Compiler Technology of Programming Languages

Chapter 7
Syntax-Directed Compilation

Prof. Farn Wang

Outlines of Chapter 7

- Overview
  - Syntax-directed translation
    - single pass compilation
    - multi-pass: syntax-directed AST generation
  - Semantic Actions and Values
  - Synthesized and Inherited Attributes
- AST Data Structures
  - A central data structure for all post-parsing activities
- AST Design and Construction

Syntax-Directed Translation

What is that ?

CFG **+** **Semantic Actions**

- In a single pass compiler, all actions, including semantic checks and code generation are completely coded in the semantic actions.
- In a multi-pass compiler, AST could be generated as an intermediate step.

Semantic Actions and Values

- Semantic Actions
  - Each production can have an associated code sequence that will execute when the production is applied.

  e.g. var_ref : id {*check if the id is declared? check if id is an array name? … etc*}

- Semantic Values

  e.g. expr : term + expr; {1) *check if the term's type is compatible with expr's type,*

  2) what is the type of expr }

Semantic Action Execution

In automatically generated parsers, the parser driver is responsible for executing the semantic actions.

Synthesized Attributes

Attributes flow from the leaves of a derivation tree toward its root. A nature way of bottom-up parsing.

Inherited Attributes

Attributes flow from the parents or the siblings. A nature way of top-down parsing

Syntax-directed translation of most programming languages requires both synthesized and inherited attributes

Attributes

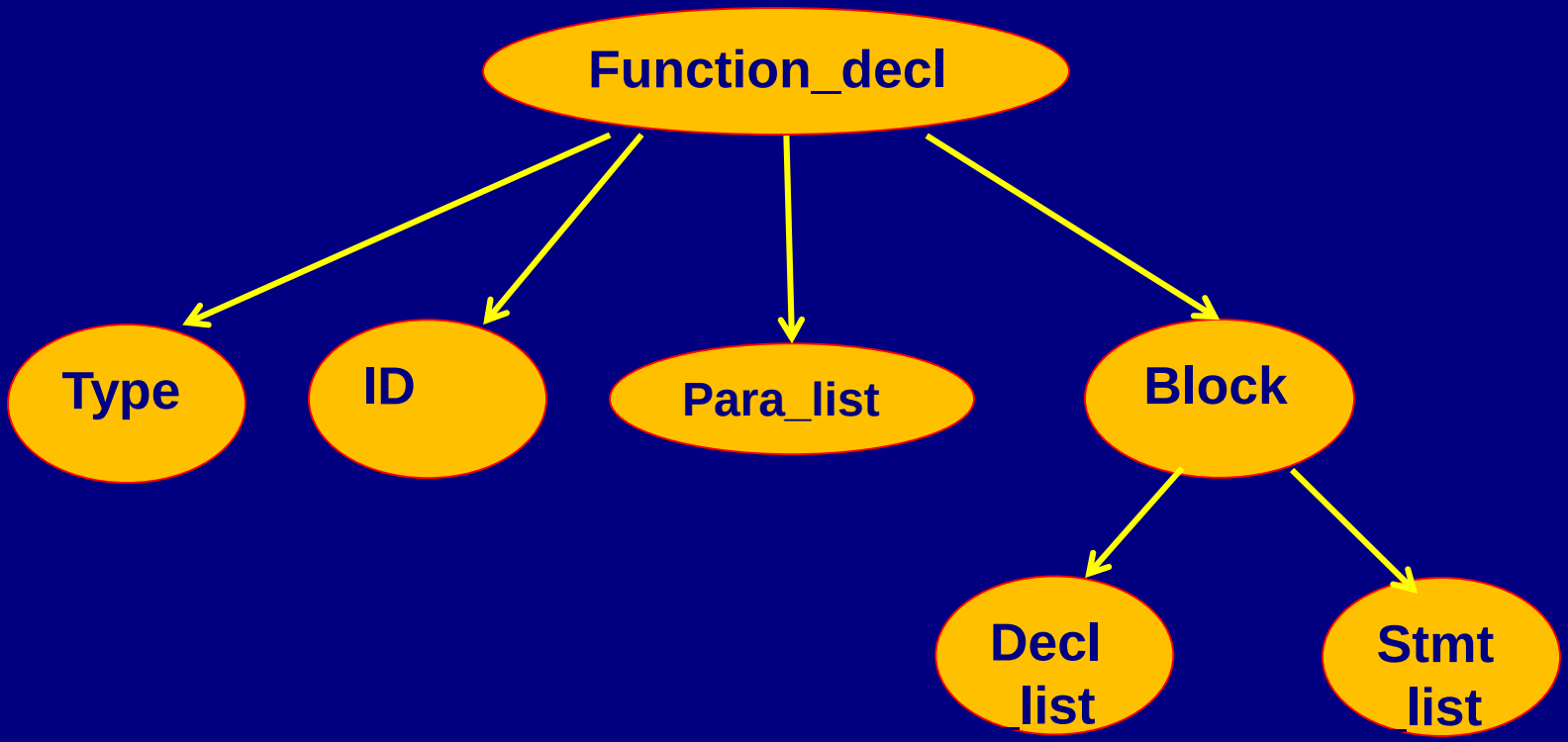
- What could be the attributes of a declared variable?

