Compiler construction in4303 – lecture 5

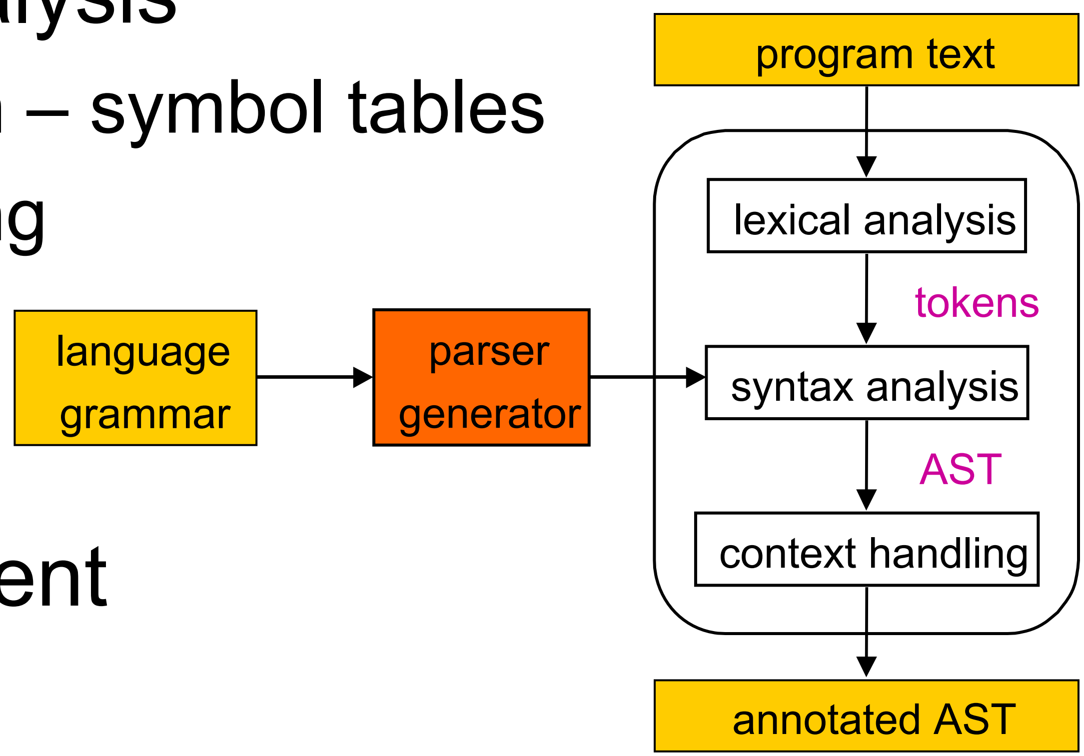
Semantic analysis

Assignment #1

Chapter 6.1

Overview

- semantic analysis
  - identification – symbol tables
  - type checking



- Lab assignment
  - yacc tutorial

Semantic analysis

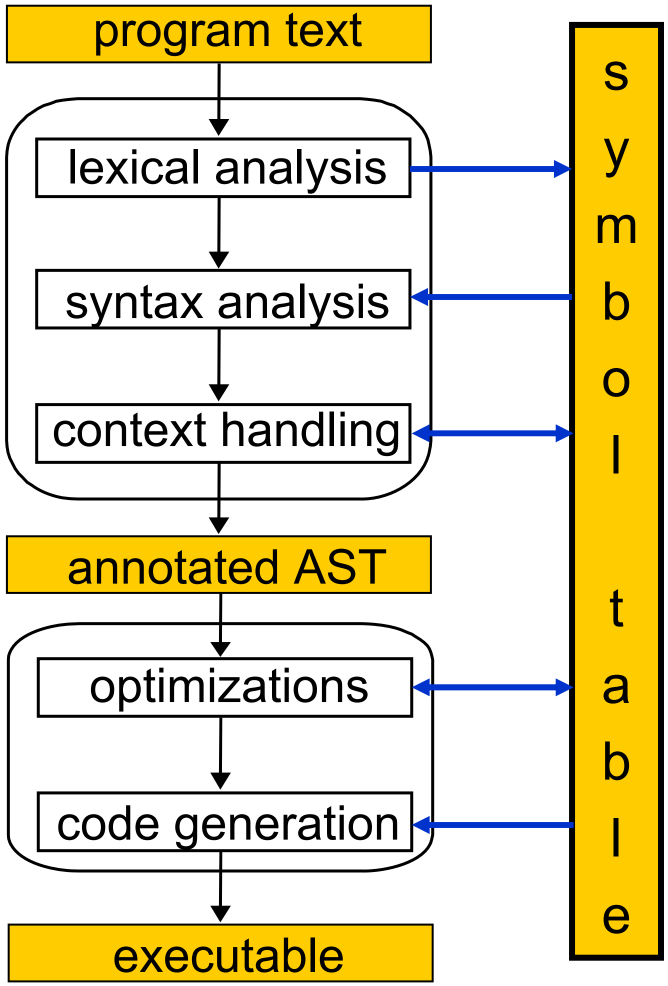
- information is scattered throughout the program
- identifiers serve as connectors

