An Overview of a Compiler - Part 1

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

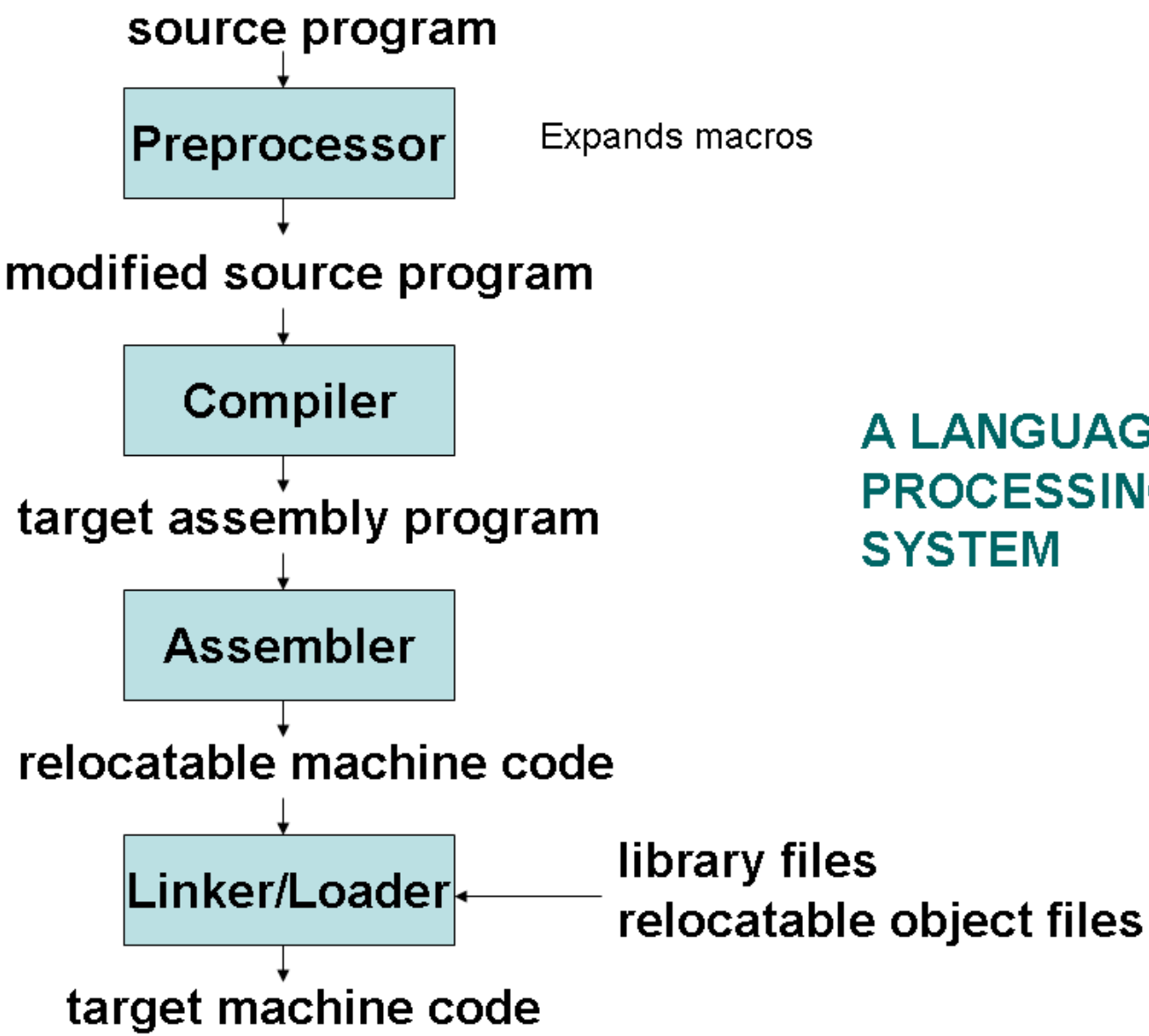
NPTEL Course on Compiler Design

Outline of the Lecture

1. Compiler overview with block diagram
2. Lexical analysis with LEX
3. Parsing with YACC
4. Semantic analysis with attribute grammars
5. Intermediate code generation with syntax-directed translation
6. Code optimization examples

