TDD55: Compilers and Interpreters

Lesson 3

Filip Strömbäck (filip.stromback@liu.se)
Department of Computer and Information Science
Linköping University

Schedule

1. Formal languages and automata theory
2. Formal languages and automata theory, Flex
3. Bison and intermediate code generation
4. Exam preparation
5. Exam preparation

Laboratory assignments

- Goal: Get some practical experience in compiler construction
- 4 assignments to complete in 4x2 sessions → non-scheduled time required
  1. Attribute Grammars and Top-Down parsing
  2. Scanner Specification
  3. Parser Generators
  4. Intermediate Code Generation

Handing in and deadline

- Demonstrate during scheduled sessions
- Then, hand in code and answers to any questions in the assignment via e-mail. One e-mail per group, subject: TDDD55: Lab n. From your LiU-email.
- Deadline: 21^{st} December

Skeleton

Copy to your home directory using:

```
cp -r ~TDDD55/lab .
```

- ~TDDD55
  - /lab
    - /doc
      - Documentation for the assignments.
    - /lab1
      - Contains all the necessary files to complete the first assignment
    - /lab2
      - Contains all the necessary files to complete the second assignment
    - /lab3-4
      - Contains all the necessary files to complete assignment three and four

Lab 1

- Some grammar rules given:
- Rewrite the grammar to LL(1)
- Add attribute rules to the grammar
- Implement the LL(1) grammar and the attributes in a C++-class named Parser. Parser shall contain a method Parse() which returns the value of a single statement in the language.

Lab 2

- Finish a scanner specification in Flex (scanner.l) by adding rules for comments, identifiers, integers and reals.

Lab 3

- Finish a parser specification in Bison (parser.y) by adding rules for expressions, conditions, function definitions, etc.
- You also need to add error productions.
- More details in lesson 3.

Lab 4

- Generate intermediate code from a parse tree.
- Finish a generator for intermediate code by adding rules for some language statements.
- More details in lesson 3.

Bison

Purpose of a parser

- Accept tokens from a scanner and verify the syntactic correctness of the program
  - Uses formal grammar for specifying the language
- Along the way, it derives information about the program and stores it as an abstract syntax tree (AST)
- The AST is an internal representation of the program and augments the symbol table

Bottom-up Parsing

- Recognize the components of a program and then combine them to form more complex constructs until a whole program is recognized
- The parse tree is then built from the bottom up, hence the name.

LR Parsing

- A specific bottom-up technique
- LR = left-to-right scan, reversed rightmost derivation
- Probably the most common and popular technique
- yacc, bison and many other parser generation tools utilize LR parsing
- Great for machines, harder for humans

Pros and Cons of LR Parsing

- Advantages
  - Accepts a wide range of grammars/languages
  - Well suited for automatic parser generation
  - Very fast
  - Generally easy to maintain
- Disadvantages
  - Error handling can be tricky
  - Difficult to use manually

Bison

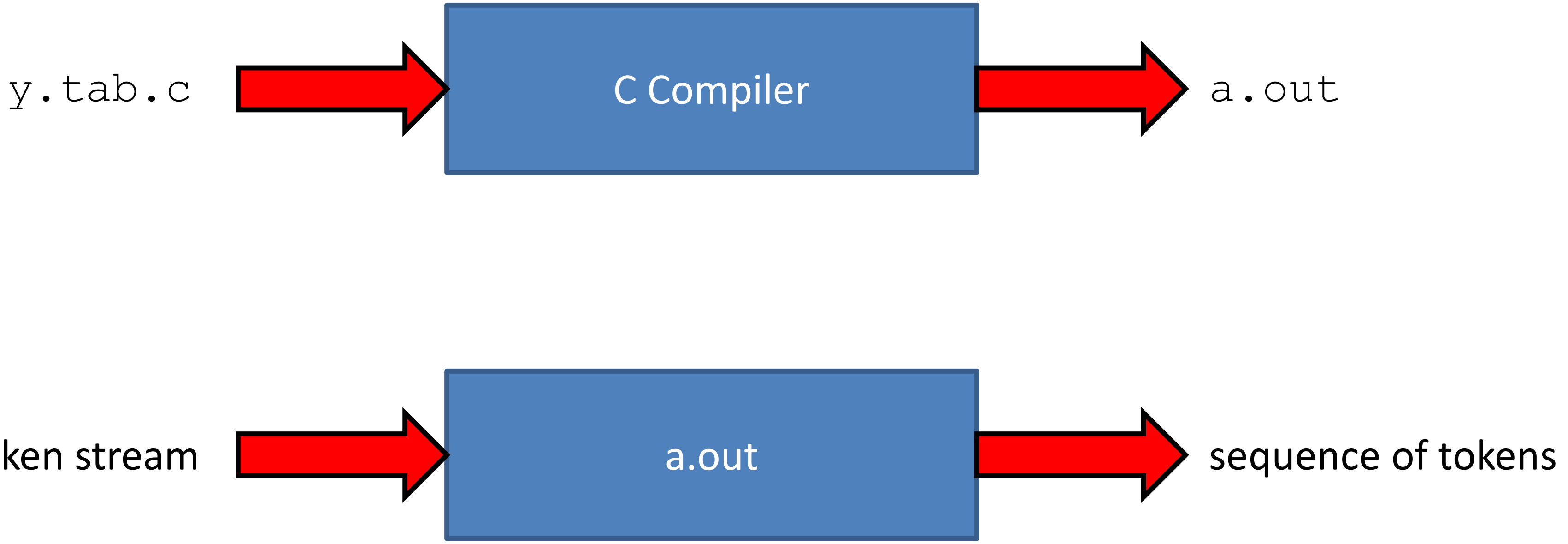
- Bison is a general-purpose parser generator that converts a grammar description of a context-free grammar into a C-program to parse the grammar
- Similar idea to flex

