COMPILERS AND INTERPRETERS Lesson 4 – TDDD16

Kristian Stavåker (kristian.stavaker@liu.se)

Department of Computer and Information Science

Linköping University

TODAY

- Introduction to the Bison parser generator tool
- Introduction to quadruples and intermediate code generation
- Hints to laboratory assignments 3 and 4

NEXT LESSON

- December14th, 15.15-17.00
- Exam preparation
- Work mostly on your own (exercises from old exams, …)
- Some example solutions
- Opportunity to ask questions

LABORATORY ASSIGNMENTS

In the laboratory exercises you should get some practical experience in compiler construction.

There are 4 separate assignments to complete in 6x2 laboratory hours. You will also (most likely) have to work during non-scheduled time.

LABORATORY ASSIGNMENTS

Lab 3 Parser Generators

Generate a parser for a Pascal-like language using the Bison parser generator

Lab 4 Intermediate Code Generation

Generate intermediate (quadruple) code from the abstract syntax tree(s)

HANDING IN AND DEADLINE

- Demonstrate the working solutions to your lab assistant during scheduled time. Hand in your solutions to theory questions on paper. Then send the modified code files to the assistant (put *TDDD16 <Name of the assignment>* in the topic field). One e-mail per group.

- Deadline for all the assignments is: **December 15, 2010** You will get 3 extra points on the final exam if you finish on time! But the '**extra credit work**' assignments in the laboratory instructions will give no extra credits this year.

BISON – PARSER GENERATOR

PURPOSE OF A PARSER

- The parser accepts tokens from the scanner and verifies the syntactic correctness of the program.
  - Syntactic correctness is judged by verification against a formal grammar which specifies the language to be recognized.
- Along the way, it also derives information about the program and builds a fundamental data structure known as parse tree or abstract syntax tree (ast).
- The abstract syntax tree is an internal representation of the program and augments the symbol table.

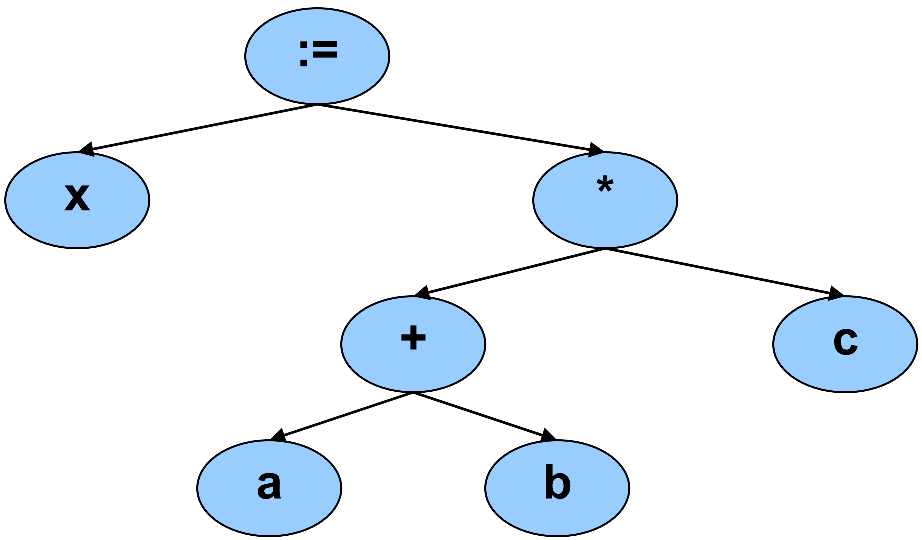
BOTTOM-UP PARSING

- 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 and up, hence the name.

BOTTOM-UP PARSING (2)

X := (a + b) * c;