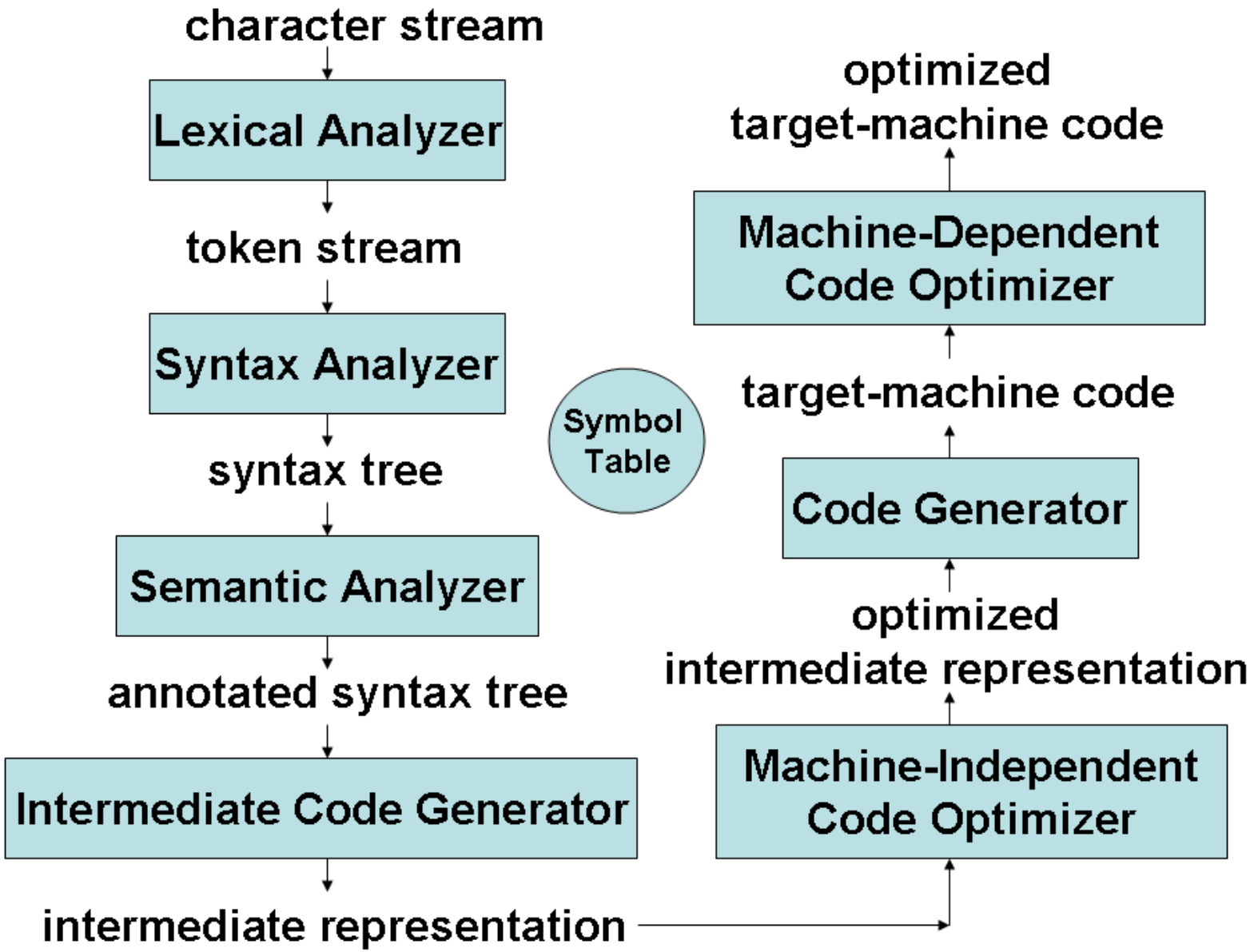
Topics 5 and 6 will be covered in Part II of the lecture

