

# CpSc 829 Advanced Compiler Topics

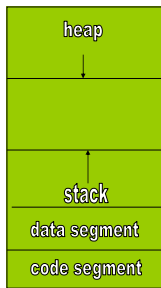
Lex and Yacc

## Tools

- lex (flex): a lexical analysis tool
- yacc (bison): parsing tool
- Btyacc, GLR



## Compiler managed memory:



## The stack of activation records (AR)



```
void fun() {
}

void foo() {
  fun();
}

main() {
  foo();
}
```

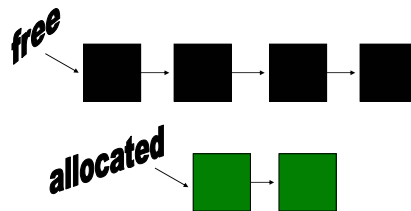
## Activation record

- Pushed onto the stack on entry to function
- popped from stack when function terminates
- storage for:
  - saved registers (pc, psw)
  - parameters
  - local variables

*stack based variables  
are called automatic*

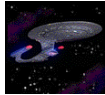
## Heap: free list for dynamic memory (pointer variables)

Usually implemented as a linked list of free nodes, where each node is a block of memory:



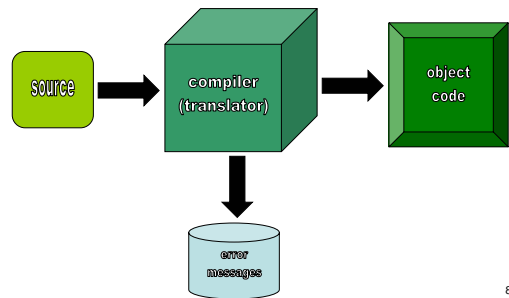
## What is a compiler?

- **Translator** from *source code* to *object code*.
- *object code* is called *target code* because it is specific to some target machine.
- compilers usually provide error and diagnostic information about the source code



7

## Compiler:



8

## The phases of compilation

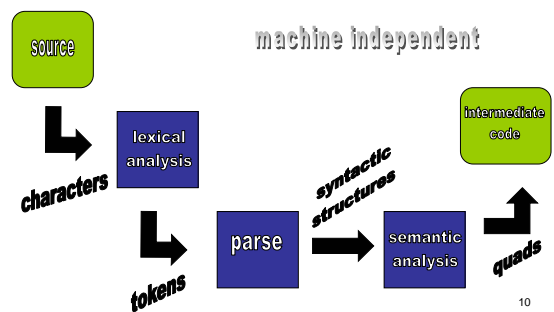
- Lexical analysis
- syntax analysis
- semantic analysis
- optimization
- code generation

The phases are easier to understand if you partition them into front end & back end.



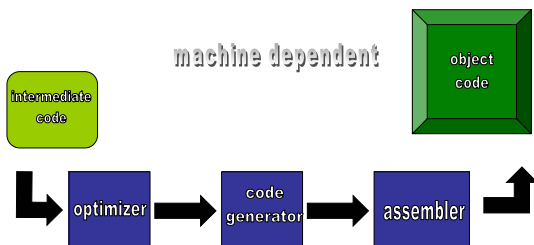
9

## compiler phases: front end



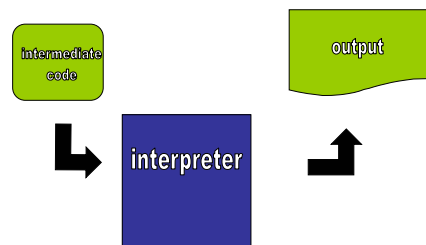
10

## Compiler phases: back end



11

## Interpreter



12

## Formal grammars facilitate construction of the first 2 phases:

- lexical analysis: analyze the source characters
- parsing: analyze tokens
- formal grammars define the structure and syntax for these two phases



13

## Formal grammars have:

- productions (grammar rules)
- symbols
  - terminals
  - non-terminals
- one non-terminal is designated as the *start symbol*



