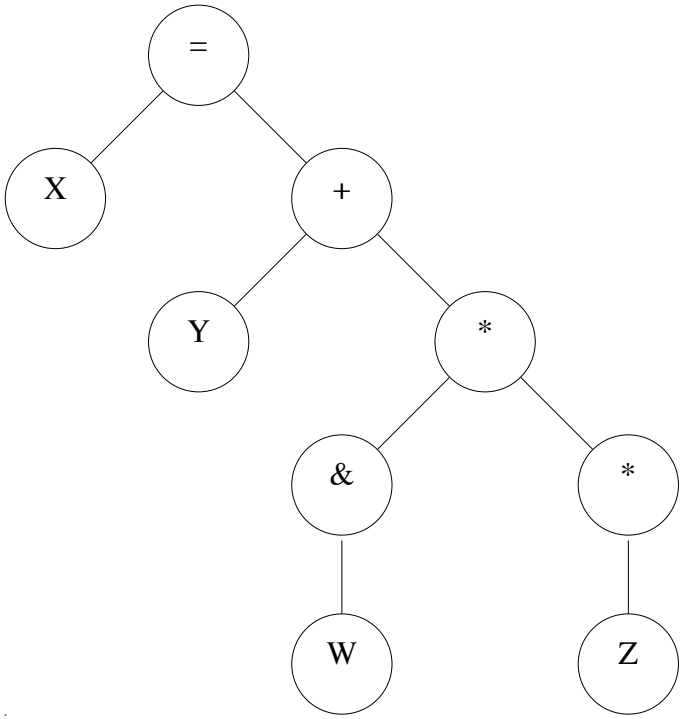
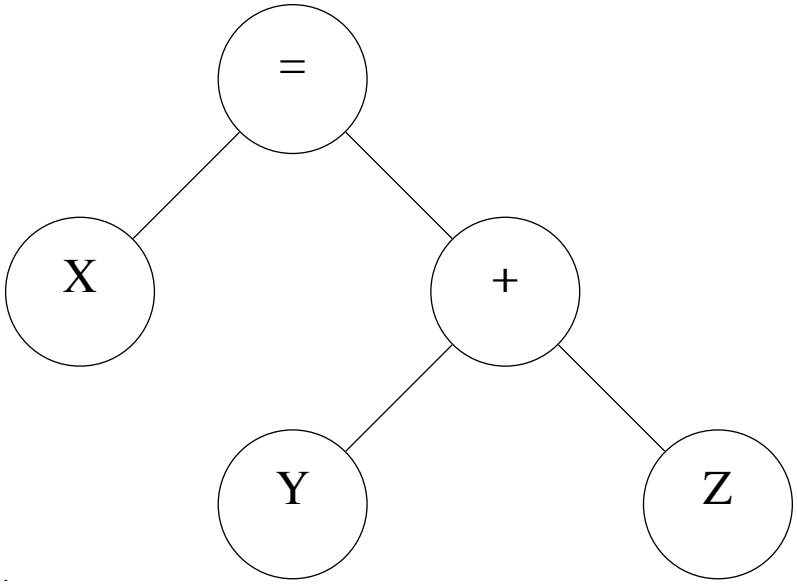
Type checking

L *vs.* R values



The actual meaning of the identifier is dependent on its context.

