAL> {
   rules to use when not in a comment
   "/*"     { yybegin(COMMENT); /* But don't return yet. */ }
}
<COMMENT> {
   "*/"     { yybegin(YYINITIAL); /* Don't return yet. */ }
   [^]      { /* Matches any character.  We still don't return. */ }
}
```

Indentation and Matching Nothing

- The start-state feature can be useful when implementing `INDENT` and `DEDENT`, but we leave it to you to figure out how.
- You are likely to face one particular problem in addition: If you have a pattern intended to match indentation, it might have to match *empty* indentation (say, at the beginning of the program).
- Unfortunately, JFlex patterns won't match empty strings.
- Fortunately, there is a ~~kludge~~ useful feature: you can contrive for a pattern to match *too much* text and then return excess text to the lexer to be reprocessed.
- In lexical actions, the call `yypushback(`N`)` will return the last N matched characters from `yytext()` to the lexer.
- We leave it to you to see where this might be helpful.

Parser Points

- Keep semantic actions simple. For the most part, you don't need much other than, e.g.,

```
statement: RETURN:r expr:e  {: RESULT = new ReturnStmt(rxleft, exright, e); :}
```

- Here, `rxleft` is "the start of the symbol labeled 'r' " and `exright` is "the end of the symbol labeled 'e' ".
- Feel free to introduce new supporting functions in the `parser code` and `action code` sections.

General Advice

- *Read the Project Documentation:* there actually is useful information there!
- *Read the Skeleton:* it gives some clues and contains work you need not do.
- *Read the Tool Documentation:* The manuals for JFlex and CUP are online.
- *Write Test Cases:* Yes, there are already some there, but it would be good to think about how to write such a test suite (and don't forget that we are holding back some tests until the deadline).
- *Use GIT:* Commit often (I used 130 commits to change the preceding solution to the Spring 2019 solution). Learn how to coordinate with your partners.
- *Meet Regularly With Your Team.* Have a clear idea of what everyone's job is.

Limitations of Context-Free Grammars

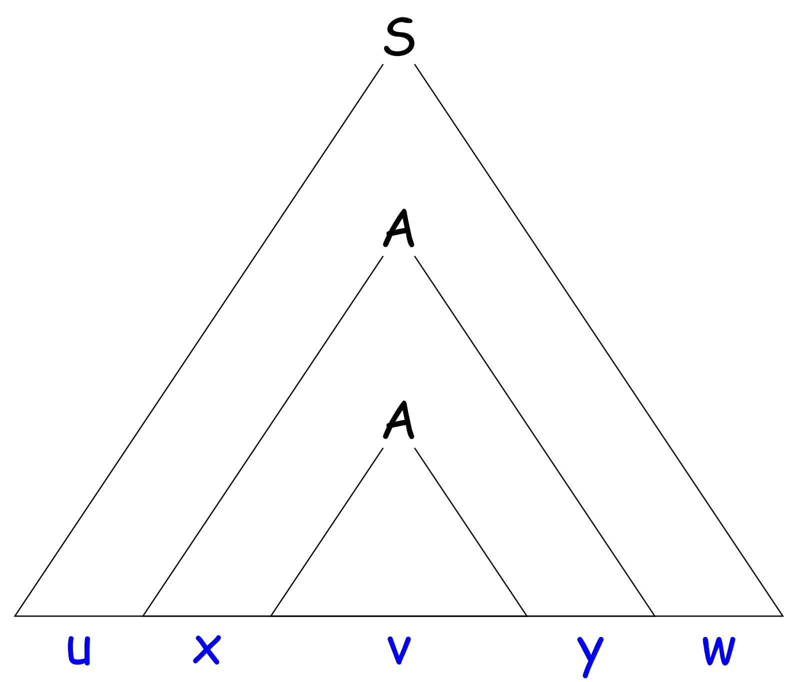
- Not all languages are context free, describable in BNF.
- This follows from an analogue of the pumping lemma for regular languages:

  The uxvyw Theorem: For every context-free grammar, G, there is some size k, such that for any string $s \in L(G)$ with $|s| \geq k$, $s = uxvyw$ where
  + $|xy| > 0$,
  + $|xvy| \leq k$, and
  + $ux^n vy^n w \in L(G)$ for all $n \geq 0$.

- So beyond a certain size, a string in the language can be "pumped" with an arbitrary number of copies of two bracketing strings.

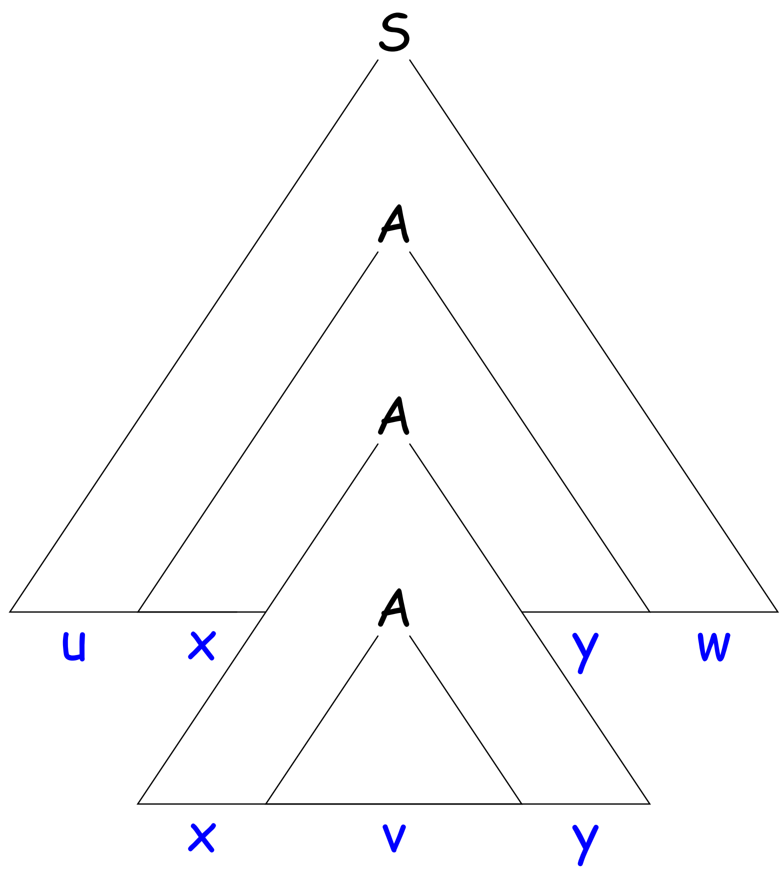
Proof Sketch

The proof involves considering parse trees. In order to parse some sufficiently large string, there must be recursion, resulting in a tree shaped like this, where the A producing v is a lowest A in the tree:



Proof Sketch (II)

By the context-free property, we can derive another string by replacing the lower A with a copy of the upper one:



...and by continuing in this fashion, pump in an arbitrary number of copies of x and y.

Overview

- Lexical analysis
  - Produces tokens
  - Detects & eliminates illegal tokens
- Parsing
  - Produces trees
  - Detects & eliminates ill-formed parse trees
- Static semantic analysis $\Longleftarrow$ *we are here*
  - Produces *decorated tree* with additional information attached
  - Detects & eliminates remaining static errors

Static vs. Dynamic

- We use the term *static* to describe properties that the compiler can determine without considering any particular execution.
  - E.g., in

    ```
    def f(x) : x + 1
    ```

    Both uses of x refer to same variable
- Dynamic properties are those that depend on particular executions in general.
  - E.g., will `x = x/y` cause an arithmetic exception?
- Actually, distinction is not that simple. E.g., after