- find defining occurrence of each applied occurrence of an identifier in the AST
  - undefined identifiers ⇒ error
  - unused identifiers ⇒ warning

- check rules in the language definition
  - type checking
  - control flow (dead code)

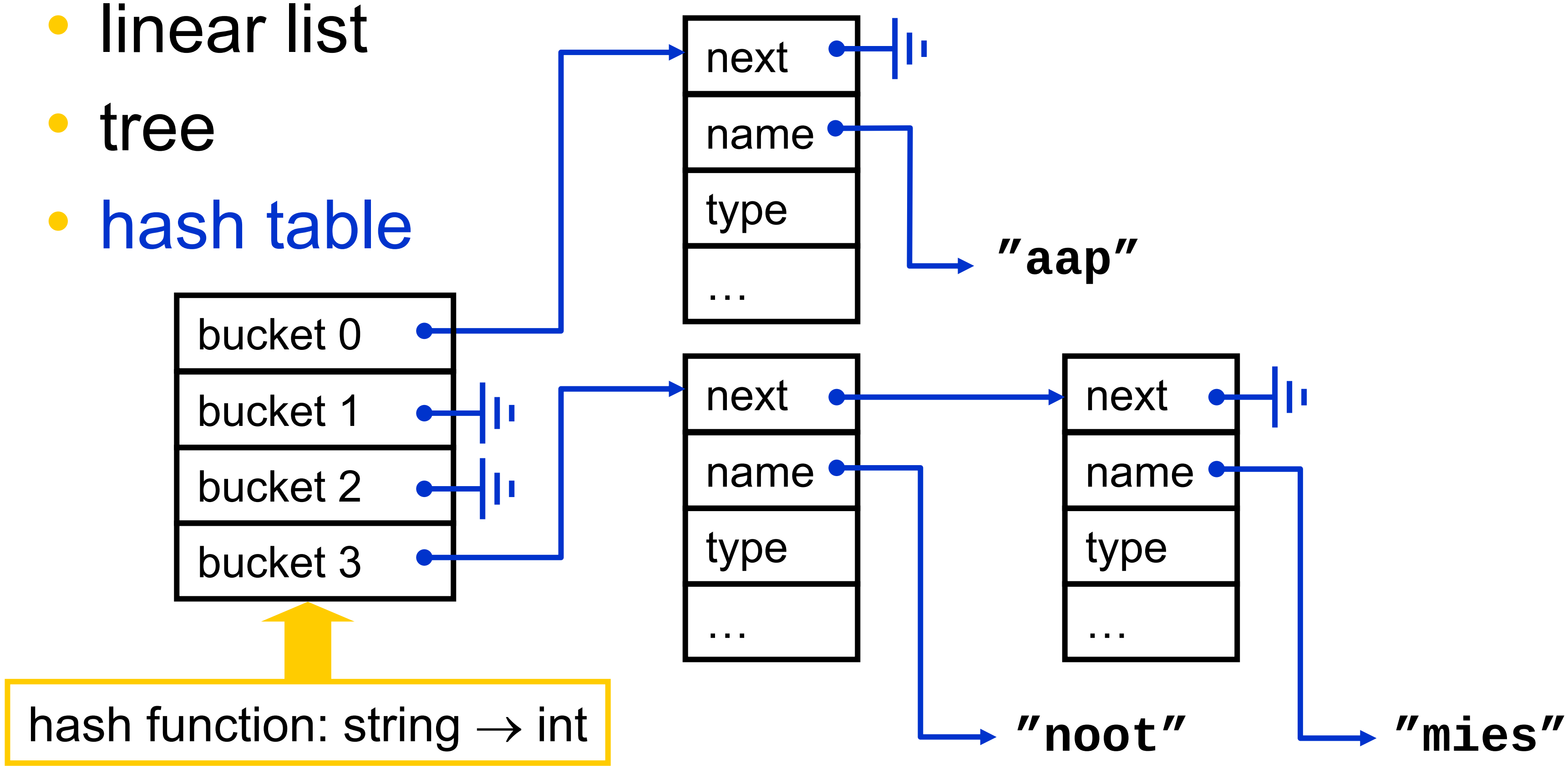
Symbol table

- global storage used by all compiler phases
- holds information about identifiers:
  - type
  - location
  - size
  -



Symbol table implementation

- extensible string-indexable array
  - linear list
  - tree
  - hash table



Identification

- different kinds of identifiers
  - variables
  - type names
  - field selectors
- name spaces
- scopes

```
typedef int i;

int j;

void foo(int j)
{
    struct i {i i;} i;

i:  i.i = 3;
    printf( "%d\n", i.i);
}
```

