Compiler Construction
Seminar 02 – TDDE66 2025

Adrian Pop (adrian.pop@liu.se)
Martin Sjölund (martin.sjolund@liu.se)

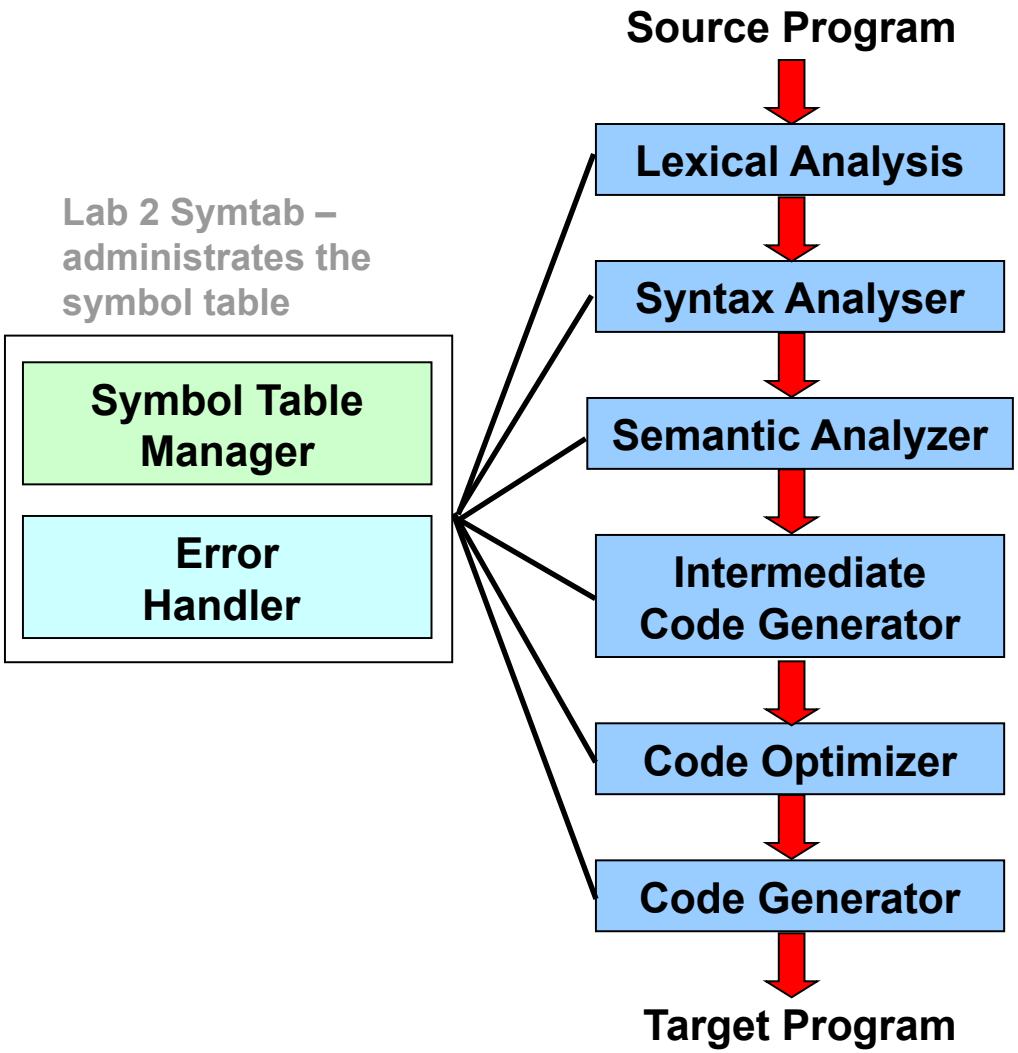
Department of Computer and Information Science
Linköping University

LABS

Lab 3 LR parsing and abstract syntax tree construction using **`bison`**

Lab 4 Semantic analysis (type checking)