  ```
  x = 3
  y = x + 2
  ```

  compiler *could* deduce that x and y are integers.
- But languages often designed to require that we treat variables only according to explicitly declared types, because deductions are difficult or impossible in general.

Typical Tasks of the Semantic Analyzer

- Find the declaration that defines each identifier instance
- Determine the static types of expressions
- Perform re-organizations of the AST that were inconvenient in parser, or required semantic information
- Detect errors and fix to allow further processing

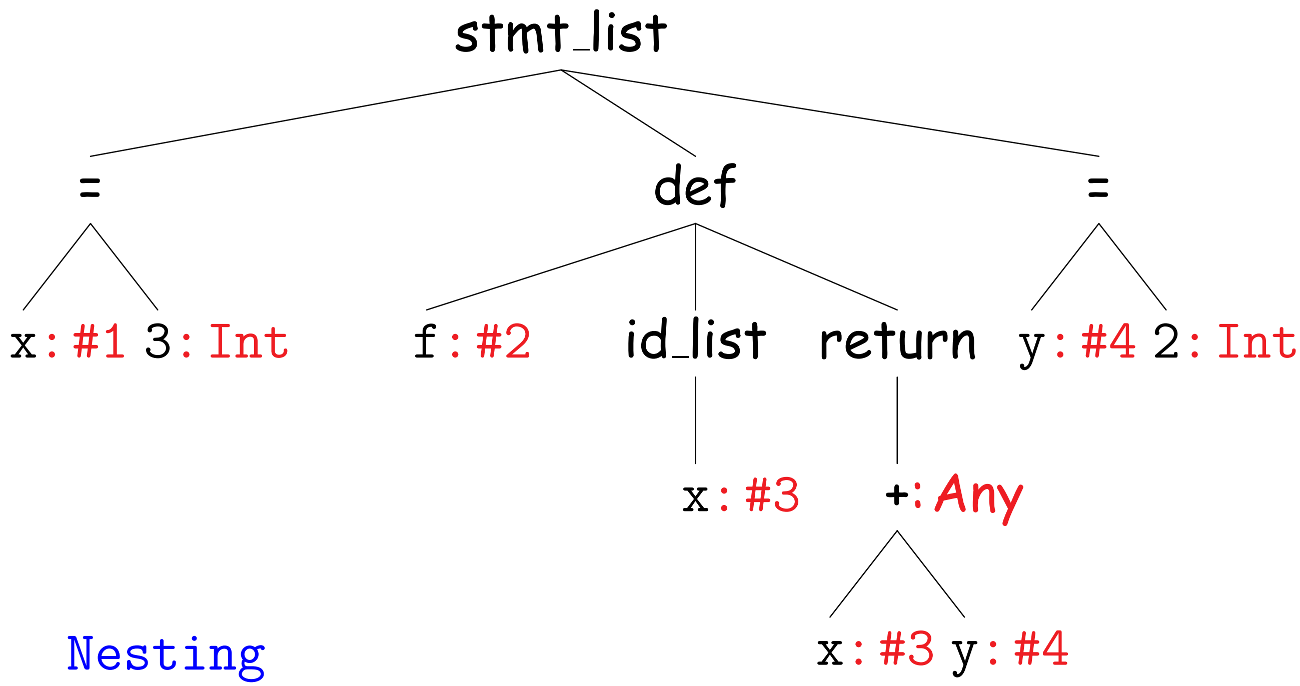
Typical Semantic Errors: Java, C++

- Multiple declarations: a variable should be declared (in the same region) at most once
- Undeclared variable: a variable should not be used without being declared.
- Type mismatch: e.g., type of the left-hand side of an assignment should match the type of the right-hand side.
- Wrong arguments: methods should be called with the right number and types of arguments. Actually subset of type mismatch.
- Definite-assignment check (Java): conservative check that simple variables assigned to before use.

Output from Static Semantic Analysis

Input is AST; output is an *annotated tree:* identifiers decorated with declarations, other expressions with type information.

