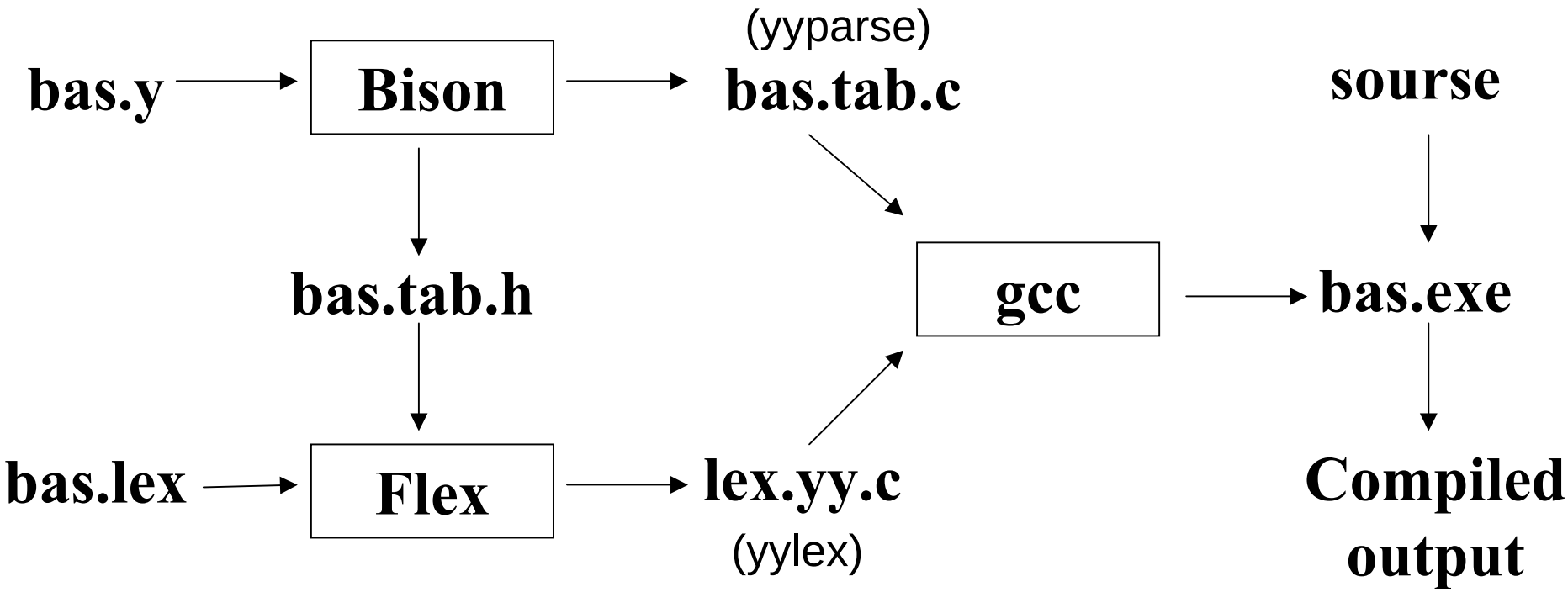
# Constructing a compiler using Flex and Bison

## lesson 3

MSc Anna Beletska, beletska@elet.polimi.it

# Building a Compiler




```
bison -d bas.y          # create bas.tab.h, bas.tab.c
flex bas.lex            # create lex.yy.c
gcc lex.yy.c bas.tab.c -obas.exe    # compile / link
```

# Building a Compiler

```
bison -d bas.y          # create bas.tab.h, bas.tab.c
flex bas.lex            # create lex.yy.c
gcc lex.yy.c bas.tab.c -obas.exe    # compile / link
```

- Bison reads the grammar description in **bas.y** and generates a parser, function **yyparse**, in file **bas.tab.c**. Included in file **bas.y** are token declarations. The **-d** option causes bison to generate definitions for tokens and place them in file **bas.tab.h**.
- Flex reads the pattern descriprions in **bas.lex**, includes file **bas.tab.h**, and generates a lexical analyzer, function **yylex**, in file **lex.yy.c**.
- Finally, the lexer and the parser are compiled and linked together to form the executable, **bas.exe**. From **main()**, we call **yyparse** to run the compiler. Function **yyparse** automatically calls **yylex** to obtain token.