PHASES OF A COMPILER



Lab 1 Scanner – manages lexical analysis

Lab 3 Parser – manages syntactic analysis, build internal form

Lab 4 Semantics – checks static semantics

Lab 6 Quads – generates quadruples from the internal form

Lab 5 Optimizer – optimizes the internal form

Lab 7 Codegen – expands quadruples to assembler

LAB 3
PARSING

SYNTAX ANALYSIS

- The parser accepts tokens from the scanner and verifies the *syntactic correctness* of the program specification
- Along the way, it also derives information about the program and builds a fundamental data structure known as **`[abstract] syntax tree`**

- The syntax tree is an internal representation of the program and augments the symbol table. The **`parse tree`** is a concrete syntax tree and is not produced by the parser

PURPOSE

- To verify the syntactic correctness of the input token stream, reporting any errors and to produce a syntax tree and certain tables for use by later phases
  - Syntactic correctness is judged by verification against a formal grammar which specifies the language to be recognized
  - Error messages are important and should be as meaningful as possible
  - Parse tree and tables will vary depending on compiler

Match token stream using manually or automatically generated parser

PARSING STRATEGIES

Two categories of parsers:

- **Top-down** parsers
- **Bottom-up** parsers

Within each of these broad categories are a number of sub strategies depending on whether leftmost or rightmost derivations are used

TOP-DOWN PARSING

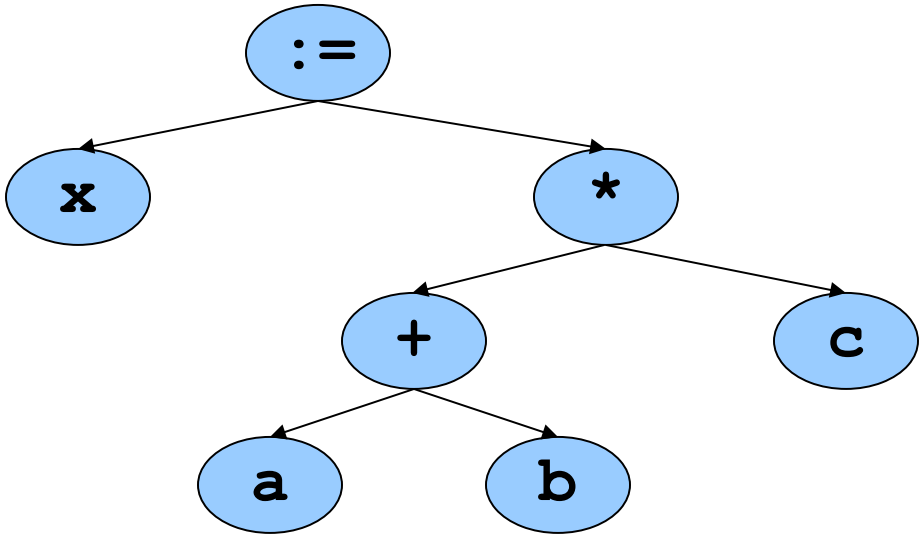
Start with a goal symbol and recognize it in terms of its constituent symbols

Example: recognize a procedure in terms of its sub-components (header, declarations, and body)

The parse tree is then built from the top (root) and down (leaves), hence the name

TOP-DOWN PARSING (cont'd)

```
X := ( a + b ) * c;
```



BOTTOM-UP PARSING

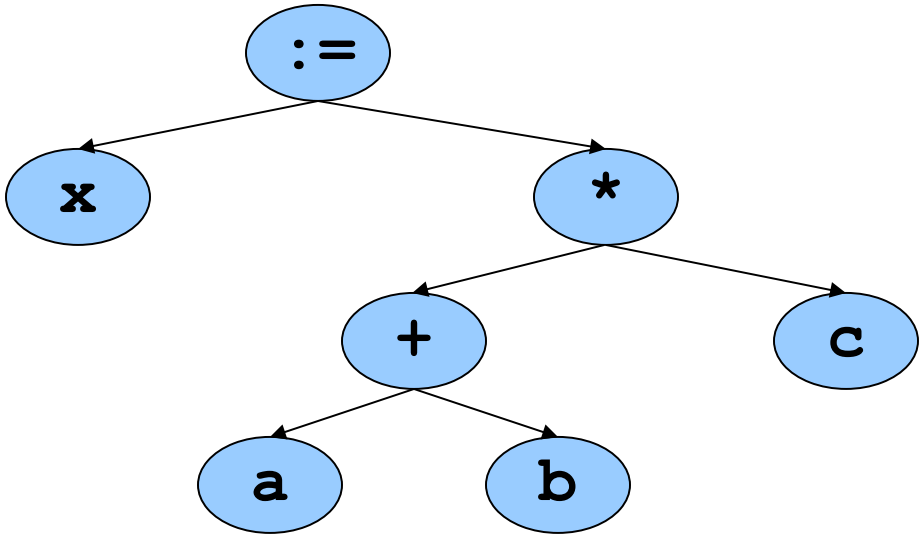
Recognize the components of a program and then combine them to form more complex constructs until a whole program is recognized

