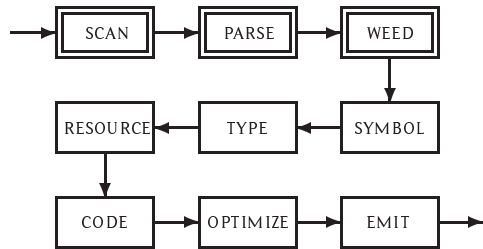


# Abstract syntax trees



A one-pass compiler only looks through the program once.

Scanning, parsing, symbol tables, type checking, and code generation happens at the same time.

A terrible methodology:

- ignores natural modularity;
- gives unnatural scope rules; and
- limits optimizations.

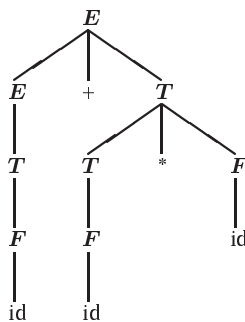
Used to be popular:

- fast (if your machine is slow); and
- space efficient (if you only have 4K).

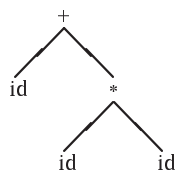
A modern compiler uses 5-15 passes.

Between passes, the program needs an intermediate representation.

An *abstract syntax tree (AST)* is essentially a parse tree:

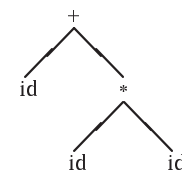


but for a more abstract grammar:



Early multi-pass compilers used *intermediate languages*.

Instead of constructing the tree:



the string:

`+(id,*(id,id))`

is written to a file.

If the compiler has  $k$  passes, then  $k - 1$  intermediate languages must be designed.

```

typedef struct EXP {
    enum {idK,intconstK,timesK,divK,plusK,minusK} kind;
    union {
        char *idE;
        int intconstE;
        struct {struct EXP *left; struct EXP *right;} timesE;
        struct {struct EXP *left; struct EXP *right;} divE;
        struct {struct EXP *left; struct EXP *right;} plusE;
        struct {struct EXP *left; struct EXP *right;} minusE;
    } val;
} EXP;

EXP *makeEXPid(char *id)
{ EXP *e;
  e = NEW(EXP);
  e->kind = idK;
  e->val.idE = id;
  return e;
}

.
.
.

EXP *makeEXPminus(EXP *left, EXP *right)
{ EXP *e;
  e = NEW(EXP);
  e->kind = minusK;
  e->val.minusE.left = left;
  e->val.minusE.right = right;
  return e;
}

```

```

%{
#include <stdio.h>
#include "tree.h"

extern char *yytext;
extern EXP *theexpression;

void yyerror() {
    printf("syntax error before %s\n",yytext);
}
%}

%union {
    int intconst;
    char *stringconst;
    struct EXP *exp;
}

%token <intconst> tINTCONST
%token <stringconst> tIDENTIFIER

%type <exp> program exp

```

```

%start program

%left '+' '-'
%left '*' '/'

%%
program: exp
        { theexpression = $1;}
;

exp : tIDENTIFIER
    { $$ = makeEXPid($1);}
| tINTCONST
    { $$ = makeEXPintconst($1);}
| exp '*' exp
    { $$ = makeEXPMult($1,$3);}
| exp '/' exp
    { $$ = makeEXPdiv($1,$3);}
| exp '+' exp
    { $$ = makeEXPplus($1,$3);}
| exp '-' exp
    { $$ = makeEXPminus($1,$3);}
| '(' exp ')'
    { $$ = $2;}
;

%%

```

### How to construct AST with flex/bison:

- design AST node structures (semantic values) (tree.h)  
kinds of nodes, associated semantic values
- write constructors for node kinds (tree.c)
- add "%union" declaration of semantic value types in the grammar file (bison)
- give types to terminals and nonterminals
 

```

%token <type> id num
%type <type> exp

```
- associate actions to each grammar rule
 

```

exp: exp '*' exp { $$ = makeEXPMult($1, $3); }

```

As mentioned before, a modern compiler uses 10–15 passes. Each pass contributes extra information to the AST nodes:

- scanner: line numbers;
- symbol tables: meaning of identifiers;
- type checking: types of expressions; and
- code generation: assembler code.

