NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 11: Type systems and attribute grammars

Michael Engel

Overview

- Type systems
  - Type checking
- Syntax-directed translation
  - Attribute grammars

Types and type systems

Semantic analysis

- Type systems can specify program behavior at a more precise level than is possible in a context-free grammar
- Type systems create a second vocabulary for describing both the ***form and behavior*** of valid programs
- Type systems yield information that cannot be obtained using the techniques of scanning and parsing

- Three distinct purposes:
  - safety
  - expressiveness
  - runtime efficiency

Type safety

 +  = ?

Semantic analysis

- Ensure that the results/parts of assignments and expressions are ***compatible*** with each other
  - Providing types for data objects and rules for type inference help the compiler with this
- (Bad?) alternatives:
  - untyped (assembly, BCPL) and weakly typed languages
  - there are ideas for a ***typed assembly language*** [1]
- Compiler performs ***type checking***
  - compiler must analyze the program and assign a type to each name and each expression
  - it must check these types to ensure that they are used in contexts where they are legal
  - unfortunate misnomer, it lumps together the separate activities of type inference and identifying type-related errors

Drawbacks of type safety

- Wirth's Pascal programming language has a (quite) strict type system [2]
- The size of an array is part of its type
  - If one declares

```
var arr10 : array [1..10] of integer;
    arr20 : array [1..20] of integer;
```

  - then `arr10` and `arr20` are arrays of 10 and 20 integers respectively
- Suppose we want to write a procedure '`sort`' to sort an integer array
- Because `arr10` and `arr20` have different types, it is ***not possible to write a single procedure that will sort them both!***

Drawbacks of type safety (2)

- Even worse, strings in Pascal are arrays of char
- Consider writing a function **`index(s,c)`** that will return the position in the string s where the character c first occurs, or zero if it does not
  - The problem is how to handle the string argument of **`index`**
  - The calls **`index('hello',c)`** and **`index('goodbye',c)`** cannot both be legal, since the strings have different lengths
- Idea: use

```
var temp : array [1..10] of char;
temp := 'hello';
n := index(temp,c);
```

- but the assignment to '**`temp`**' is illegal because '**`hello`**' and '**`temp`**' are of different lengths!

Drawbacks of safety (3)

Semantic analysis

- Practical (?!?) solutions:
  - define family of routines with a member for each possible string size!
  - or define all strings (including constant strings like **`'define'`**) to have the same length → ***used in practice!***

```
type string = array [1..MAXSTR] of char;
```

- This wastes a lot of memory (especially on the small machines Pascal was developed on)
- Wirth himself uses this in his compilers, e.g. in Pascal-S [3]:

```
word[beginsym ]:= 'begin     '; word[endsym   ]:= 'end       ';
word[ifsym    ]:= 'if        '; word[thensym  ]:= 'then      ';
word[elsesym  ]:= 'else      '; word[whilesym ]:= 'while     ';
word[dosym    ]:= 'do        '; word[casesym  ]:= 'case      ';
word[repeatsym]:= 'repeat    '; word[untilsym ]:= 'until     ';
word[forsym   ]:= 'for       '; word[tosym    ]:= 'to        ';
word[downtosym]:= 'downto    '; word[notsym   ]:= 'not       ';
```

Expressiveness

Semantic analysis

- Types allow to specify behavior more precisely than is possible with context-free rules
- Example: *operator overloading*
  - gives *context-dependent meanings* to an operator
  - example: operator "`+`" for `int`, `float`, `double`, `string`, ...

```
int x = 1,
    y = 2, z;
z = x + y;
// z = 3
```

```
double x = 1.2,
       y = 2.3, z;
z = x + y;
// z = 3.5
```

```
string x = "Hello";
string y = "World";
z = x + y;
// z = "HelloWorld"
```

That doesn't work in C, of course...

- An untyped language might have to provide lexically different operators for each case
  - e.g. BCPL: "`+`" for `int`s, "`#+`" for `float`s

Generating Better Code

Semantic analysis

- Defining types provides detailed information about every expression in the program
- Example:
  - runtime type analysis and conversion for untyped languages
  - static generation of correct assembly statements
- Runtime type checking requires a runtime representation for type
  - each variable has a value field and a tag field => overhead!
- Knowing types at compile time allows generation of efficient code

Type of			(Pseudo) assembler code
a	b	a+b	
int	int	int	**add** r_a, r_b => r_{a+b}
int	float	float	**i2f** f_a => r_{a_f} **fadd** r_{a_f}, r_b => r_{a_f+b}
int	double	double	**i2d** f_a => r_{a_d} **dadd** r_{a_d}, r_b => r_{a_f+d}

Generating Better Code

Semantic analysis

If types are known at runtime only, the compiler has to insert ***runtime type conversions*** into the generated code

```
// partial code for "a+b => c"
if (tag(a) = integer) then
  if (tag(b) = integer) then
    value(c) = value(a) + value(b);
    tag(c) = integer;
  else if (tag(b) = real) then
    temp = ConvertToReal(a);
    value(c) = temp + value(b);
    tag(c) = real;
  else if (tag(b) = …) then
    // handle all other types…
else
  signal runtime type fault
…
```

```
else if (tag(a) = real) then
  if (tag(b) = integer) then
    temp = ConvertToReal(b);
    value(c) = value(a) + temp;
    tag(c) = real;
  else if (tag(b) = real) then
    value(c) = value(a) + value(b);
    tag(c) = real;
  else if (tag(b) = …) then
    // handle all other types…
else
    signal runtime type fault
else if (tag(a) = …) then
  // handle all other types…
else
  signal illegal tag value;
```

Components of a type system

Base types: directly supported by most processors

- *Numbers*: limited-range **integers** (e.g., $-2^{-31} \ldots 2^{31}-1$)
  approximate real-numbers (**floating point**)
  - Often, underlying hardware implementation influences availability of number types (e.g. "int" in C)
- *Characters*: traditionally, support for 7 or 8 bit ASCII characters
  more recently, UTF16 (Windows), UTF8 (common)
- *Booleans*: values `TRUE` and `FALSE` + logic operators (`and`, `xor`, …)

Other possible base types (examples)

- Lisp provides a recursive basic type for *lists* (=> Lisp machines)
- Complex numbers (DSP compilers) or vectors of numbers

Compound and constructed types

Semantic analysis

Combinations of elements of the base type

- *Arrays*: groups together multiple elements of the same type (base or compound), e.g. array with 10 integers `int a[10]`
  - many languages support *multi-dimensional* arrays: `int a[10]`
- *Strings*: some languages treat strings as compound types
  - most common: character strings, sometimes bit strings
- A true string differs from an array type in several important ways
  - can have operations like concatenation, translation, and computing the length
  - can be compared, e.g. in lexicographic order: "bar" < "foo"
- *Enumerated types*: giving (successive) numbers to named elements, e.g. weekdays, months or colors

```
enum weekday {Mon, Tue, Wed, Thu, Fri, Sat, Sun} // Mon < Wed
```

Compound and constructed types

Semantic analysis

- *Structures (records)*: group together multiple objects of *arbitrary type*
  - elements (members) of the structure are typically given explicit names, e.g. in structures for a parse tree for a compiler:

```
struct Node1 {
  struct Node1 *left;
  unsigned Operator;
  int Value
}
```

```
struct Node2 {
  struct Node2 *left;
  struct Node2 *right;
  unsigned Operator;
  int Value
}
```

- The type of a structure is the ordered product of the types of the individual elements that it contains
  - Type of a `Node1`: (`Node1 *`) × `unsigned` × `int`
  - Type of a `Node2`: (`Node2 *`) × (`Node2 *`) × `unsigned` × `int`
- These new types should have the same essential properties that a base type has

Compound and constructed types

Semantic analysis

- *Pointers*: abstract memory addresses that let the programmer manipulate arbitrary data structures
  - save an address and later examine the object that it addresses
  - often created when objects are created (`new` or `malloc`)
- Some languages provide an operator that returns the address of an object (`&` operator in C)
- Some languages restrict pointer assignment to "equivalent" types
  - protect from using a pointer to type t to reference a type s
- Some languages allow direct manipulation of pointers
  - arithmetic on pointers, including autoincrement and autodecrement, allow the program to construct new pointers
- Useful, but dangerous (especially with unexperienced programmers)
  - arbitrary pointers make reasoning about programs harder

Type equivalence

Semantic analysis

When does a language allow assignments/operations between different types? Two general approaches exist:

- *name equivalence*: two types are equivalent if and only if they have the same name
  - programmer can select any name for a type
  - if the programmer chooses different names, the language and its implementation should honor that deliberate act

```
typedef int length;
typedef int height;
length l;
height h = 42;
l = h; // not allowed
```

- *structural equivalence* asserts that two types are equivalent if and only if they have the same structure
  - two objects are interchangeable if they consist of the same set of fields, in the same order, and those fields all have equivalent types

```
struct {
   int x; int y;
} pixel;
struct {
   int temp; int humidity;
} weather;
weather = pixel; // OK
```

Inference rules

Semantic analysis

Inference rules specify, for each operator, the mapping between the operand types and the result type

