4 Abstract Syntax

(4. **Abstract Syntax, p. 88**)

- Whenever an LR parser chooses to *reduce* a CFG rule, a piece of valid input syntax has just been recognized.

- In a compiler, recognition is *not* enough. The recognized piece either needs to be

  ① **interpreted**, i.e., executed in some (abstract) sense, or

  ② **remembered**, i.e., transformed into a data structure suitable to further process the recognized input.

- The compiler uses **semantic actions** attached to the CFG rules. A semantic action assigns a **semantic value** to a grammar symbol.

  – **N.B.**: the semantic value of terminal symbols has already been assigned by the lexer (e.g., `flex`: `... yylval.num = atoi (yytext); ...`).

- The semantic action associated with the CFG rule

$$S \quad \rightarrow \quad S_1 \; S_2 \; \cdots \; S_k$$

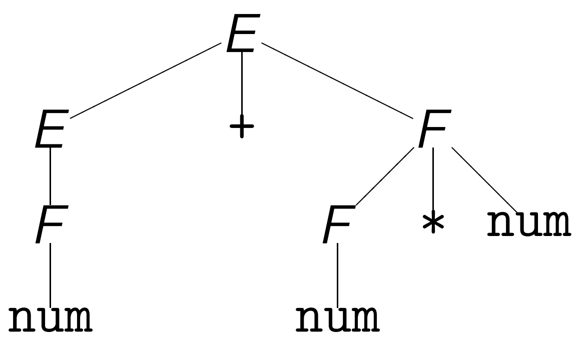
computes the **semantic value** $\$_S$ of non-terminal S which—in general—will depend on the semantic values of $\$_{S_1}, \ldots, \$_{S_k}$:

$$\$_S \quad = \quad f(\$_{S_1}, \ldots, \$_{S_k}) \ .$$

  - Parser generator `bison` enables us to code the semantic action function f in C.
  - In a compiler, f will typically construct an **abstract parse tree** out of the smaller parse tree fragments $\$_{S_1}, \ldots, \$_{S_k}$.

- Why does this work? When we apply f to compute $\$_S$, are we guaranteed that the $\$_{S_1}, \ldots, \$_{S_k}$ are already availble?
  - Yes, because the reductions during an LR parse occur in the (virtual) parse tree in *postorder* (i.e., *bottom-up, left-to-right*).
  - **Example:** consider the CFG below and the parse tree for input `num + num * num`:

$$
\begin{aligned}
E &\rightarrow E + F \\
E &\rightarrow F \\
F &\rightarrow F * \texttt{num} \\
F &\rightarrow \texttt{num}
\end{aligned}
$$



  - Number the non-terminals in this parse tree in the order in which an LR parser would reduce the non-terminals.

- In a `bison` generated parser, attach the semantic actions as a C code fragment enclosed in `{ ··· }` just next to the CFG rule.
  - `bison` syntax: $\$_S \equiv \$\$, \quad \$_{S_i} \equiv \$i.$

- **Example:** `bison` expression grammar. A "desktop calculator" *interpreting* the expressions: the semantic actions actually carry out the arithmetics specified by the input string:

`bison` **input file (excerpt)**

```
%union { int num; string bin; }
%token <num> NUM
%token <bin> BINARY
%token PLUS TIMES
%left  PLUS
%left  TIMES
%type  <num> Exp

%%

Exp : NUM                { $$ = $1; }
    | BINARY             { $$ = (int) strtol ($1, 0, 2); /* convert binary digits */ }
    | Exp PLUS Exp       { $$ = $1 + $3; }
    | Exp TIMES Exp      { $$ = $1 * $3; }
    ;
```

- **Remarks:**
  - The %union declaration enumerates all possible types of semantic values which occur in the grammar.
  - The %token <*t*> and %type <*t*> declarations assign the type of the %union variant *t* to terminals and non-terminals, respectively.

- An LR parser implements references to semantic values ($\$i$) with the help of a **semantic value stack**.
  - The semantic value stack is operated *in parallel to* the LR state stack.
  - In the semantic action of a CFG rule of the form

$$S \quad \to \quad S_1\ S_2\ \cdots\ S_k$$

  (k symbols on the right-hand side), $\$i$ refers to the semantic value of the stack element located $k - i$ positions below the stack top.