Example: recognize a procedure from its sub-components (header, declarations, and body)

The parse tree is then built from the bottom and up, hence the name

BOTTOM-UP PARSING (cont'd)

```
x := ( a + b ) * c;
```



Parsing techniques

A number of different parsing techniques are commonly used for syntax analysis, including:

- Recursive-descent parsing
- LR parsing
- Operator precedence parsing
- Many more …

LR PARSING

A specific bottom-up technique

- LR stands for **`Left->right`** scan, **`Rightmost`** derivation
- Probably the most common & “popular” parsing technique
- **`yacc`**, **`bison`**, and many other parser generation tools utilize LR parsing
- Great for machines, not so good for humans …

+ AND – FOR LR

➔ Advantages of LR:

- Accept a wide range of grammars/languages
- Well suited for automatic parser generation
- Very fast
- Generally easy to maintain

➔ Disadvantages of LR:

- Error handling can be tricky
- Difficult to use manually

bison AND yacc USAGE

bison is a general-purpose parser generator that converts a grammar description for an **LALR**(1) context-free grammar into a **C** program to parse that grammar

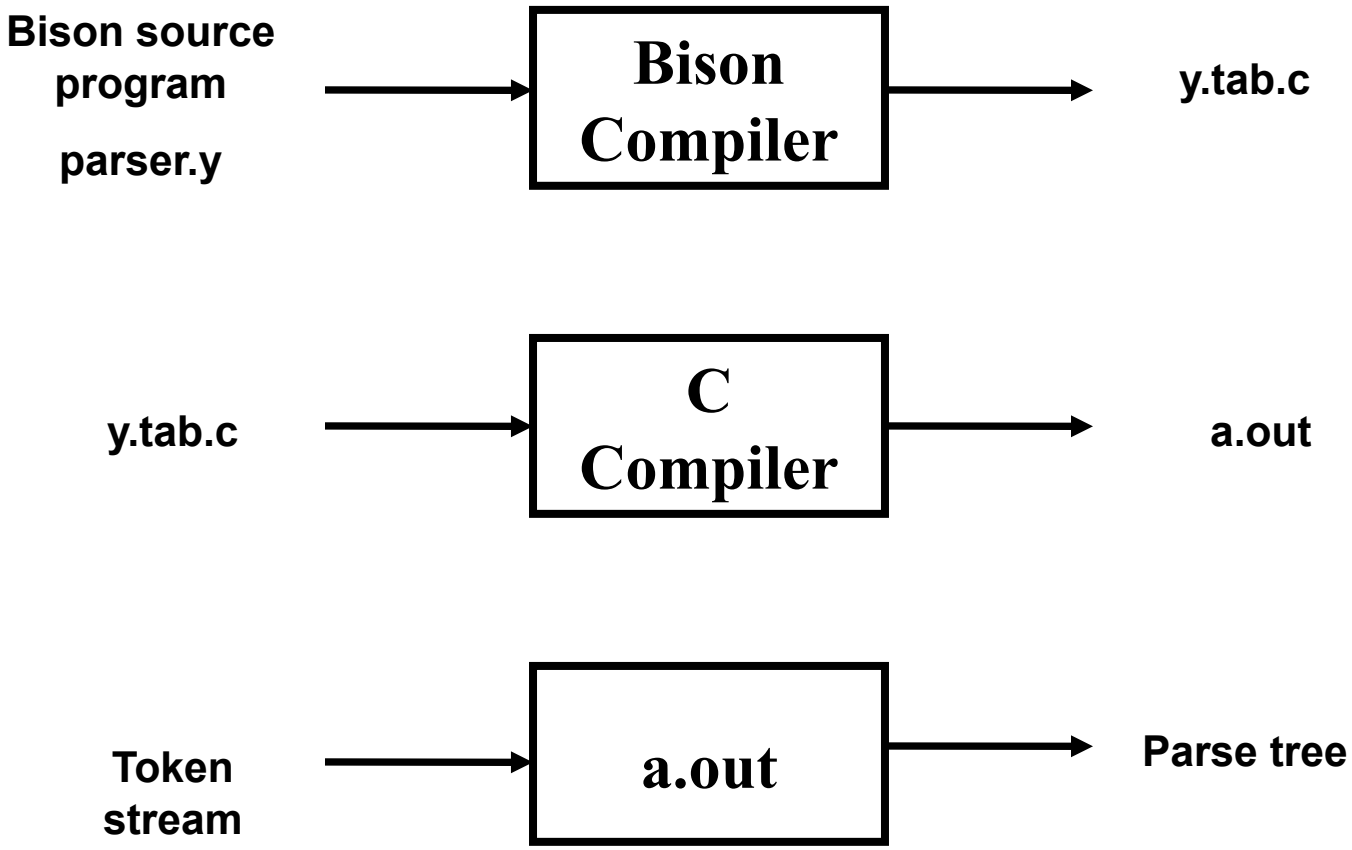
bison AND yacc USAGE

One of many parser generator packages

Yet Another Compiler Compiler

- Really a poor name, is more of a parser compiler
- Can specify actions to be performed when each construct is recognized and thereby make a full fledged compiler but its the user of **bison** that specify the rest of the compilation process.
- Designed to work with **flex** or other automatically or hand generated “lexers”

bison USAGE


Bison source program
parser.y
Bison Compiler
y.tab.c
y.tab.c
C Compiler
a.out
Token stream
a.out
Parse tree

bison SPECIFICATION

A **bison** specification is composed of 4 parts

```
%{
   /* C declarations */
%}
   /* Bison declarations */

%%

   /* Grammar rules */

%%