Type compatibility



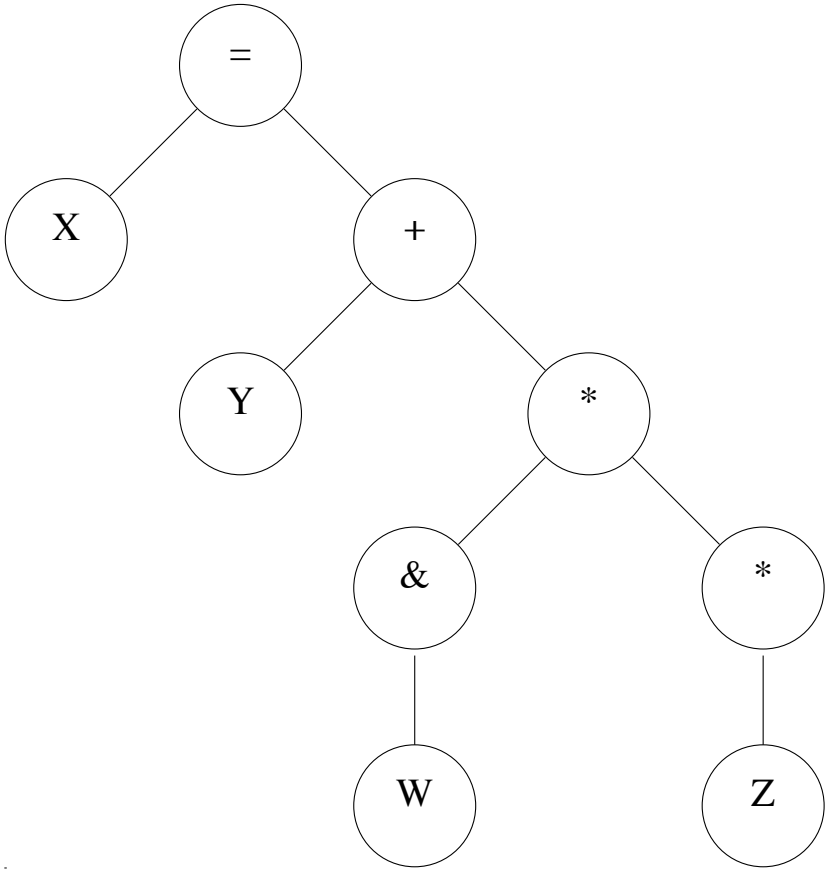
The meaning of $+$ in the above program depends on the types of Y, Z, and X. In languages that allow operator overloading, even the meaning of $+$ becomes suspect.

Notice the dual role of the operator $*$ (in the C language), which is nicely disambiguated using the proper grammar.

Left and right values of identifiers

Named for their interpretation with respect to "=", the *left* value of an identifier is its location while the *right* value is the contents of the identifier.

L *vs.* R values



The positioning of identifiers with respect to various operators in C indicates which value is desired:

X=	**The storage location of X**
=Y	**The value stored at Y**
∗ Z	**The value at Z, treated as a storage location**
& W	**The address of W, treated as a value**

C left and right values

Form	Expects	Produces
a=b	LV(a), RV(b)	RV
$*$ c	RV(c)	LV
&d	LV(d)	RV

$$
\begin{array}{lcl}
\mathbf{S} & \rightarrow & \mathbf{L = R} \\
 & | & \mathbf{R} \\
\mathbf{L} & \rightarrow & \mathbf{id} \\
 & | & * \mathbf{R} \\
\mathbf{R} & \rightarrow & \mathbf{L} \\
 & | & \mathbf{\&\ L} \\
 & | & \mathbf{int}
\end{array}
$$

This grammar produces structures where the interpretation of left and right values is clear.

Moreover, the rule R$\rightarrow$L is applied when a left value "becomes" a right value through dereferencing.

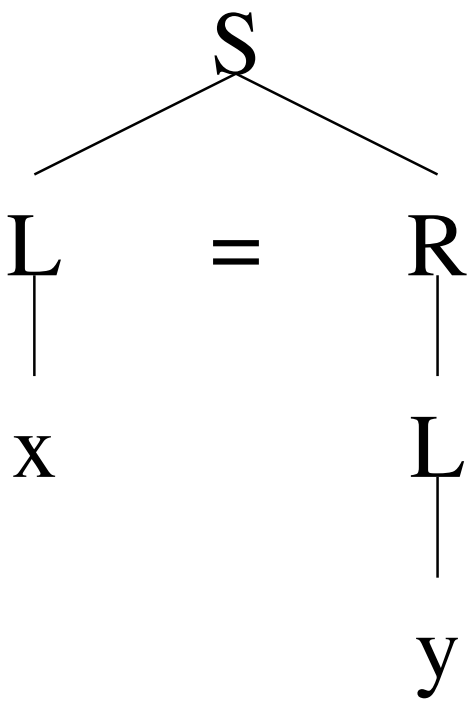
The grammar correctly precludes strings like "3=x" and "&z=y".