Example: int A[100][100], B;

- Type:
  - int, float, double, boolean, char, struct, union
- Array or scalar?
- Scope?

Synthesized Attributes

- What could be the attributes of a function?
  - Type, ID, Parameter list, Exception list, Register used, ….

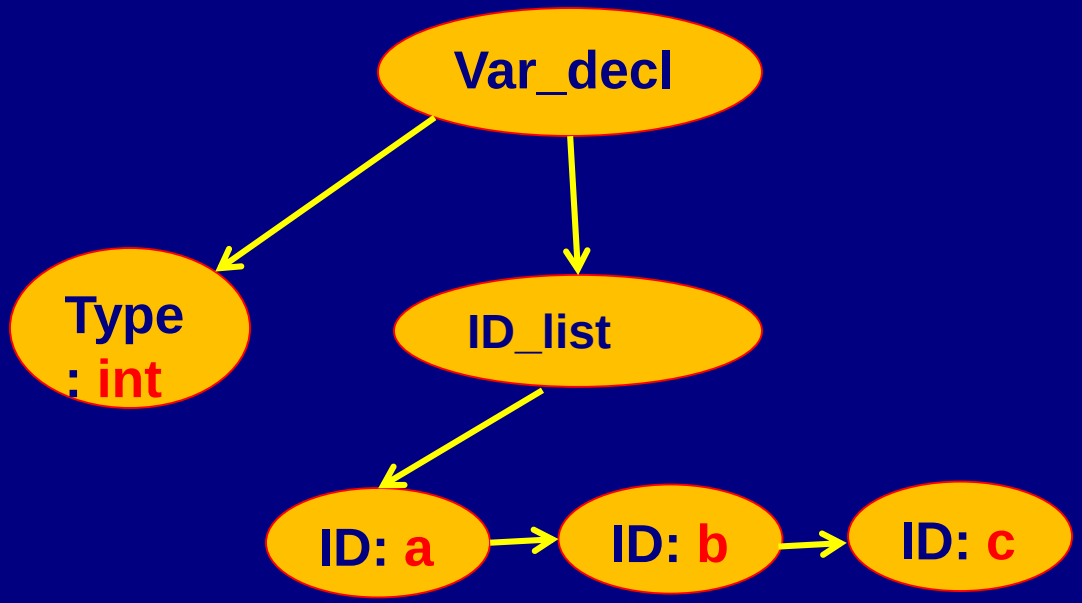


Inherited Attributes

- Attributes inherited from parents or siblings

  Example: CFG var_decl : type id_list ;

  C int a,b,c;



Value Stack in YACC/Bison

Attributes cannot be just collected at token level
e.g. array info, struct info, function, loop info, …
Value stack is to assist on attributes collection and propagation

Parser stack is respon-sible for ***syntax*** analysis

parser stack (states)