```
x = 3
def f (x):
   return x+y
y = 2
```



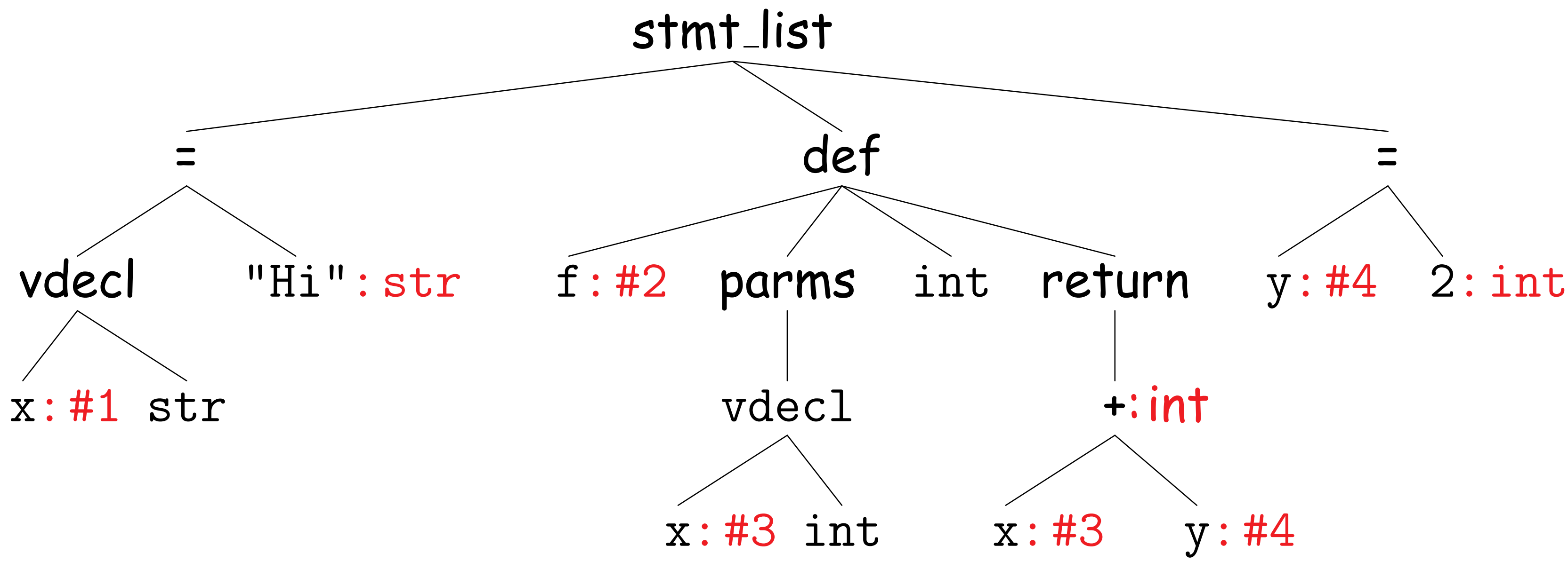
	Id	Type	Nesting
#1:	x,	Any,	0
#2:	f,	Any->Any,	0
#3:	x,	Any,	1
#4:	y,	Any,	0

Output from Static Semantic Analysis (II)

- Analysis has added objects we'll call *symbol entries* to hold information about instances of identifiers.
- In this example, `#1: x, Any, 0` denotes an entry for something named 'x' occurring at the outer lexical level (level 0) and having static type Any.
- For other expressions, we annotate with static type information.
- These symbol entry decorations might be attached directly to the AST or stored separately in symbol tables and looked up: it's all a matter of representation.

Output from Static Semantic Analysis for Chocopy

Chocopy (like Java, C++) is *statically typed*, so we can have more specific information in symbols.




