CFGs and syntax

- Having established a list of tokens, we need to describe the syntax rules for valid ways to string them together
- Our CFG will describe the ways in which parts of a program are defined in terms of sequences of token types, e.g. the syntax rules for variable declarations, the syntax rules for assignment statements, etc
- For each component that can be built, we'll provide rules for all the different valid forms of construction
- We'll borrow the yacc syntax for our CFG rules

Basic rule format

- A rule shows the name for the type of component being described (e.g. var_declaration) then a : then the sequence of token types required, then end the rule with a ;
- e.g. suppose we had defined tokens named IDENTIFIER, INT, CHAR, FLOAT, SEMICOLON, then rules might look like

```
data_type: CHAR ;
data_type: INT ;
data_type: FLOAT ;
```

- Components can be built up of other components

```
var_declaration: data_type IDENTIFIER SEMICOLON ;
```

Collapsing rules with or |

- In cases where there are multiple ways to build a component, we can use a single rule and separate the different constructions with | (or)

```
data_type: CHAR ;
data_type: INT ;
data_type: FLOAT ;
```

- Could be replaced with

```
data_type: CHAR | INT | FLOAT ;
```

Describing a program components

- We’ll have a name for each component, which it will either be a token or a *non-terminal* component composed of a sequence of tokens
- Non-terminals are used to describe parts of the program in abstract terms, e.g. to describe a for loop, or a function declaration, or a variable declaration, etc
- We’ll have a generic starting non-terminal to describe the entire program, e.g. something like program or start
- Our rule set has to describe all the ways to get from the starting non-terminal to a final valid sequence of tokens

Example: a simple language

- Suppose our tokens are: identifiers (one or more alphabetic), positive integers (one or more digits), an assignment operator (=), the keywords begin and end, and the addition operator (+), the period (.)
- Assume we have a regex for each, our CFG uses names for the token types: IDENTIFIER, INTEGER, ASSIGN, BEGIN, END, PLUS, STOP
- Programs start with begin, finish with end, and can have one or more assignment statements inside
- Assignment statements look like identifier = expression .
- Expressions can be integers, identifiers, or expr + expr

Valid sample program

- A valid sample program might be

```
begin
x = 27 .
end
```

- Another valid program might be

```
begin
foo = 123 .
x = 17 + foo + 100 .
end
```

Developing a rule set

- Let's use `program` as our starting non-terminal, `assign_stmt` as the non-terminal for an assignment statement, and `expression` as the non-terminal for an expression
- We'll need a non-terminal to represent an entire list of statements, so let's use `stmt_list`
- We can now start building the rule collection
- Our program as a whole is a begin, followed by a statement list, followed by an end, i.e.

```
program: BEGIN stmt_list END
```

Rule set, continued

- A statement list is a single assignment statement, or an assignment statement then more statements

```
stmt_list: assign_stmt | assign_stmt stmt_list
```

- An assignment statement is an identifer, the assignment operator, an expression, and a period

```
assign_stmt: IDENTIFIER ASSIGN expression STOP
```

- An expression is an identifier, an integer or expr + expr

```
expression: IDENTIFIER | INTEGER |
            expression PLUS expression
```

The whole grammar

- Thus our complete grammar (assuming we’ve handled the tokens’ regular expressions separately) is:

```
program: BEGIN stmt_list END
stmt_list: assign_stmt | assign_stmt stmt_list
assign_stmt: IDENTIFIER ASSIGN expression STOP
expression: IDENTIFIER | INTEGER |
            expression PLUS expression
```

Derivations: checking validity

- To see if a program is valid under a grammar, we (or the tool) searches for a way to generate that program using the grammar rules
- If a program cannot be generated under the grammar rules then it cannot be a valid program
- If a program **can** be generated under the grammar rules, then the sequence of rules applied tell us what the components of the program are (e.g. a variable declaration, followed by a function definition, followed by a function call)

Derivation example

- A derivation for our first sample program

```
begin
x = 27 .
end
```

- The steps in the derivation would be

```
Program -> BEGIN stmt_list END
stmt_list -> assign_stmt
assign_stmt -> IDENTIFIER ASSIGN INTEGER STOP
```

And, for the regular expressions resolving the tokens:

```
IDENTIFIER -> x      ASSIGN -> =
INTEGER -> 27        STOP -> .
```