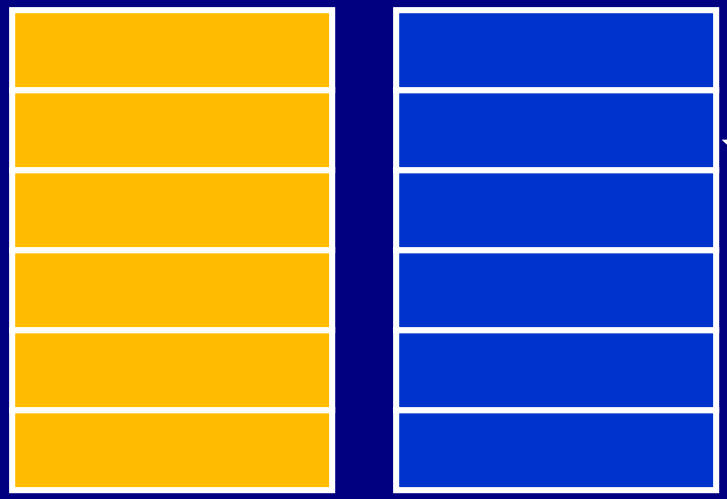
value stack (semantic record)

Value stack is respon-sible for ***semantic*** analysis

Value Stack in YACC

% union defined in the 1st section in parser.y

Parser stack is respon-sible for ***syntax*** analysis



parser stack (states)

value stack (semantic record)

Value stack is respon-sible for ***semantic*** analysis

Semantic Record

- Why variant record?
  - each non-terminal may have different attributes, e.g. function, block, statement, expression,…
  - variant record saves space
  - value stack is just an array of pointers – pointer to the variant record (i.e. the union)
  - parse stack is just an array of states, each state implicitly refers to a grammar symbol.

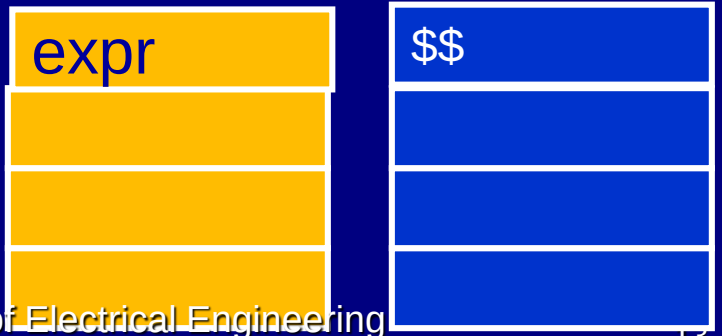
Parser.y layout

```
%union {
    int num;
    char* lexeme; …}
%token  <lexeme> ID
%token  <num> const
%type  <num> expr term
%%
expr : expr ‘+’ term    $$ = eval($1,$3);
term : ID | const
%%
int eval (char c, int a,b) {}
```

Parser.y layout

```
%union {
    int num;
    char* lexeme; …}
%token  <lexeme> ID
%token  <num> const
%type  <num> expr term
%%
expr : expr ‘+’ term
term : ID | const
%%
int eval (char c, int a,b) {}
```

$$ = eval($1,$3);



Synthesized Attributes

e.g.

```
expr      :      expr  add_op   term
{ $$-> type = (($1->type==FLOAT)||($3-
>type==FLOAT))?FLOAT : INT;}
```

e.g.

```
dim_decl : [ cexpr ]
            { initialize attribute values for $$}
            | dim_decl [ cexpr ]
            { accumulate attribute values for $$}
```

- Inherited Attributes

e.g.

var_decl : type init_id_list ;

int a=0,b,c=1;

The type of the id_list is **int**

int is the attribute of type

This (inherited) attribute flows from the left siblings to the right siblings.

Rule Cloning (section 7.2.2)

- A similar sequence of strings may need different treatment, for example:

  3+4*5 and a+b*c are both expressions

  and could be handled by nonterminal ***expr***
- In our C– syntax, we clone the expr non-terminal as
  - Regular expr and Constant expr
- Non-terminal ***expr*** and ***cexpr***
- ***expr*** parses an expression, ***cexpr*** translates a constant expression.
- When you declare an array A[3+4*5], we would like to handle 3+4*5 as a constant expression.

Rule Cloning (cont.)

- dim_decl : [***cexpr***]
  | dim_decl [***cexpr***]
- relop_factor : ***expr***
  | ***expr*** rel_op ***expr***
- ***cexpr : cexpr + mexpr***
  | ***cexpr – mexpr***
  | ***mexpr***
- ***expr : expr + term***
  | ***expr – term***
  | ***term***

Rule Cloning (cont.)

