A compiler pass is a traversal of the program. A compiler phase is a group of related passes. A one-pass compiler scans the program only once. It is naturally single-phase. The following all happen at the same time:

- scanning
- parsing
- weeding
- symbol table creation
- type checking
- resource allocation
- code generation
- optimization
- emitting

This is a terrible methodology:

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

However, it used to be popular:

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

A modern multi-pass compiler uses 5–15 phases, some of which may have many individual passes: you should skim through the optimization section of ‘man gcc’ some time!

A multi-pass compiler needs an intermediate representation of the program between passes.

We could use a parse tree, or concrete syntax tree (CST):

or we could use a more convenient abstract syntax tree (AST), which is essentially a parse tree/CST but for a more abstract grammar:
Instead of constructing the tree:

```
+  
/ 
id id
```

a compiler can generate code for an internal compiler-specific grammar, also known as an intermediate language.

Early multi-pass compilers wrote their IL to disk between passes. For the above tree, the string `+(id,*(id,id))` would be written to a file and read back in for the next pass.

It may also be useful to write an IL out for debugging purposes.

If the compiler has $k$ phases that change the fundamental representation of the program, then $k - 1$ intermediate languages must be designed.

Examples of modern intermediate languages:

- Java bytecode
- C, for certain high-level language compilers
- Jimple, a 3-address representation of Java bytecode specific to Soot that you learn about in COMP 621
- Simple, the precursor to Jimple that Laurie Hendren created for McCAT
- Gimple, the IL based on Simple that gcc uses

In this course, you will generally use an AST as your IR without the need for an explicit IL.

Note: somewhat confusingly, both industry and academia use the terms IR and IL interchangeably.

```c
$ cat tree.h  # AST construction for Tiny language
... 
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;
}
```

```c
$ cat tiny.y  # Tiny parser that creates EXP *theexpression
... 
%union {
  int intconst;
  char *stringconst;
  struct EXP *exp;
}
%token <intconst> tINTCONST
%token <stringconst> tIDENTIFIER
%type <exp> program exp

%{
  
#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

  [...]
```
Abstract syntax trees (9)

%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; }
;
%%

Abstract syntax trees (10)

Constructing an AST with flex/bison:
- AST node kinds go in tree.h
  enum {idK, intconstK, timesK, divK, plusK, minusK} kind;
- AST node semantic values go in tree.h
  struct {struct EXP *left; struct EXP *right;} minusE;
- Constructors for node kinds go in tree.c
  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;
  }
- Semantic value type declarations go in tiny.y
  %union {
    int intconst;
    char *stringconst;
    struct EXP *exp;
  }
- (Non-)terminal types go in tiny.y
  %token <intconst> tINTCONST
  %token <stringconst> tIDENTIFIER
  %type <exp> program exp
- Grammar rule actions go in tiny.y
  exp : exp '-' exp { $$ = makeEXPminus ($1, $3); }

Abstract syntax trees (11)

A “pretty” printer:

$ cat pretty.h
#include <stdio.h>
#include "pretty.h"

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;
}
}

Abstract syntax trees (12)

The following pretty printer program:

$ cat main.c
#include "tree.h"
#include "pretty.h"

void yyparse();
EXP *theexpression;

void main()
{ yyparse();
  prettyEXP(theexpression);
}

will on input:
a*(b-17) + 5/c
produce the output:
((a*(b-17))+(5/c))
As mentioned before, a modern compiler uses 5–15 phases. Each phase contributes extra information to the IR (AST in our case):

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

**Example:** adding line number support.

First, introduce a global `lineno` variable:

```c
$ cat main.c
[...]
int lineno;
void main() {
    lineno = 1; /* input starts at line 1 */
    yyparse();
    prettyEXP(theexpression);
}
```

Second, increment `lineno` in the scanner:

```c
$ cat tiny.l # modified version of previous exp.l
%
#include "y.tab.h"
#include <string.h>
#include <stdlib.h>
extern int lineno; /* declared in main.c */
%
[y]
{"/t}" /* ignore */; /* no longer ignore \n */
\n lineno++; /* increment for every \n */
[...]
```

Third, add a `lineno` field to the AST nodes:

```c
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;
```

Fourth, set `lineno` in the node constructors:

```c
extern int lineno; /* declared in main.c */
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;
}
```

The SableCC 2 grammar for our Tiny language:

```plaintext
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