Derivation example 2

- Consider our second program

```
begin
foo = 123 .
x = 17 + foo + 100 .
end
```

- The derivation steps might start like

```
program -> stmt_list
stmt_list -> assign_stmt stmt_list
stmt_list -> assign_stmt
```

Deriv example 2 continued

- For the first assignment statement

```
assign_smt -> IDENTIFIER ASSIGN expression STOP
expression -> INTEGER
```

- For the second assignment statement

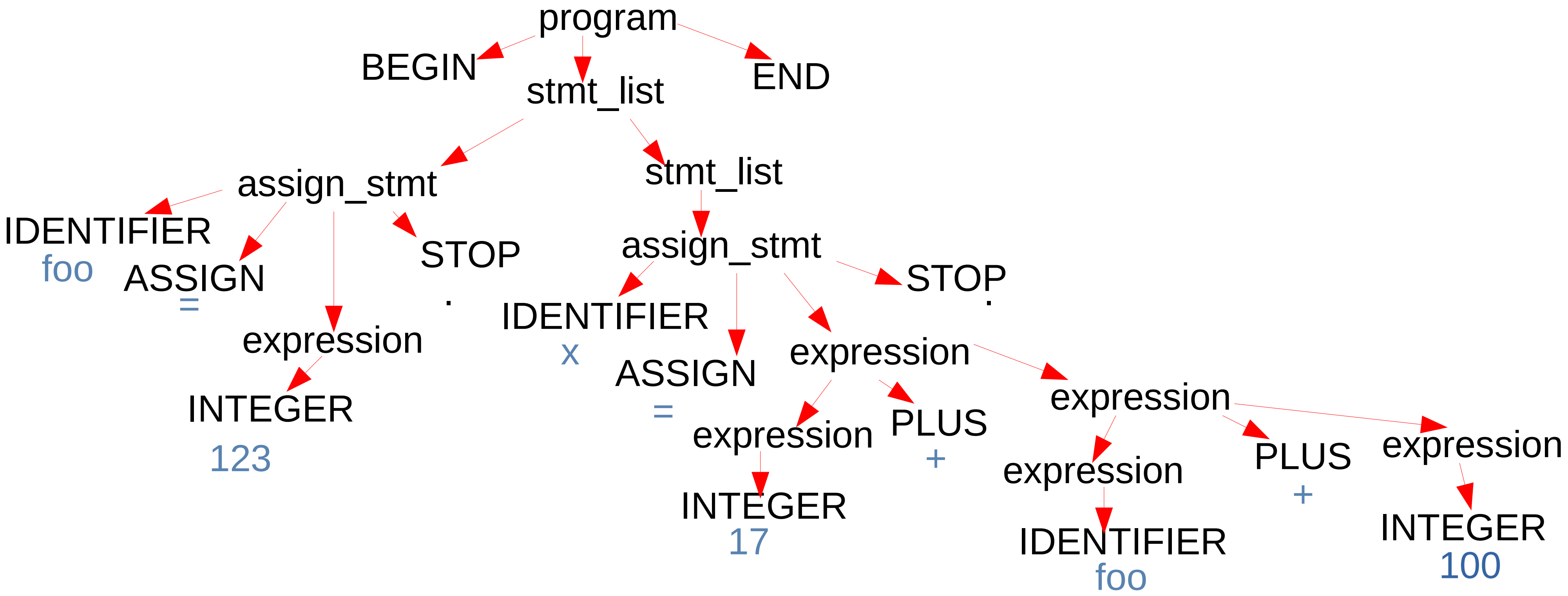
```
assign_stmt -> IDENTIFIER ASSIGN expression STOP
expression -> expression PLUS expression
```

- Then (arbitrarily) resolving the expressions left-to-right

```
expression -> INTEGER
expression -> expression PLUS expression
expression -> IDENTIFIER
expression -> INTEGER
```

Derivation trees, program meaning

- We can also represent the derivations as a tree, e.g.



program

BEGIN

stmt_list

END

assign_stmt

stmt_list

IDENTIFIER

foo

ASSIGN

=

expression

INTEGER

123

STOP

.

assign_stmt

IDENTIFIER

x

ASSIGN

=

expression

STOP

.