- For some cases, the mapping is simple:
  - e.g., an assignment has one operand and one result: result (LHS) must have type compatible with RHS
- Often, relationship between operand types and result types is specified as recursive function on the type of the expression tree
  - the result type of an operation is a function of the types of its operands, e.g. specified using a table
  - compilers often recognize certain combinations of mixed-type expressions and automatically insert appropriate conversions

+	int	float	double
int	int	float	double
float	float	float	double
double	double	double	double

Attribute grammars

Semantic analysis

- Context-free grammar augmented with a set of rules
- Each symbol in the derivation (or parse tree) has a set of named values, or attributes
- The rules specify how to compute a value for each attribute
  - Attribution rules are functional; they uniquely define the value

Example grammar:

```
Number → Sign List
Sign   → +
       | -
List   → List Bit
       | Bit
Bit    → 0
       | 1
```

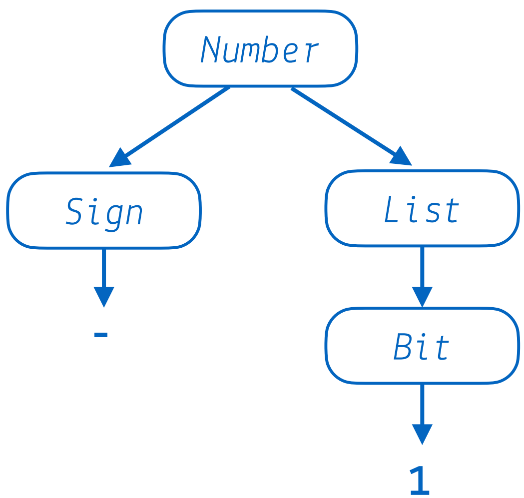
This grammar describes signed binary numbers

We will augment it with rules that compute the decimal value of each valid input string

Examples

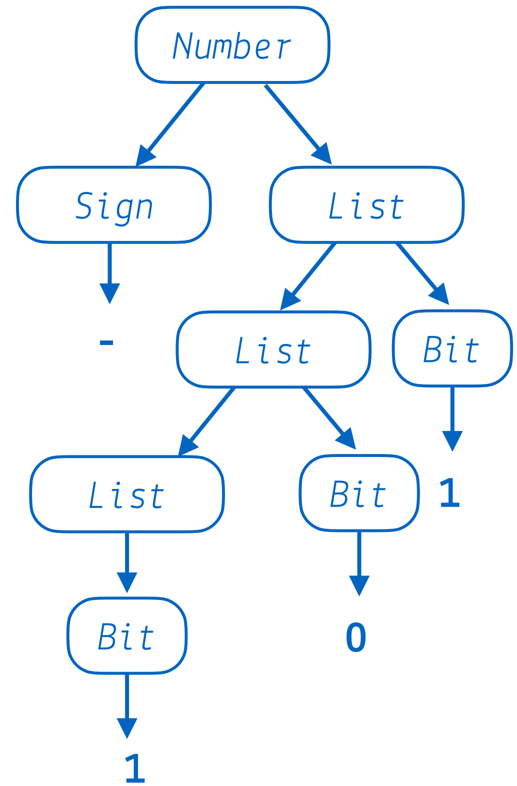
For "-1":

Number → *Sign List*
→ *Sign Bit*
→ *Sign* **1**
→ **- 1**



For "-101":

Number → *Sign List*
→ *Sign Bit*
→ *Sign List* **1**
→ *Sign List Bit* **1**
→ *Sign List* **0 1**
→ *Sign Bit* **0 1**
→ *Sign* **1 0 1**
→ **- 1 0 1**



Building attribute grammars

Semantic analysis

Add rules to compute the decimal value of a signed binary number

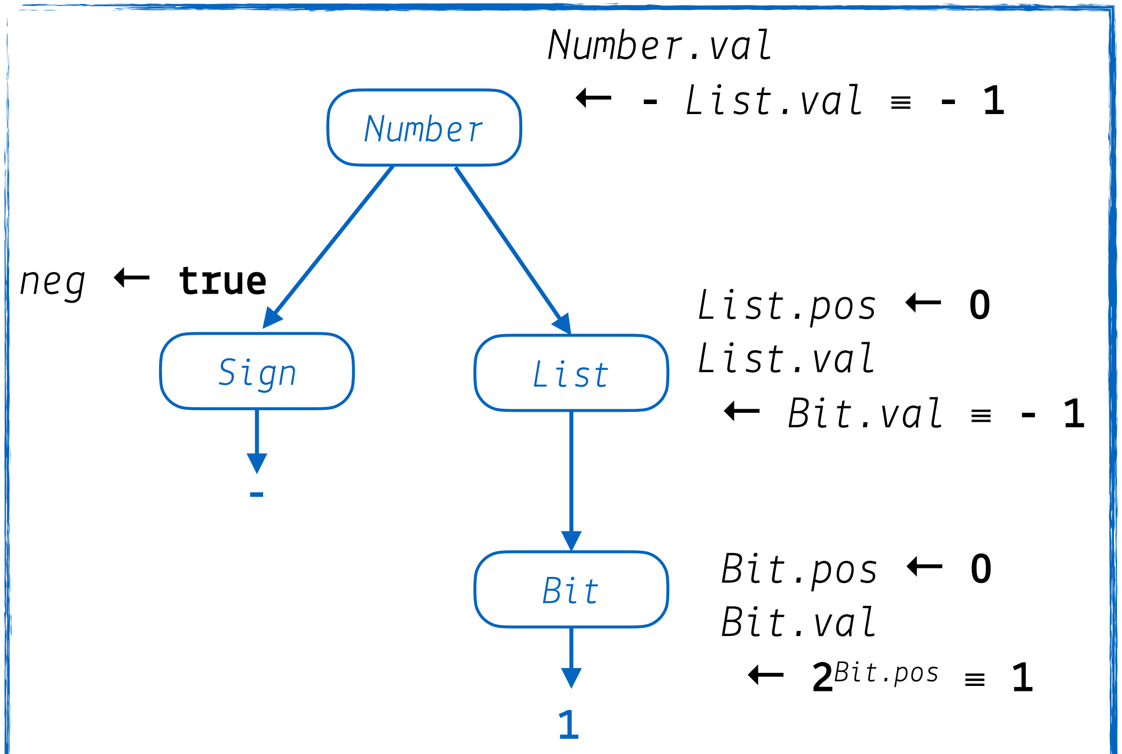
Production	Attribution rules
Number → *Sign List*	List.pos ← 0 if Sign.neg then Number.val ← - List.val else Number.val ← List.val
Sign → +	Sign.neg ← false
| -	Sign.neg ← true
$List_0$ → $List_1$ *Bit*	$List_1$.pos ← $List_0$.pos + 1 Bit.pos ← $List_0$.pos $List_1$.val ← $List_1$.val + Bit.val
| *Bit*	Bit.pos ← List.pos List.val ← Bit.val
Bit → 0	Bit.val ← 0
| 1	Bit.val ← $2^{\text{Bit.pos}}$

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

Attribute grammar for example 1

Semantic analysis

For "`-1`":



One possible evaluation order:

1. *List.pos*
2. *Sign.neg*
3. *Bit.pos*
4. *Bit.val*
5. *List.val*
6. *Number.val*

Other orders are possible

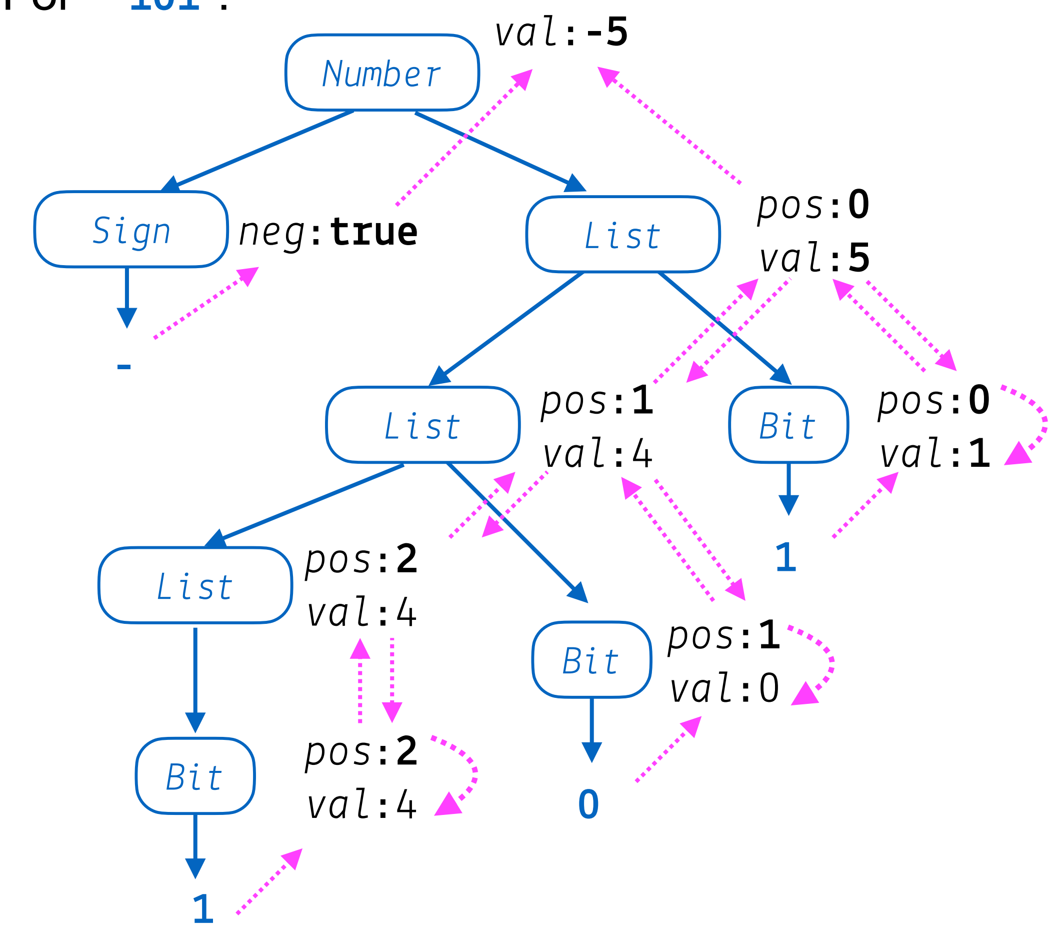
Evaluation order must be consistent with the attribute dependence graph