```plaintext
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 terminals, non-terminals and production alternatives:

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

### SableCC populates an entire directory structure:

```plaintext
tiny/
|--analysis/ Analysis.java
    | AnalysisAdapter.java
    | DepthFirstAdapter.java
    | ReversedDepthFirstAdapter.java
    |
|--lexer/ Lexer.java lexer.dat
    | LexerException.java
    |
|--node/ Node.java TEol.java ... TId.java
    | PExp.java PFactor.java PTerm.java
    | APlusExp.java ...
    | AMultFactor.java ...
    | AParenTerm.java ...
    |
|--parser/ parser.dat Parser.java
    | ParserException.java ...
    |
|-- custom code directories, e.g. symbol, type, ...
```

### Given some grammar, SableCC generates a parser that in turn builds a concrete syntax tree (CST) for an input program.

A parser built from the Tiny grammar creates the following CST for the program `a+b*c`:

```
Start
  | APlusExp
  / \ AFactorExp AMultFactor
  | / \ ATermFactor ATermFactor AIdTerm
  | / AIdTerm AIdTerm c
  | a b
```

This CST has many unnecessary intermediate nodes. Can you identify them?
We only need an abstract syntax tree (AST) to operate on:

```
APlusExp
/    \
AIdExp AMultExp
|    |      
a AIdExp AIdExp  b  c
```

Recall that bison relies on user-written actions after grammar rules to construct an AST.

As an alternative, SableCC 3 actually allows the user to define an AST and the CST $\rightarrow$ AST transformations formally, and can then translate CSTs to ASTs automatically.

Extending Tiny productions with CST $\rightarrow$ AST 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)};
```

AST for the Tiny expression 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;
```

AST rules have the same syntax as rules in the Production section except for CST $\rightarrow$ AST transformations (obviously).

A CST production alternative for a plus node:

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

needs extending to include a CST $\rightarrow$ AST transformation:

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

cst_exp {-> exp} on the LHS specifies that the CST node cst_exp should be transformed to the AST node exp.

{-> New exp.plus(cst_exp.exp, factor.exp)} on the RHS specifies the action for constructing the AST node.

exp.plus is the kind of exp AST node to create. cst_exp.exp refers to the transformed AST node exp of cst_exp, the first term on the RHS.
5 types of explicit RHS transformation (action):

1. Getting an existing node:
   \( \{\text{paren}\} \ l_{\text{par}} \ \text{cst}_{\text{exp}} \ r_{\text{par}} \ \rightarrow \ \text{cst}_{\text{exp}.\text{exp}} \)

2. Creating a new AST node:
   \( \{\text{cst}_{\text{id}}\} \ \text{id} \ \rightarrow \ \text{New} \ \text{id}.(\text{id}) \)

3. List creation:
   \( \{\text{block}\} \ l_{\text{brace}} \ \text{stm}^* \ r_{\text{brace}} \ \rightarrow \ \text{New} \ \text{stm}.\text{block}([\text{stm}]) \)

4. Elimination (but more like nullification):
   \( \{\rightarrow \ \text{Null} \} \)

5. Empty (but more like deletion):
   \( \{\rightarrow \} \)

Writing down straightforward, non-abstracting \( \text{CST} \rightarrow \text{AST} \) transformations can be tedious.
Can depend on implicit transformations:
\[ \text{prod} = \text{elm}1 \ \text{elm}2^* \ \text{elm}3^+ \ \text{elm}4^? \]
This is equivalent to:
\[ \text{prod}(\rightarrow \ \text{prod}) = \text{elm}1 \ \text{elm}2^* \ \text{elm}3^+ \ \text{elm}4^? \]
\[ \rightarrow \ \text{New} \ \text{prod}.\text{prod}([\text{elm}1.\text{elm}1], [\text{elm}2.\text{elm}2], [\text{elm}3.\text{elm}3], \text{elm}4.\text{elm}4)) \]

More SableCC 3 documentation:
- [http://sablecc.sourceforge.net/documentation.html](http://sablecc.sourceforge.net/documentation.html)

The JOOS compiler has the AST node types:

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>CLASSFILE</th>
<th>CLASS</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIELD</td>
<td>TYPE</td>
<td>LOCAL</td>
</tr>
<tr>
<td>CONSTRUCTOR</td>
<td>METHOD</td>
<td>FORMAL</td>
</tr>
<tr>
<td>STATEMENT</td>
<td>EXP</td>
<td>RECEIVER</td>
</tr>
<tr>
<td>ARGUMENT</td>
<td>LABEL</td>
<td>CODE</td>
</tr>
</tbody>
</table>

with many extra fields:

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

The JOOS constructors are as we expect:

```c
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;
}
```

```c
STATEMENT *makeSTATEMENTwhile(EXPR *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:

```c
/* ignore */;
\nlineno++; /* ignore */;
abstract return tABSTRACT;
boolean return tBOOLEAN;
break return tBREAK;
byte return tBYTE;

. .