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 great for humans …

PROS AND CONS LR PARSING

- 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

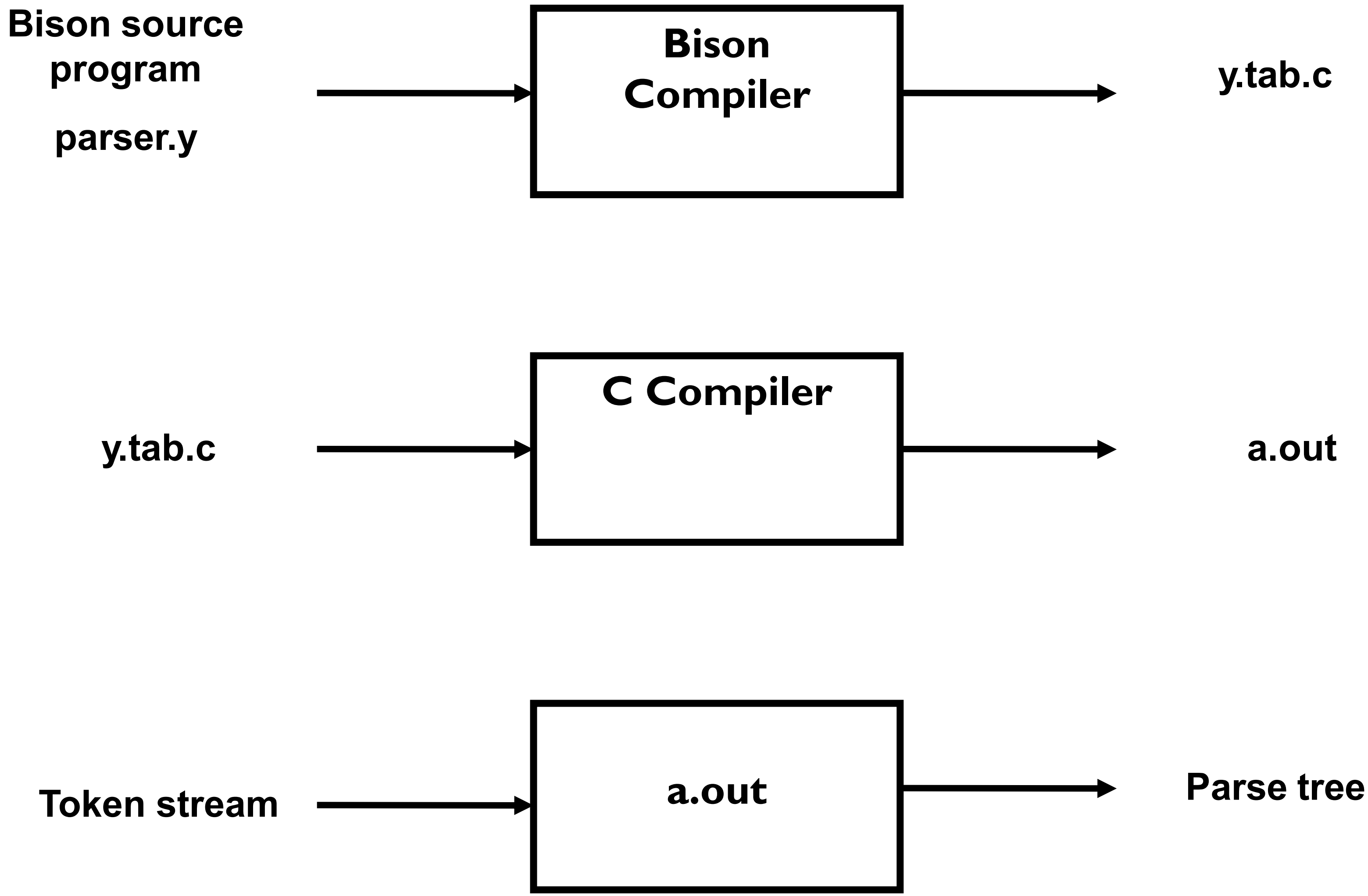
BISON

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

BISON (2)

- Input: a specification file containing mainly the grammar definition
- Output: a C source file containing the parser
- The entry point is the function int yyparse();
  - yyparse reads tokens by calling yylex and parses until
    - end of file to be parsed, or
    - unrecoverable syntax error occurs
  - returns 0 for success and 1 for failure

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 FILE

- A Bison specification is composed of 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 isn't needed, then the %{ and %} delimiters that bracket this section can be omitted

BISON DECLERATIONS

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

GRAMMAR RULES

- Contains one or more Bison grammar rule, and nothing else.
- Example:
  - expression : expression '+' term { $$ = $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.

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 needed before the definition of yyparse.
- The definitions of yylex and yyerror often go here.

SYNTAX ERRORS

- Error productions can be added to the specification
- 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 1:
  - functionCall : ID '(' paramList ')'
    | ID '(' error ')'

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

USING BISON WITH FLEX (2)

- 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 integrated scanner included

BISON EXAMPLE 1 (1/2)

```
%{
#include <ctype.h> /* standard C declarations here */
// extern int yylex();
}%
%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 1 (2/2)

```
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;
}
```

BISON EXAMPLE 2 – MID-RULES

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

The same as:

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

BISON EXAMPLE 3 (1/2)

```
/* Infix notation calculator--calc */

%{
#define YYSTYPE double
#include <math.h>
%}

/* BISON Declarations */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation        */

/* Grammar follows */
%%
```

BISON EXAMPLE 3 (2/2)

```
input:    /* empty string */
        | input line
;
line:     '\n'
        | exp '\n'  { printf ("\t%.10g\n", $1); }
;
exp:      NUM                { $$ = $1;         }
        | exp '+' exp        { $$ = $1 + $3;    }
        | exp '-' exp        { $$ = $1 - $3;    }
        | exp '*' exp        { $$ = $1 * $3;    }
        | exp '/' exp        { $$ = $1 / $3;    }
        | '-' exp  %prec NEG { $$ = -$2;        }
        | exp '^' exp        { $$ = pow ($1, $3); }
        | '(' exp ')'        { $$ = $2;         }
;
%%
```

INTERMEDIATE CODE GENERATION

INTERMEDIATE LANGUAGE

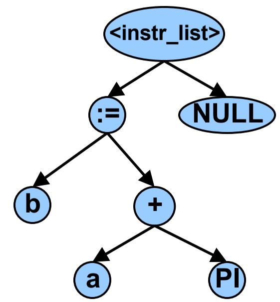
- Is closer to machine code without being machine dependent.
- Can handle temporary variables.
- Means higher portability, intermediate code can easier be expanded to assembly code.
- Offers the possibility of performing code optimizations such as register allocation.

INTERMEDIATE LANGUAGE (2)

- Why 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

GENERATION OF INTERMEDIATE CODE

