

Computer Science 160

Translation of Programming Languages

Instructor: Christopher Kruegel
(based on material by Tim Sherwood)

Context-Sensitive Analysis

Context-Sensitive Analysis

UC Santa Barbara

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;
  call 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”

Beyond Syntax

UC Santa Barbara

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 the expression “x * y + z” type-consistent?
- 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” and “q” refer to the same memory location?
- Is “x” defined before it is used?

These are beyond a CFG

Beyond Syntax

UC Santa Barbara

These questions are part of context-sensitive analysis

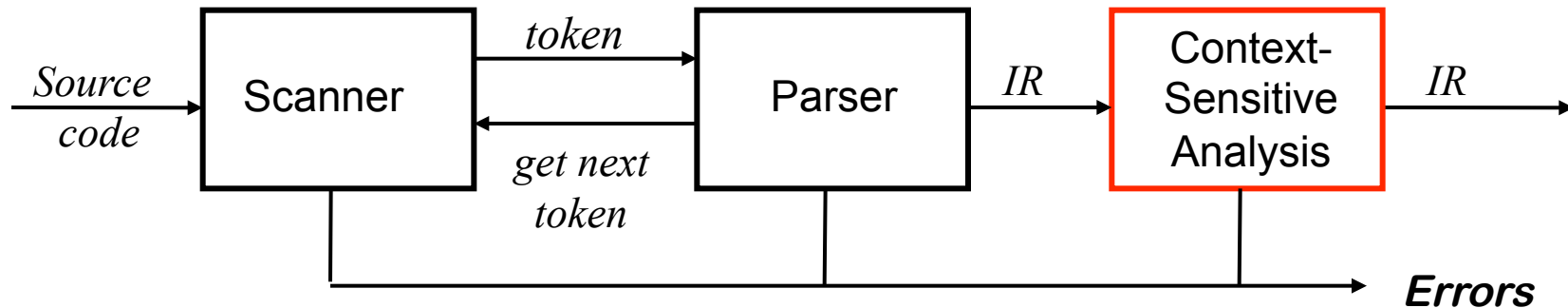
- Answers depend on values, not just tokens
 - answers depend on attributes of tokens
- Questions and answers involve non-local information
 - variable declarations, procedures
- Answers may involve computation

How can we answer these questions?

- Use formal methods
 - Context-sensitive grammars
 - Attribute grammars (semantic rules do not have side effects)
 - Use ad-hoc techniques
 - Symbol tables
 - Ad-hoc code (use semantic rules that can have side effects)
-