Knuth suggested a data-flow model for evaluation [4]:

- Independent attributes first
- Others in order as input values become available

Attribute grammar for example 2

For "**-101**":



This is the complete attribute dependence graph for "**-101**"

It shows the flow of all attribute values in the example

Some flow downward
→ ***inherited attributes***

Some flow upward
→ ***synthesized attributes***

A rule may use attributes in the parent, children, or siblings of a node

Applying the rules

- Attributes associated with nodes in parse tree
- Rules are value assignments associated with productions
- Attribute is defined once, using local information
- Label identical terms in production for uniqueness
- Rules & parse tree define an attribute dependence graph
  - Graph must be non-circular

This produces a high-level, functional specification

Synthesized attribute

- Depends on values from children

Inherited attribute

- Depends on values from siblings & parent

The attribute dependence graph is a specification for the *computation*, not an algorithm

Using attribute grammars

Semantic analysis

Attribute grammars can specify context-sensitive actions

- Take values from syntax
- Perform computations with values
- Insert tests, logic, …

Synthesized attributes

- Use values from children & constants
- S-attributed grammars
- Evaluate in a single bottom-up pass

Good match to LR parsing

Inherited attributes

- Use values from parent, constants & siblings
- Directly express context
- Can rewrite to avoid them
- Thought to be more natural

Not easily done at parse time

We want to use both kinds of attributes

Evaluation methods

Dynamic, dependence-based methods

- Build the parse tree
- Build the dependence graph
- Topological sort the dependence graph
- Define attributes in topological order

Rule-based methods (treewalk)

- Analyze rules at compiler-generation time
- Determine a fixed (static) ordering
- Evaluate nodes in that order

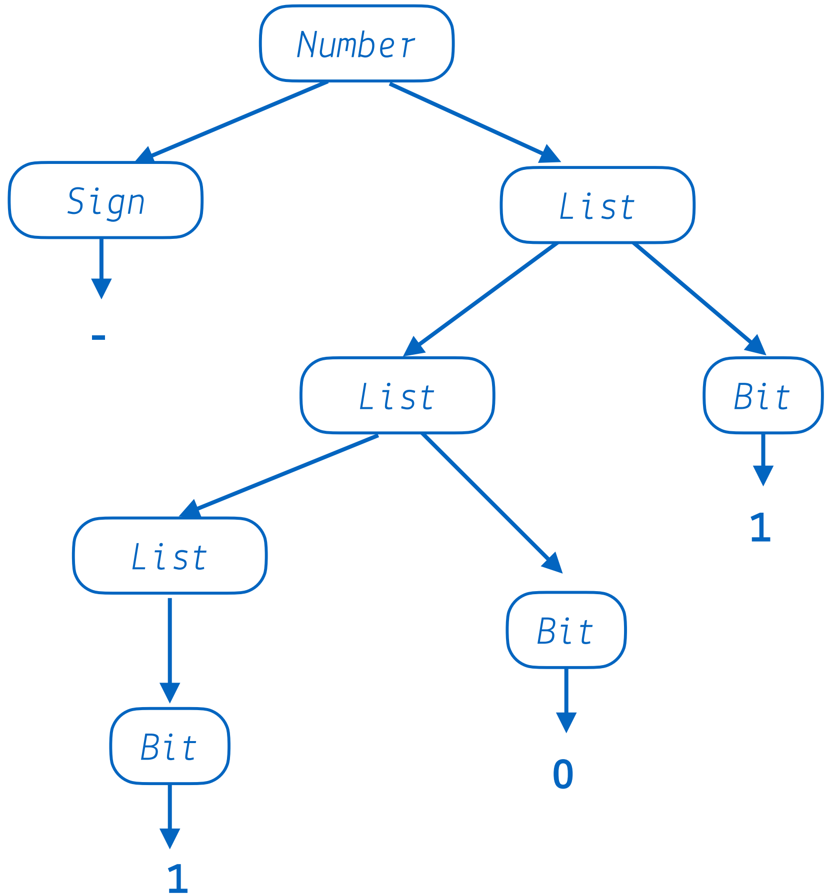
Oblivious methods (passes, dataflow)

- Ignore rules & parse tree
- Pick a convenient order (at design time) & use it

Back to the example

Semantic analysis

For "**-101**":

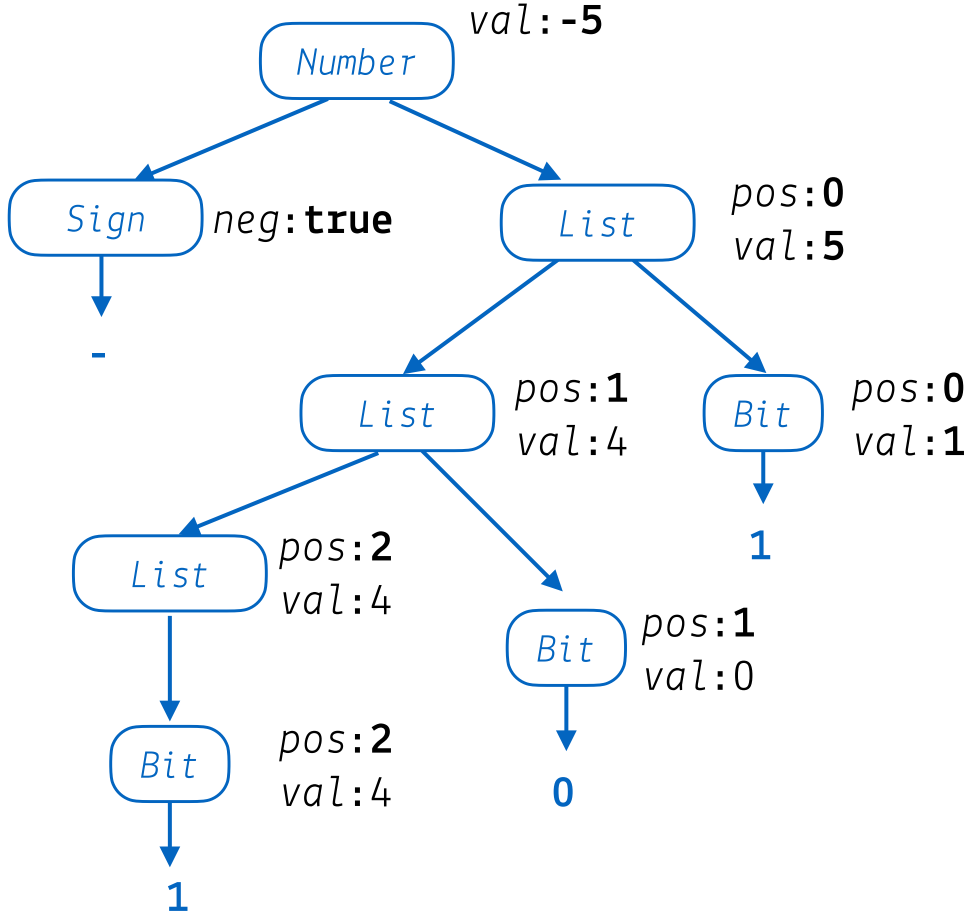


Syntax tree

Back to the example

Semantic analysis

For "**-101**":