The AST types must include extra fields:

```
typedef struct EXP {
    int lineno;
    enum {idK,intconstK,timesK,divK,plusK,minusK} kind;
    union {
        char *idE;
        int intconstE;
        struct {struct EXP *left; struct EXP *right;} timesE;
        struct {struct EXP *left; struct EXP *right;} divE;
        struct {struct EXP *left; struct EXP *right;} plusE;
        struct {struct EXP *left; struct EXP *right;} minusE;
    } val;
} EXP;
```

Line numbers in the scanner:

```
%{
#include "y.tab.h"
#include <string.h>

extern int lineno;
}%

%%
[ \t]+ /* ignore */;
\n      lineno++;

"*"      return '>';
"/"      return '/';
"+"      return '+';
"-"      return '-';
"("      return '(';
")"      return ')';

0|([1-9][0-9]*)
{ yylval.intconst = atoi(yytext);
  return tINTCONST;
}

[a-zA-Z_][a-zA-Z0-9_]*
{ yylval.stringconst =
  (char *)malloc(strlen(yytext)+1);
  sprintf(yylval.stringconst,"%s",yytext);
  return tIDENTIFIER;
}

%%
```

Line numbers in the constructors:

```
extern int lineno;

EXP *makeEXPid(char *id)
{ EXP *e;
  e = NEW(EXP);
  e->lineno = lineno;
  e->kind = idK;
  e->val.idE = id;
  return e;
}

EXP *makeEXPintconst(int intconst)
{ EXP *e;
  e = NEW(EXP);
  e->lineno = lineno;
  e->kind = intconstK;
  e->val.intconstE = intconst;
  return e;
}

.
.
.

EXP *makeEXPminus(EXP *left, EXP *right)
{ EXP *e;
  e = NEW(EXP);
  e->lineno = lineno;
  e->kind = minusK;
  e->val.minusE.left = left;
  e->val.minusE.right = right;
  return e;
}
```

A “pretty” printer:

```
void prettyEXP(EXP *e)
{ switch (e->kind) {
    case idK:
        printf("%s",e->val.idE);
        break;

    case intconstK:
        printf("%i",e->val.intconstE);
        break;

    case timesK:
        printf("(");
        prettyEXP(e->val.timesE.left);
        printf("*");
        prettyEXP(e->val.timesE.right);
        printf(")");
        break;

    .
    .
    .

    case minusK:
        printf("(");
        prettyEXP(e->val.minusE.left);
        printf("-");
        prettyEXP(e->val.minusE.right);
        printf(")");
        break;

    }
}
```

## The program:

```
#include "tree.h"
#include "pretty.h"

int lineno;

void yyparse();

EXP *theexpression;

void main()
{ lineno = 1;
  yyparse();
  prettyEXP(theexpression);
}
```

## will on input:

```
a*(b-17) + 5/c
```

## produce the output:

```
((a*(b-17))+(5/c))
```

## SableCC grammar for our TINY expression language:

```
Package tiny;
```

## Helpers

```
tab    = 9;
cr     = 13;
lf     = 10;
digit  = ['0'..'9'];
lowercase = ['a'..'z'];
uppercase = ['A'..'Z'];
letter = lowercase | uppercase;
idletter = letter | '_' ;
idchar  = letter | '_' | digit;
```

## Tokens

```
eol    = cr | lf | cr lf;
blank  = ' ' | tab;
star   = '*';
slash  = '/';
plus   = '+';
minus  = '-';
l_par  = '(';
r_par  = ')';
number = '0' | [digit-'0'] digit*;
id     = idletter idchar*;
```

## Ignored Tokens

```
blank,eol;
```

## Productions

```
exp =
  {plus}   exp plus factor |
  {minus}  exp minus factor |
  {factor} factor;

factor =
  {mult}   factor star term |
  {divd}   factor slash term |
  {term}   term;

term =
  {paren}  l_par exp r_par |
  {id}     id |
  {number} number;
```

## SableCC generates subclasses of the 'Node' class for terminal, non-terminals and alternatives:

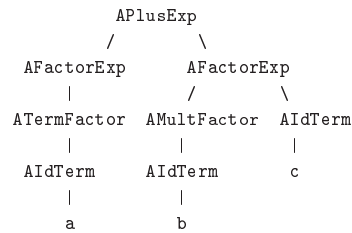
- node classes for terminals: 'T' followed by terminal name:  
TEol, TNumber, TId, etc...
- node classes for non-terminals: 'P' followed by non-terminal name:  
PExp, PFactor, and PTerm
- node classes for alternatives: 'A' followed by alternative name and non-terminal name: APlusExp (extends PExp), ANumberTerm (extends PNumber), etc..