```
x: str = "Hi"
y: int = 2
def f(x: int) -> int:
   return x+y
```

	Id	Type	Nesting
#1:	x,	str,	0
#2:	f,	int->int,	0
#3:	x,	int,	1
#4:	y,	int,	0

Output from Static Semantic Analysis: Classes

- In Python (dynamically typed), can write

```
class A(object):
   def f(self): return self.x

a1 = A(); a2 = A()   # Create two As
a1.x = 3; print a1.x # OK
print a2.x           # Error; there is no x
```

  so can't say much about attributes (fields) of A.

- In Java, C, C++ (statically typed), analogous program is illegal, even without second print (the class definition itself is illegal).
- So in statically typed languages, symbol entries for classes would contain dictionaries mapping attribute names to types.

Scope Rules: Binding Names to Symbol Entries

- *Scope of a declaration:* section of text or program execution in which declaration applies
- *Declarative region:* section of text or program execution that bounds scopes of declarations (we'll say "*region*" for short). (Others use the term "scope" for what I'm calling a declarative region. I use a separate term, since I think it is a distinct concept.)
- If scope of a declaration defined entirely according to its position in source text of a program, we say language is *statically scoped.*
- If scope of a declaration depends on what statements get executed during a particular run of the program, we say language has *dynamically scoped.*

Scope Rules: Name$\Longrightarrow$Declaration is One-to-Many

- In most languages, can declare the same name multiple times, if its declarations
  - occur in different declarative regions, or
  - involve different kinds of names.
  - Examples from Java?, C++?

Scope Rules: Nesting

- Most statically scoped languages (including C, C++, Java) use:

  Algol scope rule: Where multiple declarations might apply, choose the one defined in the *innermost* (most deeply nested) declarative region.

- Often expressed as "inner declarations *hide* (or *shadow*) outer ones."
- Variations on this: Java disallows attempts to hide local variables and parameters.

Scope Rules: Declarative Regions

- Languages differ in their definitions of declarative regions.
- In Java, variable declaration's effect stops at the closing '}', that is, each function body is a declarative region.
- What others?
- In Python, modules, function headers and their bodies, lambda expressions, comprehensions (of lists, sets, and dictionaries) and *generator expressions* make up declarative regions, but nothing smaller. Just one `x` in this program:

```
def f(x):
    x = 3
    for x in range(6):
        print(x)
    print(x)
```

It prints 0–5 and then 5 again.