   /* Additional C code */
```

Looks like **flex** specification, doesn't it?
Similar function, tools, look and feel

C DECLARATIONS

- Contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules
- Copied to the beginning of the parser file so that they precede the definition of **`yyparse`**
- Use **`#include`** to get the declarations from a header file. If **C** declarations isn't needed, then the `%{` and `%}` delimiters that bracket this section can be omitted

bison DECLARATIONS

- Contains declarations that define *terminal* and *non-terminal* symbols, and specify precedence

GRAMMAR RULES

- Contains one or more **`bison`** *grammar rules*, and nothing else

- There must always be at least one grammar rule, and the first `%%` (which precedes the grammar rules) may never be omitted even if it is the first thing in the file

Additional C code

- Copied verbatim to the end of the parser file, just as the **C** declarations section is copied to the beginning
- This is the most convenient place to put anything that should be in the parser file but isn't need before the definition of **`yyparse`**
- The definitions of **`yylex`** and **`yyerror`** often go here

bison EXAMPLE

```
%{
#include <ctype.h> /* standard C declarations here */
}%
%token DIGIT /* bison declarations */
%%
/* Grammar rules */
line : expr '\n'          {  printf { "%d\n", $1 };  }     ;
expr : expr '+' term      {  $$ = $1 + $3;  }
     | term                                               ;
term : term '*' factor    {  $$ = $1 * $3;  }
     | factor                                             ;
```

bison EXAMPLE

```
factor : '(' expr ')'     {  $$ = $2;  }
       | DIGIT                                          ;
%%
/* Additional C code */

void yylex () {
   /* A really simple lexical analyzer */
   int c;
   c = getchar ();
   if ( isdigit (c) ) {
      yylval = c - '0' ;
      return DIGIT;
   }
   return c;
}
```