SableCC generates file structures which look like following:

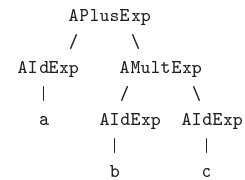
```
tiny/
|--analysis/  Analysis.java
|             AnalysisAdapter.java
|             DepthFirstAdapter.java
|             ReversedDepthFirstAdapter.java
|
|--lexer/     Lexer.java lexer.dat
|             LexerException.java
|
|--node/      Node.java TEol.java TNumber.java ...
|             PExp.java PFactor.java PTerm.java
|             APlusExp.java, ..., AMultFactor, ...,
|             AParenTerm, ...
|
|--parser/    parser.dat Parser.java
|             ParserException.java, ...
|
|--your code directories ...
```

Given previous grammar, SableCC generates a parser which builds a concrete syntax tree (CST) for an input program.

For example, the CST of the program "a + b \* c":



The CST has many redundant intermediate nodes, we only need an Abstract Syntax Tree (AST) to operate on:



Bison uses actions after grammar rules. A compiler writer puts code in actions for constructing ASTs.

SableCC allows a user define AST in a grammar file, then it translates CST to AST automatically.

Define AST for the EXP language:

```
Abstract Syntax Tree
exp =
{plus}      [l]:exp [r]:exp
| {minus}    [l]:exp [r]:exp
| {mult}     [l]:exp [r]:exp
| {divd}     [l]:exp [r]:exp
| {id}       id
| {number}   number;
```

Here, a new section, "*Abstract Syntax Tree*", has been added after the production section. AST rules have the same syntax as rules in production section, except:

- no transformation (DO NOT write `exp {-> foo}` in the AST section);

AST example:

```
exp =
{plus}      [l]:exp [r]:exp
.....
| {number}   number;

stm =
.....
| {block}    [stmts]:stm*;
```

The productions of the EXP language are expanded with transformations:

Productions

```
cst_exp {-> exp} =
  {cst_plus}    cst_exp plus factor
                {-> New exp.plus(cst_exp.exp, factor.exp)}
| {cst_minus}   cst_exp minus factor
                {-> New exp.minus(cst_exp.exp, factor.exp)}
| {factor}      factor {-> factor.exp};

factor {-> exp} =
  {cst_mult}    factor star term
                {-> New exp.mult(factor.exp, term.exp)}
| {cst_divd}    factor slash term
                {-> New exp.divd(factor.exp, term.exp)}
| {term}        term {-> term.exp};

term {-> exp} =
  {paren}       l_par cst_exp r_par {-> cst_exp.exp}
| {cst_id}      id {-> New exp.id(id)}
| {cst_number}  number {-> New exp.number(number)};
```

In the production section, a CST node, matched by

```
cst_exp = {cst_plus} cst_exp plus factor
```

, is transformed to an AST node using the following rule:

```
cst_exp {-> exp} =
  {cst_plus} cst_exp plus factor
                {-> New exp.plus(cst_exp.exp, factor.exp)}
```

The left-hand side, `cst_exp {-> exp}`, says that a CST node (*cst\_exp*) should be transformed to an AST node (*exp*).

The transformation at the right-hand side, `{-> New exp.plus(cst_exp.exp, factor.exp)}`, gives the action for constructing the AST node.

The *cst\_exp.exp* refers to the transformed AST node of the first term, *cst\_exp*, in the right-hand side of the production rule.

There are FIVE types of transformations (actions) that can happen on the right-hand side of a production:

- getting an existing identifier (token) :: *ident*  

```
{paren} l_par cst_exp r_par {-> cst_exp.exp}
```
- creating a new AST node :: *New* ...  

```
{cst_id} id {-> New exp.id(id)}
```
- list creation :: *[elm1, elm2,...]*  

```
{block} l_brace stm* r_brace {-> New stm.block([stm])}
```
- elimination :: *Null*
- empty transformation :: `{-> }`

It might be tedious to write down AST transformations for all CST productions even if a CST production has the same structure as its AST counterpart.

SableCC3 allows implicit transformations in these cases. For example, a production without explicit transformation:

```
prod = elm1 elm2* elm3+ elm4?;
```

is equivalent to

```
prod{-> prod} = elm1 elm2* elm3+ elm4?
{-> New prod.prod(elm1.elm1, [elm2.elm2],
                  [elm3.elm3], elm4.elm4)};
```