Context-Sensitive Analysis

UC Santa Barbara



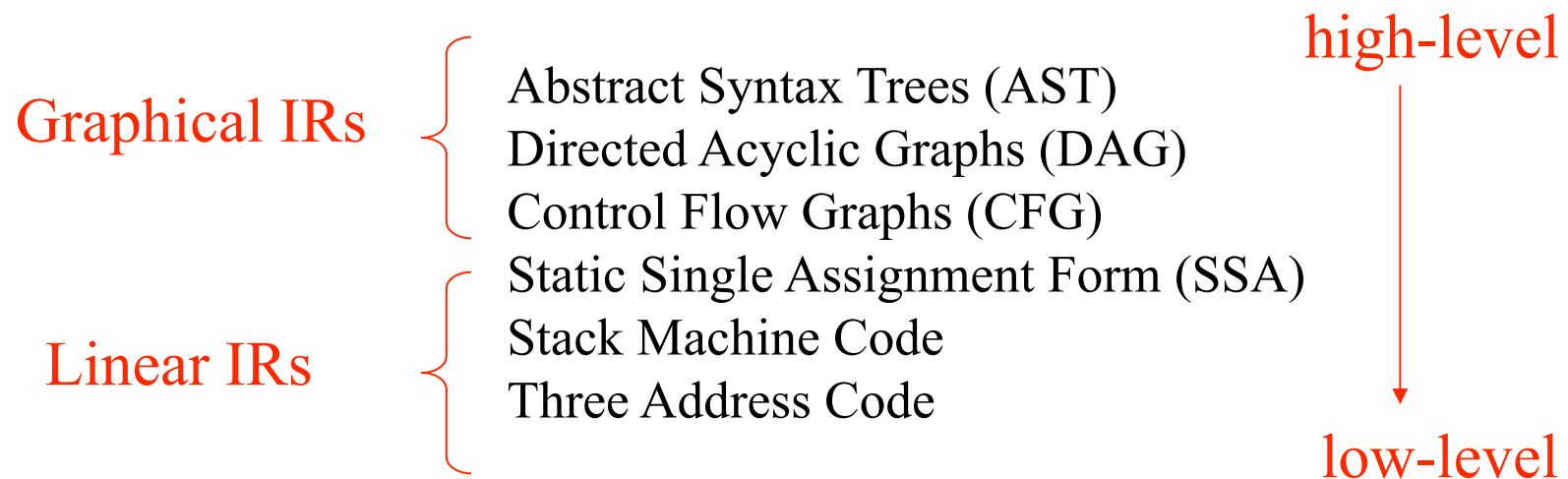
Context-Sensitive Analysis

- Properties we want to check involve non-local information
 - type-checking, variable declarations, procedure declarations
- Can be implemented as a traversal on the abstract syntax tree
- Can be integrated to parsing phase using ad-hoc translation schemes
- Formalisms such as attribute grammars can be used
- Ad-hoc techniques are more common than the formal techniques

Intermediate Representations: Overview

UC Santa Barbara

- There is more than one way to represent code as it is being generated, analyzed, and optimized.



Abstract Syntax Trees: Overview

UC Santa Barbara

- A compiler must do more than to recognize whether a sentence belongs to the language of a grammar in the form of syntax parse trees (in other words, generation of parse tree top-down or bottom-up for context-free grammars).
 - Additionally, semantic actions of the parser productions check the types (building symbol tables), and later semantic analysis and intermediate code generation, optimization and target code generation are performed.
 - The notion of abstract syntax is due to *John McCarthy*, 1963, who designed the abstract syntax for Lisp (no need to deal with all those parenthesis)
-

Abstract Syntax Trees: Overview

UC Santa Barbara

- The abstract syntax in the form of derivation (parse) trees separates syntax from semantics
 - It might be possible to write an entire compiler that fits within the semantic action phrases of a Yacc / Bison parser (i.e., syntax-driven translation, e.g., like in a simple calculator interpreter). However, such a compiler is difficult to read and maintain and forces the compiler to analyze the program in exactly the order it is parsed
-

Difference between Parse Trees and ASTs

UC Santa Barbara

- To improve modularity, it is better to separate issues of syntax (parsing) from issues of semantics (type-checking and translation to target code).
- A parse tree is the dynamic data structure that later phases of the compiler can traverse: where each leaf corresponds to each token of the input, and one internal node corresponds for each grammar rule reduced during the parse. We may have a concrete and an abstract parse tree:
 - A *concrete parse tree* represents the concrete syntax of the source languages where many punctuation tokens are redundant, has extra non-terminal symbols and extra productions for factoring, elimination of left recursion, elimination of ambiguity
 - An *abstract parse tree* eliminates a lot of this redundant (but needed by the parser) information from concrete parse tree. An *abstract syntax* makes a clean interface between the parser and later phases of a compiler, and takes the form of the abstract parse tree.

Example: Abstract versus Concrete Syntax

UC Santa Barbara

- *Example:* Abstract syntax tree - the parser uses the concrete syntax to generate a simplified abstract tree corresponding to the ambiguous context free grammar (not good for a parser), but good enough for further phases of compilation.

Example concrete syntax

```
S      → Expr
Expr   → Term Expr'
Expr'  → + Term Expr'
        | - Term Expr'
        | ε
Term   → Factor Term'
Term'  → * Factor Term'
        | / Factor Term'
        | ε
Factor → num
        | id
        | ( Expr )
```

Example abstract syntax

```
S      → Expr
Expr   → Expr + Expr
        | Expr - Expr
        | Expr * Expr
        | Expr / Expr
        | id
        | num
```