- **Example:** Using the `bison` desktop calculator grammar shown before and input `1 + %010 * 3`, trace `bison`'s semantic value stack:[16]

Stack	*Input*	*Action*
	`1 + %010 * 3 $`	*shift*
[`NUM` 1]	`+ %010 * 3 $`	*reduce* `Exp` → `NUM`
[`Exp` 1]	`+ %010 * 3 $`	*shift*
[`Exp` 1] [`PLUS`]	`%010 * 3 $`	*shift*
[`Exp` 1] [`PLUS`] [`BINARY` `"010"`]	`* 3 $`	*reduce* `Exp` → `BINARY`
[`Exp` 1] [`PLUS`] [`Exp` 2]	`* 3 $`	*shift*
[`Exp` 1] [`PLUS`] [`Exp` 2] [`TIMES`]	`3 $`	*shift*
[`Exp` 1] [`PLUS`] [`Exp` 2] [`TIMES`] [`NUM` 3]	`$`	*reduce* `Exp` → `NUM`
[`Exp` 1] [`PLUS`] [`Exp` 2] [`TIMES`] [`Exp` 3]	`$`	*reduce* `Exp` → `Exp TIMES Exp`
[`Exp` 1] [`PLUS`] [`Exp` 6]	`$`	*reduce* `Exp` → `Exp PLUS Exp`
[`Exp` 7]	`$`	*accept*

[16] In this example, `%010` indicates a binary number represented by token type `BINARY`.

bison based interpreter for SLP

```
%{

  #include <string.h>
  #include <assert.h>

  typedef char *string;

  typedef struct table *Table_;
  struct table { string id; int value; Table_ tail; };

  /* construct a new variable table entry */
  Table_ Table (string id, int value, Table_ tail)
  {
    Table_ t = (Table_) malloc (sizeof (*t));

    t->id    = id;
    t->value = value;
    t->tail  = tail;

    return t;
  }

  /* table of all variables used in SLP program */
  Table_ vars = NULL;

  /* lookup variable with name id (stop if not found) */
  int lookup (Table_ table, string id)
  {
    assert (table);

    if (! strcmp (id, table->id))
      return table->value;
    else return lookup (table->tail, id);
  }

  /* insert entry (id, value) into variable table */
  Table_ update (Table_ table, string id, int value)
  {
    return Table (id, value, table);
  }
%}
```

```
%union { int num; string id; }

%token <num> INT
%token <id>  ID
%token ASSIGN PRINT LPAREN RPAREN SEMICOLON COMMA
       PLUS MINUS TIMES DIV
%left SEMICOLON
%left PLUS MINUS
%left TIMES DIV

%type <num> exp

%%

prog  : stm
      ;

stm   : stm SEMICOLON stm
      | ID ASSIGN exp      { vars = update (vars, $1, $3); }
      | PRINT LPAREN exps RPAREN { printf ("\n"); }
      ;

exps  : exp                      { printf ("%d ", $1); }
      | exps COMMA exp           { printf ("%d ", $3); }
      ;

exp   : INT                      { $$ = $1; }
      | ID                       { $$ = lookup (vars, $1); }
      | exp PLUS exp             { $$ = $1 + $3; }
      | exp MINUS exp            { $$ = $1 - $3;}
      | exp TIMES exp            { $$ = $1 * $3; }
      | exp DIV exp              { $$ = $1 / $3; }
      | stm COMMA exp            { $$ = $3; }
      | LPAREN exp RPAREN        { $$ = $2; }
      ;

%%
```

4.1 Abstract Parse Trees

- To compile most of the features of non-toy programming languages, the strict postorder parse tree traversal of an LR parser is way too restrictive.