# More Flex

- Function **yylex** has a return type of int, and returns the token value.
- Values associated with the token are returned by Flex in variable **yylval**.

```
[0-9]+          {
                        yylval = atoi (yytext);
                        return INTEGER;
```

- The type of **yylval** is determined by **YYSTYPE**. The default type is integer. Token values 0-255 are reserved for character values.

```
[-+]            return *yytext;             /* return operator */
```

returns the character value for minus or plus. Note that we placed the minus sign first so that it wouldn't be mistaken for a range designator.

# More Bison

The grammar is automatically checked by Bison to assure that:

- There are rules (productions) for all non-terminals.
- All non-terminals must be reachable from the axiom.
- The grammar is not ambiguous.
- The grammar is LALR(1) parsable.

# Recursion

**When specifying a list, we may do so using**

**Left recursion:**

```
list:
        item
        |   list  ‘,’  item
        ;
```

**Right recursion:**

```
list:
        item
        |   item ‘,’ list
        ;
```

**If right recursion is used, all the items on the list are pushed on the stack. After the last item is pushed, we start reducing.**

**With left recursion, we never have more than three terms on the stack, since we reduce as we go along. For this reason, it is advantageous to use left recursion.**

# Value stack type

- **Along with the parsing state stack, Bison maintains a parallel stack of attribute value associated with each token and non-terminal in the parsing stack.**
- **The type of an attribute is defined as YYSTYPE, which is int by default, but can be redefined within the specification in two ways:**

1.
```
typedef struct att_type {
    char *sval;
    int ival;
};
#define YYSTYPE att_type
```

2.
```
%union {
    char *sval;
    int ival;
}
```

# Grammar ambiguity

**If the grammar is ambiguos, it can be disambiguated using disambiguating rules.**

# If-Else Ambiguity

**A shift-reduce conflict that frequently occurs involves the if-else construct. Assume we have the following rules:**

**stmt:**

**IF expr stmt**
**| IF expr stmt ELSE stmt**
**...**

**And the following state:**

**IF expr stmt IF expr stmt ELSE stmt**

**We need to decide if we should shift the ELSE, or reduce the „IF expr stmt" at the top of the stack.**

**The modern programming languages pair an ELSE with the most recent unpaired IF: for the shift-reduce conflicts Bison shifts.**

# If-Else Ambiguity

**Although Bison does the right thing, it also issues a shift-reduce warning message. To remove the message, give IF-ELSE a higher precedence that the simple IF statement:**

**%nonassoc IFX**
**%nonassoc ELSE**

**Stmt:**

**IF expr stmt %prec IFX**
**| IF expr stmt ELSE stmt**

The altered rule precedence affects how conflicts involving that rule are resolved

**(context dependent precedence)**

# What Bison cannot parse?

**There are unambiguos grammars that Bison cannot parse. This is due to the limitations of the LALR(1) technique. For example, the following unambiguos grammar is not LALR(1)-parsable:**

| | | |
|---|---|---|
| **S** | **→** | **A a b** |
| | **\|** | **B a c** |
| **A** | **→** | **d \| e** |
| **B** | **→** | **d \| f** |

**A LALR(1) parser fails to parse the sentense *dab*, for two lookahead symbols are needed to select the right production for reduction. Indeed, after shifting *d*, the lookahead token is *a*. The parser cannot tell whether to reduce *d* by A → d or by B → d since *a* belongs to both FOLLOW(A) and FOLLOW(B), which amounts to reduce/reduce conflict.**

# Error Control

**Error control consists of three related activities:**

- Detection. In Bison,  detection is accomplished when given the current parsing state in top of the stack and the next input token, no shift or reduce action is possible. Empty entries are taken as error entries.

- Reporting. In Bison, reporting is done via calls to yyerror(). It will print the message *„parse error”* by default.

- Recovery.

