Context-sensitive Analysis

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
  int a, b, c, d;
{ … }

fee() {
  int f[3],g[0],h,i,j,k;
   char *p;
  fie(h,i,“ab”,j, k);
  k = f * i + j;
  h = g[17];
  printf(“<%s,%s>.\n”,p,q);
  p = 10;
}
```

What is wrong with this program?

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
 int a, b, c, d;
{ … }

fee() {
 int f[3],g[0],h,i,j,k;
  char *p;
 fie(h,i,“ab”,j, k);
 k = f * i + j;
 h = g[17];
 printf(“<%s,%s>.\n”, p,q);
 p = 10;
}
```

What is wrong with this program?

- **declared g[0], used g[17]**
- **wrong number of args to fie()**
- **“ab” is not an int**
- **wrong dimension on use of f**
- **undeclared variable q**
- **10 is not a character string**

All of these are

“deeper than syntax”

To generate code, we need to understand its meaning !

Beyond Syntax

To generate code, the compiler needs to answer many questions

- Is "x" a scalar, an array, or a function? Is "x" declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of "x" does each use reference?
- Is "x" defined before it is used?
- Is the expression "x * y + z" type-consistent?

These are beyond a Context Free Grammar

Beyond Syntax

To generate code, the compiler needs to answer many questions

- In "*a*[i,j,k]", does *a* have three dimensions?
- Where can "z" be stored? *(register, local, global, heap, static)*
- How many arguments does "fie()" take?
- Does "*p" reference the result of a "malloc()" ?
- Do "p" & "q" refer to the same memory location?

These are beyond a Context Free Grammar

Beyond Syntax

These questions are part of context-sensitive analysis

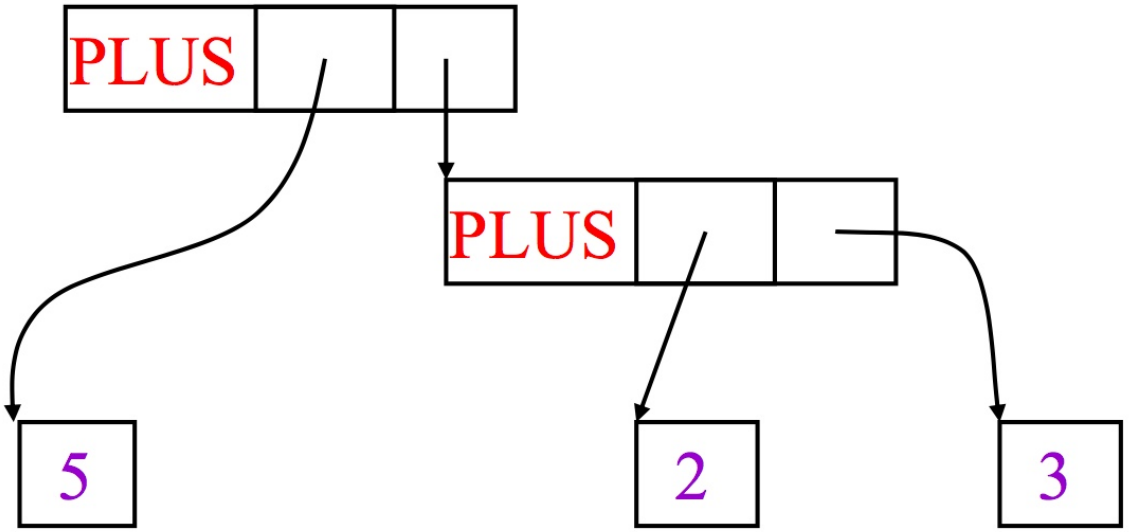
- Questions & answers involve non-local information
- Answers may involve computation

How can we answer these questions?

- Use formal methods
  - → Attribute grammars?
    - Also known as attributed CFG or syntax-directed definitions
- Use *ad-hoc* techniques
  - → Symbol tables
  - → *Ad-hoc* code **(action routines)**

In scanning & parsing, formalism won; different story here.

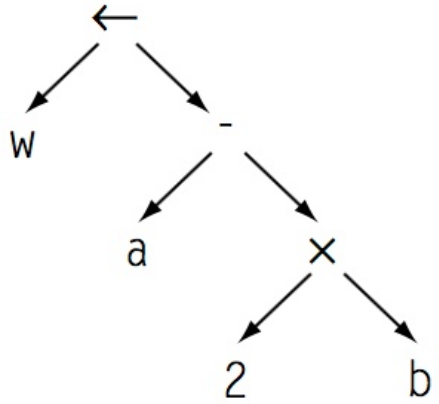
Example of an Abstract Syntax Tree



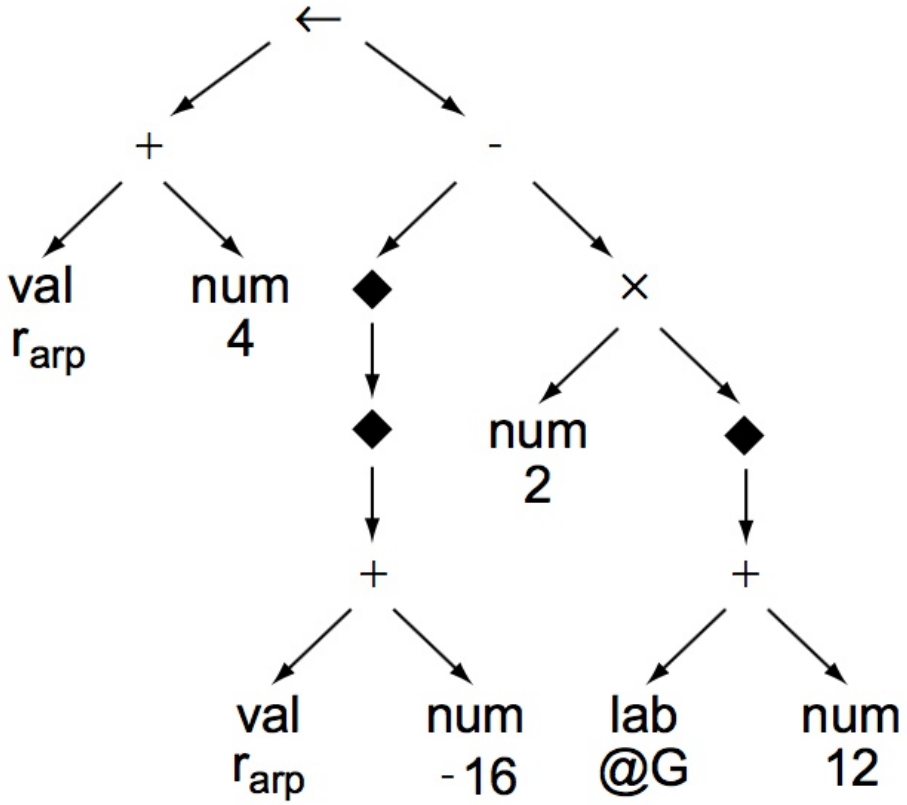
- Also captures the nesting structure
- But <u>abstracts</u> from the concrete syntax

  ↦ more compact and easier to use
- An important data structure in a compiler

More Abstract Syntax Tree Examples



(a) Source-Level AST



(b) Low-Level AST

ASTs and Parsing

- AST can be built doing a bottom-up parse
- The construction procedure is rather simple
  - Create appropriate tree nodes for each element in the grammar
  - for each node, carry sufficient essential information of program fragments it represents

What type of trees they have built by using CUP?

Read the code of the skeleton to understand what they are operating on

Inheritance Graph