- dim_decl : [***cexpr***]
  | dim_decl [***cexpr***]
- relop_factor : ***expr***
  | ***expr*** rel_op ***expr***
- ***cexpr : cexpr + mexpr*** ***{$$=$1 + $3}***
  | ***cexpr – mexpr*** ***{$$=$1-$3}***
  | ***mexpr*** ***{$$ = $1}***
- ***expr : expr + term*** ***{makenode…}***
  | ***expr – term***
  | ***term***

Forcing Semantic Actions (section 7.2.3)

- Bottom-up parsers normally are prepared to execute semantic actions on reductions.
- If a semantic action is desired on the shift of some symbol *X* , a unit production can be inserted

Example:

B → X1 X2 … Xn

If you want an action to take when X2 is shifted,

you may modify the grammar as follows:

B → X1 A … Xn

A → X2 {*action*}

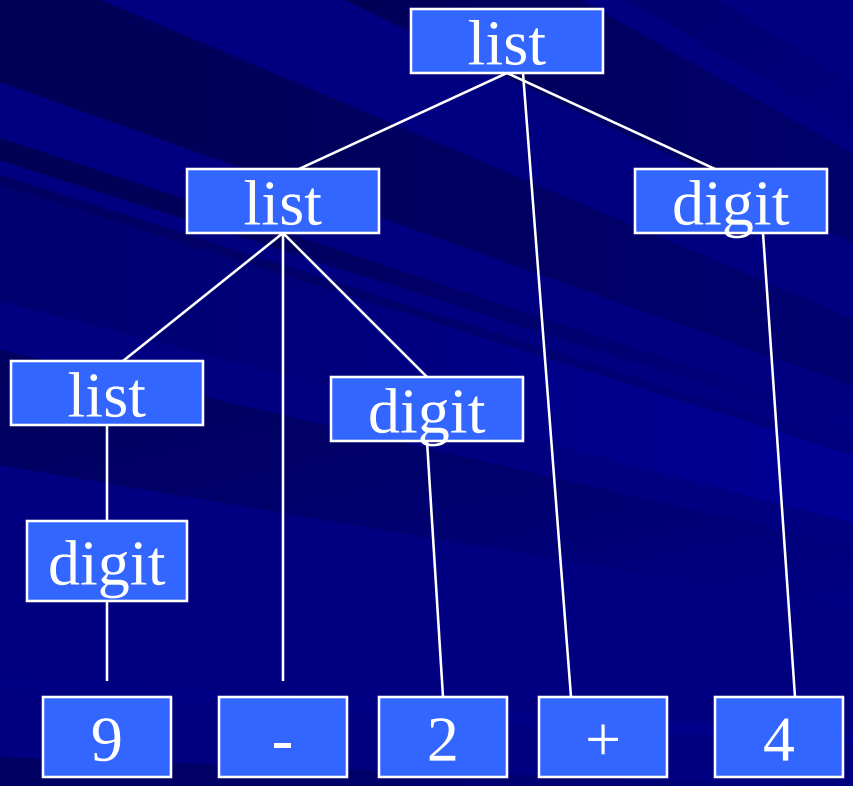
What if you just want to have an action to take between X1 and X2?

You may insert a Lamda production A → λ

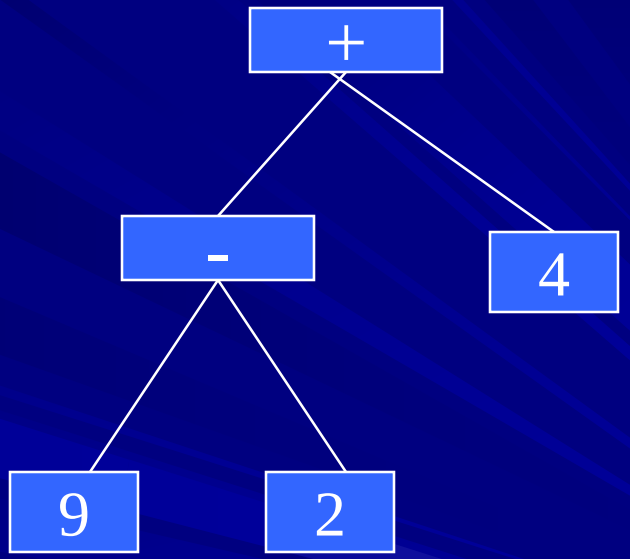
Abstract Syntax Trees

- Many compiler's tasks can be performed in a single pass via *syntax-directed translation*.
- Modern software practices discourage implementing so much functionality in a single component.
  - difficult to understand, extend, and maintain.
- A multi-pass compilation approach is to define an IR. Each of the semantic analysis, program optimization, and code generation work will be treated in separate passes. AST is such an IR.
- Our project takes multi-pass compilation.