How it works

Bison generates a C source file, y.tab.c

How it works

y.tab.c is compiled and linked with required libraries to produce an executable, which is the parser



Bison specifications

Bison specifications are divided into 4 parts

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

  /* Grammar rules */

%%

  /* Additional C code */
```

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 aren’t needed, then the %{ and %} can be omitted

Bison declarations

- Contains:
  - Declarations that define terminal and non-terminal symbols
  - Data types of semantic values of various symbols
  - Specify precedence

Bison specifications

Bison specifications are divided into 4 parts

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

  /* Grammar rules */

%%

  /* Additional C code */
```

Grammar rules

- Contains one or more Bison grammar rules, and nothing else.
- Example:
  - expression : expression ‘+’ expression { $$ = $1 + $3; };
- 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

Bison specifications

Bison specifications are divided into 4 parts

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

  /* Grammar rules */

%%

  /* Additional C code */
```

Additional C code

- Copied verbatim to the end of the parser file, just as the C declarations are copied to the beginning
- This is the most convenient place to put anything that should be in the parser file but isn’t needed before the definition of yyparse
- For example, yylex() and yyerror() often go here

Example 1

```
%{
#include <ctype.h>
#define YYSTYPE double
int yylex();
%}
%token DIGIT
%%
line : expr '\n'      { printf("%d\n", $1); };
expr : expr '+' term { $$ = $1 + $3; }
     | term           { $$ = $1; };
term : term '*' fact { $$ = $1 * $3; }
     | fact           { $$ = $1; };
fact : '(' expr ')'  { $$ = $2; }
     | DIGIT;
```

Example 1 (cont)

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

Example 2

```
thing: A { printf(“seen an A”); } B;
```

Same as

```
thing: A fakename B;
fakename: /* empty */ { printf(“seen an A”); };
```

Example 3

```
%{
#define YYSTYPE double
#include <math.h>
%}
%token NUM
%left '-' '+'
%left '*' '/'
%right '^'
```

Example 3 (cont)

```
%%
input : /* empty string */ | input line;
line  : '\n'
      | expr '\n'     { printf("\t%.10f\n", $1); };
expr  : NUM           { $$ = $1; }
      | expr '+' expr { $$ = $1 + $3; }
      | expr '-' expr { $$ = $1 - $3; }
      | expr '*' expr { $$ = $1 * $3; }
      | expr '/' expr { $$ = $1 / $3; }
      | expr '^' expr { $$ = pow($1, $3); }
      | '(' expr ')'  { $$ = $1; };
%%
```

Syntax Errors

- Error productions can be added
- They help the compiler to recover from syntax errors and to continue to parse
- In order for the error productions to work, we need at least one valid token after the error symbol.
- Example
  - ```
    functionCall : ID '(' paramList ')'
                 | ID '(' error ')';
    ```
- Recover from syntax errors by discarding tokens until it reaches the valid token.
    ```