Table construction for this grammar fails for SLR because "$=$" can follow an R.

But there's no sentential form that begins "R="; the R must be preceded by an $*$ as in "$*$R=". The LR(1) construction can create a suitable parse table. The grammar is also LALR(1) (YACC can handle this grammar).

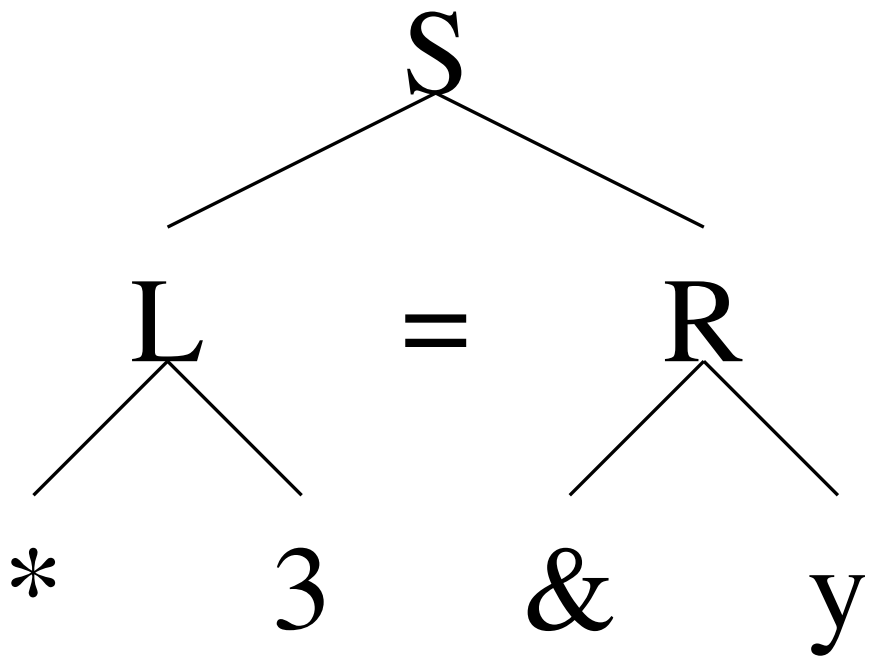
Examples

x=y



Push x
Push y
Fetch
Store

∗3=&y

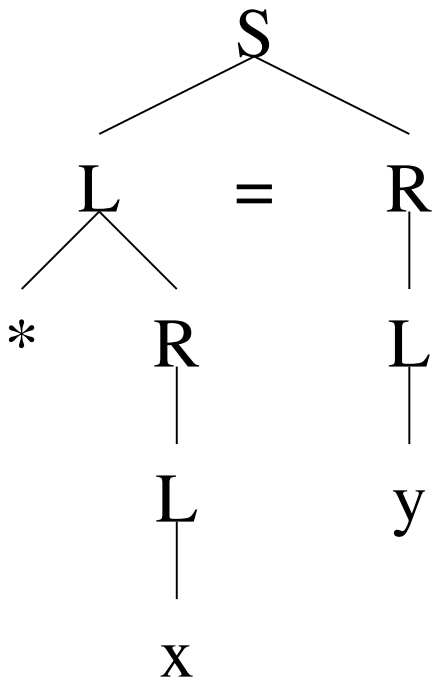


Push 3
Push y
Store

The ∗ and & have no effect on code generation: they merely change the type of an expression. The language design is biased towards the most prevalent "x=y" form.