Language Processing System

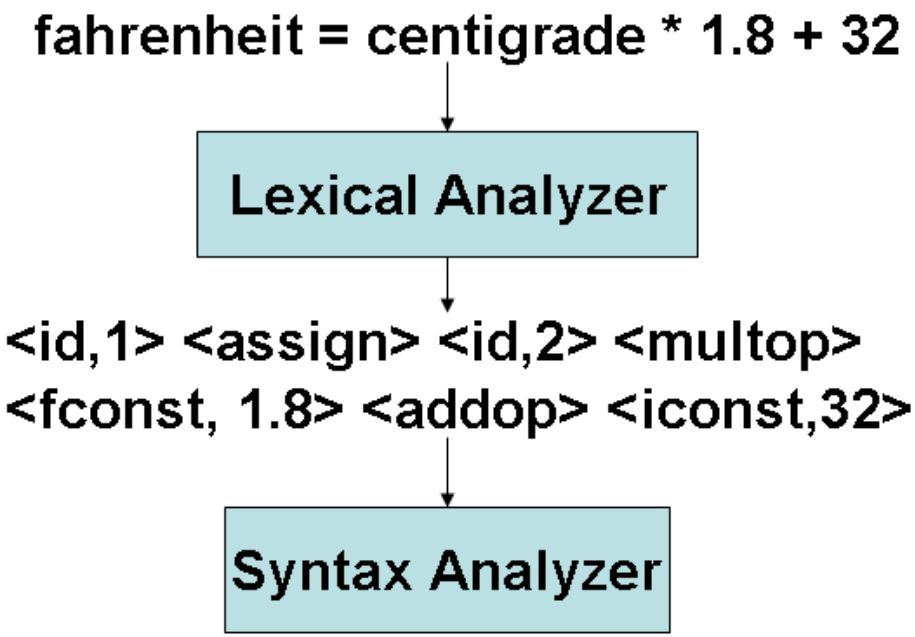


A LANGUAGE PROCESSING SYSTEM

Compiler Overview



Translation Overview - Lexical Analysis



Lexical Analysis

- LA can be generated automatically from regular expression specifications
  - LEX and Flex are two such tools
- Tokens of the LA are the terminal symbols of the parser
- LA is usually called to deliver a token when the parser needs it
- Why is LA separate from parsing?
  - Simplification of design - software engineering reason
  - I/O issues are limited LA alone
  - LA based on finite automata are more efficient to implement than pushdown automata used for parsing (due to stack)

LEX Example

```
%%
[A-Z]+
%%
yywrap(){}
main(){yylex();}
```

Form of a LEX File

- LEX has a language for describing regular expressions
- It generates a pattern matcher for the REs described
- General structure of a LEX program
  {definitions}
  %%
  {rules}
  %%
  {user subroutines}
- A LEX compiler generates a C-program **lex.yy.c** as output

Definitions Section

- Definitions Section contains definitions and included code
  - Definitions are like macros and have the following form: **name translation**

```
digit [0-9]
number {digit} {digit}*
```

  - Included code is all code included between **%{** and **%}**

```
%{
    float number; int count=0;
%}
```

Rules Section

- Contains patterns and C-code
- A line starting with white space or material enclosed in %{ and %} is C-code
- A line starting with anything else is a pattern line
- Pattern lines contain a pattern followed by some white space and C-code
  $\{pattern\} \quad \{action\ (C-code)\}$
- C-code lines are copied verbatim to the the generated C-file
- Patterns are translated into NFA which are then converted into DFA, optimized, and stored in the form of a table and a driver routine
- The action associated with a pattern is executed when the DFA recognizes a string corresponding to that pattern and reaches a final state

LEX Example 1

```
number [0-9]+\.?|[0-9]*\.[0-9]+
name [A-Za-z][A-Za-z0-9]*
%%
[ ] {/* skip blanks */}
{number} {sscanf(yytext,"%lf",&yylval.dval);
                    return NUMBER;}
{name} {struct symtab *sp =symlook(yytext);
                    yylval.symp = sp; return NAME;}
"++" {return POSTPLUS;}
"--" {return POSTMINUS;}
"$" {return 0;}
\n|. {return yytext[0];}
```