Note: **bison** uses **yylex**, **yylval**, etc - designed to be used with **flex**

bison EXAMPLE

```
expr ::=                                          term
     ::=                                 expr   + term
     ::=                          expr   + term   + term
     ::= term   + ... + term   + term   + term   + term

term ::=                                          factor
     ::=                                 term   * factor
     ::=                          term   * factor * factor
     ::= factor * ... * factor * factor * factor * factor

factor ::= DIGIT
        ::= ( expr )
        ::= ( term + term + ... + term )
        ::= ( factor * ... factor + term + ... term )
        ::= ...

DIGIT ::= [0-9]
```

bison EXAMPLE

```
|'(' '1' '*' '3' '+' '2' ')' '*' '5' '\n'
```

```
line ::= |expr '\n'
```

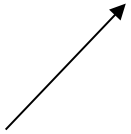
bison EXAMPLE

```
'('|'1' '*' '3' '+' '2' ')' '*' '5' '\n'
```

```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '('|expr ')'
```

bison EXAMPLE

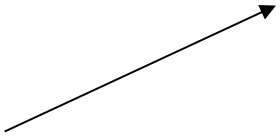
```
'(' '1'|'*' '3' '+' '2' ')' '*' '5' '\n'
```



```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'
  expr ::=  term
  term ::=  factor
factor ::=  DIGIT|
```

bison EXAMPLE

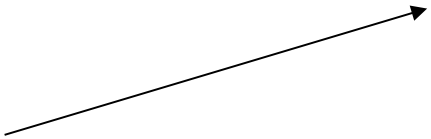
```
'(' '1' '*'|'3' '+' '2' ')' '*' '5' '\n'
```



```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'
  expr ::=  term
  term ::=  term '*'|factor
```

bison EXAMPLE

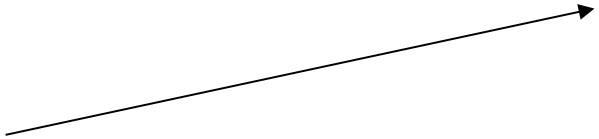
```
'(' '1' '*' '3'|'+' '2' ')' '*' '5' '\n'
```



```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'
  expr ::=  term
  term ::=  term '*' factor
factor ::=  DIGIT|
```

bison EXAMPLE

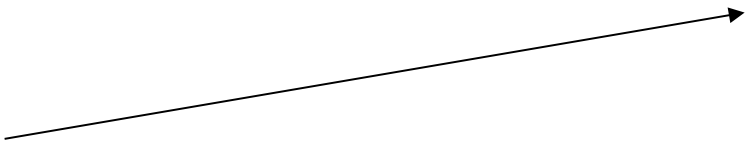
```
'(' '1' '*' '3' '+'|'2' ')' '*' '5' '\n'
```



```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'
  expr ::=  expr '+'|term
  term ::=  term '*' factor
```

bison EXAMPLE

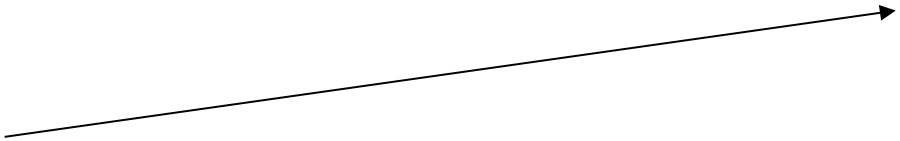
```
'(' '1' '*' '3' '+' '2'|')' '*' '5' '\n'
```



```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'
  expr ::=  expr '+' term
  term ::=  factor
factor ::=  DIGIT|
```

bison EXAMPLE

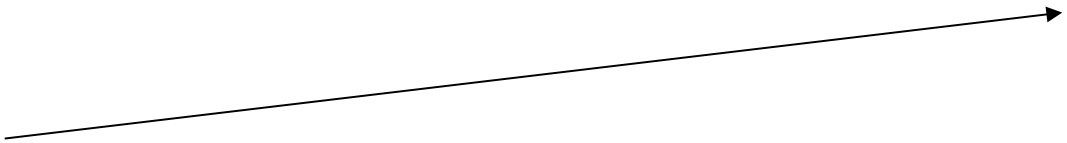
```
'(' '1' '*' '3' '+' '2' ')'|'*' '5' '\n'
```



```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  factor
factor ::=  '(' expr ')'|
  expr ::=  expr '+' term
  term ::=  factor
```

