Predictive Parsing

Dr William Harrison

Spring 2017

HarrisonWL@missouri.edu

CS4430/7430 – Compilers I

Announcement

- Midterm 1 on March 1 (Wednesday).

Today

- Continuing the second phase
  - "parsing" or grammatical analysis
    - discovers the real structure of a program and represents it in a computationally useful way
- "Predictive" parsing
  - also called "recursive descent" parsing

Parsing so far…

- A **language** is a set of strings
- Some languages may be described with a **context-free grammar**
  - Terminals: tokens from the lexer
  - Non-terminals: have production rules in our grammar
- A **parser** for a grammar/language determines whether a string belongs to a language or not
  - Parsing discovers a **derivation** (if one exists).
  - This derivation will let us build our **parse tree**.
- Grammars can be **ambiguous**
  - Admit several valid parses
  - Can transform grammar to
    - remove ambiguity (if necessary)
    - make it easier to parse

Review: Simple CFG

$S \rightarrow$ **`if`** E **`then`** S **`else`** S
$S \rightarrow$ **`begin`** S L
$S \rightarrow$ **`print`** E

$L \rightarrow$ **`end`**
$L \rightarrow$ **`;`** S L

$E \rightarrow$ num **`=`** num

Review: Derivation

S → **`if`** E **`then`** S **`else`** S
S → **`begin`** S L
S → **`print`** E

L → **`end`**
L → **`;`** S L

E → num **=** num

S → **`begin`** S L
→ **`begin print`** E L
→ **`begin print 1=1`** L
→ **`begin print 1=1 end`**

Review: Derivation

S → **if** E **then** S **else** S
S → **begin** S L
S → **print** E

L → **end**
L → **;** S L

E → num **=** num

S → **begin** S L
→ **begin print** E L
→ **begin print 1=1** L
→ **begin print 1=1 end**

∴ this string is in language(S)

Review: Parse Trees from Derivations

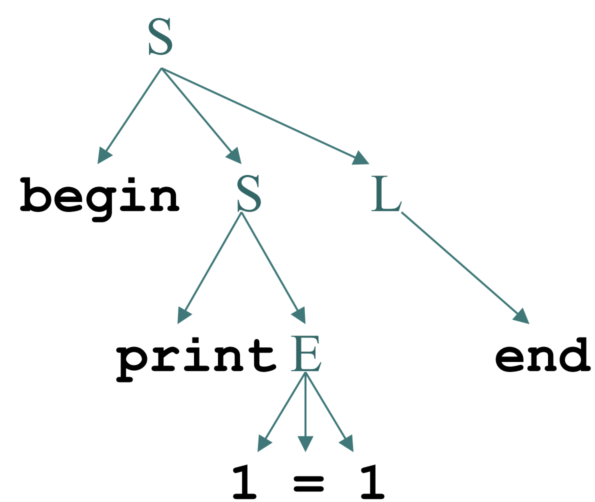
$S \rightarrow$ **if** E **then** S **else** S
$S \rightarrow$ **begin** S L
$S \rightarrow$ **print** E

$L \rightarrow$ **end**
$L \rightarrow$ **;** S L

$E \rightarrow$ num **=** num

$S \rightarrow$ **begin** S L
$\rightarrow$ **begin print** E L
$\rightarrow$ **begin print 1=1** L
$\rightarrow$ **begin print 1=1 end**

Parse Tree Associated with Derivation



The Process of Constructing Parsers

1. **First** define the language syntax
   - i.e., both it's lexical and grammatical syntax
     - produce a CFG capturing the language designer's intent
2. **Important:** Depending on the form of the CFG, there are different techniques and tools for producing parsers
   - These grammar forms are called “grammar hierarchies” or “language classes” in the literature
3. **If** a grammar is in one of these forms, then you're constructing a parser is straightforward (and you're mostly done)
   - **otherwise,** go to step 1.

Two (of many possible) Forms of Derivation

Leftmost derivation

- Always expand the leftmost non-terminal

S	*S → S ; S*
<u>S</u> ; S	*S → id := E*
id := <u>E</u> ; S	*E → num*
id := num ; <u>S</u>	*S → id := E*
id := num ; id := <u>E</u>	
…	

Rightmost derivation

- Always expand the rightmost non-terminal

S	*S → S ; S*
S ; <u>S</u>	*S → id := E*
S ; id := <u>E</u>	*E → E + E*
S ; id := E + <u>E</u>	*S → (S , E)*
…	
id := num ; id := E + (S , <u>E</u>)	
…	

Grammar Hierarchies: LL(k) vs. LR(k)