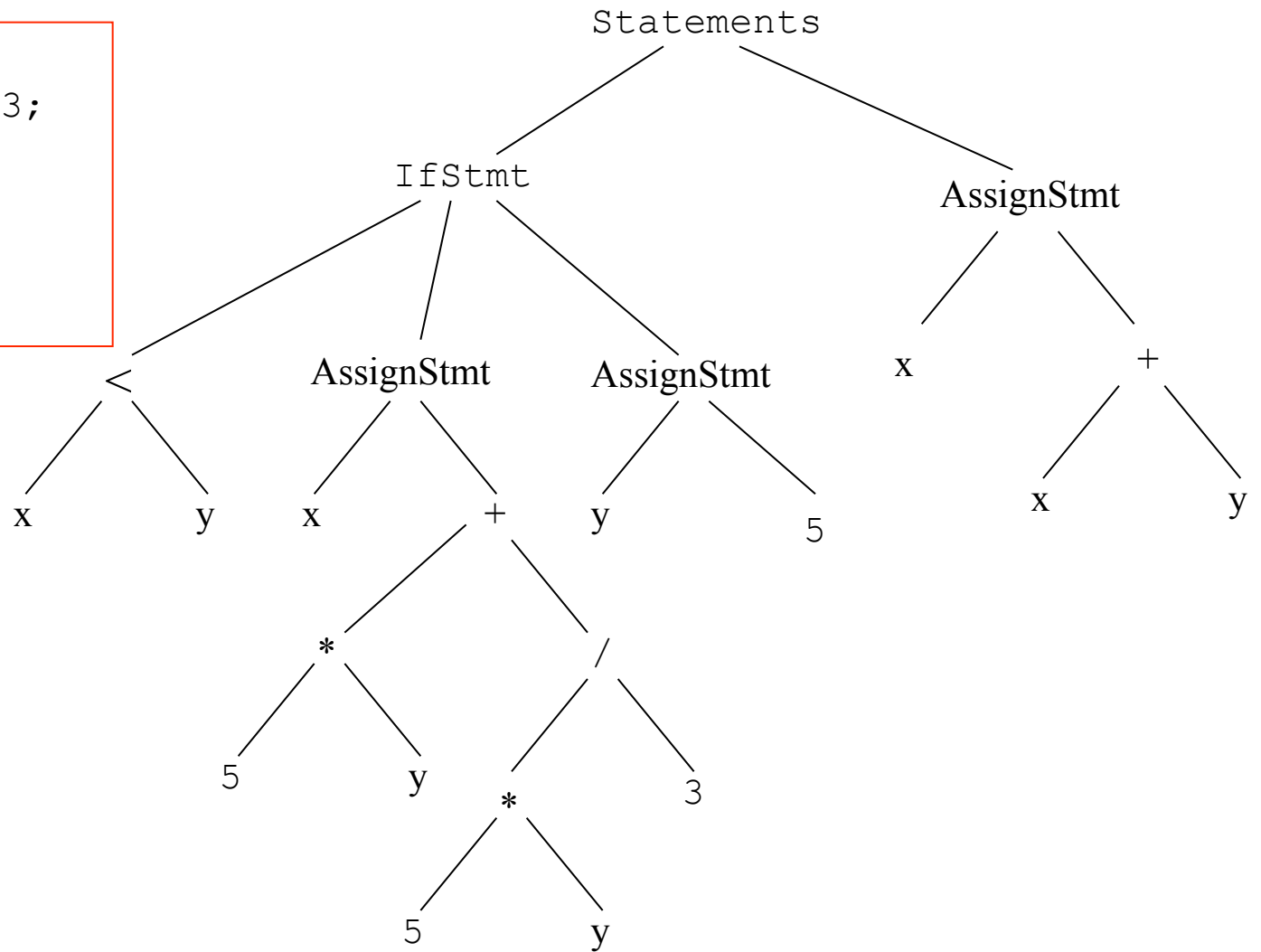


Much simpler, no punctuation is required, just enough to describe the structure of a program

Abstract Syntax Trees (ASTs)

UC Santa Barbara

```
if (x < y)
  x = 5*y + 5*y/3;
else
  y = 5;
x = x+y;
```



Object Oriented ASTs

UC Santa Barbara

- A tree is described by one or more abstract classes, corresponding to a symbol in the grammar
- Each abstract class is extended by one or more sub-classes, one for each grammar rule
- For each non-trivial symbol on the right hand side, there is a field in the corresponding class
- Each class has a constructor that initializes all fields and is thereafter immutable

```
# CDEF file for lang
```

```
Program ==> *Expr
```

```
Expr:Add ==> Expr Expr
```

```
Expr:Sub ==> Expr Expr
```

```
Expr:Div ==> Expr Expr
```

```
Expr:Mult ==> Expr Expr
```

```
Expr:Ident ==> SymName
```

```
Expr:Num ==> Primitive
```

```
# these classes should not be generated automagically
```

```
SymName external "symtab.hpp"
```

```
Primitive external "primitive.hpp"
```