. "!=" return tNEQ;  /* ignore */;
"&&" return tAND;
"||" return tOR;
"+" return '+';
\n0|([1-9]\[0-9\]*) {yylval.intconst = atoi(yytext);
true {yylval.boolconst = 1;  /* ignore */;
false {yylval.boolconst = 0;  /* ignore */;
\n"(\"[\"
\n0|([1-9]\[0-9\]*) {yylval.intconst = atoi(yytext);
return tINTCONST;)
true {yylval.boolconst = 1;  /* ignore */;
false {yylval.boolconst = 0;  /* ignore */;
```

Building LALR(1) lists:

```c
formals : /* empty */
. ($$ = NULL;)
| neformals
. ($$ = $1;)
;
```

```c
neformals : formal
. ($$ = $1;)
| neformals ,' formal
. ($$ = $3; $$->next = $1;)
;
```

```c
formal : type tIDENTIFIER
. ($$ = makeFORMAL($2,$1,NULL;))
;
```

The lists are naturally backwards.

Using backwards lists:

```c
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);
  }
}
```

What effect would a call stack size limit have?
The JOOS grammar calls for:

\[
\text{castexpression} : '(', \text{identifier} ')', \text{unaryexpressionnotminus}
\]

but that is not LALR(1).

However, the more general rule:

\[
\text{castexpression} : '(', \text{expression} ')', \text{unaryexpressionnotminus}
\]

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

\[
\text{castexpression} : '(', \text{expression} ')', \text{unaryexpressionnotminus}
\]

\[
\{\text{if ($2->\text{kind}\neq \text{idK}) \text{yyerror("identifier expected")};
$$ = \text{makeEXPcast($2->\text{val}.\text{idE}.\text{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 big and complicated.

In these cases we can try using a more liberal grammar which accepts a slightly larger language. A separate phase can then weed out the bad parse trees.

**Example:** disallowing division by constant 0:

\[
\begin{align*}
\text{exp} : & \text{tIDENTIFIER} \\
& | \text{tINTCONST} \\
& | \text{exp '('* exp} \\
& | \text{exp '/' pos} \\
& | \text{exp '+' exp} \\
& | \text{exp '-' exp} \\
& | \text{'} exp '^{}'} \\
& \\
\text{pos} : & \text{tIDENTIFIER} \\
& | \text{tINTCONSTPOSITIVE} \\
& | \text{exp '('* exp} \\
& | \text{exp '/' pos} \\
& | \text{exp '+' exp} \\
& | \text{exp '-' exp} \\
& | \text{'} pos '} '
\end{align*}
\]

We have doubled the size of our grammar.

This is not a very modular technique.

Instead, weed out division by constant 0:

```c
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 of JOOS programs:

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

  ```java
  int i;
  int j;
  int b; /* illegal */
  b=1;
  ```

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

```java
  boolean foo (Object x, Object y) {
    if (x.equals(y))
      return true;
  } /* illegal */
```

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

Weeding bad local declarations:

```java
  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;
          }
          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;
      }
    }
  }
```

Weeding missing returns:

```java
  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 ifS:
          return 0;
        case ifelse:
          return weedSTATEMENTreturns(s->val.ifelseS.thenpart) &&
          weedSTATEMENTreturns(s->val.ifelseS.elsepart);
        case whileS:
          return 0;
        case blockS:
          return weedSTATEMENTreturns(s->val.blockS.body);
        case superconsS:
          return 0;
      }
    }
  }
```

The testing strategy for a parser that constructs an abstract syntax tree \( T \) from a program \( P \) usually involves a pretty printer. If \( \text{parse}(P) \) constructs \( T \) and \( \text{pretty}(T) \) reconstructs the text of \( P \), 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 for parser correctness.