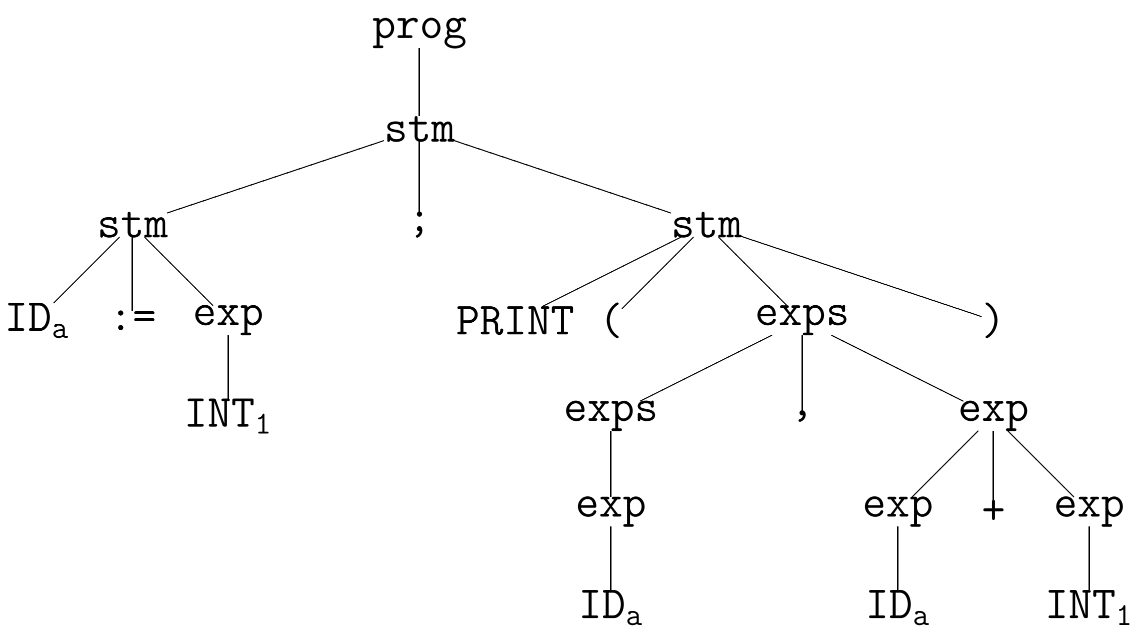
- Additionally, if we try to squeeze all compiler functionality (semantic analysis, code generation, optimization, ...) into the semantic actions, we probably end up with a maintenance nightmare.

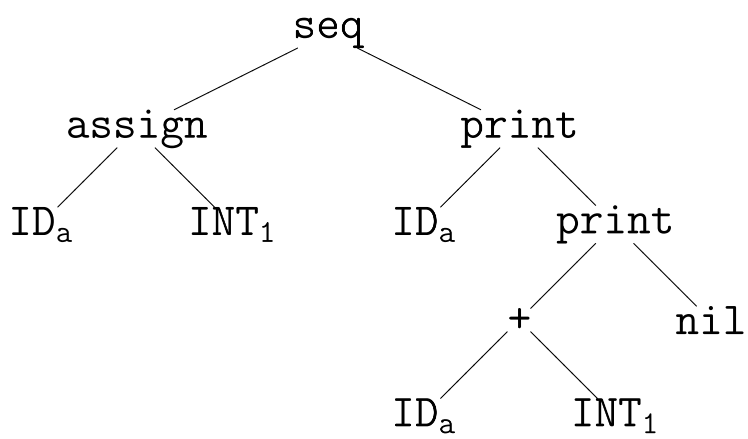
- Issues of syntax (parsing) and semantics need to be **separated**.

- The parsing phase thus uses the semantic actions to construct an **abstract parse tree** which is then communicated to the subsequent compiler phases.

- **Concrete vs. abstract parse trees:** *the* concrete and *an* abstract parse tree for the SLP input program

```
a := 1; print (a, a + 1)
```





  - **Punctuation characters** in concrete parse tree do *not* contribute to meaning of the program (*anymore*).
  - The concrete parse tree depends too much on the **structure of the grammar**.

bison based abstract syntax tree construction

```
%%
prog  : stm                            { $$ = $1; }
      ;

                                              seq
stm   : stm SEMICOLON stm              { $$ =  /   \  ; }
                                            $1     $3

                                             assign
      | ID ASSIGN exp                  { $$ =  /   \  ; }
                                            $1     $3

      | PRINT LPAREN exps RPAREN       { $$ = $3; }
      ;

                                             print
exps  : exp                            { $$ =  /   \   ; }
                                            $1    nil

                                             print
      | exp COMMA exps                 { $$ =  /   \  ; }
                                            $1     $3
      ;

exp   : INT                            { $$ = $1; }

      | ID                             { $$ = $1; }

                                               +
      | exp PLUS exp                   { $$ =  /   \  ; }
                                            $1     $3

                                               -
      | exp MINUS exp                  { $$ =  /   \  ; }
                                            $1     $3

                                               *
      | exp TIMES exp                  { $$ =  /   \  ; }
                                            $1     $3

                                               /
      | exp DIV exp                    { $$ =  /   \  ; }
                                            $1     $3

                                              eseq
      | stm COMMA exp                  { $$ =  /   \  ; }
                                            $1     $3

      | LPAREN exp RPAREN              { $$ = $2; }
      ;
%%
```

