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.

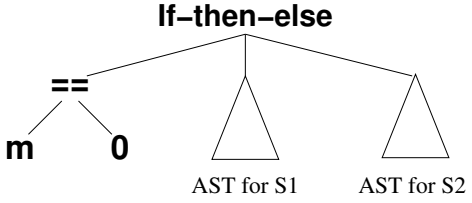
An Example:

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:

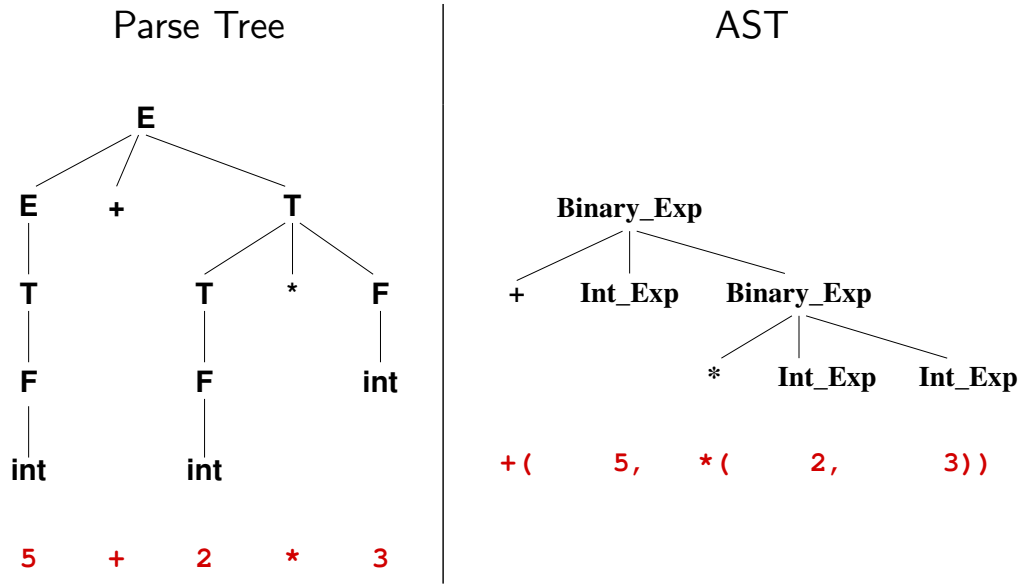


Construction of Abstract Syntax Trees

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 $\{a, f(a), f(f(a)), f(f(f(a))), \ldots\}$ can be represented by the grammar:

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

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

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

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

$$
\begin{array}{lcll}
E & \longrightarrow & E_1 \texttt{ + } T & \{\ E.\texttt{ast} = \texttt{BINARY\_EXPR(}\ \texttt{PLUS},\ E_1.\texttt{ast}, T.\texttt{ast})\ \} \\
E & \longrightarrow & T & \{\ E.\texttt{ast} = T.\texttt{ast};\ \} \\
\vdots & & & \\
F & \longrightarrow & \texttt{(} E \texttt{)} & \{\ F.\texttt{ast} = E.\texttt{ast};\ \} \\
F & \longrightarrow & \texttt{int} & \{\ F.\texttt{ast} = \texttt{INT\_EXPR(int.val};\ \}
\end{array}
$$

Actions and AST: Another Example

$$
\begin{array}{lcll}
S & \longrightarrow & \texttt{if } E\ S_1 \texttt{ else } S_2 & \\
 & & & \{\ S.\texttt{ast} = \texttt{IF\_STMT(}\ E.\texttt{ast}, S_1.\texttt{ast}, S_2.\texttt{ast})\ \} \\
S & \longrightarrow & \texttt{return } E & \{\ S.\texttt{ast} = \texttt{RETURN\_STMT(}\ E.\texttt{ast})\}
\end{array}
$$

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

$S \longrightarrow \texttt{g}(S, S)$
$S \longrightarrow \texttt{h}(T)$

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

$S \longrightarrow \texttt{if}\ E\ S_1\ \texttt{else}\ S_2$
$\quad \{\ S.\texttt{ast} = \texttt{IF\_STMT(}\ E.\texttt{ast}, S_1.\texttt{ast}, S_2.\texttt{ast}\texttt{)}\ \}$

$S \longrightarrow \texttt{return}\ E \quad \{\ S.\texttt{ast} = \texttt{RETURN\_STMT(}\ E.\texttt{ast}\texttt{)}\}$

$S \longrightarrow \texttt{if}\ E\ S_1\ \texttt{else}\ S_2$
$\quad \{\ S.\texttt{ast} = \texttt{malloc(sizeof(struct Statement))};$
$\quad\quad S.\texttt{ast.stmt.kind} = \texttt{IF};$
$\quad\quad S.\texttt{ast.stmt.ifstmt.expr} = E.\texttt{ast};$
$\quad\quad S.\texttt{ast.stmt.ifstmt.then} = S_1.\texttt{ast};$
$\quad\quad S.\texttt{ast.stmt.ifstmt.else} = S_2.\texttt{ast};\ \}$

$S \longrightarrow \texttt{return}\ E \quad \{\ S.\texttt{ast} = \texttt{new\_ast\_node(Statement)};$
$\quad\quad S.\texttt{ast.stmt.kind} = \texttt{RETURN};$
$\quad\quad S.\texttt{ast.stmt.return.expr} = E.\texttt{ast}\}$

Tree Data Structures in OOP languages

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

$S \longrightarrow$ b	`class S {...};  // abstract base class`
$S \longrightarrow g(S, S)$	`class S_b: public S {...};  //derived classes`
$S \longrightarrow h(T)$	`class S_g: public S {`
	`  public:`
	`    S *g_1;`
	`    S *g_2;`
	`    ...`
	`}`
	`class S_h: public S {`
	`  public:`
	`    T *h;`
	`    ...`
	`}`

C++ Implementation of AST Construction

$S \longrightarrow$ `if` $E\ S_1$ `else` S_2
{ S.ast = IF_STMT(E.ast, S_1.ast, S_2.ast) }

$S \longrightarrow$ `return` E
{ S.ast = RETURN_STMT(E.ast)}

$S \longrightarrow$ `if` $E\ S_1$ `else` S_2
{ S.ast = new IF_Statement(E.ast, S_1.ast, S_2.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;

$S \longrightarrow$ `return` E
{S.ast = new RETURN_Statement(E.ast); }