

Semantics

As mentioned we normally break things into syntax

- involving structure

and semantics

- involving the meaning of those structures

For pragmatic reasons we typically draw the line between syntax and semantics at things that can be described by CFG (syntax) versus not (sem)

- structure of expression is syntax
- meaning of expression (e.g., float vs. int add) is typically considered semantics

Note: its possible to do some semantics in the CFG

- could do (some) type inference / checking

Example: could syntactically disallow: "abc" / 5

- but this would make grammar much larger
- can't do this for everything (decl before use)
- this is not the easiest way to do this

Lexical analysis / scanning (based on RE) and parsing (based on CFGs) take care of syntax

What about semantics?

Normally break semantics into two parts:

Static semantics

meaning of the static program

- at compile time

Example: (Java)

```
int a, b;
```

```
...
```

```
System.out.println("abc" + a*b)
```

means: ??

A lot of this is what we think of as type checking / analysis

Note that this is still talking about what makes a properly structured program

Dynamic semantics

meaning/behavior of program at runtime

No universally accepted way to express / define semantics (static or dynamic)

- depends a lot on what you need to do with the "meaning"

Static semantics is very important to the language translator

Need to know enough about meaning of constructs to translate them into another representation
machine/assembly language or
structures suitable for interpretation

Need to enforce the type (and other) constraints imposed by the language

- e.g., can't assign real value to int in Pascal
(can in C, but implies "insertion" of
otherwise implicit conversion operation)

Example of a non-type semantic restriction(s): ??

Impl (static) semantic analysis in translator

Attribute grammars

Augmentation of CFG to include aspects of static semantics

- invented by Donald Knuth (who also invented LR parsing, which is most powerful linear time method)

Each symbol (terminal or non-terminal) may have information attached to it

- attributes (basically a variable)
- meaning contained in attributes

Example: indicate the type of an expression

Each production in the grammar may define a set of attribute evaluation functions

(AKA semantic functions)

- define how to compute an attribute's value from attributes of other symbols appearing in the production

Example attribute grammar (signed binary numbers)