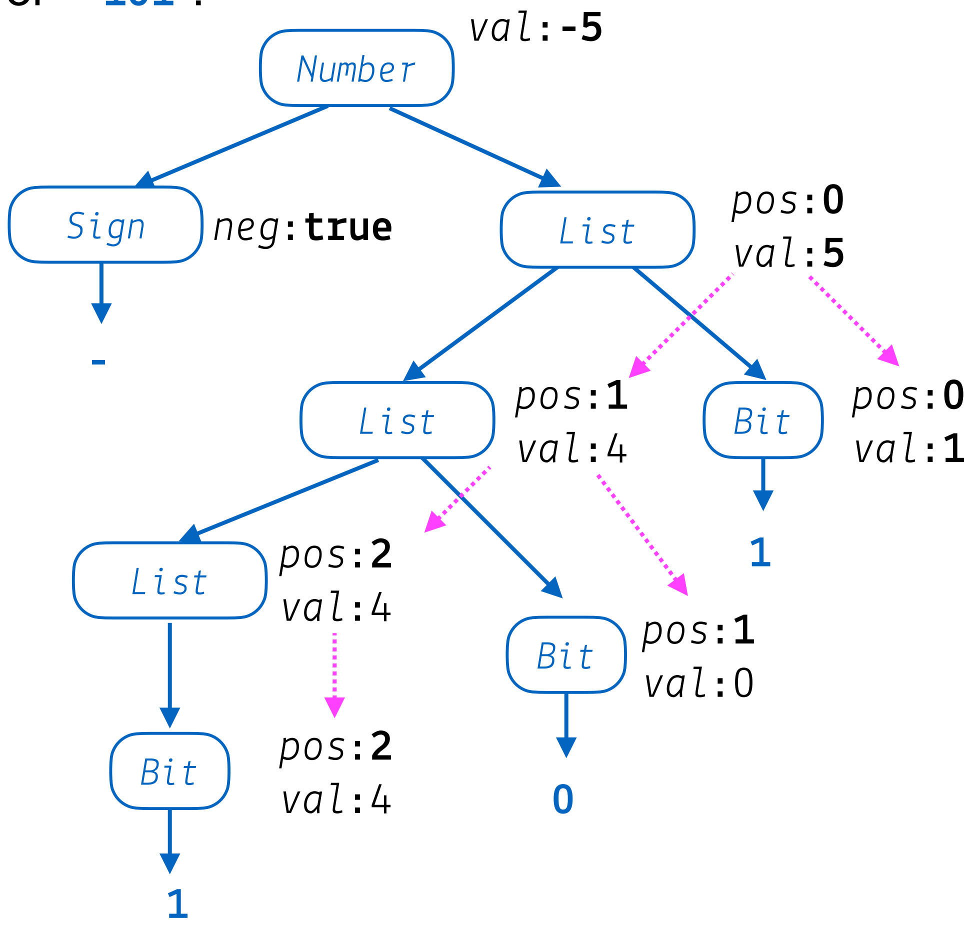


Attributed syntax tree

Back to the example

Semantic analysis

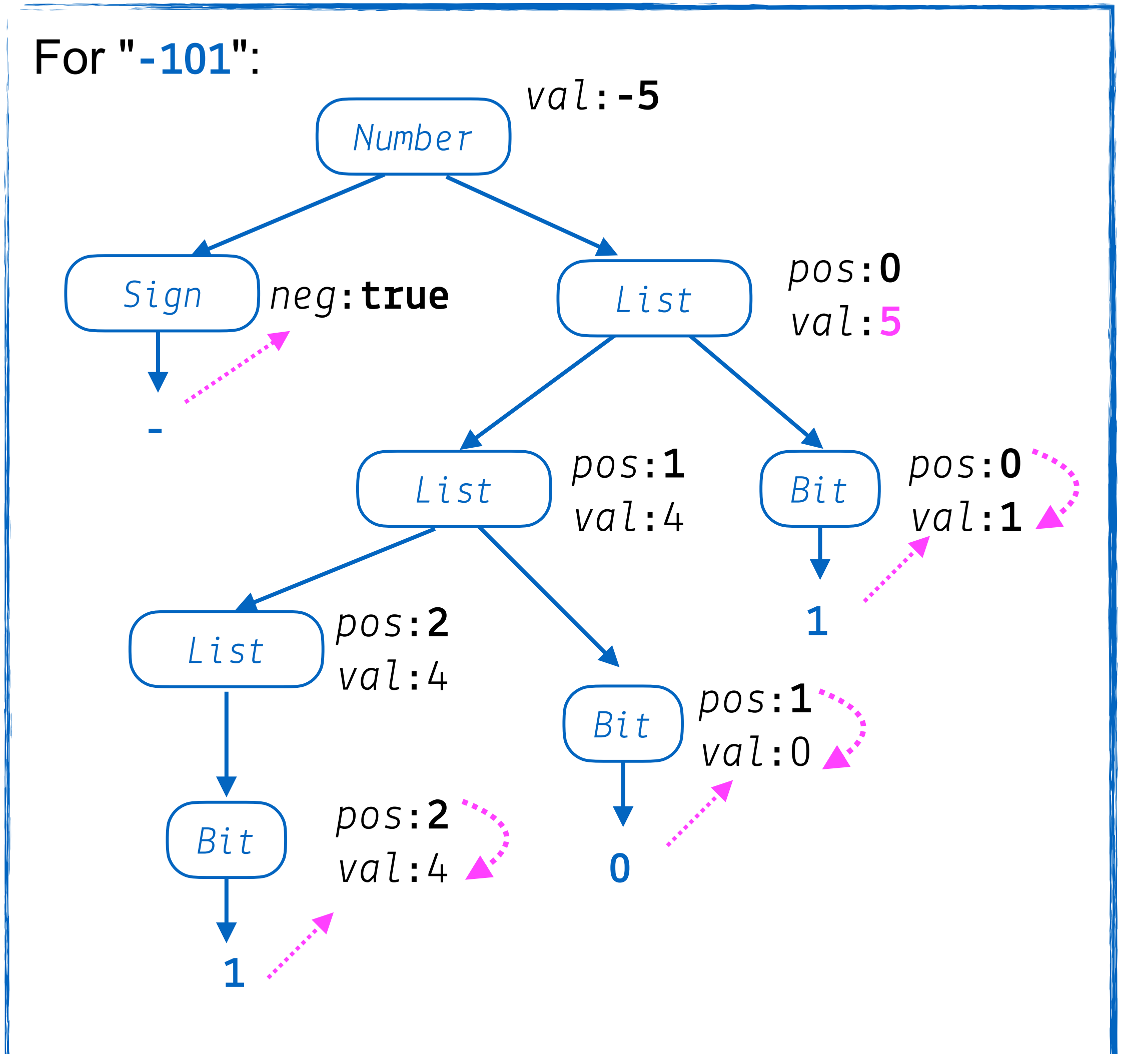
For "-101":