14

## Noam Chomsky: grammar hierarchy

- type 0: **free**  
productions:  $u \rightarrow v$ ,  $u$  &  $v$  arbitrary strings
- type 1: **context-sensitive**  
productions:  $uXw \rightarrow uvw$
- type 2: **context-free**  
productions:  $X \rightarrow v$
- type 3: **regular**  
productions:  $X \rightarrow a$ , or  $X \rightarrow aY$



more  
powerful

15

## The bottom of the hierarchy: regular expressions

- Useful for pattern matching
- easiest grammar to understand
- can be used to **specify tokens**

16

## tokens

- terminal symbols in the source language
  - reserved words: if else while
  - constants: 25 3.5
  - special symbols: ( ; : ?
- can be specified by **regular expressions**

17

## operators for regular expressions:

- + means one or more repetitions
- \* means zero or more repetitions
- | means or
- parens are used for grouping
- one character followed by another:  
**concatenation**

18

Which of the following are specified by  $a+b+$

- aaabbb
- abbb
- aaa
- bbb
- bbaa
- Aaaaaaaaaaaaaa
- ab
- $\lambda$



19

Which of the following strings are specified by  $0^*1^*$

- 000111
- 000
- 111
- 1100
- 1111111111
- $\lambda$

20

Which of the following strings are specified by  $(0 | 1)^*$

- 000
- 111
- 000111
- 111000
- 101010
- 121212
- $\lambda$

21

Describe the following regular expressions:

- $01^*0$
- $(0|1)^*$
- $0^*1^*$
- $a+b+$
- $[A-Z]^*$
- $[a-zA-Z][a-zA-Z0-9]^*$
- $[0-9]^+$



22

The *lex* tool:

- a tool that uses *regular expressions* to specify strings
- *flex*: free version of *lex* (GNU)



23

The *lex* tool:

- basic operations:
  - concatenation  
 $xy$  the pattern consists of  $x$  followed by  $y$
  - alternation  
 $x | y$  the pattern consists of either  $x$  or  $y$
  - arbitrary repetition  
 $x^*$  --  $x$  repeated 0 or more times  
 $x^+$  --  $x$  is repeated 1 or more times  
it's really  $xx^*$

24

## lex notation

- character classes [0-9]
- not operator ^ [^0-9] matches a non-digit
- period . matches anything but newline
- ^x match x at beginning of line only
- x\$ match x at end of line only
- "x" match the string x
- "x\*" matches x followed by an asterisk
- \x if x is an operator, match x itself  
or x can be an escape, like \n

25

## Examples of lex expressions:

[xz]	x or y
[x-z]	x, y or z
.	matches any character except end of line
x   y	x or y

26

## The lex tool:

- lex program has the form:

```

definitions
%%
Rules & definitions
%%
user functions
    
```

*end of section 1*

*end of section 2*

27

## Overview of lex file:

```

%{
// any variables/functions that you want
// included in the lex specification
%}

%%

<definitions>

%}
<functions>
    
```

*regular expressions*

*C or C++ code*

*<actions>*

28

## A lex file has 3 sections, separated by %%

```

%{ // items here are inserted into lex.yy.c
#include "y.tab.h"
...
%}

digit      [0-9]
digits     {digit}+
letter     [a-zA-Z]

%%

"+"       { return PLUSTK; }

%%

int yywrap() { return 1; }
    
```

*definitions go here*

*rules go in this section*

*lex is returning a token*

*functions go here*

29

```

%{
    int line_count = 0, word_count = 0;
%}
%%
[a-zA-Z]+  ++word_count;
"\n"      ++line_count;
.          ;
%%
int yywrap() { return 1; }
    
```