LEX Example 2

```
%{
FILE *declfile;
%}

blanks [ \t]*
letter  [a-z]
digit [0-9]
id ({letter}|_)({letter}|{digit}|_)*
number {digit}+
arraydeclpart {id}"["{number}"]"
declpart ({arraydeclpart}|{id})
decllist ({declpart}{blanks}","{blanks})*
                    {blanks}{declpart}{blanks}
declaration (("int")|("float")){blanks}
                    {decllist}{blanks};
```

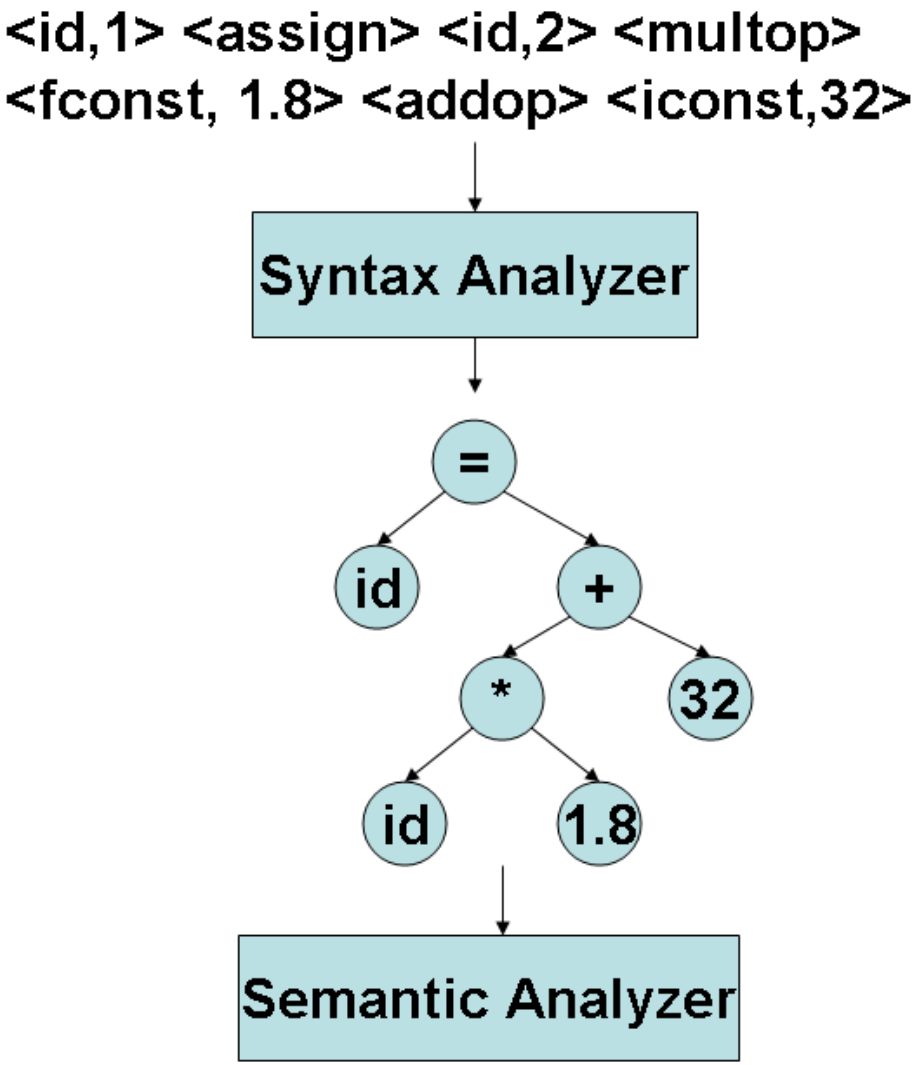
LEX Example (contd.)

```
%%
{declaration} fprintf(declfile,"%s\n",yytext);
%%

yywrap(){
fclose(declfile);
}
main(){
declfile = fopen("declfile","w");
yylex();
}
```