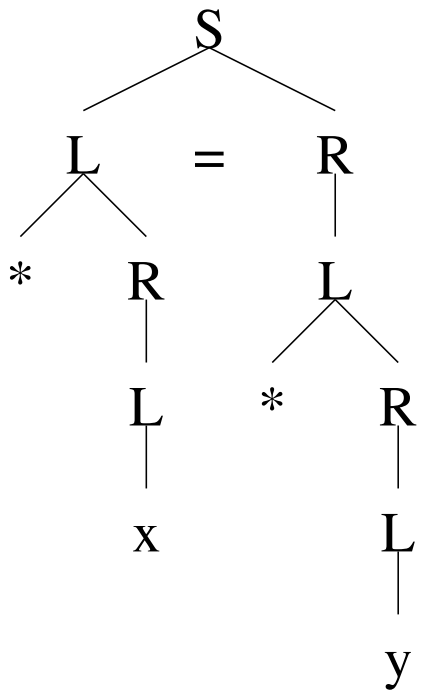
Examples

∗x=y



Push x
Fetch
Push y
Fetch
Store

∗x=∗y



Push x
Fetch
Push y
Fetch
Fetch
Store

More on left and right values

Unfortunately, the syntactic rules for C do not allow a grammar-based approach to left and right value disambiguation. However, the rules related to $=$, &, and $*$ can be applied just as easily to the parse tree, using attribute grammars or an additional (bottom-up) pass over the tree.

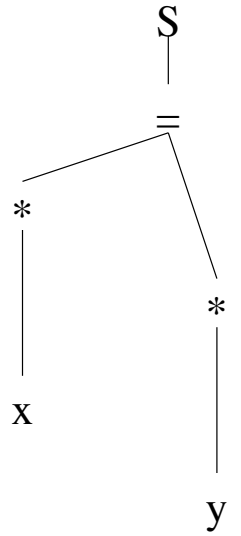
SetV($node, kind$): asserts the left or right valuedness of $node$.

ExpectLV($node$): expects that $node$ is a left value.

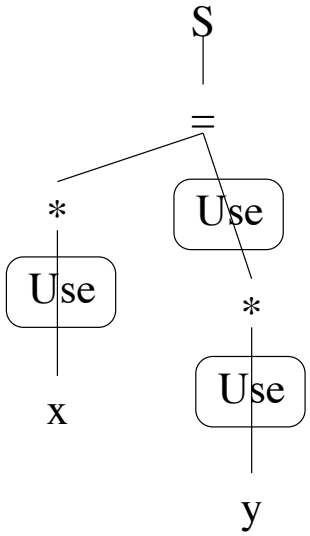
ExpectRV($node$): expects that $node$ is a right value.

Convert($node, how$): attempts to convert $node$ into a left or right value.

As our grammar indicates, the only conversion that makes sense is a left to right value conversion, which is basically a dereference. In a call-by-reference language, however, a right value could become a left value by introducing a temporary. Show below are the parse trees before and after the extra bottom-up pass.



Form	Expects	Produces
a=b	LV(a), RV(b)	RV
$*$ c	RV(c)	LV
&d	LV(d)	RV



Data types and compatible operations

The type checking phase of a compiler is traditionally responsible for establishing the semantic well-formedness of operations and data. Where language standards allow flexibility (some would say sloppiness) with respect to type consistency, compilers are charged with introducing implicit or explicit conversion operations to allow operations on otherwise unsuitable data.

Most languages offer a host of basic types, which are usually (though not always) supported by target instruction sets:

- **integers;**
- **floating point;**
- **Boolean-valued $\{\,\mathbf{true}, \mathbf{false}\,\}$;**
- **character.**

Most languages also allow the introduction of new types based on old ones:

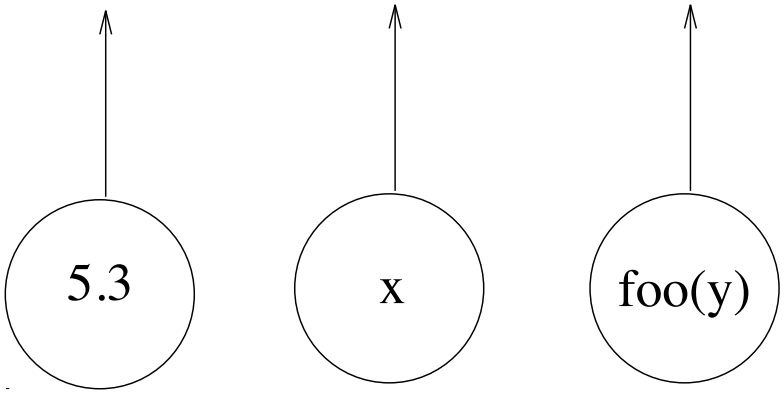
- **tuples (records, structs);**
- **maps (functions, arrays);**
- **sets.**

Proponents of *strongly-typed* languages, where data and operations must adhere rigidly to type consistency, claim that when soundly checked at compile-time, their programs are less likely to contain bugs.