expression

INTEGER

17

PLUS

+

expression

expression

IDENTIFIER

foo

PLUS

+

expression

INTEGER

100

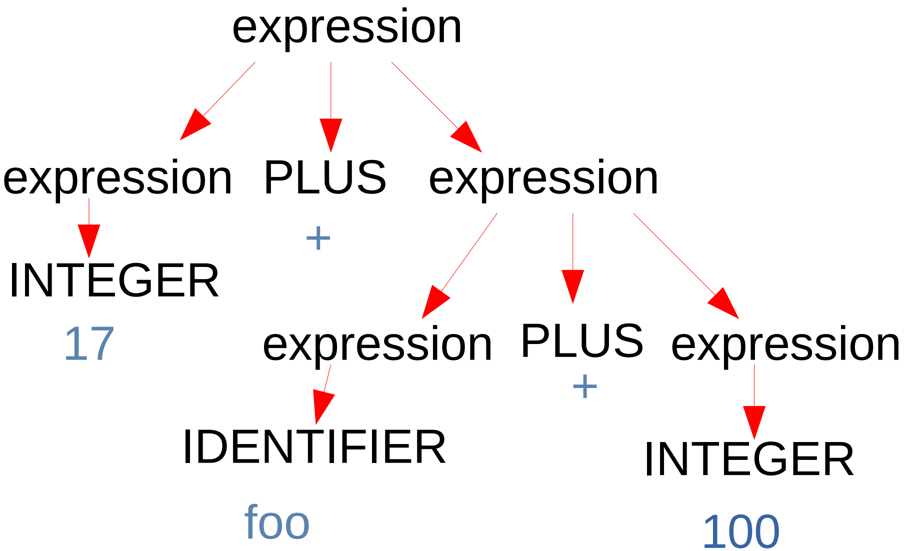
Ambiguous grammars

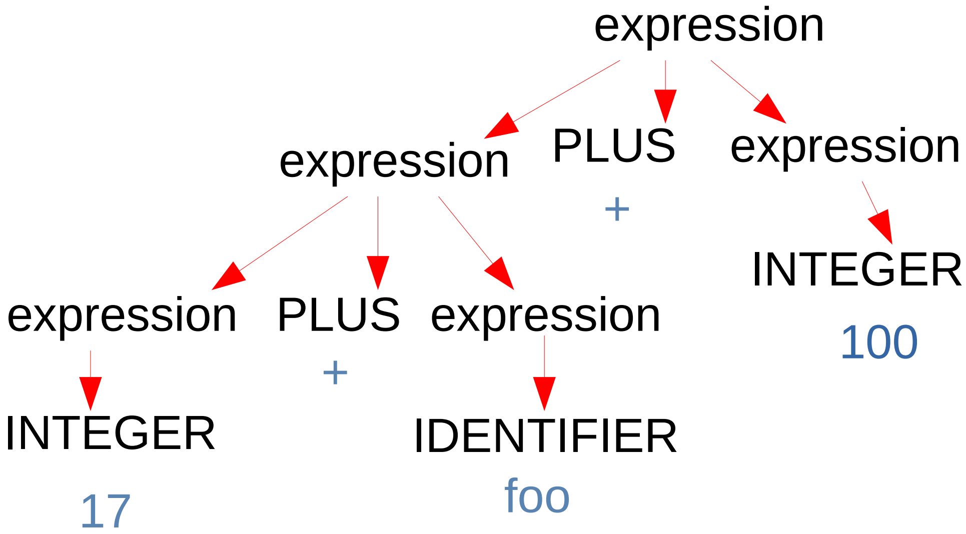
- If there is more than one way to generate a particular program under the grammar then there are multiple possible interpretations about what the structure of the program is
- The grammar is called ambiguous
- Not a good thing: e.g. one compiler might pick one derivation while a different compiler picks another, and the same source code could thus produce executables that behave differently

Example: ambiguous grammar

- We can demonstrate our sample grammar was ambigous by showing a second, different, valid derivation tree for the program from example 2
- The difference will be in the expression for the second statement: the first time we expanded the `expression` non-terminals from left to right, this time we'll expand them in the opposite direction