Examples of declarations:
int a, b[10], c, d[25];
float k[20], l[10], m,n;

Translation Overview - Syntax Analysis



Parsing or Syntax Analysis

- Syntax analyzers (parsers) can be generated automatically from several variants of context-free grammar specifications
  - LL(1), and LALR(1) are the most popular ones
  - ANTLR (for LL(1)), YACC and Bison (for LALR(1)) are such tools
- Parsers are deterministic PDAs and cannot handle context-sensitive features of programming languages; e.g.,
  - Variables are declared before use
  - Types match on both sides of assignments
  - Parameter types and number match in declaration and use
- Syntax tree need not be produced explicitely by the parser if semantic analysis is carried out simultaneously with parsing
  - However, this may not be possible in languages such as C++ which cannot be semantically validated in a single pass

Form of a YACC file

- YACC has a language for describing context-free grammars
- It generates an LALR(1) parser for the CFG described
- Form of a YACC program
  %{ declarations – optional
  %}
  %%
  rules – compulsory
  %%

  programs – optional
- YACC uses the lexical analyzer generated by LEX to match the **terminal symbols** of the CFG
- YACC generates a file named **y.tab.c**

YACC Example: LEX Specification

```
number [0-9]+\.?|[0-9]*\.[0-9]+
name [A-Za-z][A-Za-z0-9]*
%%
[ ] {/* skip blanks */}
{number} {sscanf(yytext,"%lf",&yylval.dval);
                    return NUMBER;}
{name} {struct symtab *sp =symlook(yytext);
                    yylval.symp = sp; return NAME;}
"++" {return POSTPLUS;}
"--" {return POSTMINUS;}
"$" {return 0;}
\n|. {return yytext[0];}
```

YACC Example: YACC Specification

```
%{
#define NSYMS 20
struct symtab {
        char *name; double value;
     }symboltab[NSYMS];
struct symtab *symlook();
#include <string.h>
#include <ctype.h>
#include <stdio.h>
%}
```

YACC Example: YACC Specification

```
%union {
    double dval;
    struct symtab *symp;
    }
%token <symp> NAME
%token <dval> NUMBER
%token POSTPLUS
%token POSTMINUS
%left '='
%left '+' '-'
%left '*' '/'
%right UMINUS
%left POSTPLUS
%left POSTMINUS
%type <dval> expr
```

YACC Example: YACC Specification

```
%%
lines: lines expr '\n' {printf("%g\n",$2);}
     | lines '\n'
     | /* empty */
     | error '\n'
       {yyerror("reenter last line:"); yyerrok; }
     ;
expr : NAME '=' expr {$1 -> value = $3; $$ = $3;}
     | NAME {$$ = $1 -> value;}
     | expr '+' expr {$$ = $1 + $3;}
     | expr '-' expr {$$ = $1 - $3;}
     | expr '*' expr {$$ = $1 * $3;}
     | expr '/' expr {$$ = $1 / $3;}
     | '(' expr ')' {$$ = $2;}
     | '-' expr   %prec UMINUS {$$ = - $2;}
```