bison EXAMPLE

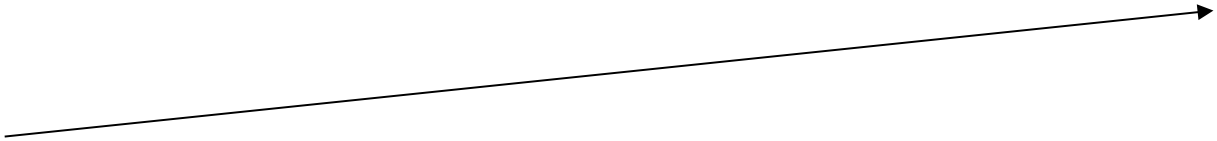
```
'(' '1' '*' '3' '+' '2' ')' '*'|'5' '\n'
```



```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  term '*'|factor
factor ::=  '(' expr ')'
```

bison EXAMPLE

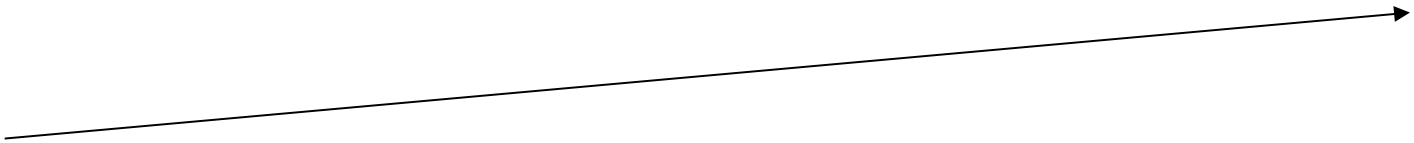
```
'(' '1' '*' '3' '+' '2' ')' '*' '5'|'\n'
```



```
  line ::=  expr '\n'
  expr ::=  term
  term ::=  term '*' factor
factor ::=  DIGIT|
```

bison EXAMPLE

```
'(' '1' '*' '3' '+' '2' ')' '*' '5' '\n'|
```



```
line ::=  expr '\n'|
expr ::=  term
term ::=  term '*' factor
```

USING bison WITH flex

bison and **flex** are obviously designed to work together

bison produces a driver program called **yylex()** (actually its included in the **flex** library **-lfl**)

- **#include "lex.yy.c"** in the third part of **bison** specification
- this gives the program **yylex** access to **bisons'** token names

USING BISON WITH FLEX

- Thus do the following:
  - `% flex scanner.l`
  - `% bison parser.y`
  - `% cc y.tab.c -ly -lfl`
- This will produce an **`a.out`** which is a parser with an integrated scanner included

ERROR HANDLING IN `bison`

Error handling in **`bison`** is provided by error productions

An error production has the general form

```
non-terminal: error synchronizing-set
```

- **`non-terminal`** where did it occur
- **`error`** a keyword
- **`synchronizing-set`** possible empty subset of tokens

When an error occurs, **`bison`** pops symbols off the stack until it finds a state for which there exists an error production which may be applied

FILES TO BE CHANGED

- **`parser.y`** is the input file to **`bison`**. This is the only file you will do most of editing in.
- **`scanner.l`** needs a small, but important change. The file **`scanner.hh`** is no longer needed since there is a file **`parser.hh`**, which will contain (among other things) the same declarations. **`parser.hh`** will be generated automatically by **`bison`**.
- Add (in this order):

```
#include "ast.h"
#include "parser.hh"
```

  and comment out

```
#include "scanner.hh"
```

  at the top of **`scanner.l`** to reflect this.

Other files of interest

- **`error.h`, `error.cc`, `symtab.hh`, `symbol.cc`, `symtab.cc`** Use your completed versions from the earlier labs.
- **`ast.hh`** contains the definitions for the AST nodes. You'll be reading this file a lot.
- **`ast.cc`** contains the implementations of the AST nodes.