- **L**eft to Right parse (no backtracking)
- **L**eftmost derivation
- ***k***-token look ahead

  ➔ LL(k)

- Needs to predict what production to use after seeing only *k* tokens
- Both hand-written (recursive descent) and built by tools.
- Considered faster to write, but may be less efficient
- Recently seen a renaissance.
  - ANTLR and javacc for Java
- May need to tweek original grammar

- **L**eft to Right parse
- **R**ightmost derivation
- ***k***-token look ahead

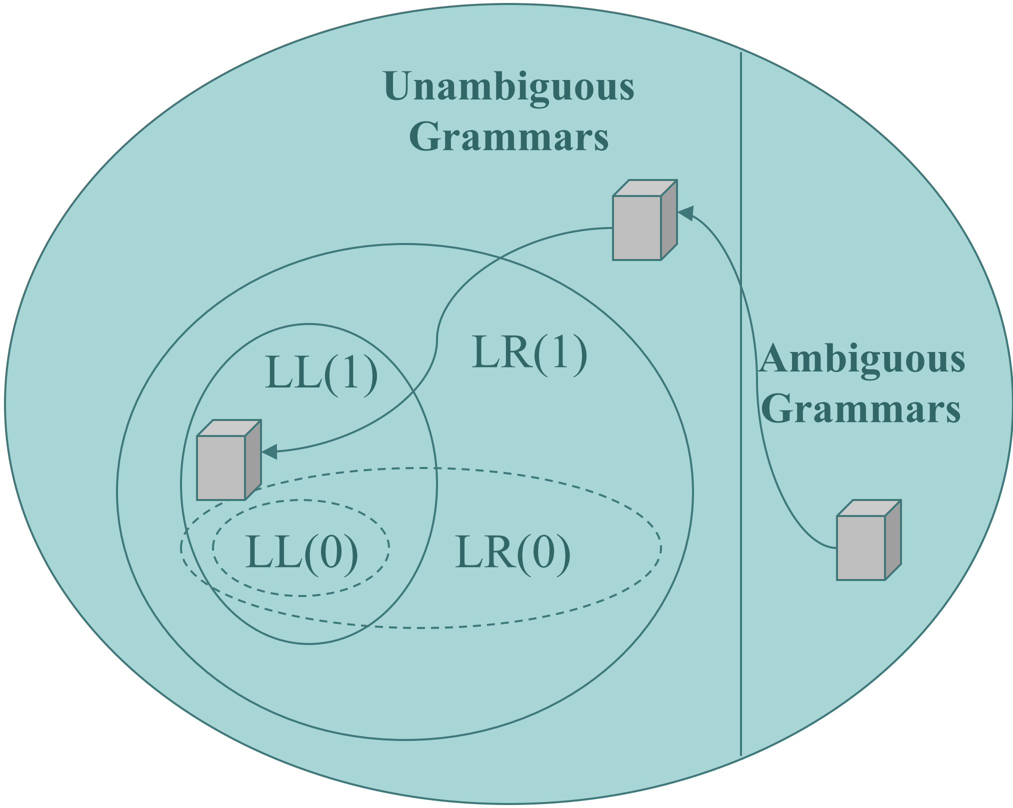
  ➔ LR(k)

- Can see the input corresponding to a specific non-terminal (and *k* tokens after) before needing to choose which production to use.
- Typically built by tools.
- Most popular for "real" parsing
  - YACC, CUP for Java, sablecc
- Also may need to tweek original grammar

Grammar Hierarchies

- Grammars describe languages
- Parsers for grammars accept/ reject
- Compiler engineers rework grammars
  - From ambiguous to unambiguous
  - Into the chosen grammar class.
- Set of all languages is partitioned by grammar classes

The Set of All Languages



Predictive Parsing

The first token on the RHS of each rule is unique.

$S \rightarrow$ **if** E **then** S **else** S
$S \rightarrow$ **begin** S L
$S \rightarrow$ **print** E

$L \rightarrow$ **end**
$L \rightarrow$ **;** S L

$E \rightarrow$ num **=** num

- a.k.a. "recursive descent"
- If the "left-most" symbol on the r.h.s. of the productions is unique, then this grammar is LL(1).