*example #1*

*this is the lex file, call it count.l*

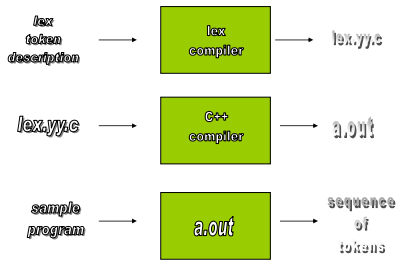
```

#include <iostream>
#include "lex.yy.c"
main() {
    yylex();
    cout << "There are: " << line_count << " lines." << endl;
    cout << "There are: " << word_count << " words." << endl;
}
    
```

*this is the main program, call it main.cpp*

30

## Using *lex*



31

To compile it and run it, you need a data file: *data.dat*

```
lex count.l
g++ main.cpp
a.out < data.dat
```

*this one redirects the input*

```
lex count.l
g++ -o count main.cpp
count data.dat
```

*this one exploits command-line parameters*

32

## Some important points: *yylex()*

- *yylex()* is a function that recognizes regular expressions.
  - *yylex()* is placed in *lex.yy.c*
  - you can view *lex.yy.c*
- Need to call *yylex()* from main. **If the lex specification returns values, may need to repeatedly call *yylex()***
- *yylex()* returns zero when eof is reached

33

## *yywrap()*

- *yywrap()* is a function that is called upon termination of *yylex()*
  - can be used to make another pass through the input
  - can be used to do clean-up or wrap-up

34

## The function *yywrap()*



```
% {
    int line_count = 0, letter_count = 0;
% }
%%
[a-zA-Z]    ++letter_count;
"\n"       ++line_count;
.          ;
%%
int yywrap() {
    cout << "Now leaving yywrap()" << endl;
    // if we return zero, we have an infinite loop!
    // if we return one, we return to main.
    return 1;
}
```

35

## More than one pattern matches?

```
%%
[0-9]+    printf("Matched\n");
9        printf("never found the 9\n");
.        ;
```



*lex takes the first match!*

36

## We could write a lex specification using definitions:

```
%{
    int line_count = 0, word_count = 0;
}%
letter      [a-zA-Z]
letters     (letter)+
%%
{letters}   ++word_count;
"\n"       ++line_count;
.          ;
%%

int yywrap() { return 1; }
```

*definitions  
go here*

37

```
% {
    int reserved_words = 0, ids = 0;
% }
```

**example #2**

```
id          [a-zA-Z][a-zA-Z0-9_]*
```

```
%%
int         ++reserved_words;
if          ++reserved_words;
else       ++reserved_words;
while      ++reserved_words;
for        ++reserved_words;
main       ++reserved_words;
{id}       ++ids;
%%
int yywrap() { return 1; }
```

*this lex program counts  
reserved words and  
identifiers*

38 38

## How can *lex* return a token?

- associate a number (*const*) with each unique token;
- write code so that *yylex()* returns the number associated with each token;
- send the token to the parser or other tool.



39

```
% {
    const INT_TOK = 1;
    const IF_TOK = 2;
    const ELSE_TOK = 3;
    const WHILE_TOK = 4;
    const FOR_TOK = 5;
    const MAIN_TOK = 6;
    const ID_TOK = 7;
% }
```

**example #3**

```
id          [a-zA-Z][a-zA-Z0-9_]*

%%
int         { return INT_TOK; }
if          { return IF_TOK; }
else       { return ELSE_TOK; }
while      { return WHILE_TOK; }
for        { return FOR_TOK; }
main       { return MAIN_TOK; }
{id}       { return ID_TOK; }
\n         ;
.          ;
%%
int yywrap() { return 1; }
```

*this is a lex specification  
to recognize C tokens*

40

40

```
#include "lex.yy.c"
#include <iostream>
```

*source program,  
input to lex*

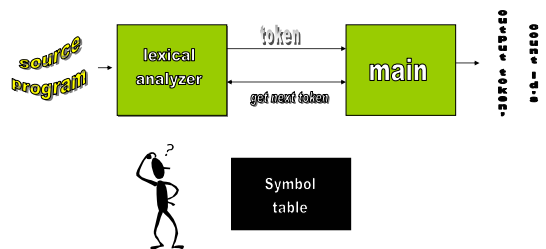
```
main() {
    int token = yylex();
    int id_count = 0;
    while ( token ) {
        cout << "The token was: " << token << endl;
        if (token == 7) ++id_count;
        token = yylex();
    }
    cout << "There were: " << id_count << " identifiers." << endl;
}
```

```
main() {
    int i = 0, j = 0;
    while (i) {
        if (i) ++i;
        else ++j;
    }
}
```