Start with a CFG

Num ::= Sign List

Sign ::= +

Sign ::= -

List ::= List Bit

List ::= Bit

Bit ::= 0

Bit ::= 1

Attributes of interest

Num

val Value of the number

Sign

neg Is the sign negative

List

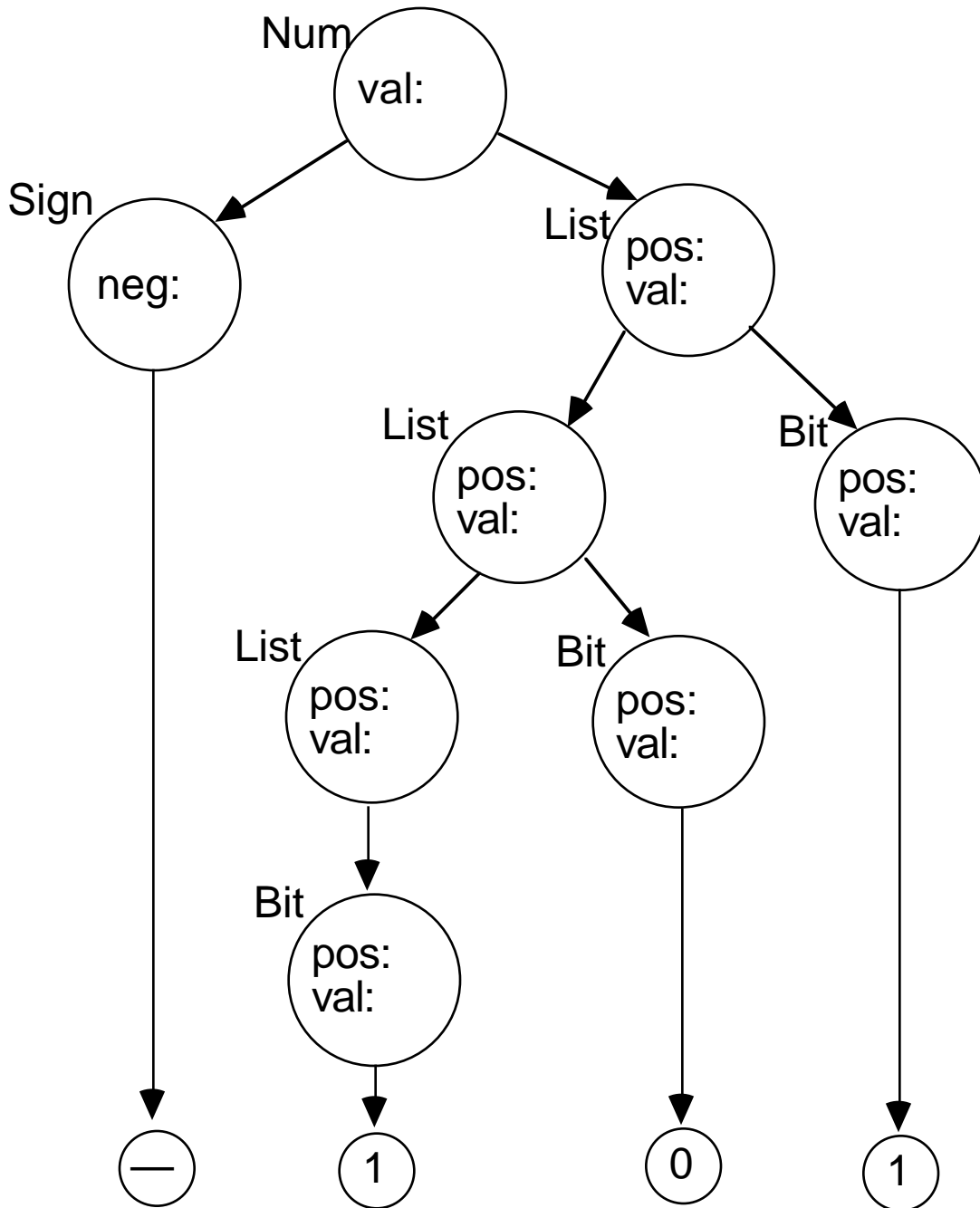
val Value of the (partial) number
pos Bit position of low order bit of
(partial) number