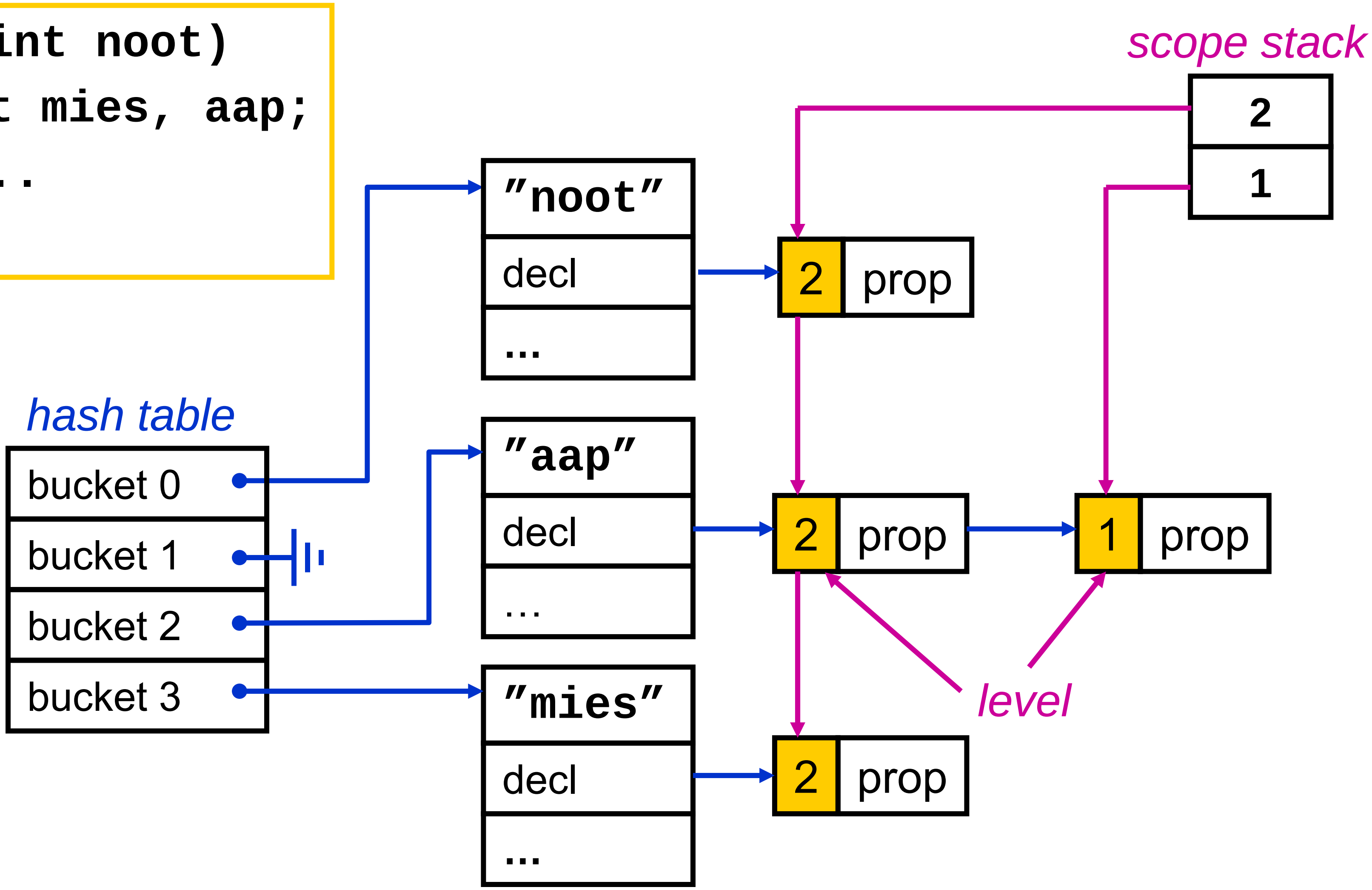
Handling scopes

Stack of scope elements

- when entering a scope a new element is pushed on the stack
- declared identifiers are entered in the top scope element
- applied identifiers are looked up in the scope elements from top to bottom
- the top element is removed upon scope exit

A scoped hash-based symbol table

```
aap( int noot)
{  int mies, aap;
   ....
}
```



Identification: complications

- overloading
  - operators: `N*2  prijs*2.20371`
  - functions: `PUT(s:STRING)  PUT(i:INTEGER)`
  - solution: yield set of possibilities (to be constrained by type checking)
- imported scopes
  - C++ scope resolution operator `x::`
  - Modula `FROM module IMPORT ...`
  - solution: stack (or merge) the new scope

Type checking

- operators and functions impose restrictions on the types of the arguments

- types
  - basic types
  - structured types
  - type names

```
typedef
    struct {
        double re;
        double im;
    } complex;
```

Forward declarations

- recursive data structures

```
TYPE Tree = POINTER TO Node;
TYPE Node = RECORD
              element : Integer;
              left, right : Tree;
            END RECORD;
```

- type information must be stored
  - type table
- type information must be resolved
  - undefined types
  - circularities

Type equivalence

- name equivalence [all types get a unique name]

```
VAR a : ARRAY [Integer 1..10] OF Real;
VAR b : ARRAY [Integer 1..10] OF Real;
```

TYPE(a) ≠ TYPE(b)

- structural equivalence [difficult to check]

```
TYPE c = RECORD i : Integer; p : POINTER TO c; END RECORD;
TYPE d = RECORD
           i : Integer;
           p : POINTER TO
                 RECORD
                   i : Integer;
                   p : POINTER to c;
                 END RECORD;
         END RECORD;
```

TYPE(c) = TYPE(d)

Coercions

- implicit data and type conversion to match operand (argument) type

```
VAR a : Real;
...
a := 5;
```

- coercions complicate identification (ambiguity)

```
  3.14 + 7
     8 + 9
```

- two phase approach
  - expand a type to a set by applying coercions
  - reduce type sets based on constraints imposed by (overloaded) operators and language semantics

Variable: value or location?