## The JOOS compiler has the AST node types:

PROGRAM	CLASSFILE	CLASS
FIELD	TYPE	LOCAL
CONSTRUCTOR	METHOD	FORMAL
STATEMENT	EXP	RECEIVER
ARGUMENT	LABEL	CODE

## with many extra fields:

```
typedef struct METHOD {
    int lineno;
    char *name;
    ModifierKind modifier;
    int localslimit; /* resource */
    int labelcount; /* resource */
    struct TYPE *returntype;
    struct FORMAL *formals;
    struct STATEMENT *statements;
    char *signature; /* code */
    struct LABEL *labels; /* code */
    struct CODE *opcodes; /* code */
    struct METHOD *next;
} METHOD;
```

## The JOOS constructors are as we expect:

```
METHOD *makeMETHOD(char *name, ModifierKind modifier,
                    TYPE *returntype, FORMAL *formals,
                    STATEMENT *statements, METHOD *next)
{
    METHOD *m;
    m = NEW(METHOD);
    m->lineno = lineno;
    m->name = name;
    m->modifier = modifier;
    m->returntype = returntype;
    m->formals = formals;
    m->statements = statements;
    m->next = next;
    return m;
}

STATEMENT *makeSTATEMENTwhile(EXP *condition,
                              STATEMENT *body)
{
    STATEMENT *s;
    s = NEW(STATEMENT);
    s->lineno = lineno;
    s->kind = whileK;
    s->val.whileS.condition = condition;
    s->val.whileS.body = body;
    return s;
}
```

## Highlights from the JOOS scanner:

```
[ \t]+      /* ignore */;
\n          lineno++;
\\\/[^\n]*  /* ignore */;
abstract    return tABSTRACT;
boolean      return tBOOLEAN;
break       return tBREAK;
byte        return tBYTE;
.
.
"!="        return tNEQ;
"&&"        return tAND;
"||"        return tOR;
"+"         return '+';
"-"         return '-';
.
.
.
0|([1-9][0-9]*) {yylval.intconst = atoi(yytext);
                  return tINTCONST;}
true         {yylval.boolconst = 1;
              return tBOOLCONST;}
false        {yylval.boolconst = 0;
              return tBOOLCONST;}
\"([^\"])*\"  {yylval.stringconst =
              (char*)malloc(strlen(yytext)-1);
              yytext[strlen(yytext)-1] = '\0';
              sprintf(yyval.stringconst,"%s",yytext+1);
              return tSTRINGCONST;}
```

## Highlights from the JOOS parser:

```
method : tPUBLIC methodmods returntype
        tIDENTIFIER '(' formals ')' '{' statements '}'
        { $$ = makeMETHOD($4,$2,$3,$6,$9,NULL); }
| tPUBLIC returntype
  tIDENTIFIER '(' formals ')' '{' statements '}'
  { $$ = makeMETHOD($3,modNONE,$3,$5,$8,NULL); }
| tPUBLIC tABSTRACT returntype
  tIDENTIFIER '(' formals ')' ';;'
  { $$ = makeMETHOD($4,modABSTRACT,$3,$6,NULL,NULL); }
| tPUBLIC tSTATIC tVOID
  tMAIN '(' mainargv ')' '{' statements '}'
  { $$ = makeMETHOD("main",modSTATIC,
                    makeTYPEvoid(),NULL,$9,NULL); }
;

whilestatement : tWHILE '(' expression ')' statement
               { $$ = makeSTATEMENTwhile($3,$5); }
;
```

## Building LALR(1) lists:

```

formals : /* empty */
        { $$ = NULL; }
        | neformals
        { $$ = $1; }
;

neformals : formal
          { $$ = $1; }
          | neformals ',' formal
          { $$ = $3; $$->next = $1; }
;

formal : type tIDENTIFIER
       { $$ = makeFORMAL($2,$1,NULL); }
;

```

The lists are naturally backwards.

## Using backwards lists:

```

typedef struct FORMAL {
    int lineno;
    char *name;
    int offset; /* resource */
    struct TYPE *type;
    struct FORMAL *next;
} FORMAL;

void prettyFORMAL(FORMAL *f)
{ if (f!=NULL) {
    prettyFORMAL(f->next);
    if (f->next!=NULL) printf(", ");
    prettyTYPE(f->type);
    printf(" %s",f->name);
  }
}

```

## The JOOS grammar calls for:

```

castexpression :
    '(' identifier ')' unaryexpressionnotminus

```

but that is not LALR(1).

However, the more general rule:

```

castexpression :
    '(' expression ')' unaryexpressionnotminus

```

is LALR(1), so we use a clever action:

```

castexpression :
    '(' expression ')' unaryexpressionnotminus
    { if ($2->kind!=idK) yyerror("identifier expected");
      $$ = makeEXPcast($2->val.idE.name,$4); }
;

```

Hacks like this only work sometimes.