Using Bison with Flex

- Bison and flex are designed to work together
- Flex produces a driver program called yylex()
  - #include "lex.yy.c" in the last part of bison specification
  - this gives the program yylex access to bison's token names

Using Bison with Flex

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

Assignment 3

Parser generators

Parser Generators

- Finish a parser specification given in *parser.y* by adding rules for expressions, conditions and function definitions, …

Functions

```
function : funcnamedecl parameters ':' type variables functions block ';'
{
  // Set the return type of the function
  // Set the function body
  // Set current function to point to the parent again
}

funcnamedecl : FUNCTION id
{
  // Check if the function is already defined, report error if so
  // Create a new function information and set its parent to current
function
  // Link the newly created function information to the current function
  // Set the new function information to be the current function
}
```

Expressions and conditions

- For precedence and associativity you can factorize the rules…
- or specify precedence and associativity at the top of the Bison specification file. Read more about this in the Bison reference.

Expressions - Example

```
expression : expression '+' term
{
  // If any of the sub-expressions are NULL,
  // set $$ to NULL
  // Create a new Plus node and return in $$
  // IntegerToReal casting might be needed
}
```

Assignment 4

Intermediate code

Intermediate Code

- Closer to machine code, but not machine specific
- Can handle temporary variables
- Means higher portability: intermediate code can easier be expanded to assembly code
- Offers the possibility of performing code optimizations such as register allocation

Intermediate Code

- Why do we use intermediate languages?
- Retargeting – Build a compiler for a new machine by attaching a new code generator to an existing front-end and middle part.
- Optimization – reuse intermediate code optimizers in compilers for different languages and different machines
- Code generation for different source languages can be combined

Intermediate Languages

- Infix notation
- Postfix notation
- Three-address code
  - Triples
  - Quadruples

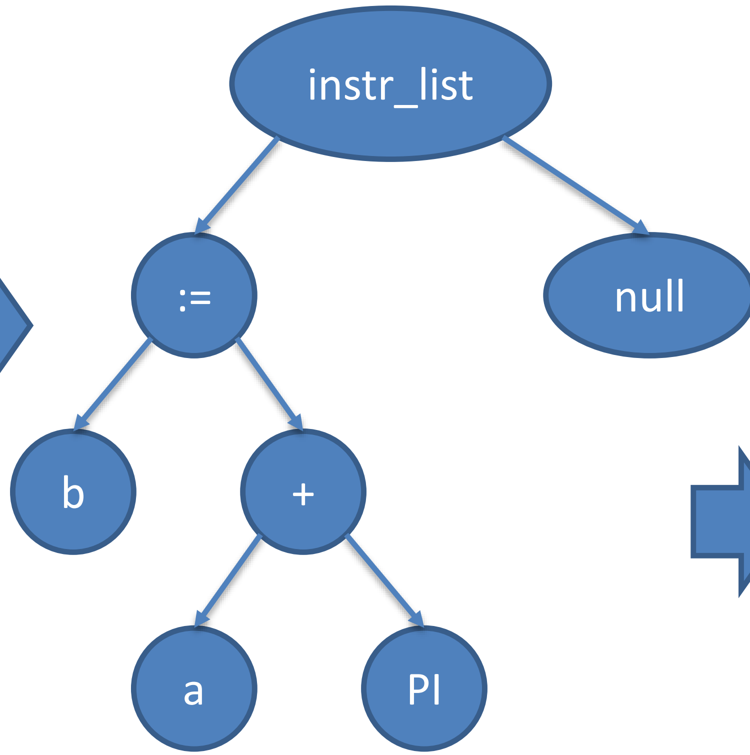
Quadruples

- We use quadruples as an intermediate language.
- An instruction has four fields:

operator	operand1	operand2	result

Generation

```
program ex;
const
   PI = 3.1415;
var
   a : real;
   b : real;
begin
   b := a + PI;
end
```



q_rplus	A	PI	$1
q_rassign	$1	-	B
q_labl	4	-	-

Quadruples

$$\boxed{(A+B)*(C+D)-E}$$

operator	operand1	operand2	result
+	A	B	T1
+	C	D	T2
*	T1	T2	T3
-	T3	E	T4

Intermediate Code Generation

- The purpose of this assignment is to learn how abstract syntax trees can be translated into machine code.
- You are to finis a generator for intermediate code (quadruples) by adding rules for some language constructs.
- You will work in *codegen.cc.*

