

# Syntax-Directed Definitions

Compiler Design

CSE 504

Last modified: Wed Apr 08 2015 at 16:10:34 EDT  
Version: 1.6 15:28:43 2015/01/25  
Compiled at 16:42 on 2015/04/08

Compiler Design

Attribute Grammars

CSE 504 1 / 13

## Semantic Analysis Phases of Compilation

- Build an Abstract Syntax Tree (AST) while parsing
- Decorate the AST with type information (type checking/inference)
- Generate intermediate code from AST
- Optimize intermediate code
- Generate final code

# Abstract Syntax Tree (AST)

Represents the syntactic structure of the input program, independent of peculiarities in the grammar.

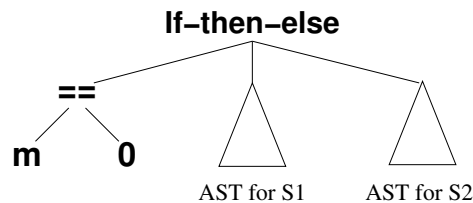
## An Example:

*Consider a statement of the form:*

*“if (m == 0) S1 else S2”*

*where S1 and S2 stand for some block of statements.*

*A possible AST for this statement is:*

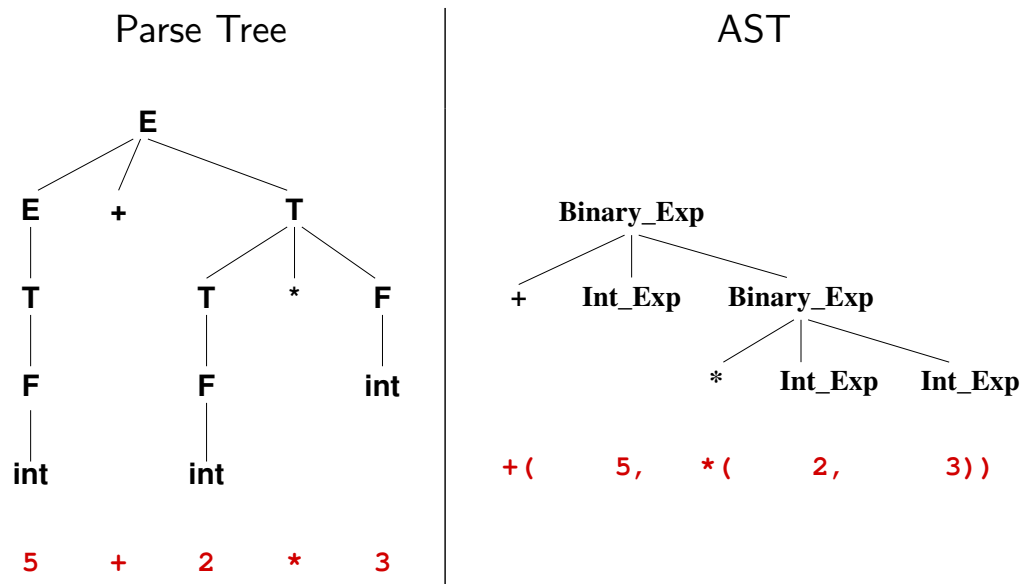


## Construction of Abstract Syntax Trees

Typically done simultaneously with parsing

- ... as another instance of syntax-directed translation
- ... for translating *concrete* syntax (the parse tree) to *abstract* syntax (AST).
- ... with AST as a *synthesized attribute* of each grammar symbol.

# Abstract Syntax Trees



## Trees & Tree Grammars

Tree grammars can be used to specify a set of **trees**.

**Example 1:** The set of trees  $\{a, f(a), f(f(a)), f(f(f(a))), \dots\}$  can be represented by the grammar:

$$\begin{aligned} T &\longrightarrow a \\ T &\longrightarrow f(T) \end{aligned}$$

**Example 2:** The set of trees  $\{b, g(b, b), g(g(b, b), b), g(b, g(b, b)), \dots\}$  can be represented by:

$$\begin{aligned} S &\longrightarrow b \\ S &\longrightarrow g(S, S) \end{aligned}$$

# AST & Tree Grammars

The set of AST's that represent the set of (syntactically correct) programs can be described by tree grammars.

```
Expression  ::  INT_EXPR(Int_val)
              |  FLOAT_EXPR(float_val)
              |  :
              |  METHOD_EXPR(Expression, Expression*)
              |  ARRAY_EXPR(Expression, Expression)
              |  FIELD_EXPR(Expression, Name)
              |  BINARY_EXPR(BinaryOp, Expression, Expression)
              |  UNARY_EXPR(UnaryOp, Expression)
              |  :
```

## Actions and AST

```
E  →  E1 + T    { E.ast = BINARY_EXPR( PLUS,  E1.ast, T.ast) }
E  →  T           { E.ast = T.ast; }
:
:
F  →  ( E )       { F.ast = E.ast; }
F  →  int          { F.ast = INT_EXPR(int.val; }
```

## Actions and AST: Another Example

```
S  →  if E S1 else S2
      { S.ast = IF_STMT( E.ast, S1.ast, S2.ast) }
S  →  return E    { S.ast = RETURN_STMT( E.ast) }
```

## Physical Representation of Trees

In C, trees can be built using *variant records* (e.g., structs and unions).

**Example:** The set of trees described by

$S \rightarrow g(S, S)$

$S \rightarrow h(T)$

can be implemented as:

```
typedef enum { G, H} S_node_kind;
struct S_node {
    S_node_kind kind;
    union {
        struct S_g {
            struct S_node *g_child1;
            struct S_node *g_child2;
        } g;
        struct T_node *h_child;
    } children;
};
```

## C Implementation of AST Construction

```
S  →  if E S1 else S2
      { S.ast = IF_STMT( E.ast, S1.ast, S2.ast) }
S  →  return E   { S.ast = RETURN_STMT( E.ast) }
```

---

```
S  →  if E S1 else S2
      { S.ast = malloc(sizeof(struct Statement));
        S.ast.stmt.kind = IF;
        S.ast.stmt.ifstmt.expr = E.ast;
        S.ast.stmt.ifstmt.then = S1.ast;
        S.ast.stmt.ifstmt.else = S2.ast; }
S  →  return E   { S.ast = new_ast_node(Statement);
                  S.ast.stmt.kind = RETURN;
                  S.ast.stmt.return.expr = E.ast }
```

## Tree Data Structures in OOP languages

The tree will be a base class, and each tree constructor will be a derived class.

|                                             |                                                                                                                                                                                                                                             |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>S  →  b S  →  g(S, S) S  →  h(T)</pre> | <pre>class S {...}; // abstract base class class S_b: public S {...}; //derived classes class S_g: public S {     public:         S *g_1;         S *g_2;         ... } class S_h: public S {     public:         T *h;         ... }</pre> |
|---------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## C++ Implementation of AST Construction

```
S  →  if E S1 else S2
      { S.ast = IF_STMT( E.ast, S1.ast, S2.ast) }
S  →  return E
      { S.ast = RETURN_STMT( E.ast) }
```

---

```
S  →  if E S1 else S2
      { S.ast = new IF_Statement(E.ast, S1.ast, S2.ast); }
      // In constructor for IF_Statement(e, s1, s2):
      // where e, s1, s2 are all of type AST*
      // this.expr = e1;
      // this.then_part = s1; this.else_part = s2;
S  →  return E
      { S.ast = new RETURN_Statement(E.ast); }
```