*this is the main program  
that uses the lex spec in  
example #3*

41

## In example #3, *lex* is working in tandem with main:



Symbol  
table

42

## Lexical analysis

- First phase in compiler implementation
- techniques apply to editors, query languages, testing and info retrieval
- problem: recognize patterns
- approaches:
  - write it by hand (*ad hoc*)
  - use a tool (*lex*)

43

## Tasks in lexical analysis

- find extraneous (error) characters
- store *names* in symbol table
- 'strip' white space
- recognize tokens

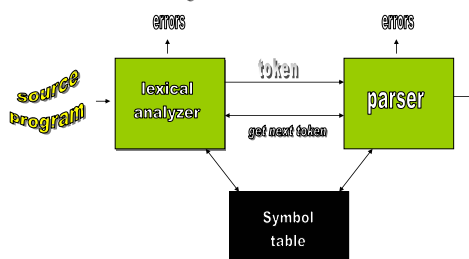
44

## Definition of *lex* tool:

- a *lexical analyzer* generator that takes a description of tokens (using regular expressions) and produces a FSM
- *flex*: free version of *lex* (GNU)

45

## lexical analyzer and parser usually work in tandem:



46

## Kinds of errors that lexical analysis can detect:

- only extraneous characters
- other errors must be detected by later stage. for example:

fi (x == y)

*lex thinks this is an identifier*

47

## What's a symbol table?

- During *lexical analysis* it's really an identifier table because it's simply a list of unique identifier strings
- during *parsing* we can store info about the id's such as type info, nesting level, size, etc.



48

## Writing a main program that uses lex to read from a file:

```
#include <iostream>
#include <fstream>
#include "lex.yy.c"

void main(int argc, char * argv[] ) {
    if (argc != 2) {
        cout << "usage: " << argv[0] << " << filename << endl; return 1;
    }
    FILE * infile;
    infile = fopen(argv[1], "r");
    if (!infile) {
        cout << "Could not open: " << argv[1] << endl; return 1;
    }
    yyin = infile;
    yylex();
}
```

**Must use C-style file syntax**

**The magic line**

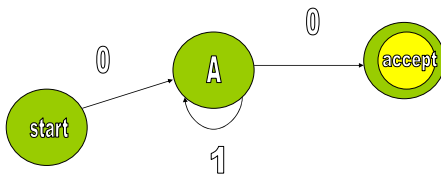
49

## `yylex()` is a finite state machine (FSM):

- states
- transition tokens
- special states:
  - start state
  - final state (acceptor)

50

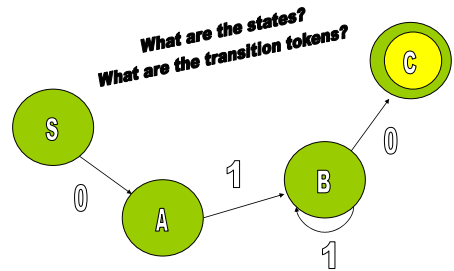
## A FSM to recognize $01^*0$



the states are start, A, accept  
the transition tokens are 0 and 1

51

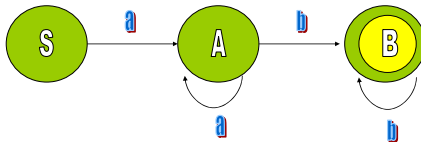
## A FSM to recognize $01+0$



**What are the states?  
What are the transition tokens?**

52

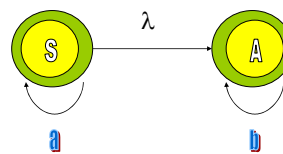
## What regular expression is accepted by the FSM below?