Inherited attributes

Back to the example

Semantic analysis



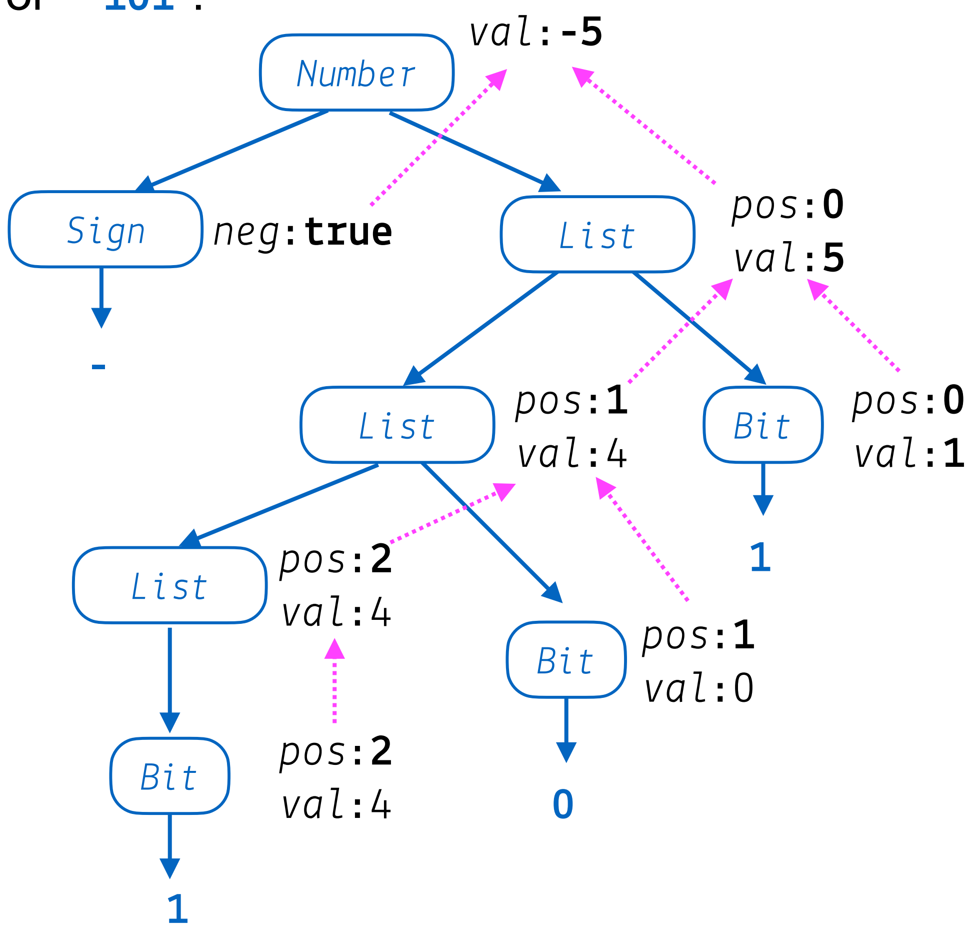
Synthesized attributes

val obtains values from children and the same node

Back to the example

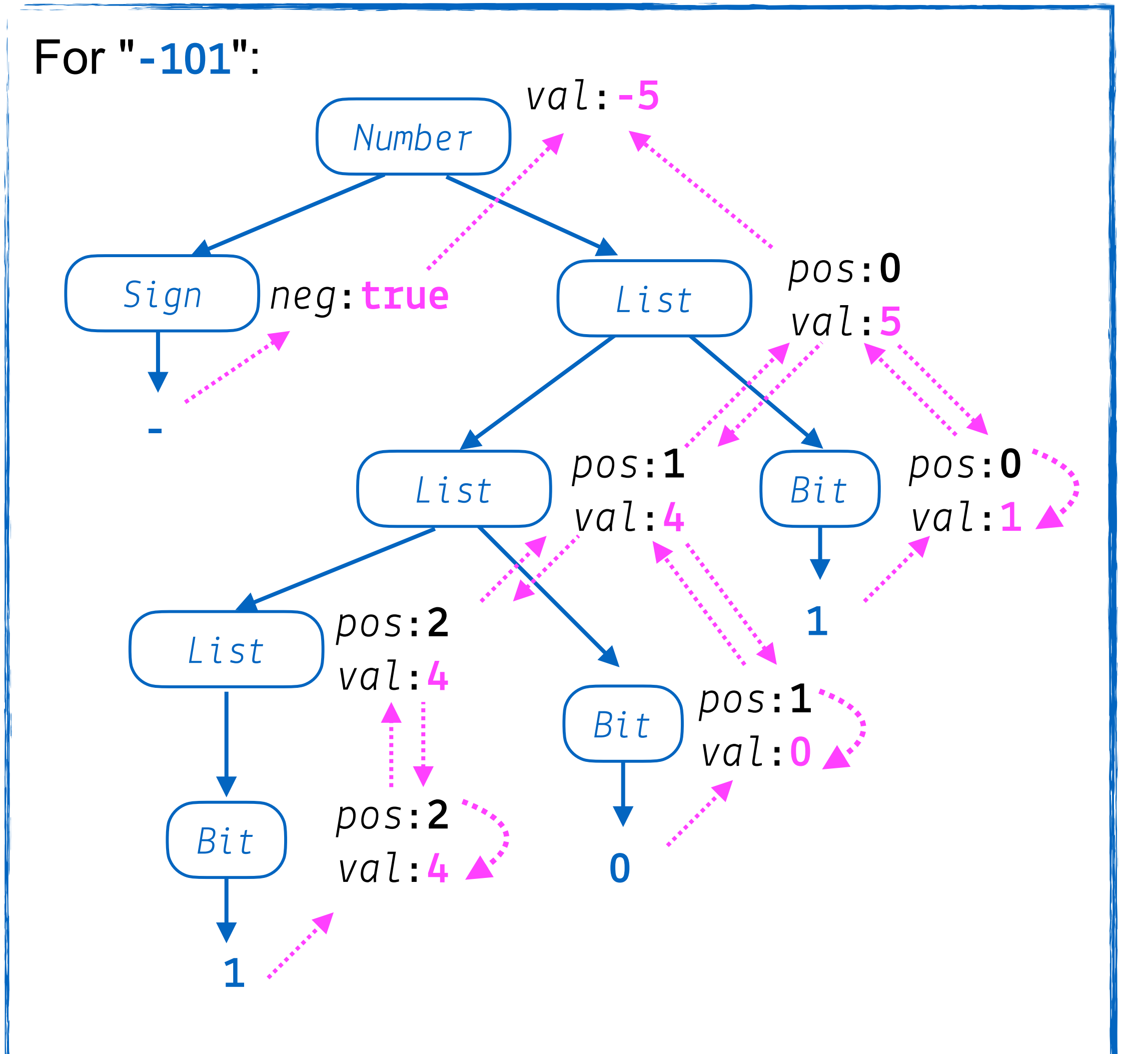
Semantic analysis

For "-101":



More synthesized attributes

Back to the example

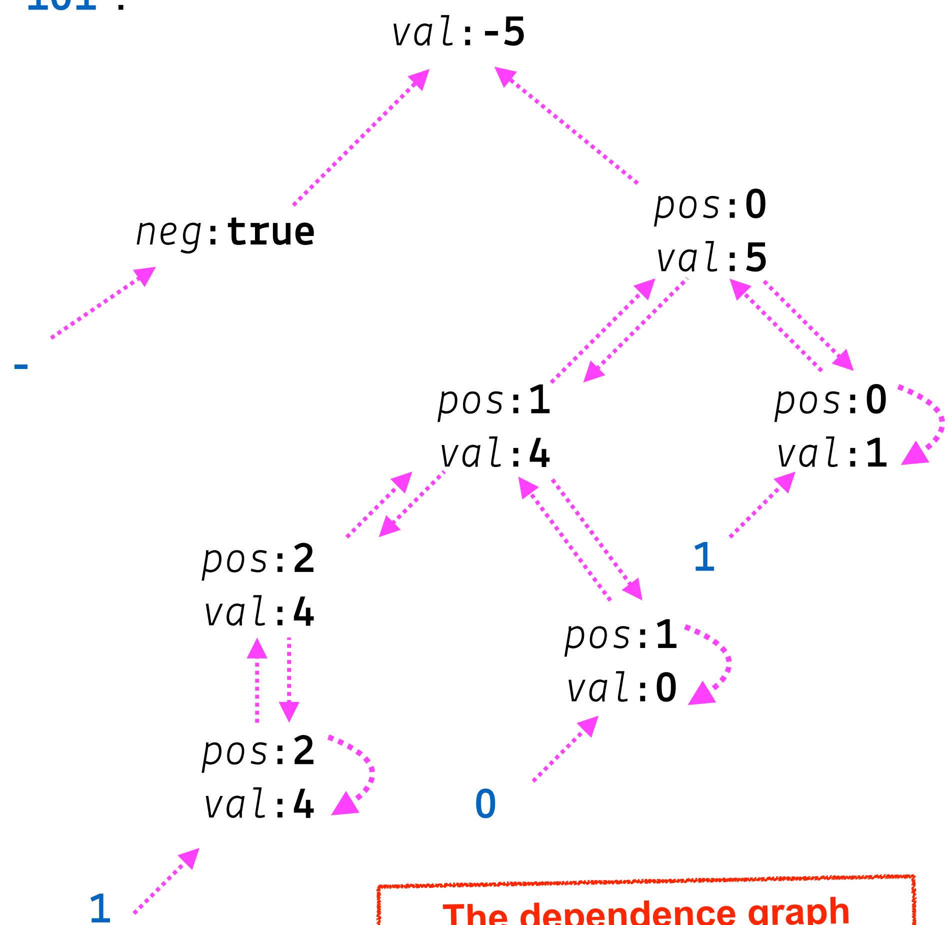


Let's show the computation…

and remove the syntax tree

Back to the example

For "-101":



The dependence graph *must* be acyclic!

All that is left is the ***attribute dependence graph***

This succinctly represents the flow of values in the problem instance

The dynamic methods sort this graph to find independent values, then work along graph edges

The rule-based methods try to discover "good" orders by analyzing the rules

The oblivious methods ignore the structure of this graph

Circularity

- We can only evaluate acyclic instances
- ***General circularity testing*** problem is inherently exponential!
- We can prove that some grammars can only generate instances with acyclic dependence graphs
    - Largest such class is “strongly non-circular” grammars (SNC) [5]
    - SNC grammars can be tested in polynomial time
    - Failing the SNC test is not conclusive

- Many evaluation methods discover circularity dynamically
  $\Rightarrow$ Bad property for a compiler to have