Binary Operations

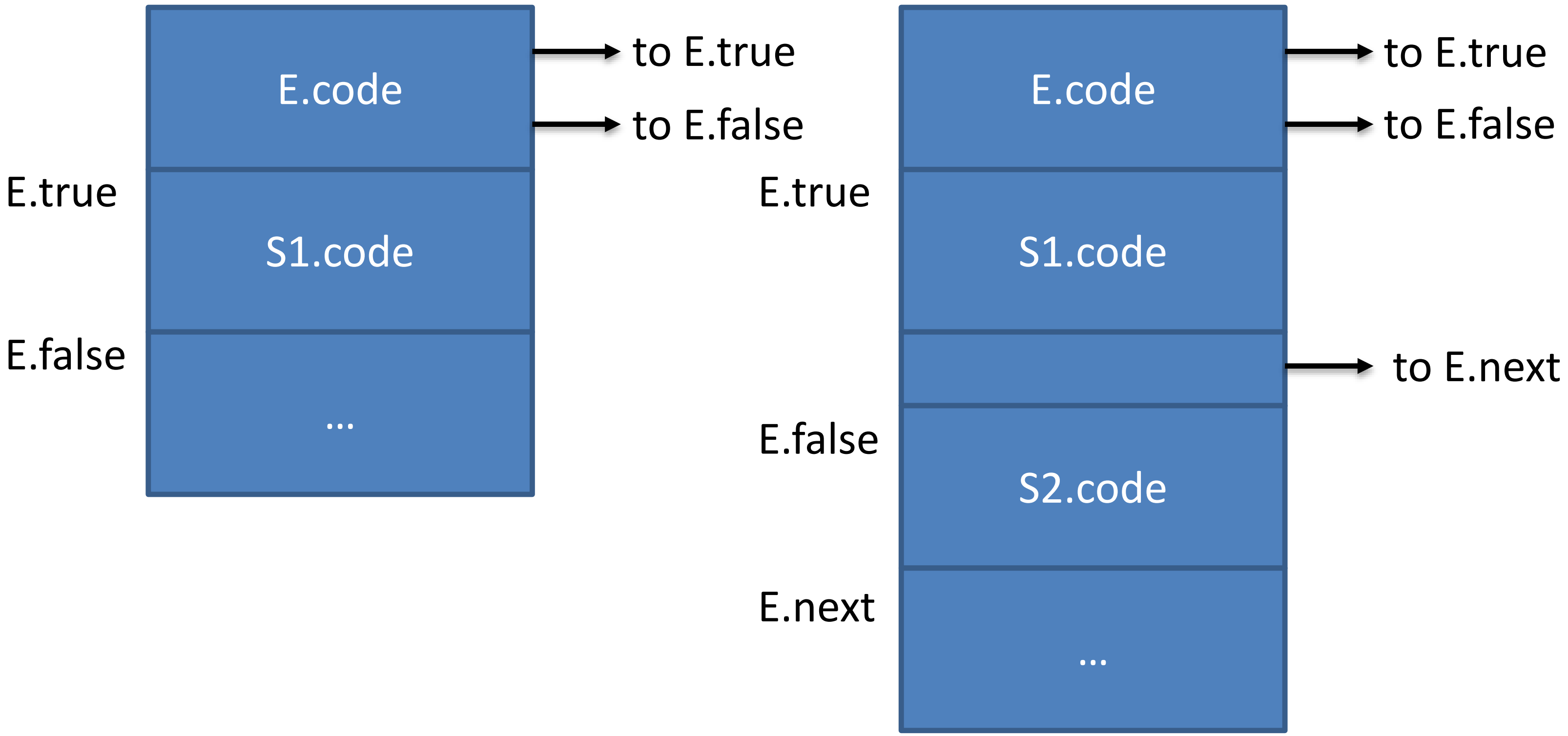
- Create code for left expression and right expression
- Generate either a realop or intop quad
  - Type of the result is the same as the type of the operands
  - You can use currentFunction->TemporaryVariable

Array References

- The absolute address is computed as follows:
  - absAddr = baseAddr + arrayTypeSize*index
- Generate code for the index expression
- You must then compute the absolute address
  - You will have to create several temporary variables
  - Create a quad for loading the size of the type to a temporary
  - Then generate iadd and imul quads
  - Finally generate either a istore or rstore quad

If Statement

- S -> if E then S1
- S -> if E then S1 else S2



While Statement

- S -> while E do S1