LL(1) Parsers

- Also called "predictive" or "recursive descent" parsing.
- Has one function for each non-terminal, and one clause for each production.

```
int tok = getToken();
void advance() { tok = getToken(); }
void eat(int t) {
        if (tok = t) { advance() } else { error(); }
}
void S(){switch(tok) {
  case IF: eat(IF); E(); eat(THEN); S();
           eat(ELSE); S(); break;

  case BEGIN: ...
}
void E() {
  eat(NUM); eat(EQ); eat(NUM);
}
...
```

S → **if** E **then** S **else** S
S → **begin** S L
S → **print** E

L → **end**
L → **;** S L

E → num **=** num

LL(1) Parser + AST creation

- Has one function for each terminal, and one clause for each production.
  - Returns the AST

```
int eatNUM() {
  if (tok = NUM) { int i = getCurrTokValue(); advance() ; return i; }
  else { error(); }
}
S parseS(){switch(tok) {
  case IF: eat(IF); E e = parseE();
           eat(THEN); S s1 = parseS();
           eat(ELSE); S s2 = parseS();
           return new S_IF(e,s1,s2)
  case BEGIN: …
}
E parseE() {
  int i1 = eatNUM();
  eat(EQ); int i2 = eatNUM();
  return new E_EQ(i1,i2);
}
…
```

S → if E then S else S
S → begin S L
S → print E

L → end
L → ; S L

E → num = num

Shortcomings of LL Parsers

- Recursive descent renders a readable parser.
  - depends on the first terminal symbol of each sub-expression providing enough information to choose which production to use.
- But consider a rec. des. parser for this grammar

E -> E + T
E -> T
T -> id

```
void E(){switch(tok) {
  case ?: E(); eat(TIMES); T(); ← no way of choosing production
  case ?: T();
  …
}
void T(){eat(ID);}
```

Table-driven Parsing

- Introduce table driven parsing
  - this is just a common implementation technique
    - that's more efficient than the procedure-driven style we've seen previously

FIRST(γ)

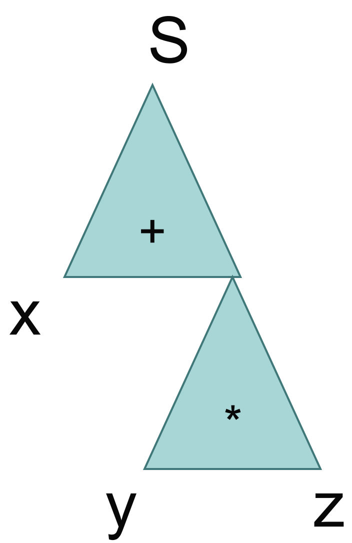
- γ is a sequence of symbols (terminal or non-terminal)
  - For example, the right hand side of a production
- FIRST returns the set of all possible terminal symbols that begin any string derivable from γ.
- Consider two productions $X \rightarrow \gamma_1$ and $X \rightarrow \gamma_2$
  - If FIRST (γ_1) and FIRST (γ_2) have symbols in common, then the prediction mechanism will not know which way to choose.
    - ➔ The Grammar is not LL(1)!
  - If FIRST (γ_1) and FIRST (γ_2) have no symbols in common, then <u>perhaps</u> LL(1) can be used.
    - We need some more formalisms.

Road map

- Determining if a grammar is LL(1)
  - First sets
  - “Follow” sets – we’ll come back to this later
- “Retrofitting” a grammar into LL
  - Using the technique of “left factoring”
  - …and eliminating “left recursion”

Example

Want a derivation tree for any program like: x + y * z



1. S → E $
2. E → T E’
3. E’ → + T E’
4. E’ → - T E’
5. E’ → λ
6. T → F T’
7. T’ → * F T’
8. T’ → / F T’
9. T’ → λ
10. F → id
11. F → num
12. F → (E)

Example First Sets

- First(S) = {id,num,(}
- First(F) = {id,num,(}
- First(E')) = {+,-,$}

1. $S \rightarrow E\ \$$
2. $E \rightarrow T\ E'$
3. $E' \rightarrow +\ T\ E'$
4. $E' \rightarrow -\ T\ E'$
5. $E' \rightarrow \lambda$
6. $T \rightarrow F\ T'$
7. $T' \rightarrow *\ F\ T'$
8. $T' \rightarrow /\ F\ T'$
9. $T' \rightarrow \lambda$
10. $F \rightarrow id$
11. $F \rightarrow num$
12. $F \rightarrow (\ E\ )$

Table-driven Predictive Parsing

assume we're given this table

	+	*	id	$
S			1	
E			2	
E’	3			5
T			6	
T’	9	7		9
F			10	

empty=error

1. S → E $
2. E → T E’
3. E’ → + T E’
4. E’ → - T E’
5. E’ → λ
6. T → F T’
7. T’ → * F T’
8. T’ → / F T’
9. T’ → λ
10. F → id
11. F → num
12. F → (E)