YACC Example: YACC Specification

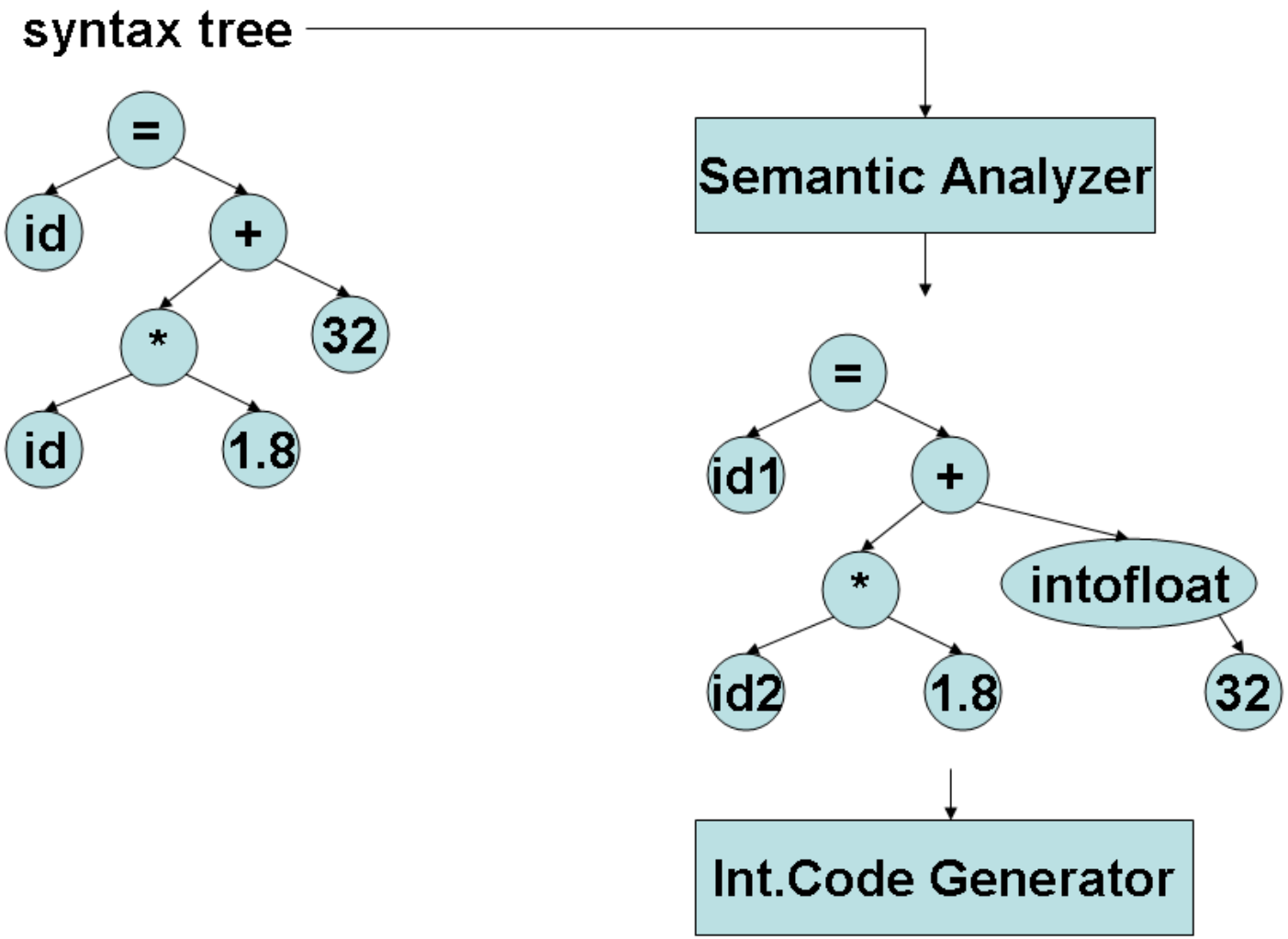
```
        | NUMBER
        | NUMBER POSTPLUS  %prec POSTPLUS
              {$$ = $1 + 1;}
        | NUMBER POSTMINUS %prec POSTMINUS
              {$$ = $1 - 1;}
        ;
%%
void initsymtab()
{int i = 0;
 for(i=0; i<NSYMS; i++) symboltab[i].name = NULL;
}
int yywrap(){return 1;}
yyerror( char* s) { printf("%s\n",s);}
main() {initsymtab(); yyparse(); }

#include "lex.yy.c"
```

YACC Example: YACC Specification

```
struct symtab* symlook(char* s)
{struct symtab* sp = symboltab; int i = 0;
 while ((i < NSYMS) && (sp -> name != NULL))
  { if(strcmp(s,sp -> name) == 0) return sp;
    sp++; i++;
  }
 if(i == NSYMS) {
   yyerror("too many symbols"); exit(1);
 }
 else { sp -> name = strdup(s);
        return sp;
      }
}
```