draw FSM for  $0+1^*$

53

## Sometimes the empty production facilitates FSM construction: $a^*b^*$

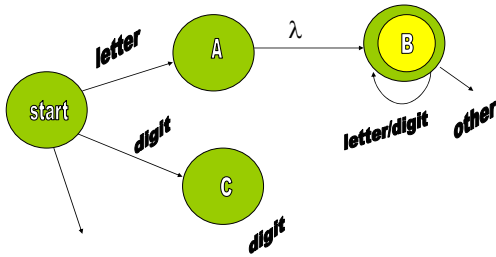


**try to construct the FSM without an empty token**

**Note that both states are accepting states!**

54

How do you *recognize* a token for a language: use a FSM



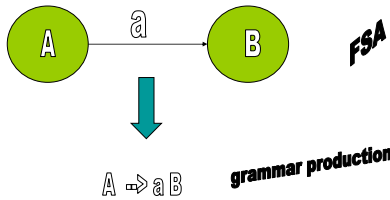
55

Translation from FSM to grammar is straightforward:

- states become non-terminal symbols (usually **upper case** letter or word)
- transition tokens become terminal symbols (usually **lower case** letter)

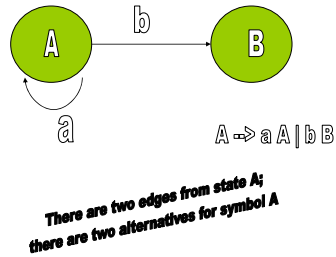
56

To construct a grammar production: take two states & the transition token



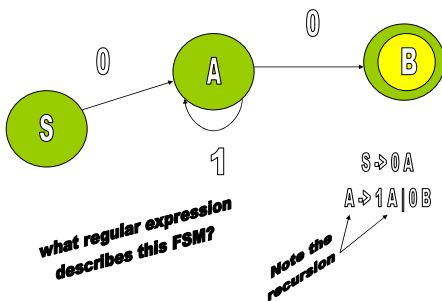
57

Each transition edge becomes an alternative in the production:



58

The FSM below can be translated into a grammar:



59

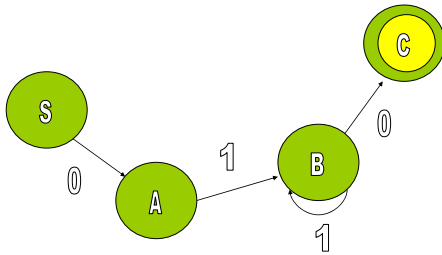
Parse trees can be used to generate strings from a grammar:

$S \rightarrow 0A$   
 $A \rightarrow 1A | 0B$   
 $B \rightarrow 0$

generate 0110

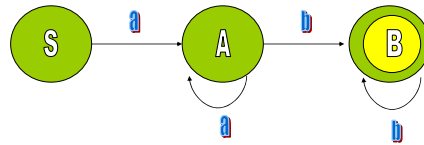
60

Write a grammar for the FSM ; also write a regular expression:



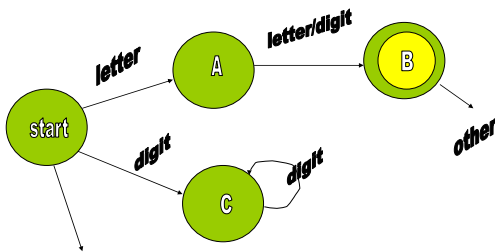
61

Write a grammar for:



62

Write a grammar for:



63

Some constructs are:

- **regular**, can be specified/recognized by:
  - regular expression
  - Finite State Machine (FSM)
  - regular grammar
- **other constructs need context-free grammar**
  - excellent for recognizing *syntactic structures* like “if” structures, “while” structures, **balanced parentheses**

64

Parsing: using grammars to recognize program structures:

- regular grammars can recognize tokens such as “if” or id’s
- regular grammars cannot recognize some program structures, such as **balanced parentheses**
  - try to construct a FSM to generate only balanced parens
- context-free grammars can recognize balanced parens, as well as language structures, such as *while*, *if* etc