Predictive Parsing example

Input tokens	Parse stack	Action
1. x + y * z $	S	predict 1

- Given left-most input token and top of stack ("x" and "S"),
  - determine the Action on the right using the parsing table
    - actions are "predict", "match", and "accept"
  - update the input and stack accordingly

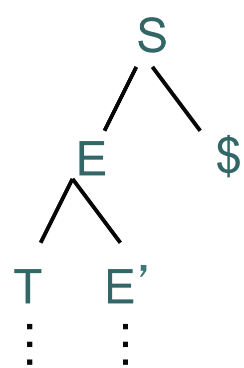
Predictive Parsing example

Input tokens	Parse stack	Action
1. x + y * z $	S	predict 1
2. x + y * z $	E $	predict 2
3. x + y * z $	T E' $	predict 6

Predictive Parsing example

	Input tokens	Parse stack	Action
1.	$\underline{x}$ + y * z $	$\underline{S}$	predict 1
2.	$\underline{x}$ + y * z $	$\underline{E}$ $	predict 2
3.	$\underline{x}$ + y * z $	$\underline{T}$ E' $	predict 6

Note that we are elaborating a parse tree for the input



Predictive Parsing example

	Input tokens	Parse stack	Action
1.	<u>x</u> + y * z $	<u>S</u>	predict 1
2.	<u>x</u> + y * z $	<u>E</u> $	predict 2
3.	<u>x</u> + y * z $	<u>T</u> E' $	predict 6
4.	<u>x</u> + y * z $	<u>F</u> T' E' $	predict 10

Predictive Parsing example

	Input tokens	Parse stack	Action
1.	x + y * z $	S	predict 1
2.	x + y * z $	E $	predict 2
3.	x + y * z $	T E' $	predict 6
4.	x + y * z $	F T' E' $	predict 10
5.	x + y * z $	id T' E' $	**match**

Now we have a terminal on top of the stack ("id"),

- **match** occurs because "x" and "id" are both terminal symbols
- we **match** by consuming a token & terminal
- here is where a parse error might occur

Predictive Parsing example

	Input tokens	Parse stack	Action
1.	x + y * z $	S	predict 1
2.	x + y * z $	E $	predict 2
3.	x + y * z $	T E' $	predict 6
4.	x + y * z $	F T' E' $	predict 10
5.	x + y * z $	id T' E' $	match
6.	+ y * z $	T' E' $	predict 9

Predictive Parsing example

	Input tokens	Parse stack	Action
1.	x + y * z $	S	predict 1
2.	x + y * z $	E $	predict 2
3.	x + y * z $	T E' $	predict 6
4.	x + y * z $	F T' E' $	predict 10
5.	x + y * z $	id T' E' $	match
6.	+ y * z $	T' E' $	predict 9
7.	+ y * z $	λ E' $	predict 3

Predictive Parsing example

	Input tokens	Parse stack	Action
1.	x + y * z $	S	predict 1
2.	x + y * z $	E $	predict 2
3.	x + y * z $	T E' $	predict 6
4.	x + y * z $	F T' E' $	predict 10
5.	x + y * z $	id T' E' $	match
6.	+ y * z $	T' E' $	predict 9
7.	+ y * z $	λ E' $	predict 3
8.	+ y * z $	+ T E' $	match
9.	y * z $	T E' $	predict 6

Predictive Parsing example

	Input tokens	Parse stack	Action
1.	x + y * z $	S	predict 1
2.	x + y * z $	E $	predict 2
3.	x + y * z $	T E' $	predict 6
4.	x + y * z $	F T' E' $	predict 10
5.	x + y * z $	id T' E' $	match
6.	+ y * z $	T' E' $	predict 9
7.	+ y * z $	λ E' $	predict 3
8.	+ y * z $	+ T E' $	match
9.	y * z $	T E' $	predict 6
10.	y * z $	F T' E' $	predict 10

Predictive Parsing example

	Input tokens	Parse stack	Action
		< … >	
10.	y * z $	F T' E' $	predict 10
11.	y * z $	id T' E' $	match
12.	* z $	T' E' $	predict 7
13.	* z $	* F T' E' $	match
14.	z $	F T' E' $	predict 10

Predictive Parsing example

Input tokens	Parse stack	Action
	< … >	
15. z $	id T' E' $	match
16. $	T' E' $	predict 9
17. $	λ E' $	predict 5
18. $	λ $	match
19. λ	λ	accept

Remaining Questions about Predictive Parsing

- We were “given” that grammar and prediction table
  - Can all grammars be given a similar table?
    - Alas, no – only LL(1) grammars
  - How did we come up with that table?
    - “First” and “Follow” sets
  - Techniques for converting **some** grammars into LL(1)
    - Eliminating left-recursion
    - Left factoring

FOLLOW sets and nullable productions

- FOLLOW(X)
  - X is a non-terminal
  - The set of terminals that can immediately follow X.
- nullable(X)
  - X is a non-terminal
  - True if, and only if, X can derive to the empty string λ.
- FIRST, FOLLOW & nullable can be used to construct predictive parsing tables for LL(1) parsers.
- Can build tables that support LL(2), LL(3), etc.