## LALR(1) and Bison are not enough, when:

- our language is not context-free;
- our language is not LALR(1); or
- an LALR(1) grammar is too large and complex.

In these cases we use a liberal grammar, which accepts a slightly larger language.

A separate pass may then weed out the bad parse trees.



## Division by constant 0 is not allowed:

```

exp : tIDENTIFIER
    | tINTCONST
    | exp '*' exp
    | exp '/' pos
    | exp '+' exp
    | exp '-' exp
    | '(' exp ')'
    ;

pos : tIDENTIFIER
    | tINTCONSTPOSITIVE
    | exp '*' exp
    | exp '/' pos
    | exp '+' exp
    | exp '-' exp
    | '(' pos ')'
    ;

```

We have doubled the size of our grammar.

Not a very modular technique.

## Weed out division by constant 0:

```

int zerodivEXP(EXP *e)
{
    switch (e->kind) {
        case idK:
        case intconstK:
            return 0;
        case timesK:
            return zerodivEXP(e->val.timesE.left) ||
                zerodivEXP(e->val.timesE.right);
        case divK:
            if (e->val.divE.right->kind==intconstK &&
                e->val.divE.right->val.intconstE==0) return 1;
            return zerodivEXP(e->val.divE.left) ||
                zerodivEXP(e->val.divE.right);
        case plusK:
            return zerodivEXP(e->val.plusE.left) ||
                zerodivEXP(e->val.plusE.right);
        case minusK:
            return zerodivEXP(e->val.minusE.left) ||
                zerodivEXP(e->val.minusE.right);
    }
}

```

A simple, modular traversal.

## Requirements to JOOS programs:

- all local variable declarations must appear at the beginning of a statement sequence:

```

int i;
int j;
i=17;
int b;
b=i;

```

- every branch through the body of a non-void method must terminate with a return statement:

```

if (x.equals(y)) return true;
else x=null;

```

These are hard or impossible to express through an LALR(1) grammar.

## Weed bad local declarations:

```

int weedSTATEMENTlocals(STATEMENT *s,int localsallowed)
{
    int onlylocalsfirst, onlylocalssecond;
    if (s!=NULL) {
        switch (s->kind) {
            case skipK:
                return 0;
            case localK:
                if (!localsallowed) {
                    reportError("illegally placed local declaration",s->lineno);
                }
                return 1;
            case expK:
                return 0;
            case returnK:
                return 0;
            case sequenceK:
                onlylocalsfirst =
                    weedSTATEMENTlocals(s->val.sequenceS.first,localsallowed);
                onlylocalssecond =
                    weedSTATEMENTlocals(s->val.sequenceS.second,onlylocalsfirst);
                return onlylocalsfirst && onlylocalssecond;
            case ifK:
                (void)weedSTATEMENTlocals(s->val.ifS.body,0);
                return 0;
            case ifelseK:
                (void)weedSTATEMENTlocals(s->val.ifelseS.thenpart,0);
                (void)weedSTATEMENTlocals(s->val.ifelseS.elsepart,0);
                return 0;
            case whileK:
                (void)weedSTATEMENTlocals(s->val.whileS.body,0);
                return 0;
            case blockK:
                (void)weedSTATEMENTlocals(s->val.blockS.body,1);
                return 0;
            case superconsK:
                return 1;
        }
    }
}

```

## Weed missing returns:

```

int weedSTATEMENTReturns(STATEMENT *s)
{ if (s!=NULL) {
    switch (s->kind) {
        case skipK:
            return 0;
        case localK:
            return 0;
        case expK:
            return 0;
        case returnK:
            return 1;
        case sequenceK:
            return weedSTATEMENTReturns(s->val.sequenceS.second);
        case ifK:
            return 0;
        case ifelseK:
            return weedSTATEMENTReturns(s->val.ifelseS.thenpart) &&
                weedSTATEMENTReturns(s->val.ifelseS.elsepart);
        case whileK:
            return 0;
        case blockK:
            return weedSTATEMENTReturns(s->val.blockS.body);
        case superconsK:
            return 0;
    }
}
}

```

The testing strategy for a parser that constructs abstract syntax trees usually involves a pretty printer.

If  $\text{parse}(P)$  constructs a tree and  $\text{pretty}(T)$  reconstructs the text, then:

$$\text{pretty}(\text{parse}(P)) \approx P$$

Even better, we have that:

$$\text{pretty}(\text{parse}(\text{pretty}(\text{parse}(P))) \equiv \text{pretty}(\text{parse}(P))$$

Of course, this is a necessary but not sufficient condition.