Implementation (ast.hpp)

UC Santa Barbara

```
typedef Program* Program_ptr;
typedef Expr* Expr_ptr;

class Visitor
{
public:
    virtual ~Visitor() {}
    virtual void visitProgramImpl(ProgramImpl *p) = 0;
    virtual void visitAdd(Add *p) = 0;
    virtual void visitSub(Sub *p) = 0;
    virtual void visitDiv(Div *p) = 0;
    virtual void visitMult(Mult *p) = 0;
    virtual void visitIdent(Ident *p) = 0;
    virtual void visitNum(Num *p) = 0;
    virtual void visitSymName(SymName *p) = 0;
    virtual void visitPrimitive(Primitive *p) = 0;
};

class Visitable
{
public:
    virtual ~Visitable() {}
    virtual void visit_children(Visitor *v) = 0;
    virtual void accept(Visitor *v) = 0;
};
```

Implementation (ast.hpp)

UC Santa Barbara

```
class Program : public Visitable {
public:
    Attribute m_attribute;
    virtual Program *clone() const = 0;
};

// Program ==> *Expr
class ProgramImpl : public Program
{
public:
    list<Expr_ptr> *m_expr_list;

    ProgramImpl(const ProgramImpl &);
    ProgramImpl &operator=(const ProgramImpl&);
    ProgramImpl(list<Expr_ptr> *p1);
    ~ProgramImpl();
    virtual void visit_children( Visitor* v );
    virtual void accept(Visitor *v);
    virtual ProgramImpl *clone() const;
    void swap(ProgramImpl &);
};
```

```
class Expr : public Visitable {
public:
    Attribute m_attribute;
    virtual Expr *clone() const = 0;
};

// Expr:Add ==> Expr Expr
class Add : public Expr
{
public:
    Expr *m_expr_1;
    Expr *m_expr_2;

    Add(const Add &);
    Add &operator=(const Add &);
    Add(Expr *p1, Expr *p2);
    ~Add();
    virtual void visit_children( Visitor* v );
    virtual void accept(Visitor *v);
    virtual Add *clone() const;
    void swap(Add &);
};
```

Implementation (ast.cpp)

UC Santa Barbara

```
/****** Add *****/
Add::Add(Expr *p1, Expr *p2) {
    m_expr_1 = p1;
    m_expr_2 = p2;
    m_attribute.lineno = yylineno;
}
Add::Add(const Add & other) {
    m_expr_1 = other.m_expr_1->clone();
    m_expr_2 = other.m_expr_2->clone();
}
Add &Add::operator=(const Add & other) {
    Add tmp(other); swap(tmp); return *this; }
void Add::swap(Add & other) {
    std::swap(m_expr_1, other.m_expr_1);
    std::swap(m_expr_2, other.m_expr_2);
}
Add::~Add() {
    delete(m_expr_1);
    delete(m_expr_2);
}
void Add::visit_children( Visitor* v ) {
    m_expr_1->accept( v );
    m_expr_2->accept( v );
}
void Add::accept(Visitor *v) { v->visitAdd(this); }
Add *Add::clone() const { return new Add(*this); }
```

Implementation (ast.cpp)

UC Santa Barbara

```
/****** ProgramImpl *****/
ProgramImpl::ProgramImpl(list<Expr_ptr> *p1) {
    m_expr_list = p1;
    m_attribute.lineno = yylineno;
}
ProgramImpl::ProgramImpl(const ProgramImpl & other) {
    m_expr_list = new list<Expr_ptr>;
    list<Expr_ptr>::iterator m_expr_list_iter;
    for(m_expr_list_iter = other.m_expr_list->begin();
        m_expr_list_iter != other.m_expr_list->end();
        ++m_expr_list_iter){
        m_expr_list->push_back( (*m_expr_list_iter)->clone() );
    }
}

[... ]

void ProgramImpl::visit_children( Visitor* v ) {
    list<Expr_ptr>::iterator m_expr_list_iter;
    for(m_expr_list_iter = m_expr_list->begin();
        m_expr_list_iter != m_expr_list->end();
        ++m_expr_list_iter){
        (*m_expr_list_iter)->accept( v );
    }
}

void ProgramImpl::accept(Visitor *v) { v->visitProgramImpl(this); }
```

Example: AST for 2+5-x

UC Santa Barbara