$X \rightarrow A\ X\ B$

$C \rightarrow X\ d$

$B \rightarrow a\ b$

FOLLOW(X) = ?

$X \rightarrow a\ B$

$X \rightarrow B$

$B \rightarrow d$

$B \rightarrow \lambda$

nullable(X) = ?

FOLLOW sets and nullable productions

- FOLLOW(X)
  - X is a non-terminal
  - The set of terminals that can immediately follow X.
- nullable(X)
  - X is a non-terminal
  - True if, and only if, X can derive to the empty string λ.
- FIRST, FOLLOW & nullable can be used to construct predictive parsing tables for LL(1) parsers.
- Can build tables that support LL(2), LL(3), etc.

$X \rightarrow A\ X\ B$

$C \rightarrow X\ d$

$B \rightarrow a\ b$

FOLLOW(X) = { d, a }

$X \rightarrow a\ B$

$X \rightarrow B$

$B \rightarrow d$

$B \rightarrow \lambda$

nullable(X) = true

Constructing the parse table

For every non-terminal X and token t:

	t
X	$X \to \gamma$

- Enter the production ($X \to \gamma$) if $t \in$ FIRST(γ),
- If X is nullable, enter the production ($X \to \gamma$) if $t \in$ FOLLOW(γ)

Constructing the parse table

What if there are more than one production?

	t
X	X → γ1,X → γ

Constructing the parse table

What if there are more than one production?

	t
X	$X \rightarrow \gamma 1, X \rightarrow \gamma$

Then the grammar cannot be parsed with “predictive parsing”, and it is **(by definition**) not LL(1).

Shortcomings of LL Parsers

- Recursive descent renders a readable parser.
  - depends on the first terminal symbol of each sub-expression providing enough information to choose which production to use.
- But consider a predictive parser for this grammar

$$E \rightarrow E + T$$
$$E \rightarrow T$$

```
void E(){switch(tok) {
  case ?: E(); eat(TIMES); T(); ← no way of choosing production
  case ?: T();
  …
}
void T(){eat(ID);}
```

Ways that grammars can fail to be LL(1) and ways to overcome them

LEFT RECURSION, COMMON PREFIXES

Defining Left Recursion

Directly Left-recursive

$$A \rightarrow^* A\beta$$

CFG is **left recursive** iff
for some non-terminal X and α β:

$$S \rightarrow^* X\,\alpha \rightarrow^* X\,\beta$$

Eliminating left recursion

- Consider this grammar
  - it's "left-recursive"

$$E \rightarrow E + T$$
$$E \rightarrow T$$

- Can not use LL(1) – why?
- Consider this alternative different grammar

$$E \rightarrow T\ E'$$
$$E' \rightarrow +\ T\ E'$$
$$E' \rightarrow \lambda$$

First(E + T) = First(T)

- Th
  - Now there is no left recursion.
  - There is a generalization for more complex grammars.

Removing Common Prefixes with "Left Factoring"

- Consider

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$
$$S \rightarrow \text{if } E \text{ then } S$$

- It's not LL(1) – why?
- We can transform this grammar, and make it LL(1).

$$S \rightarrow \text{if } E \text{ then } S\ X$$
$$X \rightarrow \lambda$$
$$X \rightarrow \text{else } S$$

** **Remark:** The resulting grammar is not as readable.*

In Class Discussion (1)

$S \rightarrow S ; S$

$S \rightarrow id := E$

$E \rightarrow E + E$

$E \rightarrow id$

$E \rightarrow num$

- How can we transform this grammar
  - So that it accepts the same language, but
    - Has no left recursion

We may have to use left factoring

In Class Discussion (2)

$S \rightarrow S1 \; ; \; S$ ⇐ was $S \rightarrow S \; ; \; S$
$S \rightarrow S1$
$S1 \rightarrow id := E$

$E \rightarrow E1 + E$ ⇐ was $E \rightarrow E+E$
$E \rightarrow E1$
$E1 \rightarrow id$
$E1 \rightarrow num$