- **`semantic.hh`** and **`semantic.cc`** contain type checking code.
- **`optimize.hh`** and **`optimize.cc`** contain optimization code.
- **`quads.hh`** and **`quads.cc`** contain quad generation code.
- **`codegen.hh`** and **`codegen.cc`** contain assembler generation code.

Other files of interest

- **`main.cc`** this is the compiler wrapper, parsing flags and the like.
- **`Makefile`** this is not the same as the last labs. It generates a file called **`compiler`** which will take various arguments (see **`main.cc`** for information). It also takes source files as arguments, so you can start using **`diesel`** files to test your compiler-in-the-making.
- **`diesel`** this is a shell script which works as a wrapper around the binary compiler file, handling flags, linking, and such things. Use it when you want to compile a **`diesel`** file. At the top of this file is a list of all flags you can send to the compiler, for debugging, printouts, symbolic compilation and the like.

LAB 4
SEMANTICS

PURPOSE

To verify the semantic correctness of the program represented by the parse tree, reporting any errors, possibly, to produce an intermediate form and certain tables for use by later compiler phases

- Semantic correctness the program adheres to the rules of the type system defined for the language (plus some other rules)
- Error messages should be as meaningful as possible
- In this phase, there is sufficient information to be able to generate a number of tables of semantic information **`identifier, type`** and **`literal tables`**

Ad-hoc confirmation of semantic rules

IMPLEMENTATION

- Semantic analyzer implementations are typically syntax directed
- More formally, such techniques are based on attribute grammars
- In practice, the evaluation of the attributes is done manually

MATHEMATICAL CHECKS

Divide by zero

Zero must be compile-time determinable constant zero, or an expression which symbolically evaluates to zero at runtime

Overflow

Constant which exceeds representation of target machine language arithmetic which obviously leads to overflow

Underflow

Same as for overflow

UNIQUENESS TESTS

In certain situations it is important that particular constructs occur only once

Declarations

within any given scope, each identifier must be declared only once

Case statements

each case constant must occur only once in the “switch”

TYPE CONSISTENCY

Some times it is also necessary to ensure that a symbol that occurs in one place occurs in others as well.

Such consistency checks are required whenever matching is required and what must be matched is not specified explicitly (i.e as a terminal string) in the grammar

This means that the check cannot be done by the parser

TYPE CHECKS

These checks form the bulk of semantic checking and certainly account for the majority of the overhead of this phase of compilation

In general the types across any given **operator** must be **compatible**

The meaning of ***compatible*** may be:

- *the same*
- *two different sizes of the same basic type*

TYPE CHECKS

Must execute the same steps as for expression evaluation

Effectively we are ”executing” the expression at compile time for type information only

This is a bottom-up procedure in the parse tree

We know

- the type of ”things” at the leaves of a parse tree corresponding to an expression
- (associated types stored in literal table for literals and symbol table for identifiers)

When we encounter a parse tree node corresponding to some operator if the operand sub-trees are leaves we know their type and can check that the types are valid for the given operator.

FILES TO BE CHANGED

- **`semantic.hh`** and **`semantic.cc`** contains type checking code implementation for the AST nodes as well as the declaration and implementation of the semantic class. These are the files you're going to edit in this lab. They deal with type checking, type synthesizing, and parameter checking.

Other files of interest

All these files are the same as in **lab 3**:

- **`parser.y`** is the input file to **`bison`**. This is the file you edited in the last lab, and all you should need to do now is uncomment a couple of calls to:

  `do_typecheck()`.
- **`ast.hh`** contains the definitions for the AST nodes.
- **`ast.cc`** contains (part of) the implementations of the AST nodes.

- **`optimize.hh`** and **`optimize.cc`** contains optimizing code.
- **`quads.hh`** and **`quads.cc`** contains quad generation code.
- **`codegen.hh`** and **`codegen.cc`** contains assembler generation code.

Other files of interest

- **`error.hh`**, **`error.cc`**, **`symtab.hh`**, **`symbol.cc`**, **`symtab.cc`**, **`scanner.l`** use your versions from the earlier labs.
- **`main.cc`** this is the compiler wrapper, parsing flags and the like.
- **`Makefile`** and **`diesel`** use the same files as in the last lab.