65

Example: grammar to generate balanced parentheses:

- $S \rightarrow (S)S \mid \lambda$ 
  - two productions
  - one *non-terminal* (the **start symbol**)
  - two *terminals*
- the *empty* production:  $\lambda$
- | means or



66

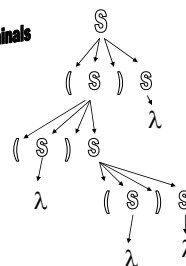
## Parsing

- (defn) deriving a *sentence* from the *start symbol* by repeated application of *productions*
- parsing can be illustrated with a tree, showing how each symbol is *derived* from other symbols
- *sentence*: list terminals, the leaves in the tree, in left-to-right order

67

## Parse tree for: ( ( ) ( ) )

all leaves are terminals



Where's the derived sentence?

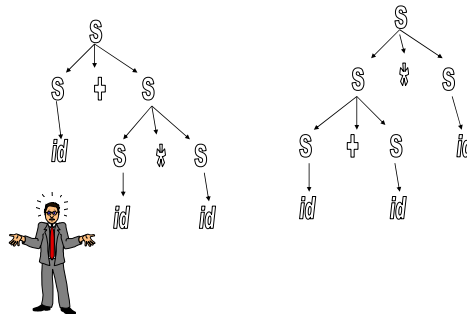
68

## Example: grammar to generate arithmetic expressions

- $S \rightarrow S + S \mid S * S \mid id$ 
  - how many productions?
  - how many non-terminals?
  - how many terminals?
- generate a tree for  $x + y * z$ :

69

## Parse trees for: $x + y * z$



70

## $S \rightarrow S + S \mid S * S \mid id$ is *ambiguous*

- two different parse trees for the same sentence!
- was  $S \rightarrow (S)S \mid \lambda$  ambiguous?
- how does a parser handle ambiguity?
- can you think of another example of ambiguity in a programming language?

71

## Techniques for handling ambiguity:

- Modify the grammar
- establish a rule & incorporate it into the parser
- modify the language
- *lex* & *yacc* have special operations

72

## An unambiguous grammar for arithmetic expressions:

- $S \rightarrow S + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow id$

Try to get two different parse trees for:  
 $x + y * z$

73

## *if* statements

- $S \rightarrow \text{if expr then stmt}$
- $S \rightarrow \text{if expr then stmt else stmt}$
- $S \rightarrow \text{other}$

Show this grammar  
is ambiguous.

74

## How do we use CFG to parse?

- Use grammar to guide writing of top-down parser (by hand)
- yacc takes a grammar and writes a parser

75

## yacc

- *yet another compiler compiler*
- constructs a bottom-up parser to recognize language constructs
- constructs must be *context free!*
- type of parser is *lalr*
- built to work with *lex*
- structure is similar to *lex*: 3-part structure

76

## grammar

- productions or rules
- rules describe valid sentences
- usual representation of a parsed sentence is a **tree**
- some rules are recursive: they refer to themselves:  $A \rightarrow 0A$

77

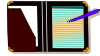
## what yacc cannot parse:

- ambiguous grammars
- grammars that require more than one look-ahead to determine a matched rule

78

## Symbol table

- data structure used to store information about the *names* used in the source program
- info:
  - the string for the *name*
  - type info for the *name*
  - scope info



79

## Review; what have we done:

- Lexical analysis: regular expressions, finite state automata, regular grammars, **lex tool**, tokens
- parsing: context free grammar, chomsky hierarchy, why regular grammars are not enuf, **yacc tool**
- constructed two examples of the use of *lex* and *yacc*: calculator & pascal grammar

80

## Limitations of the two examples

- Both were calculators
- Neither allowed for **different types of data** or for iteration statements: simply integer arithmetic
- To allow for different types of data, we can incorporate inheritance and polymorphism into our symbol table construction