A circular attribute grammar

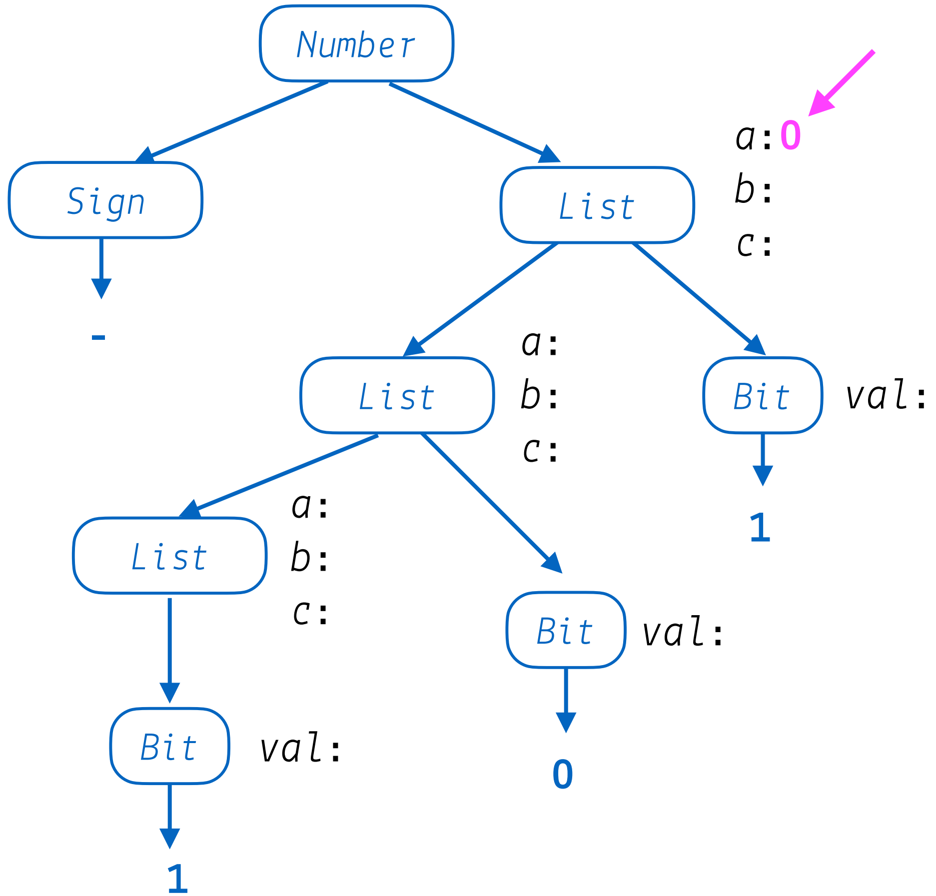
Semantic analysis

Production	Attribution rules
Number → *Sign List*	$List.a \leftarrow 0$
List₀ → *List₁ Bit*	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
| *Bit*	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
Bit → 0	$Bit.val \leftarrow 0$
| 1	$Bit.val \leftarrow 1$

The circularity is in the attribution rules, not the underlying CFG

Circular grammar example

For "-101":

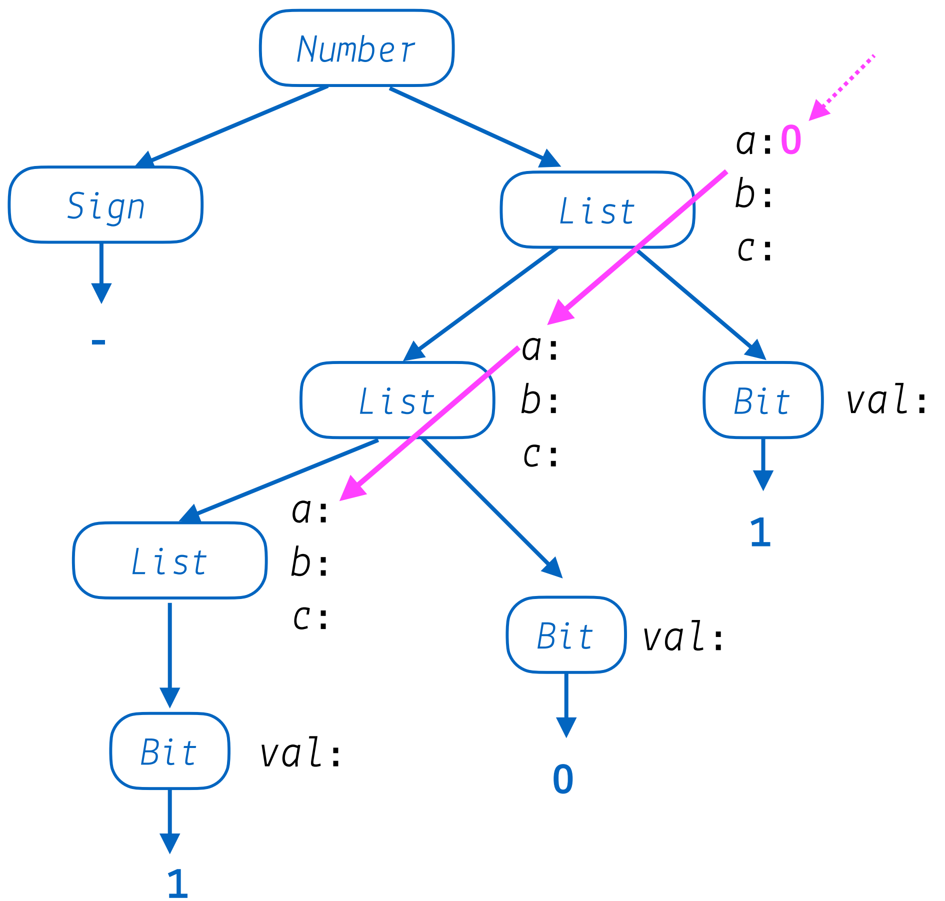


Production	Attribution rules
Number → *Sign List*	List.a ← 0
List$_0$ → *List*$_1$ *Bit*	List$_1$.a ← List$_0$.a + 1 List$_0$.b ← List$_1$.b List$_1$.c ← List$_1$.b + Bit.val
| *Bit*	List$_0$.b ← List$_0$.a + List$_0$.c + Bit.val
Bit → 0	Bit.val ← 0
| 1	Bit.val ← 1

Circular grammar example

Semantic analysis

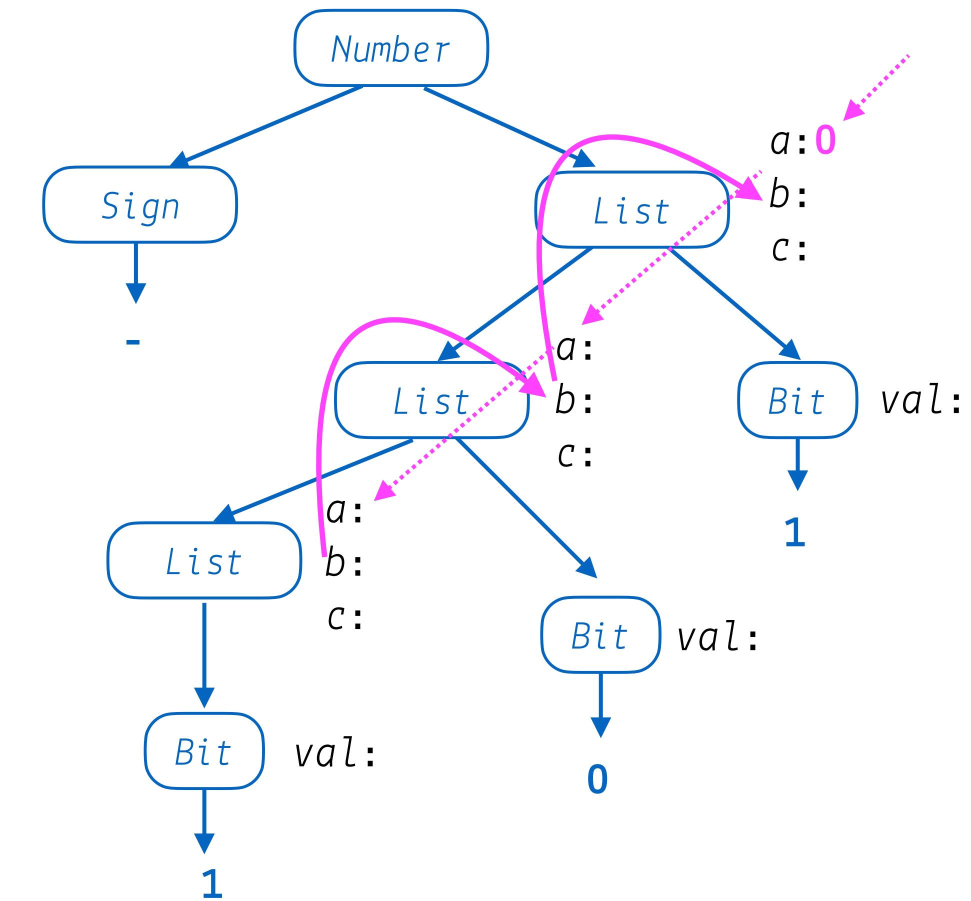
For "-101":



Production	Attribution rules
Number → *Sign List*	List.a ← 0
List$_0$ → *List*$_1$ *Bit*	List$_1$.a ← List$_0$.a + 1 List$_0$.b ← List$_1$.b List$_1$.c ← List$_1$.b + Bit.val
| *Bit*	List$_0$.b ← List$_0$.a + List$_0$.c + Bit.val
Bit → 0	Bit.val ← 0
| 1	Bit.val ← 1