Prefix elimination doesn't work here

- Write a grammar that accepts the same language, but
  - **Is not ambiguous**
  - Has no left recursion

Class Discussion (2)

S → S1 S2
S1 → id := E
S2 → ; S1 S2
S2 →

E → E1 E2
E1 → id
E1 → num
E2 → + E1 E2
E2 →

- Write a grammar that accepts the same language, but
  - . . .
  - **Has no left recursion**

Before: (# is a terminal)

X → X # Y
X → Y

After:

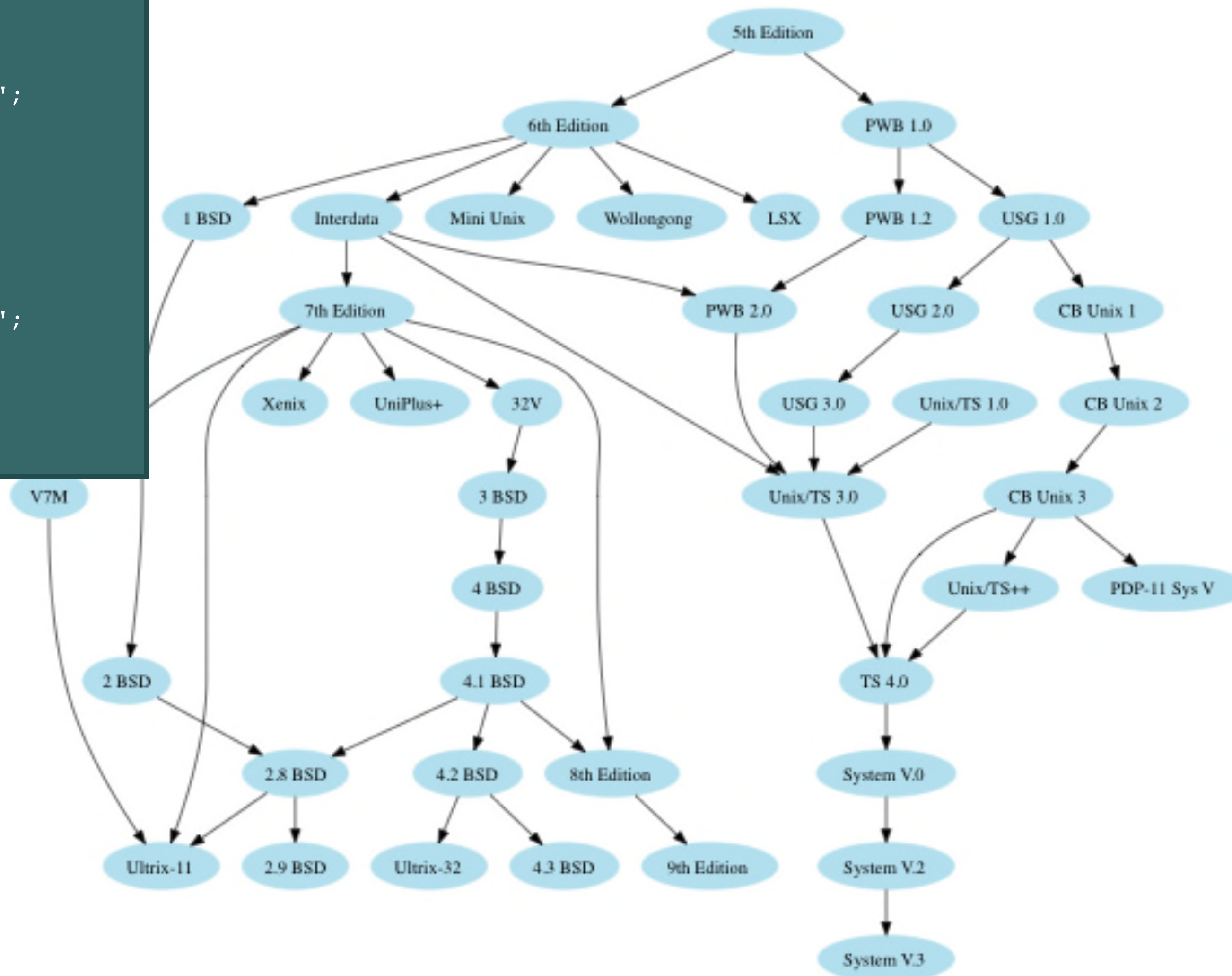
X → Y X2
X2 → # Y X2
X2 →

This final grammar is LL(1).