4.1.1 Positions (Source Code Coordinates)

- The generation of good **error messages** for *non-syntactical* (i.e., semantical) errors in the program becomes harder with abstract syntax trees.
  - The abstract parse tree retains no information about the **positions** (source code start line/column, end line/column) of the program fragments it represents.
  - A message about a semantical error like "`variable x has illegal type`" in a 10 000 lines program will frustrate any programmer.

> **Idea:**
> ① attach source code positions (just like semantic value) to tokens in lexer (`flex`: `... yylloc = ...`) and
> ② for a non-terminal S, attach the "bounding box" of the source code positions of the symbols of the right-hand side of an S rule.

  - In `bison`[17] semantic actions, use `@`i and `@$` to access/assign the source code position of grammar symbol S_i and the left-hand side non-terminal, respectively.
  - `bison` carries out the "bounding box" computation by default.

[17] Enable `bison`'s source code position stack via the `%locations` declaration or the `--locations` option.

4.2 Abstract Syntax for Tiger

- The abstract parse trees (also: *abstract syntax*) for Tiger feature node types that reflect the constructs of the Tiger language.

Abstract syntax for Tiger, `absyn.h`

```
...

A_var   A_SimpleVar (A_pos pos, S_symbol sym);
A_var   A_FieldVar (A_pos pos, A_var var, S_symbol sym);
A_var   A_SubscriptVar (A_pos pos,
                        A_var var, A_exp exp);

A_exp   A_VarExp (A_pos pos, A_var var);
A_exp   A_IntExp (A_pos pos, int i);
A_exp   A_CallExp (A_pos pos,
                   S_symbol func, A_expList args);
A_exp   A_OpExp (A_pos pos,
                 A_oper oper, A_exp left, A_exp right);
A_exp   A_SeqExp (A_pos pos, A_expList seq);
A_exp   A_AssignExp (A_pos pos, A_var var, A_exp exp);
A_exp   A_IfExp (A_pos pos,
                 A_exp test, A_exp then, A_exp elsee);
A_exp   A_WhileExp (A_pos pos, A_exp test, A_exp body);
A_exp   A_ForExp (A_pos pos,
                  S_symbol var, A_exp lo, A_exp hi,
                  A_exp body);
A_exp   A_LetExp (A_pos pos, A_decList decs, A_exp body);
A_dec   A_FunctionDec (A_pos pos, A_fundecList function);
A_dec   A_VarDec (A_pos pos,
                  S_symbol var, S_symbol typ, A_exp init);

A_field A_Field (A_pos pos, S_symbol name, S_symbol typ);
A_fieldList A_FieldList (A_field head, A_fieldList tail);

A_expList A_ExpList (A_exp head, A_expList tail);

...
```