Different expression derivations



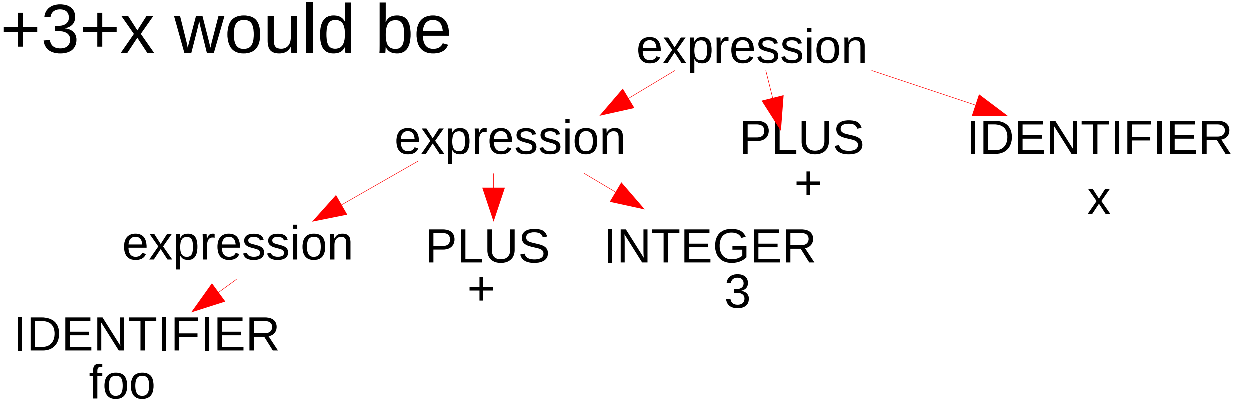


Meaning: 17 + (foo + 100)

Meaning: (17 + foo) + 100

Eliminating ambiguity

- We can structure our grammar rules to enforce which terms to expand next, e.g. instead of expr -> expr + expr we could use
- Expr -> expr PLUS INTEGER | expr PLUS IDENTIFIER
- thus it would finalize the term to the right of the +, so foo+3+x would be



Order of operations: associativity

- the grammar rules we pick must reflect our desired order of operations, both precendence and associativity
- `expr -> expr PLUS INTEGER` implies the rightmost `PLUS` is evaluated last, which means order of evaluation is left to right (typically what we want)
- `expr -> INTEGER PLUS expr` implies the leftmost `PLUS` is evaluated last, i.e. + operations would evaluate right to left (not usually what we want for +, but might be the desired order for assignment, e.g. for things like x = y = z;)

Order of ops: precedence

- We want higher precedence operations to be “lower” in the derivation tree, so they get performed first, e.g. for x+y*z what we want is effectively x+(y*z), and for x*y+z what we want is effectively (x*y)+z
- To get this effect, we can create separate non-terminals for the different precedence levels of expression, and have the grammar rules finalize the lower precedence operations earlier in the derivation

Example: + and *

- We’ll introduce two expression types: `add_expr` and `mult_expr`, and have our derivations process every `add_expr` first so they’re “higher” in the tree

```
expr -> add_expr
add_expr --> add_expr PLUS mult_expr
           | add_expr PLUS mult_expr
           | mult_expr
```

- ie there will be no way for a mult_expr to lead back to an add_expr, so our derivations are forced to deal with every `PLUS` before any `MULT`