Parse Tree and Syntax Tree

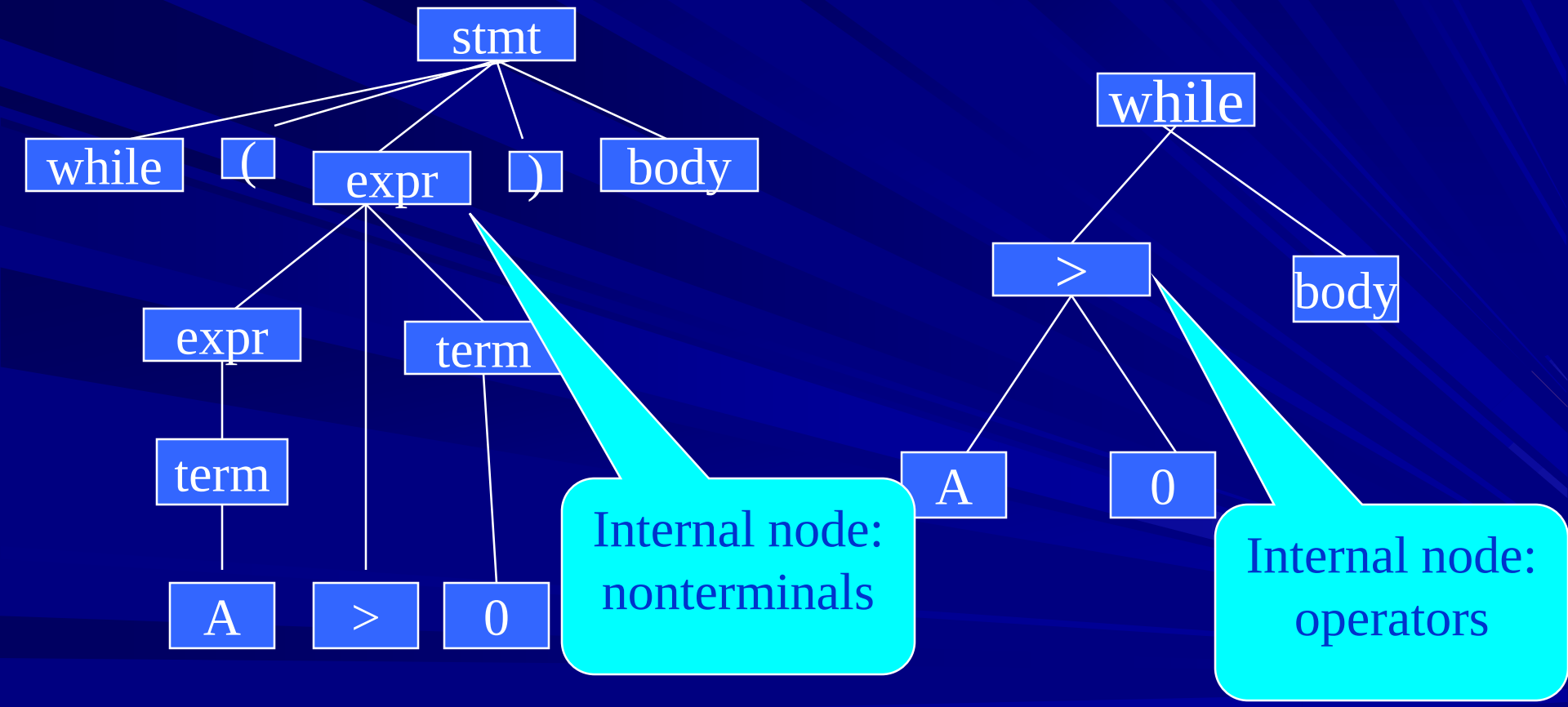


Parse Tree



Syntax Tree

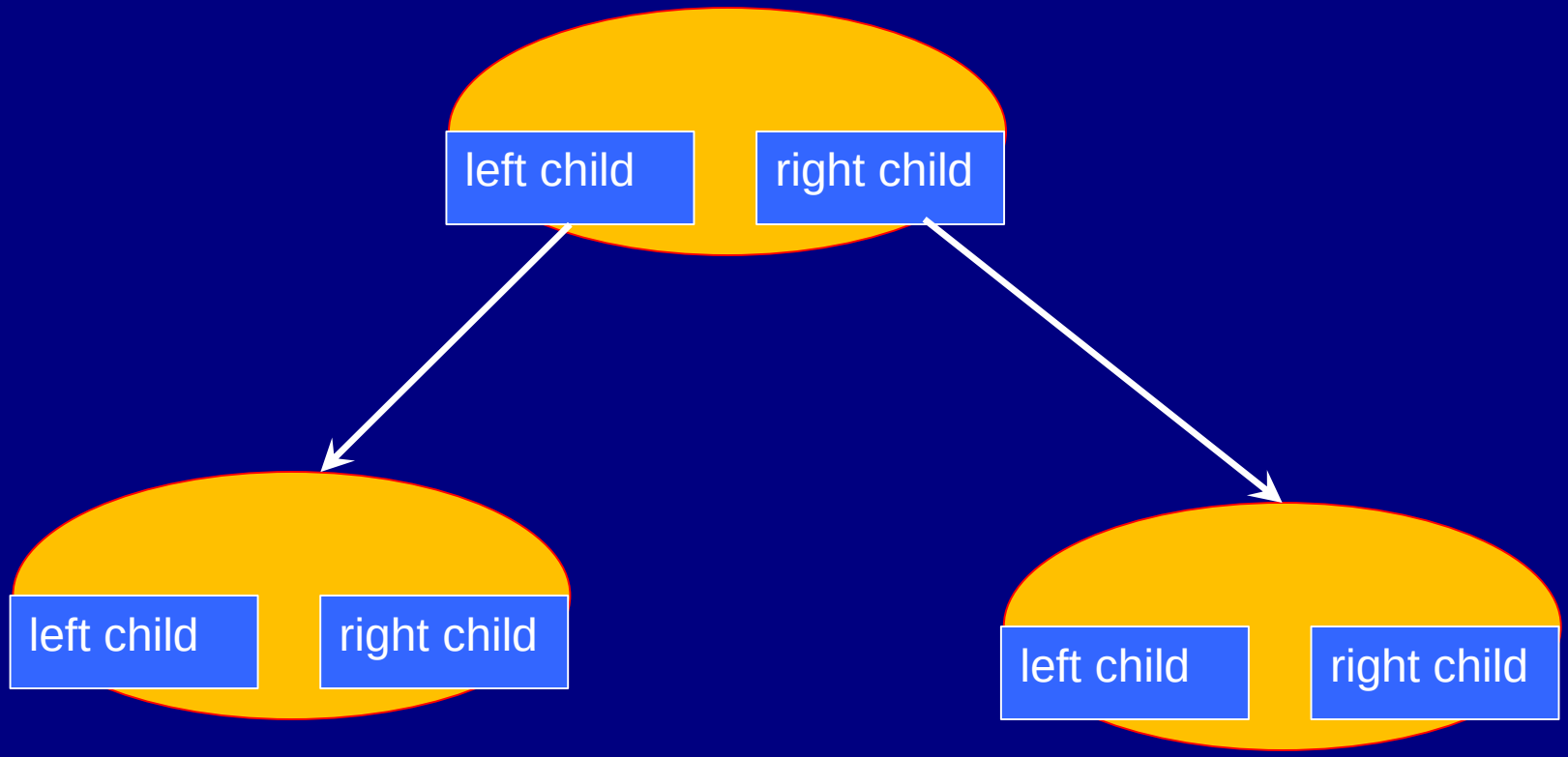
Concrete Tree and Abstract Tree



Parse Tree

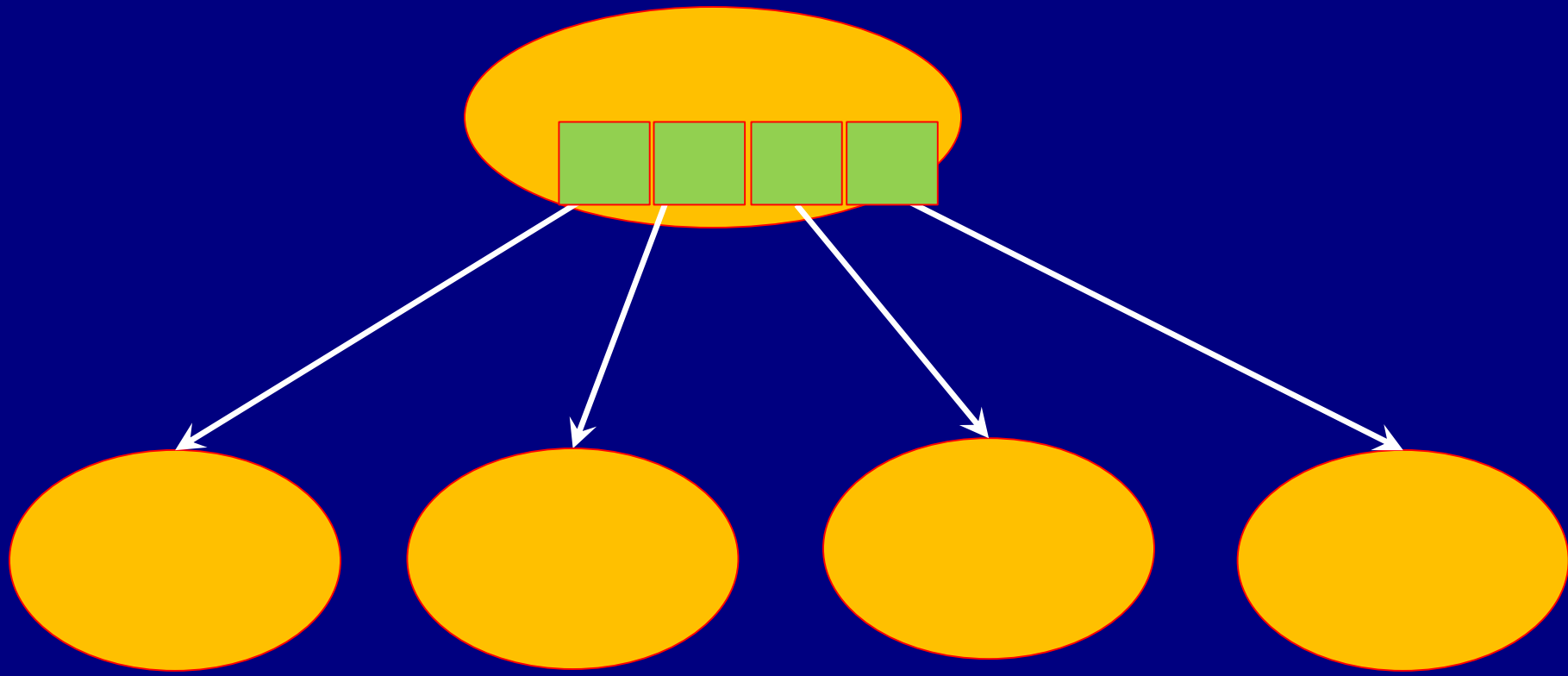
Syntax Tree

Tree Data Structure

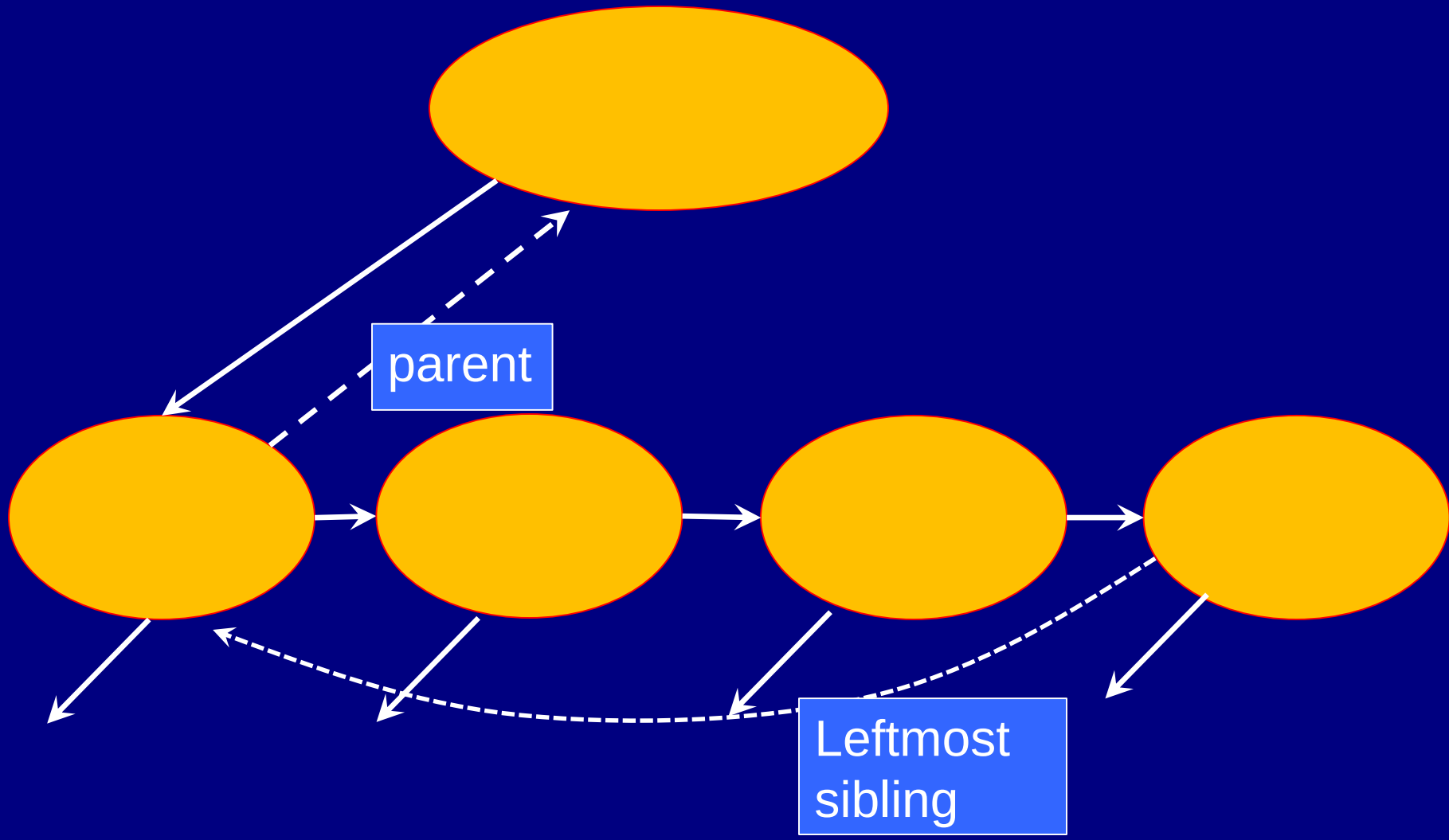


A Typical binary Tree

AST Data Structure



AST Data Structure



Parser.y layout

```
%union {
    int num;
    char* lexeme; …}
%token  <lexeme> ID
%token  <num> const
%type  <num> expr
%%
expr : expr ‘+’ term
term : ID | const
%%
int eval (char c, int a,b) {}
```

```
{$$=Allocate(expr);
Makechild($$,$1);
Makesibling($1,$3);
$$->sem_val.op_type =
PLUS };
```

Infrastructure for creating ASTs

- makeNode(t)

  makeNode(int n) instantiates a node that represents an integer n.

  makeNode(Symbol s) instantiates a node for a symbol s. Methods must be included to set and get the symbol table entry for s.

  makeNode(Operator o) instantiates a node for an operation o.

- adoptChildren(x,y)
- makeSiblings(x,y)
- makeFamily(op, kid1,kid2,kid3…)

Construction Example

- See page 291 for a complete example

Start → Stmt $ { return(ast) }

Stmt → id = E { result ← makeFamily (=,id.var, E.expr);}

Stmt → if (E) Stmt fi {result ← makeFamily (if,E.expr, Stmt.s, makeNode());}

AST Structures