Circular grammar example

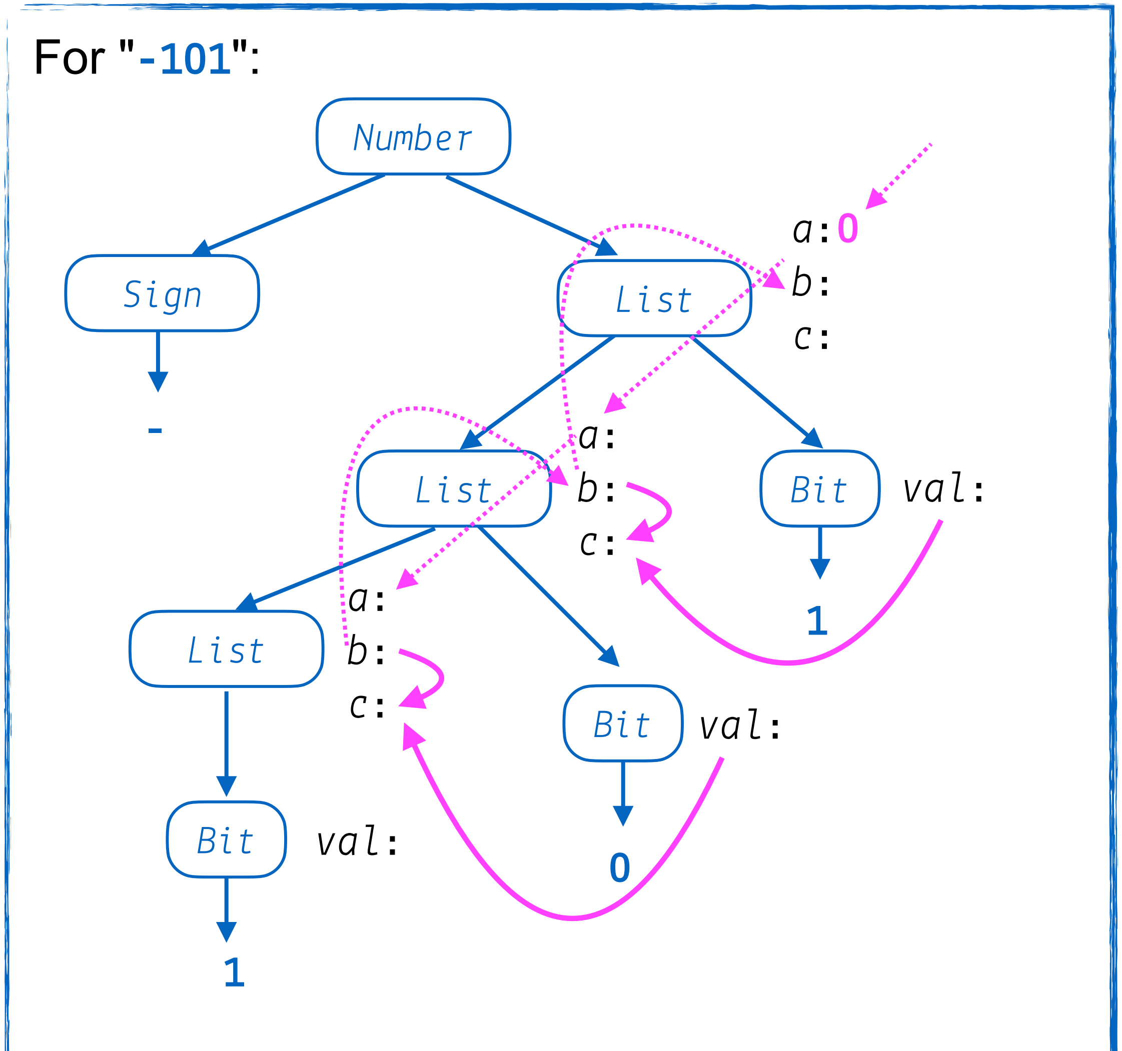
For "-101":



Production	Attribution rules
Number → *Sign List*	List.a ← 0
$List_0$ → $List_1$ *Bit*	$List_1$.a ← $List_0$.a + 1 $List_0$.b ← $List_1$.b $List_1$.c ← $List_1$.b + Bit.val
| *Bit*	$List_0$.b ← $List_0$.a + $List_0$.c + Bit.val
Bit → 0	Bit.val ← 0
| 1	Bit.val ← 1

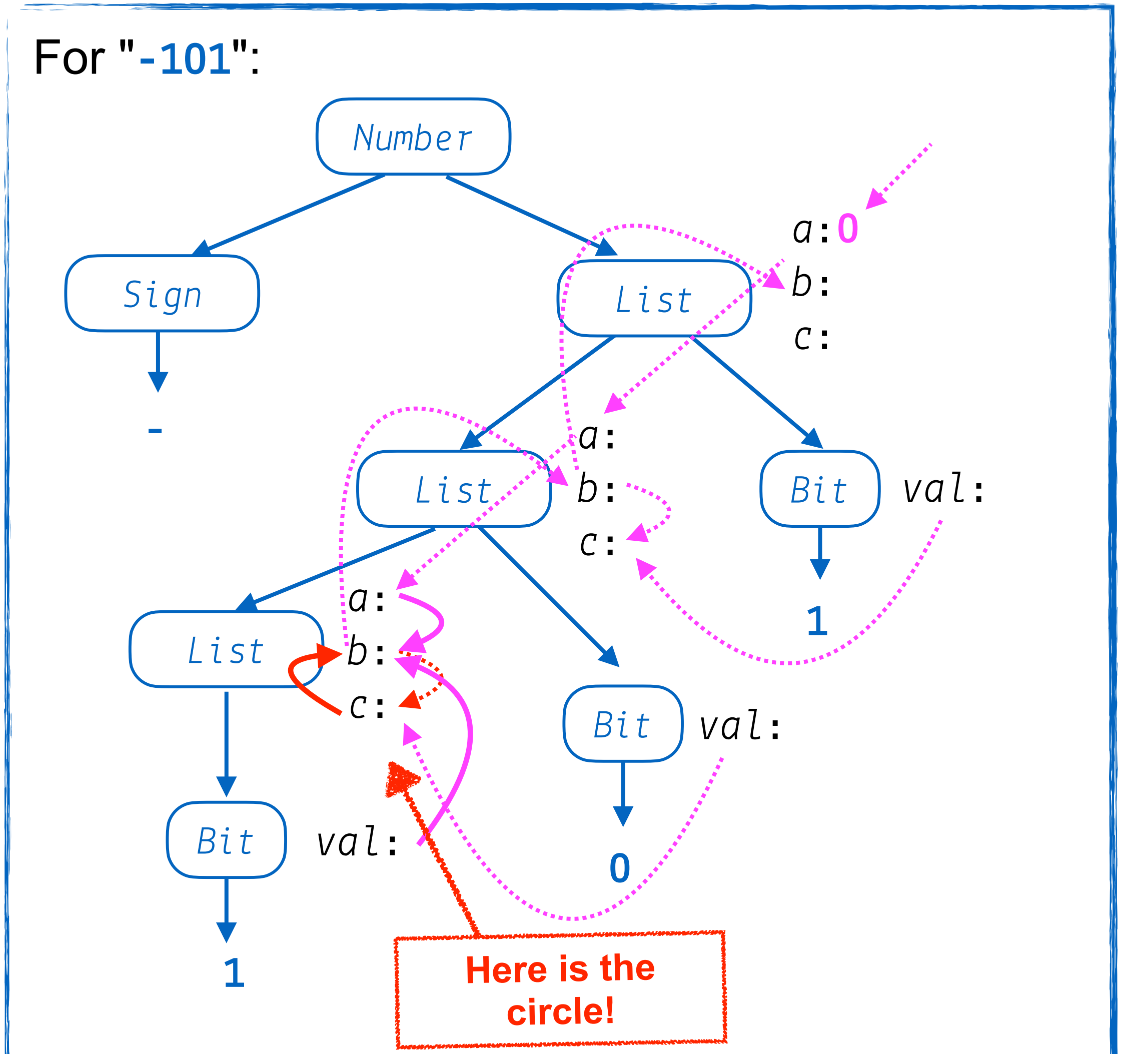
Circular grammar example

Semantic analysis



Production	Attribution rules
Number → *Sign List*	List.a ← 0
$List_0$ → $List_1$ *Bit*	$List_1$.a ← $List_0$.a + 1 $List_0$.b ← $List_1$.b $List_1$.c ← $List_1$.b + Bit.val
| *Bit*	$List_0$.b ← $List_0$.a + $List_0$.c + Bit.val
Bit → 0 | 1	Bit.val ← 0 Bit.val ← 1