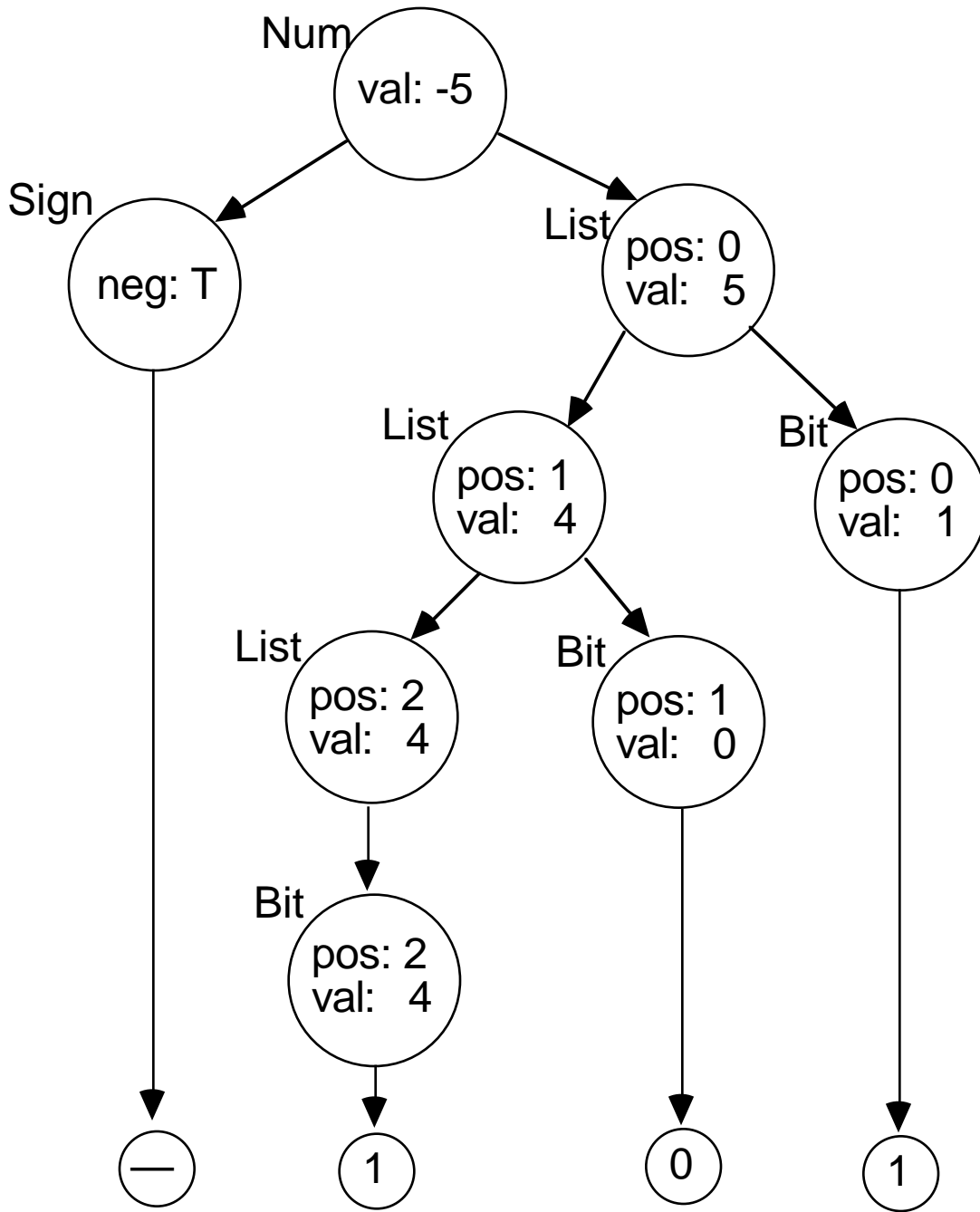
Bit

val Value of the bit
pos Position of the bit

Grammar with semantic functions

$$\frac{\text{Num} ::= \text{Sign List}}{\text{List.pos} = 0}$$
$$\text{Num.val} = \text{if Sign.neg}$$
$$\quad \text{then } - \text{List.val}$$
$$\quad \text{else List.val}$$
$$\frac{\text{Sign} ::= +}{\text{Sign.neg} = \text{false}}$$
$$\frac{\text{Sign} ::= -}{\text{Sign.neg} = \text{true}}$$
$$\frac{\text{List0} ::= \text{List1 Bit}}{\text{List1.pos} = \text{List0.pos} + 1}$$
$$\text{Bit.pos} = \text{List0.pos}$$
$$\frac{\text{List} ::= \text{Bit}}{\text{List.val} = \text{Bit.val}}$$
$$\text{Bit.pos} = \text{List.pos}$$
$$\frac{\text{Bit} ::= 0}{\text{Bit.val} = 0}$$
$$\frac{\text{Bit} ::= 1}{\text{Bit.val} = 2^{\text{Bit.pos}}}$$





More about attribute grammars

Attributes can be broken into two categories:

Synthesized attributes

- attributes of the parent (LHS of production) which are defined in terms of attributes from the children
 - information flowing up the tree
- (which attributes from example are synth?)

Inherited attributes

- attributes of the children (RHS of production) which are defined in terms of attributes from the parent and siblings
- information flowing down (and sideways)

But what about:

Bit ::= 0
Bit.val = 0

Intrinsic attributes

- comes from constant or "external" source
- considered a synthesized attribute

Note: attribute grammar does not say anything about order of evaluation (just as grammar does not specify order of derivation)

For practical reasons, we often place restrictions on attribute grammars used for translation

Normally require AGs to be acyclic:

No attributes will directly or indirectly depend on themselves

- basic "well-formed" property
- but actually hard to check statically
- can restrict class further to make checking easier

Other common restrictions:

S-Attributed

- only synthesize attributes
- can we do example as S-Attributed?