- two usages of variables

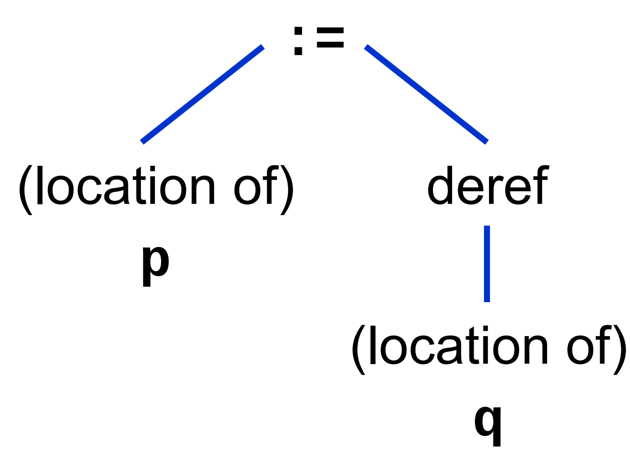
  rvalue: value

  lvalue: location

```
VAR p : Real;
VAR q : Real;
...
p := q;
```

- insert coercion to dereference variable
- checking rules:

	expected	
found	lvalue	rvalue
lvalue	-	deref
rvalue	ERROR	-



Exercise (3 min.)

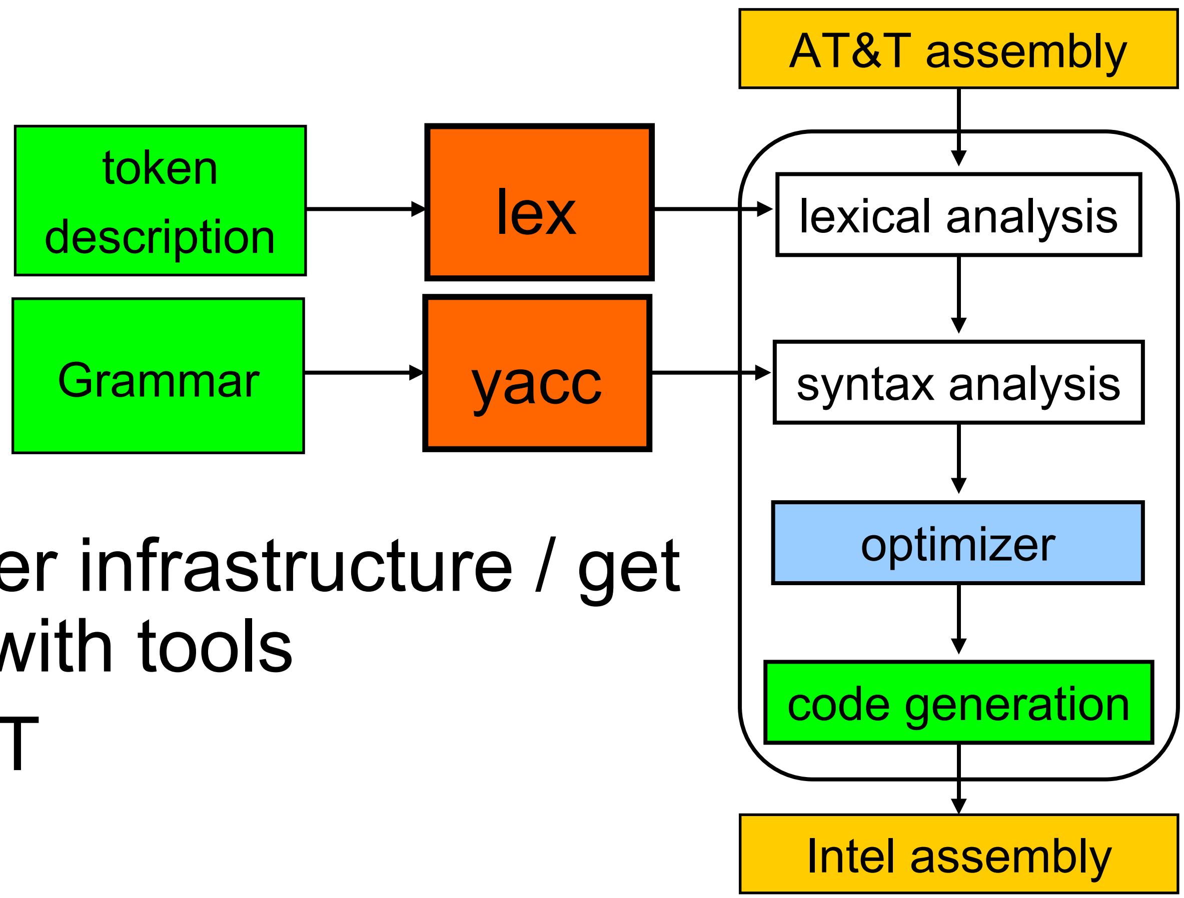
complete the table

expression construct	result kind (lvalue/rvalue)
constant	rvalue
variable	
identifier (non-variable)	
&lvalue	
*rvalue	
V[rvalue]	
rvalue + rvalue	
lvalue = rvalue	