Circular grammar example

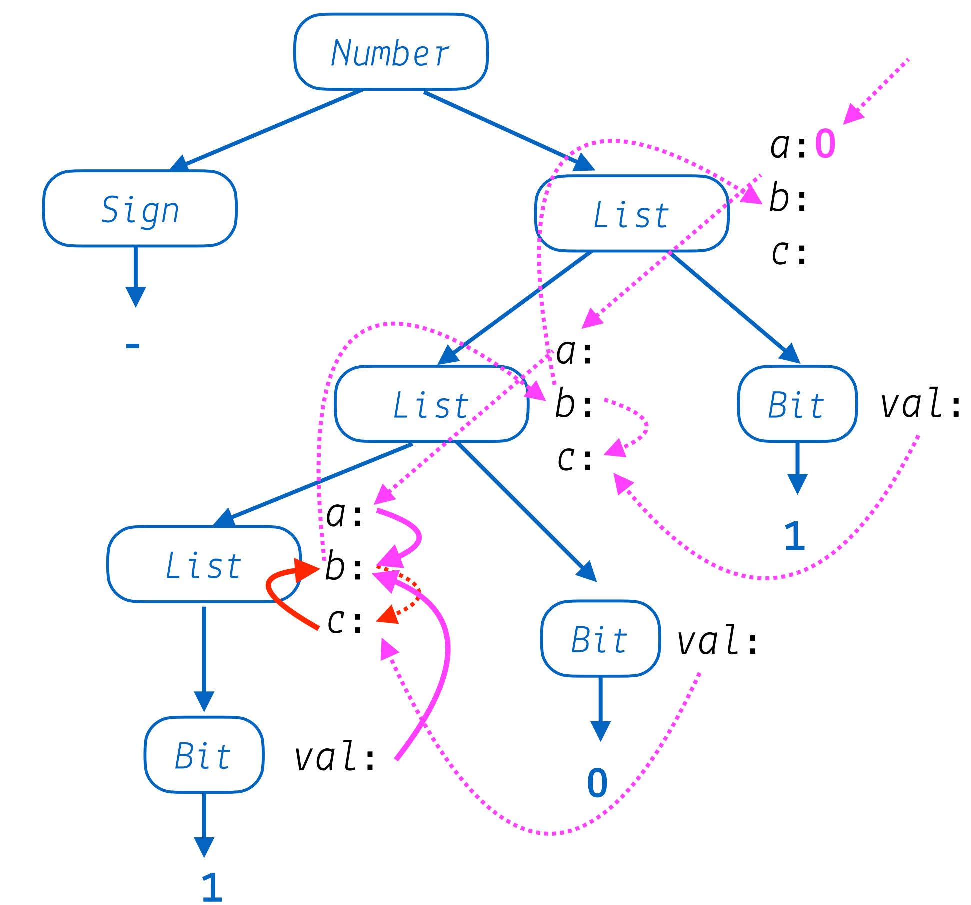


Production	Attribution rules
Number → *Sign List*	List.a ← 0
$List_0$ → $List_1$ *Bit*	$List_1$.a ← $List_0$.a + 1 $List_0$.b ← $List_1$.b $List_1$.c ← $List_1$.b + Bit.val
| *Bit*	$List_0$.b ← $List_0$.a + $List_0$.c + Bit.val
Bit → 0	Bit.val ← 0
| 1	Bit.val ← 1

Circular grammar example

Semantic analysis

For "-101":



Production	Attribution rules
Number → *Sign List*	List.a ← 0
$List_0$ → $List_1$ *Bit*	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
| *Bit*	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
Bit → 0 | 1	Bit.val ← 0 Bit.val ← 1

Here is the circle!

Circles – the point

- Circular grammars have indeterminate values
    - Algorithmic evaluators will fail
- Noncircular grammars evaluate to a unique set of values
- Circular grammar might give rise to noncircular instance
    - Probably shouldn't bet the compiler on it…

⇒ Should (undoubtedly) use provably noncircular grammars

Remember, we are studying AGs to gain insight

- We should avoid circular, indeterminate computations
- If we stick to provably noncircular schemes, evaluation should be easier

An extended attribute grammar ex.

Grammar for a basic block

```
1   Block0  → Block1 Assign
2            | Assign
3   Assign  → Ident = Expr ;
4   Expr0   → Expr1 + Term
5            | Expr1 - Term
6            | Term
7   Term0   → Term1 * Factor
8            | Term1 / Factor
9            | Factor
10  Factor  → ( Expr )
11           | Number
12           | Ident
```

Let's estimate cycle counts (again)

- Each operation has a COST
- Add them, bottom up
- Assume a load per value
- Assume no reuse

Simple problem for an attribute grammar

A quick look at basic blocks

Semantic analysis

Code in a ***basic block***

- has ***one entry point*** (at its start), so no code inside the block is the destination of a jump instruction anywhere in the program
- has ***one exit point***, so only the last instruction can cause the program to begin executing code in a different basic block

- This implies:
  whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order

```
i = 1;
j = 1;
k = 0;

while (k < 100) {
    if (j < 20) {
        j = i;
        k = k+1;
    } else {
        j = k;
        k = k+2;
    }
}
return j;
```

Source code

B1
```
i = 1;
j = 1;
k = 0;
```

B2
```
while (k < 100) {
```

B3
```
    if (j < 20) {
```

B4
```
        j = i;
        k = k+1;
```

B5
```
        j = k;
        k = k+2;
```

B6
```
return j;
```

Basic Blocks B1–B6

The code may be source code, assembly code or some other sequence of instructions

An extended example

Semantic analysis

Grammar for a basic block

1	$Block_0$	→	$Block_1$ $Assign$	`Block₀.cost ←`	`Block₁.cost + Assign.cost`
2		|	$Assign$	`Block₀.cost ←`	`Assign.cost`
3	$Assign$	→	$Ident$ **=** $Expr$ **;**	`Assign.cost ←`	**COST(store)** + `Expr.cost`
4	$Expr_0$	→	$Expr_1$ **+** $Term$	`Expr₀.cost ←`	`Expr₁.cost` + **COST(add)** + `Term.cost`
5		|	$Expr_1$ **-** $Term$	`Expr₀.cost ←`	`Expr₁.cost` + **COST(sub)** + `Term.cost`
6		|	$Term$	`Expr₀.cost ←`	`Term.cost`
7	$Term_0$	→	$Term_1$ ***** $Factor$	`Term₀.cost ←`	`Term₁.cost` + **COST(mul)** + `Factor.cost`
8		|	$Term_1$ **/** $Factor$	`Term₀.cost ←`	`Expr₁.cost` + **COST(div)** + `Factor.cost`
9		|	$Factor$	`Term₀.cost ←`	`Factor.cost`
10	$Factor$	→	**(** $Expr$ **)**	`Factor.cost ←`	`Expr.cost`
11		|	$Number$	`Factor.cost ←`	**COST(LoadImm)**
12		|	$Ident$	`Factor.cost ←`	**COST(Load)**

An extended example (contd.)

Properties of the example grammar

- All attributes are synthesized ⇒ so-called S-attributed grammar
- Rules can be evaluated bottom-up in a single pass
    - Good fit to bottom-up, shift/reduce parser
- Easily understood solution
- Seems to fit the problem well

What about an improvement?

- Values are loaded only once per block (not at each use)
- Need to track which values have been already loaded

A better execution model

Load tracking adds complexity

- But, most of it is in the “copy rules”
- Every production needs rules to copy Before & After

```
Factor  → ( Expr )      Factor.cost   ← Expr.cost
                        Expr.before   ← Factor.before
                        Factor.after  ← Expr.after
        | Number        Factor.cost   ← COST(LoadImm)
                        Factor.after  ← Factor.before
        | Ident         If (Ident.name ∉ Factor.before)
                          then Factor.cost   ← COST(Load)
                               Factor.after  ← Factor.before
                                             ∪ {Ident.name}
                          else Factor.cost   ← 0
                               Factor.after  ← Factor.before
```

A better execution model

Adding ***load tracking***

- This needs sets `Before` and `After` for each production
- Must be initialized, updated, and passed around the tree

An example production:

4	$Expr_0 \rightarrow Expr_1 + Term$	Expr_0	$\leftarrow$	$\text{Expr}_1\text{.cost} + \textbf{COST(add)} + \text{Term.cost}$
		$\text{Expr}_1\text{.before}$	$\leftarrow$	$\text{Expr}_0\text{.before}$
		Term.before	$\leftarrow$	$\text{Expr}_1\text{.before}$
		$\text{Expr}_1\text{.after}$	$\leftarrow$	Term.after

- These copy rules multiply rapidly
- Each creates an instance of the set
- Lots of work, lots of space, lots of rules to write

An even better model

What about accounting for finite register sets?

- Before & After must be of limited size
- Adds complexity to Factor → Identifier
- Requires more complex initialization

Jump from tracking loads to tracking registers is small

- Copy rules are already in place
- Some local code to perform the allocation

…and its extensions

Tracking loads

- Introduced Before and After sets to record loads
- Added ≥ 2 copy rules per production
- Serialized evaluation into execution order
- Made the whole attribute grammar large & cumbersome

Finite register set

- Complicated one production (Factor → Identifier)
- Needed a little fancier initialization
- Changes were quite limited

Why is one change hard and the other one easy?

Summing it up

- Non-local computation needed lots of supporting rules
- Complex local computation was relatively easy

The problems

- Copy rules increase cognitive overhead
- Copy rules increase space requirements
    - Need copies of attributes
    - Can use pointers, for even more cognitive overhead
- Result is an attributed tree
    - Must build the parse tree
    - Either search tree for answers or copy them to the root

⇒ ***in practice, ad-hoc solutions are used (see previous lecture)***

What’s next?

Semantic analysis

- Three-address code and intermediate representations

References

[1] Crary, K., et al. (1999)
TALx86: A realistic typed assembly language.
ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA.

[2] Kernighan, Brian W. (1984)
Why Pascal is not my favorite programming language.
Computer Science Technical Report 100, Bell Laboratories, Murray Hill, NJ, USA, July 1981.
Available online at http://cm.bell-labs.com/cm/cs/cstr.

[3] Wirth, Niklaus (1981)
Pascal-S: A Subset and its Implementation.
In Pascal - The Language and its Implementation 1981: 199-259

[4] Knuth, D.E. (1990)
The genesis of attribute grammars.
In: Deransart P., Jourdan M. (eds) Attribute Grammars and their Applications.
Lecture Notes in Computer Science, vol 461. Springer, Berlin, Heidelberg

[5] Kennedy, K., Warren, S.K. (1976)
Automatic generation of efficient evaluators for attribute grammars.
In: Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages, POPL 1976, pp. 32–49. ACM, New York