Visualizing Abstract Syntax Trees

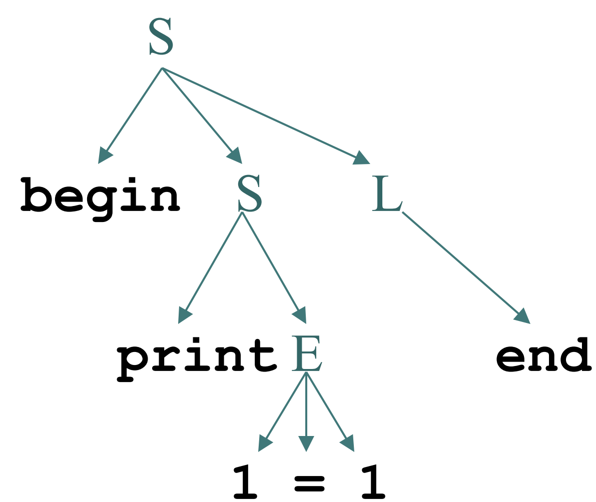
- We're going to use GraphViz
  - freely downloadable from graphviz.org
  - works on linux, mac, and windows
- We will use to visualize ASTs
  - input is text file
  - format is pretty easy
  - check their website for a number of examples
- I will assume that you have a working installation of this somewhere or another

```
digraph unix {
size="6,6";
node [color=lightblue2, style=filled];
   "5th Edition" -> "6th Edition”;
   "5th Edition" -> "PWB 1.0";
   "6th Edition" -> "LSX";
   "6th Edition" -> "1 BSD";
   "6th Edition" -> "Mini Unix";
   "6th Edition" -> "Wollongong";
   "6th Edition" -> "Interdata";
   "Interdata" -> "Unix/TS 3.0";
   "Interdata" -> "PWB 2.0”;
   "Interdata" -> "7th Edition";
   "7th Edition" -> "8th Edition";
   "7th Edition" -> "32V";
   "7th Edition" -> "V7M";
   "7th Edition" -> "Ultrix-11";
   "7th Edition" -> "Xenix";
   "7th Edition" -> "UniPlus+";
   "V7M" -> "Ultrix-11";
   "8th Edition" -> "9th Edition";
   "1 BSD" -> "2 BSD";
   "2 BSD" -> "2.8 BSD";
            <deleted for space>
}
```

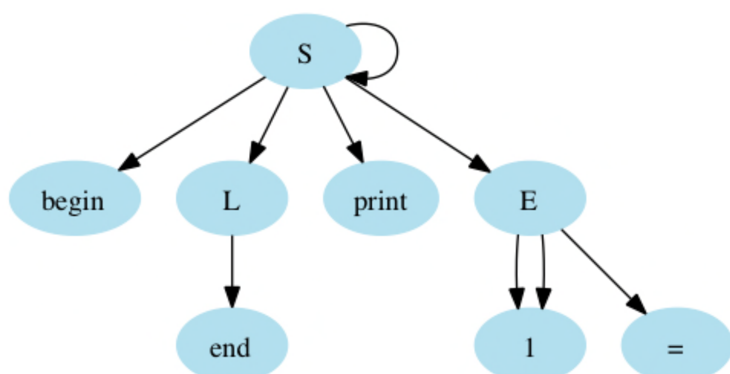


GraphViz & Parse Trees

```
digraph derivation {
size="6,6";
node [color=lightblue2, style=filled];
   "S" -> "begin";
   "S" -> "S";
   "S" -> "L";
   "S" -> "print";
   "S" -> "E”;
   "L" -> "end”;
   "E" -> "1”;
   "E" -> "=";
   "E" -> "1";
}
```