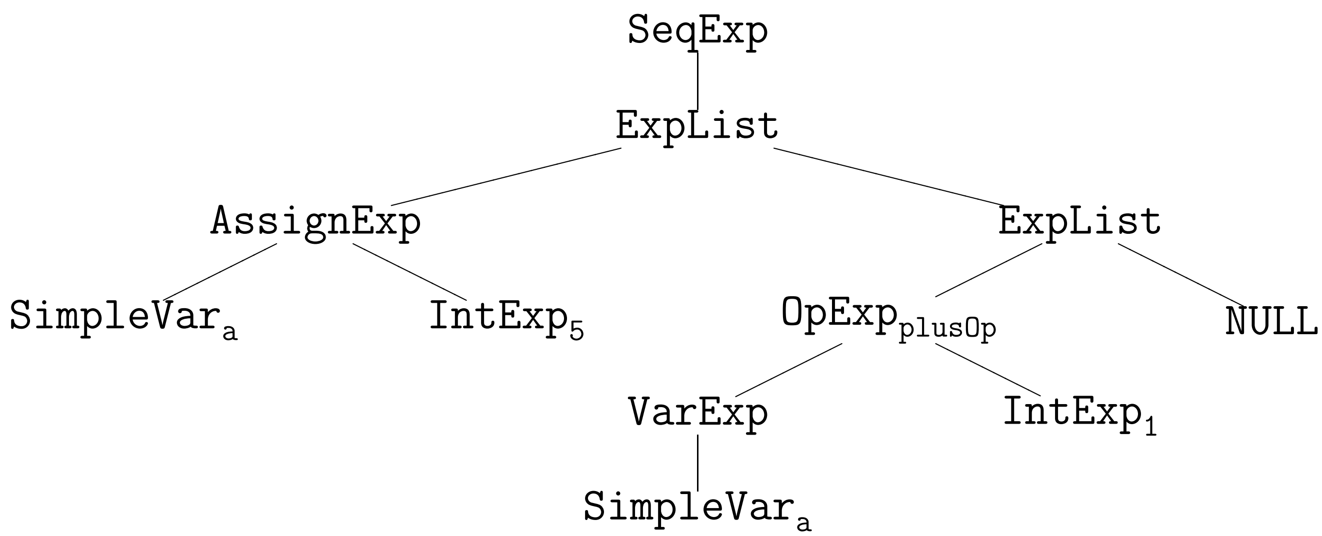
- The abstract syntax tree for the Tiger expression (evaluating to 6)

```
(a := 5; a + 1)
```

would be constructed via

C code fragment (omitting positions)

```
A_SeqExp (
    A_ExpList (A_AssignExp (A_SimplVar (S_Symbol ("a")),
                            A_IntExp (5)),
               A_ExpList (A_OpExp (A_plusOp,
                                   A_VarExp (A_SimpleVar (S_Symbol ("a"))),
                                   A_IntExp (1)),
                          NULL)))
```



- To facilitate the implementation of the following (semantic) phases of the compiler, it is sensible to design the abstract parse tree structure such that semantic units are reflected in a single subtree (if this is possible).

  - **Example:** type declarations in Tiger may be *mutually recursive* (the same is true for function declarations):

    Tiger code fragment

```
let
   type tree   = { key: int; children: forest }
   type forest = { head: tree; tail: forest }

   var t : tree = nil
in
   t
end
```

  - This is reflected in the abstract syntax by the `A_TypeDec` constructor that takes a *list* (`A_NametyList`) of type declarations not just a single declaration.
    All declarations in the list may refer to each other.

Abstract syntax tree for mutual recursive types example

```
A_LetExp (
  A_DecList (
    A_TypeDec (
      A_NametyList (
        A_Namety (S_Symbol ("tree"),
          A_Recordty (A_FieldList (
                        A_Field (S_Symbol ("key"), S_Symbol ("int")),
                        A_FieldList (
                          A_Field (S_Symbol ("children"), S_Symbol ("forest")),
                          NULL)))),
        A_NametyList (
          A_Namety (S_Symbol ("forest"),
            A_Recordty (A_FieldList (
                          A_Field (S_Symbol ("head"), S_Symbol ("tree")),
                          A_FieldList (
                            A_Field (S_Symbol ("tail"), S_Symbol ("forest")),
                            NULL)))),
          NULL))),
    A_DecList (
      A_VarDec (S_Symbol ("t"), S_Symbol ("tree"), A_NilExp ()),
      NULL)),
  A_VarExp (A_SimpleVar (S_Symbol ("t")))
)
```

N.B. `S_Symbol` constructs variable/function/type identifiers (*symbols*).

4.2.1 Simplification

- The more complex the abstract syntax tree structure (the more *distinct node types* we use), the more complex will subsequent compiler phases become.

  > **Rule:** if we can save abstract syntax tree node types (e.g., by *equivalently* reformulating expressions), then do so.

- For Tiger, several such **simplifications** can be done, e.g.:

  - e_1 `|` e_2 $\equiv$ `if` e_1 `then 1 else` e_2
  - e_1 `&` e_2 $\equiv$ `if` e_1 `then` e_2 `else 0`
  - `-` e $\equiv$ `0 -` e

  (Save the otherwise necessary `A_BoolExp` and `A_UnaryExp` node types.)

- Aggressive application of this technique can pay off. The compiler then merely operates on some sort of *minimal core language*.