Translation Overview - Semantic Analysis



Semantic Analysis

- Semantic consistency that cannot be handled at the parsing stage is handled here
- Type checking of various programming language constructs is one of the most important tasks
- Stores type information in the symbol table or the syntax tree
  - Types of variables, function parameters, array dimensions, etc.
  - Used not only for semantic validation but also for subsequent phases of compilation
- Static semantics of programming languages can be specified using attribute grammars
- Semantic analyzers can be generated automatically from attributed translation grammars
- If declarations need not appear before use (as in C++), semantic analysis needs more than one pass

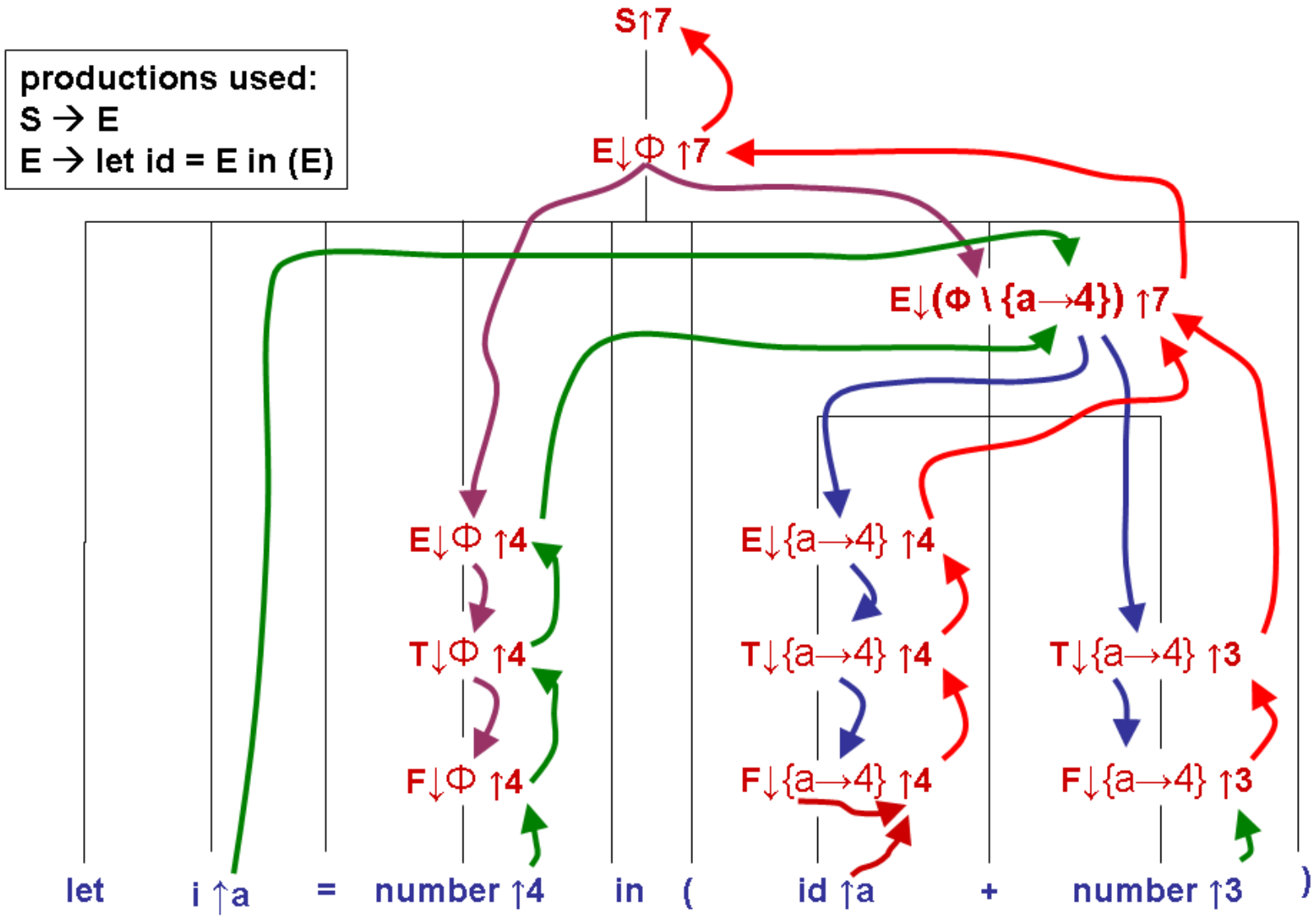
Example of an Attribute Grammar

- Let us first consider the CFG for a simple language
  1. $S \longrightarrow E$
  2. $E \longrightarrow E + T \mid T \mid \mathit{let}\ \mathit{id} = E\ \mathit{in}\ (E)$
  3. $T \longrightarrow T * F \mid F$
  4. $F \longrightarrow (E) \mid \mathit{number} \mid \mathit{id}$
- This language permits expressions to be nested inside expressions and have scopes for the names
  - *let A = 5 in ((let A = 6 in (A*7)) - A)* evaluates correctly to 41, with the scopes of the two instances of *A* being different
- It requires a scoped symbol table for implementation
- The next slide shows an abstract attribute grammar for the above language and the slide following it shows an implementation of the abstract AG using a YACC-style translation grammar
- Abstract AGs permit both inherited and synthesized attributes, whereas YACC-style grammars permit only synthesized attributes

An Abstract Attribute Grammar

1. $S \longrightarrow E \; \{E.symtab \downarrow := \phi; S.val \uparrow := E.val \uparrow\}$