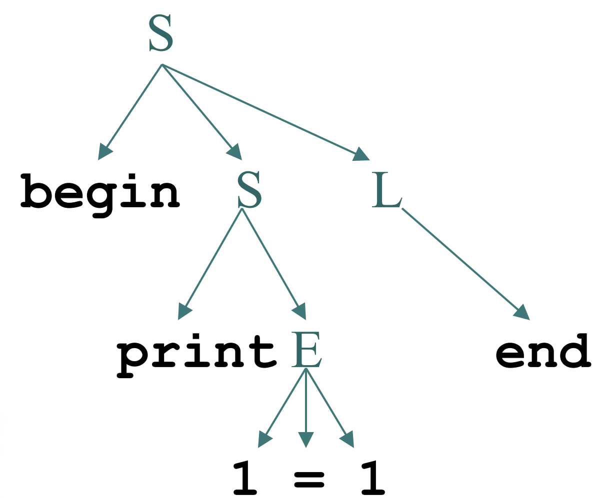


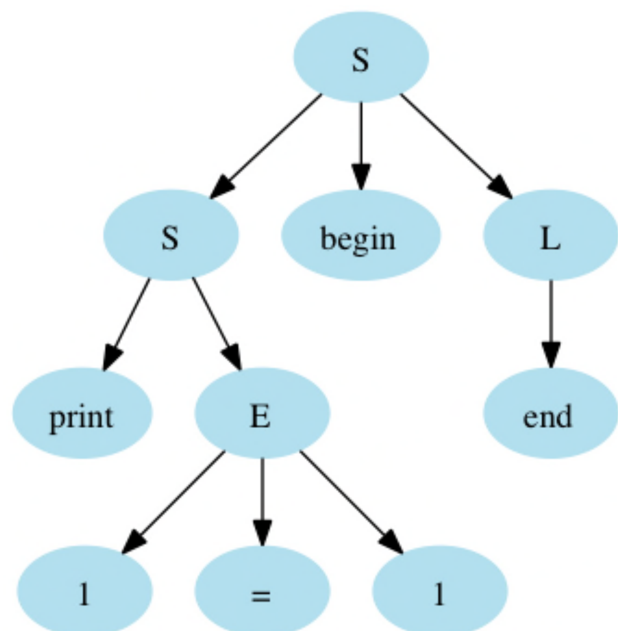
By default, label is name, so there is only one S, etc.



GraphViz & Parse Trees

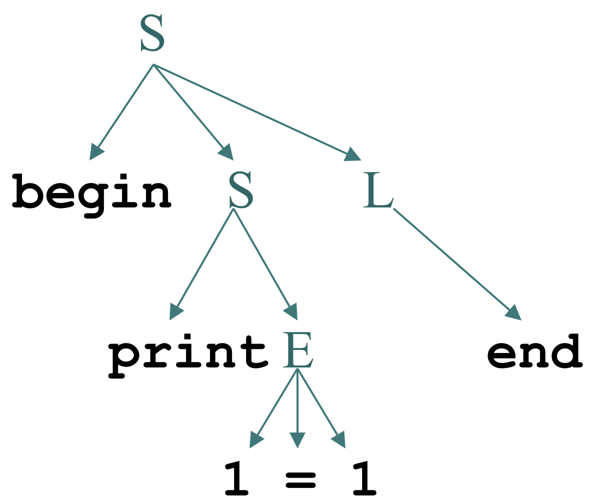
```
digraph derivation {
   size="6,6";
   node [color=lightblue2,
style=filled];
   s0 [label="S"];
   s1 [label="S"];
   s0 -> "begin";
   s0 -> s1;
   s0 -> "L";
   s1 -> "print";
   s1 -> "E";
   "L" -> "end";
   one0 [label="1"];
   "E" -> one0;
   eq [label="="];
   "E" -> eq;
   one1 [label="1"];
   "E" -> one1;
}
```

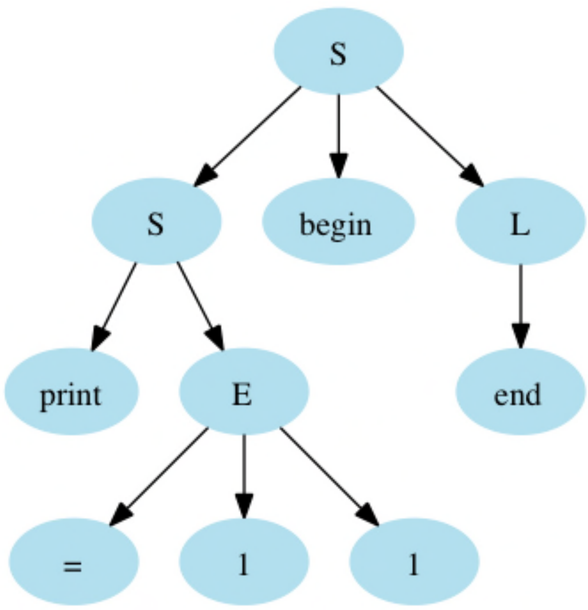




GraphViz & Parse Trees

```
digraph derivation {
   size="6,6";
   node [color=lightblue2,
style=filled];
   s0 [label="S"];
   s1 [label="S"];
   eq [label="="];
   s0 -> "begin";
   s0 -> s1;
   s0 -> "L";
   s1 -> "print";
   s1 -> "E";
   "L" -> "end";
   one0 [label="1"];
   "E" -> one0;
   "E" -> eq;
   one1 [label="1"];
   "E" -> one1;
}
```





N.b., order matters.