Example + and * continued

- Now we can process the mult operations

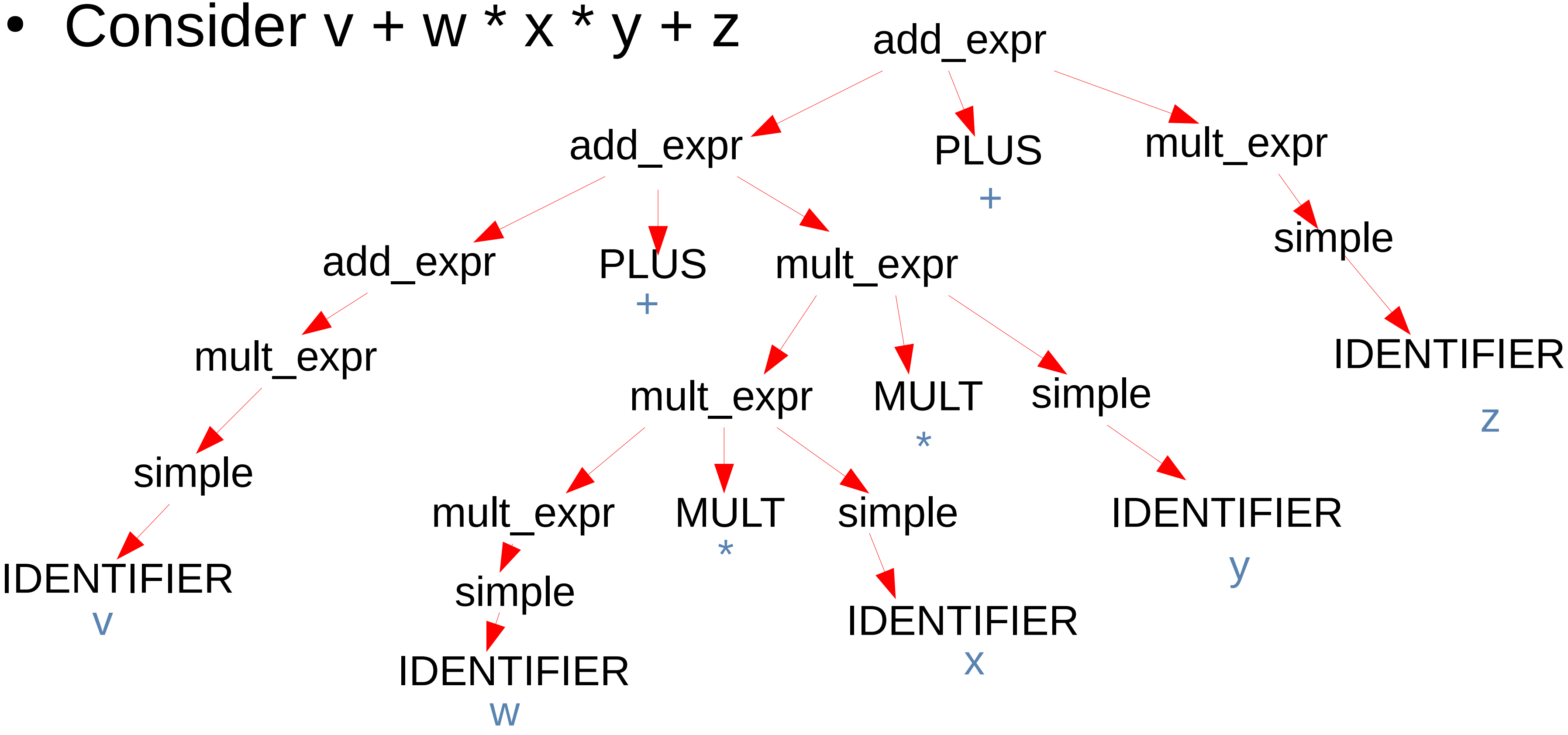
```
mult_expr --> mult_expr MULT simple
            | simple
Simple --> INTEGER
         | IDENTIFIER
```

- Note that if an expression was just an integer (or just an identifier) the derivation now goes

```
expr -> add_expr -> mult_expr -> simple -> INTEGER
```

Example: derivation tree

- Consider v + w * x * y + z


add_expr
add_expr
PLUS
+
mult_expr
add_expr
PLUS
+
mult_expr
simple
mult_expr
mult_expr
MULT
*
simple
IDENTIFIER
z
simple
mult_expr
MULT
*
simple
IDENTIFIER
y
IDENTIFIER
v
simple
IDENTIFIER
x
IDENTIFIER
w

Adding handling of parenthesis

- Generally the ( ) are regarded as highest precedence, and working from the “outside” in, so these have to be reflected in our grammar rules
- For our “simple” rule from the previous example, we can add our bracket checker

```
Simple --> INTEGER
         | IDENTIFIER
         | LBRACKET expr RBRACKET
```

- Thus the content inside the brackets is treated as a normal top-level expression, assuming `LBRACKET` and `RBRACKET` are “(“ and “)”

“Real” languages

- You can see the lex tokenization for C at

  `www.lysator.liu.se/c/ANSI-C-grammar-l.html`

- Similarly, you can see the yacc syntax parsing for C at

  `www.lysator.liu.se/c/ANSI-C-grammar-y.html`

- While it takes some time to follow through the sequences, the ideas have all been covered!