```
program example;
const
  PI = 3.14159;
var
  a : real;
  b : real;
begin
  b := a + PI;
end.
```




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

INTERMEDIATE LANGUAGES

- Various types of intermediate code are:
  - Infix notation
  - Postfix notation
  - Three address code
    - Triples
    - Quadruples

QUADRUPLES

- You will use quadruples as intermediate language where an instruction has four fields:

operator	operand1	operand2	result

QUADRUPLES

(A + B) * (C + D) - E

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

HINTS LABORATORY ASSIGNMENT 3

PARSER GENERATORS

- Finnish a parser specification given in a *parser.y* bison file, by adding rules for expressions, conditions and function definitions, ...

FUNCTIONS

- Outline:

```
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 current function
}  ;
```

EXPRESSIONS

- For precedence and associativity you can factorize the rules for expressions …

  or

- you can specify precedence and associativy at the top of the Bison specification file, in the *Bison Declarations* section. Read more about this in the Bison reference(s).

EXPRESSIONS (2)

- Example with factoring:

```
expression : expression '+' term
{
 // If any of the sub-expressions is NULL, set $$ to NULL
// Create a new Plus node but IntegerToReal casting might be needed
}
|
...
```

CONDITIONS

- For precedence and associativity you can factorize the rules for conditions …

  or

- you can specify precedence and associativy at the top of the Bison specification file, in the *Bison Declarations* section. Read more about this in the Bison reference(s).

HINTS LABORATORY ASSIGNMENT 4

INTERMEDIATE CODE GENERATION

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

BINARY OPERATIONS

- In *BinaryGenerateCode*:
  - Generate code for left expression and right expression.
  - Generate either a *realop* or *intop* quadruple
    - For relations the type of the result is always integer
    - Otherwise the type of the result is the same as the type of the operands
    - You can use *currentFunction->TemporaryVariable*

ARRAY REFERENCES

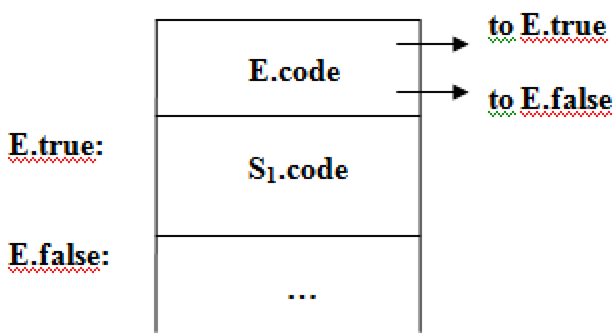
- The absolute address is computed as follows:
  - $absAdr = baseAdr + arrayTypeSize * index$

ARRAY REFERENCES (2)

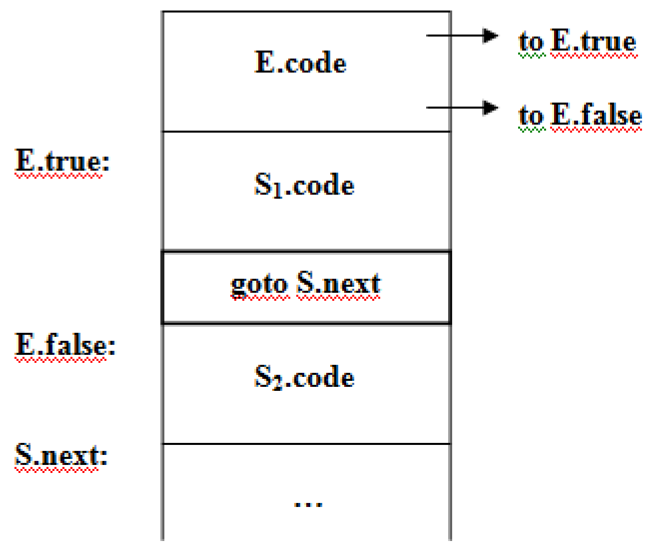
- Generate code for the index expression
- You must then compute the absolute memory address
  - You will have to create several temporary variables (of integer type) for intermediate storage
  - Generate a quadruple *iaddr* with *id* variable as input for getting the base address
  - Create a quadruple for loading the size of the type in question to a temporary variable
  - Then generate *imul* and *iadd* quadruples
  - Finally generate either a *istore* or *rstore* quadruple

IF STATEMENTS

- **S → if E then S_1**
- **S → if E then S_1 else S_2**



if - then



if – then - else

WHILE STATEMENT

- **S → while E do S1**