81

## After front-end: **code generation**

- why do we need it?
  - our calculator cannot easily process repetition statements
  - machines (computers) only understand *object code*
- Most compilers first generate **intermediate code**
  - better to modularize compilation
  - a portable front end is desirable

82

## Intermediate code:

- Many forms and varieties
  - three address code (quadruples)
  - two address code
  - *rtl* -- used by GNU
  - simply construct and decorate the parse tree
- complex arithmetic statements are converted to many simple statements ==> use temporary locations ==> **registers**

83

## Three address code (quads)

- It's a **quad** because there are 4 fields
  - one operator field
  - three operand fields
- It's **three address** because each of the 3 operands will likely include an address
- Typically, the operand fields must include more info than the address, for example, the type of the operand (int, real, const, temp)

84

## Sample C program:

```
main(){
  int a, b, c, d;
  a=0;
  b=3;
  c=6;
  d = a - b + c - 8 - a + 20 - c;
  printf(b);
}
```

85

## Corresponding quads:

ASSGN	<-3, 0>	<-6, -6>	< 2, 0>
ASSGN	<-3, 3>	<-6, -6>	< 2, 1>
ASSGN	<-3, 6>	<-6, -6>	< 2, 2>
SUBI	< 2, 0>	< 2, 1>	<-1, 1>
ADDI	<-1, 1>	< 2, 2>	<-1, 2>
SUBI	<-1, 2>	<-3, 8>	<-1, 3>
SUBI	<-1, 3>	< 2, 0>	<-1, 4>
ADDI	<-1, 4>	<-3, 20>	<-1, 5>
SUBI	<-1, 5>	< 2, 2>	<-1, 6>
ASSGN	<-1, 6>	<-6, -6>	< 2, 3>
VPRM	< 2, 1>	<-6, -6>	<-6, -6>
CALL	<-6, -6>	<-6, -6>	<-5, printf>
NOARGS	<-3, 1>	<-3, 0>	<-6, -6>

86

## How to generate quads

- Insert semantic actions into the parser
- Need a symbol table to be able to determine the type of variables
- To handle different types of variables, use inheritance and polymorphism

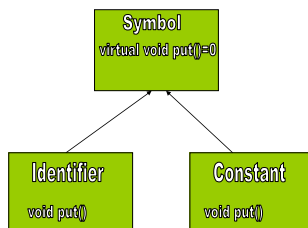
87

## How does polymorphism help?

- permits symbol table to maintain different types of data
- use a base class: Symbol
- derived classes for each type of data:
  - constant
  - identifier
  - temporary

88

## Inheritance hierarchy:



89

## To use yacc:

- must declare the types of items
- yacc is a C based tool, i.e., need a union, the polymorphic counterpart

90

## Semantic action for calculator:

```
Simple_expr: Term
  | PLUSTK Term
  | MINUSTK Term
  | Simple_expr Add_op Term
  {
    if ($2 == PLUSTK) $$ = $1 + $3;
    else $$ = $1 - $3;
  }
  | Simple_expr Add_op error
  ;
```

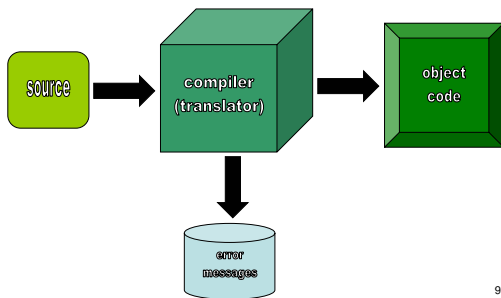
91

## Semantic action to generate code:

```
Simple_expr: Term
  | PLUSTK Term
  | MINUSTK Term
  | Simple_expr Add_op Term
  {
    symbol = install_temp();
    if ($2 == PLUSTK)
      generate(ADDI, $1, $3, symbol);
    else
      generate(SUBI, $1, $3, symbol);
    $$ = symbol;
  }
  | Simple_expr Add_op error
  ;
```

92

## Back to basics: What's a compiler?



93