2. $E_1 \longrightarrow E_2 + T \; \{E_2.symtab \downarrow := E_1.symtab \downarrow;$
   $E_1.val \uparrow := E_2.val \uparrow + T.val \uparrow; T.symtab \downarrow := E_1.symtab \downarrow\}$
3. $E \longrightarrow T \; \{T.symtab \downarrow := E.symtab \downarrow; E.val \uparrow := T.val \uparrow\}$
4. $E_1 \longrightarrow let \; id = E_2 \; in \; (E_3)$
   $\{E_1.val \uparrow := E_3.val \uparrow; E_2.symtab \downarrow := E_1.symtab \downarrow;$
   $E_3.symtab \downarrow := E_1.symtab \downarrow \setminus \{id.name \uparrow \rightarrow E_2.val \uparrow\}\}$
5. $T_1 \longrightarrow T_2 * F \; \{T_1.val \uparrow := T_2.val \uparrow * F.val \uparrow;$
   $T_2.symtab \downarrow := T1.symtab \downarrow; F.symtab \downarrow := T_1.symtab \downarrow\}$
6. $T \longrightarrow F \; \{T.val \uparrow := F.val \uparrow; F.symtab \downarrow := T.symtab \downarrow\}$
7. $F \longrightarrow (E) \; \{F.val \uparrow := E.val \uparrow; E.symtab \downarrow := F.symtab \downarrow\}$
8. $F \longrightarrow number \; \{F.val \uparrow := number.val \uparrow\}$
9. $F \longrightarrow id \; \{F.val \uparrow := F.symtab \downarrow [id.name \uparrow]\}$

Attrbute Flow



An Attributed Translation Grammar

```
1. S --> E  { S.val := E.val }
2. E --> E + T  { E(1).val := E(2).val + T.val }
3. E --> T  { E.val := T.val }
/* The 3 productions below are broken parts
   of the prod.: E --> let id =  E in (E) */
4. E --> L B  { E.val := B.val; }
5. L --> let id = E { //scope initialized to 0;
         scope++; insert (id.name, scope, E.val) }
6. B --> in (E) { delete_entries (scope);
          scope--; B.val := E.val }
7. T --> T * F  { T(1).val := T(2).val * F.val }
8. T --> F  { T.val := F.val }
9. F --> (E) { F.val := E.val }
10. F --> number  { F.val := number.val }
11. F --> id { F.val := getval (id.name, scope) }
```