L-Attributed

- Information flows either bottom up (synthesized) or left to right

L-Attributed (cont.)

Specifically, given production: $A ::= X_1 X_2 \dots X_n$

Each inherited attribute of X_i may depend on (be defined in terms of) only

a) inherited attributes of A , or

b) attributes of X_j for $1 \leq j < i$
(attributes to the left)

L-Attributed are suitable for evaluation "on the fly" during a left-to-right parse

Dynamic Semantics

No universally accepted form

- which is more useful depends on what you are using it for

Three major techniques for defining / describing dynamic semantics

Operational

Axiomatic

Denotational

(Actually four: most popular is English text, but that is not a formal definition, hence has problems.)

Operational Semantics

Define meaning of constructs in terms of another (lower-level, often machine-like) language

- could in theory be a real machine
- more useful to use a virtual machine that can be easily translated into specific target machines

Meaning/effect of executing a program should be the same as executing its defined equivalent in the lower level language

Need two things for operational semantics

- Translation of PL constructs into chosen lower level language
- Virtual machine for lower-level language

Most famous use of operational semantics: PL/I
"Vienna Definition Language"
Too complex to be useful

Generally, operational semantics are most useful for translator writers

Axiomatic semantics

Most useful for reasoning about programs (e.g., proofs of correctness)

Meaning of program constructs (typically statements) defined by axioms and inference rules

Based on reasoning with pre- and post-conditions

Pre- and post-conditions are logical assertions that are true at certain points in the program (before and after statements)

General form: $\{ P \} S \{ Q \}$

where P and Q are predicates

and S is a program statement

Example: $\{ \dots \} \text{sum} := 2 * x + 1; \{ \text{sum} > 1 \}$

Typically make a post condition following the last statement of the program which indicates the program is correct

Work backwards to derive pre-conditions under which that post-condition holds

Generally looking for the weakest precondition which will cause the desired post-condition to hold true

- "weakest" means least specific (most general)

$\{ ?? \}$ $\text{sum} := 2 * x + 1;$ $\{\text{sum} > 1\}$

$\{ x > 10 \}$ will cause post condition to be true, but so will $\{ x > 5 \}$, and $\{ x > 1000 \}$

Weakest precondition is $\{ x > 0 \}$

Axiomatic definition gives you a set of axioms and inference rules to reason about how pre-conditions are related to post-conditions

Axioms

- Statements of fact
- Simple transformations of predicates

Example:

$\{ P \} x := E; \{ Q \}$

Axiom: $P = Q_{x \rightarrow E}$
(Q rewritten replacing x with E)

$\{ ?? \}$	$a := b * b - 20; \{ a < 10 \}$
$\{ b * b - 20 < 10 \}$	$a := b * b - 20; \{ a < 10 \}$
$\{ b * b < 30 \}$	$a := b * b - 20; \{ a < 10 \}$
$\{ \text{abs}(b) < \text{sqrt}(30) \}$	$a := b * b - 20; \{ a < 10 \}$

Inference Rules

"Rules of consequence"

Rules that allow logical deductions

Expressed with "bar" notation

$$\frac{S_1, S_2, \dots, S_n}{S}$$

Which means if all of $S_1 \dots S_n$ are true then we can deduce that S is true

Examples:

$$\frac{\{P\} S \{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\} S \{Q'\}}$$

$$\frac{\{P_1\} S \{P_2\}, \{P_2\} S \{P_3\}}{\{P_1\} S_1; S_2 \{P_3\}}$$

Denotational semantics

Based on defining translation of program into a mathematical construct — a "denotation"

Two parts:

- definition of what the mathematical object looks like
 - often an expression in the Lambda calculus
- function that maps PL constructs to the mathematical object

(see the book for an example)

One way to get the intuition for this is to think of it as operational semantics where the translation is expressed in a particular way (a function) and the target low-level language is a set of mathematical constructs (e.g., the Lambda Calculus)

Why do this?

There are rigorous ways of manipulating mathematical objects, don't have this for program concepts