# Error Messages

**A nice compiler gives the user meaningful error message. For example, not much information is conveyed by the following message:**

**syntax error**

**If we track the line number in Flex, then we can at least give the user a line number:**

```
void yyerror (char *s) {
        fprintf (stderr, „line %d: %s\n”, yylineno, s);
}
```

# Error Messages

**When Bison discovers a parsing error, default action is to call yyerror, and then return from yylex with a return value of one. A more graceful action flushes the input stream to a statement delimiter, and continues to scan:**

```
stmt:
        ';'
        | expr ';'
        | PRINT expr ';'
        | VARIABLE '=' expr ';'
        | WHILE '(' expr ')' stmt
        | IF '(' expr ')' stmt %prec IFX
        | IF '(' expr ')' stmt ELSE stmt
        | '{' stmt_list '}'
        | error ';'
        | error '}'
        ;
```

The error token is a special feature of Bison that will match all input until the tokem following error is found.

For this example, when Bison detects an error in a statement it will call **yyerror**, flush input up to the next semicolon or brace, and resume scanning.

# Inherited attributes

At any point in a syntax tree we can determine the attributes of a node based on the attributes of its children. Consider the rule:

**expr: expr ‘+’ expr { $$ = $1 + $3; }**

Since we are parsing bottom-up, the values of both operands are available, and we can determine the value associated with the left-hand side. An inherited attribute of a node depends on the value of a parent or sibling node. The following grammar defines a C variable declaration:

```
decl:     type varlist
type:     INT | FLOAT
varlist:
          VAR                 { setType ($1, $0); }
          | varlist ‘,’ VAR   { setType ($3, $0); }
```

# Inherited attributes

**decl:** **type varlist**
**type:** **INT | FLOAT**
**varlist:**

| | |
|---|---|
| **VAR** | **{ setType ($1, $0); }** |
| **\| varlist ‘,’ VAR** | **{ setType ($3, $0); }** |

And here is a sample parse:

. INT VAR
INT . VAR
type . VAR
type VAR .
type varlist .
decl .

When we reduce **VAR** to **varlist**, we should annotate the symbol table with the type of the variable. However, the **type** is buried in the stack. This problem is resolved by indexing back into the stack. Recall that **$1** designated the first term on the right-hand side. We can index backwards, using **$0, $-1**, and so on. In this case, **$0** will do just fine.

# Embedded Actions

Rules in Bison can contain embedded actions:

```
list:     item1 { do_item1 ($1); }   item2 { do_item2 ($3); }   item3
```

Note that the actions take a slot in the stack, so **do_item2** must use **$3** to reference **item2**. Actually, this grammar is transformed by Bison into the following:

```
list:                item1   _rule01   item2   _rule02   item3
_rule01:             { do_item1 ($0); }
_rule02:             { do_item2 ($0); }
```

# Bison’s variables and functions

| | |
|---|---|
| yyerror() | This function *must be supplied by the user*. It calls yyparse() on a parse error. Used for error reporting. |
| yyparse() | The parser function. It returns an integer value: zero if there is success or non-zero if unable to parse the token sentence provided by yylex(). |
| yylval | Global variable of type YYSTYPE that contains the value pushed on a *shift* – i.e., a token attribute value, so it must be set by yylex(), usually to a copy of yytext (yytext is modified each time a token pattern matches the input) |

# Bison’s variables and functions

| | |
|---|---|
| yyval | Global variable of type YYSTYPE that contains the value pushed on a *goto* – i.e., the value $$. |
| yynerrs | Global int variable that holds the number of errors found. It is automatically incremented by yyparse() on each call to yyerror(). |
| yyerrok; | It is an action that instructs the parser to continue in normal mode, as if recovered from an error. |

# Bison’s variables and functions

YYRECOVERING()

If the user calls his own error-reporting routine whose name is other than yyerror(), such routine should use the Bison macro YYRECOVERING() to test if the parser is trying to resynchronize, in which case no error messages should be printed:

```
warning (char *msg)
{
        if (YYRECOVERING()) return;
        ...
        /* print error */
}
```

# Bison’s variables and functions

yydebug — Global int variable that should be set to 1 if YYDEBUG has been defined and we want to output trace information. It should be declared as extern in main() and set to the proper value there.

```
main()
{
        #ifdef  YYDEBUG
        extern int yydebug;
        #endif
        ...
        yydebug = 1;
        ...
}
```