Type checking

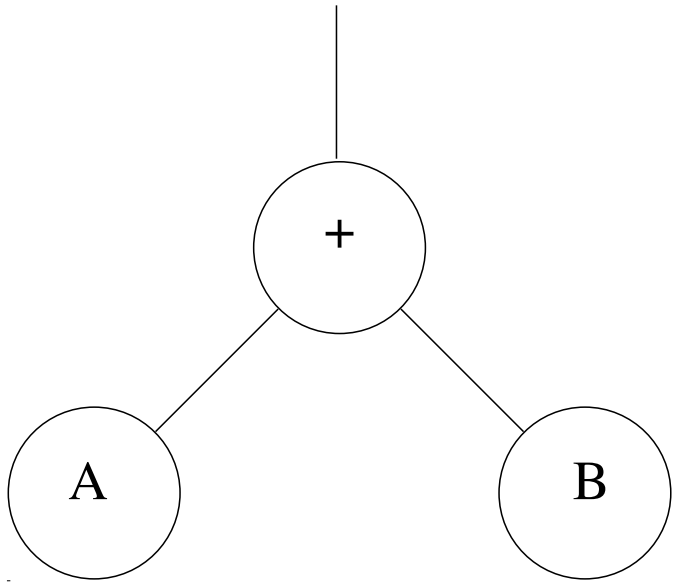
As with left and right value determination, type checking can be performed as a bottom-up pass over the parse (or abstract syntax) tree.

A straightforward (*i.e.*, highly localized) scheme operates as follows. At the tree's leaves are found the atomic elements such as constants, identifiers, and function calls. Each of these asserts its type, based on syntactic ("x" *vs.* 'x') or contextual (declared) information.



At each internal node X

1. **the subtrees of X are checked for type compatibility: this depends on the operation contained in X;**
2. **conversion operations are inserted as necessary;**
3. **the type of X is determined.**



Simple type checking

+	int	float	char
int	**int**	**float**	**int**
float	**float**	**float**	**float**
char	**int**	**float**	**int**

=	int	float	char
int	**int**	**int**	**int**
float	**float**	**float**	**float**
char	**char**	**char**	**char**

The arithmetic operators tend to find the grandest type suitable for performing the operation, while the assignment operators insist that the assigned value matches the type of its destination.

Name *vs.* structural type equivalence

```
typedef int t
int x
t y
⋮
x = (int) y
```

```
enum { a=1, b=2, c=3} foo;
⋮
foo = foo + 1;
M[(int) foo] = 'x';
```

The C language has *cast* operations, that assert the type of an expression. But conversions can also occur in uncast expressions, which can lead to confusion.

Are the above casts necessary? It depends on whether we regard type equivalence as a *structural* property or as a property of the name used in the declaration. In the above examples, `x`, `y`, and `foo` are all structurally represented as an integer.

The use of $+$ on `foo` could also be problematic, if an `enum` data type cannot be the target of $+$.

Pointers are an interesting example, since they are all structurally the same. Good language design and programming practice suggest distinguishing between pointers to different types.

C castigates those who fail to cast between pointers of different types.

Representing types [33]

The lineage of a type can be represented without reference to specific type names. A bit-vector representation is convenient for construction and for comparison:

Each of the types extenders is assigned a bit pattern from k bits:

Type	Pattern
ptr	01
array	10
func	11

So that a pointer to type X is represented as

01X

Note that this scheme does not track array index types or function parameter types.

Wisely leaving one pattern free, we now assign the base types:

Type	Pattern
void	0000
char	0001
int	0010
float	0011

Declaration	Type representation
`int x`	0010
`int **x[]`	01 01 10 0010
`char (*(*x())[])()`	11 01 10 01 11 0001

The last entry is a function that returns a pointer to an array of pointers to functions that return characters.