V stands for lvalue or rvalue

Assignment

Create a peephole optimizer for x86 assembly

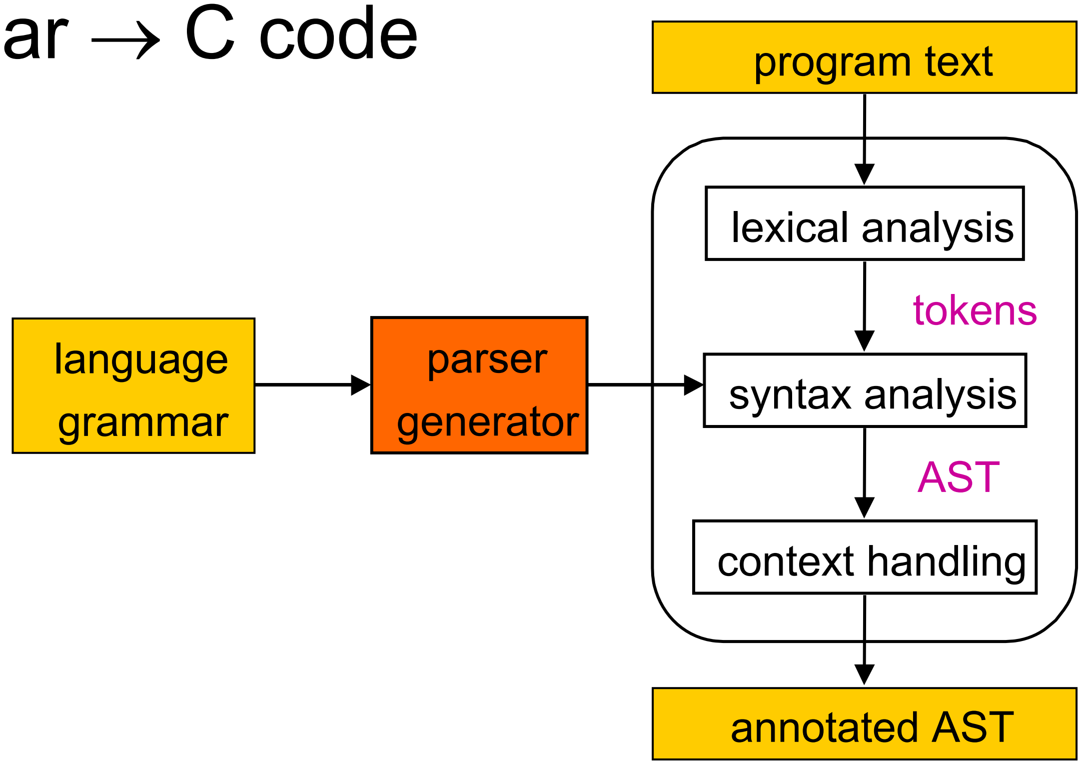


1. build compiler infrastructure / get acquainted with tools
2. optimize AST

Case study: desktop calculator

yacc (bison): parser generator for UNIX

- LALR(1) grammar → C code



Yet another compiler compiler

- format of the yacc input file:

```
definitions    <——  tokens + properties
%%
rules          <——  grammar rules + actions
%%
user code      <——  auxiliary C-code
```

Yacc-based expression interpreter

- input file

```
%token DIGIT

%%
line : expr '\n'            { printf("%d\n", $1);}
     ;
expr : expr '+' expr        { $$ = $1 + $3;}
     | expr '*' expr        { $$ = $1 * $3;}
     | '(' expr ')'         { $$ = $2;}
     | DIGIT
     ;
%%
```

grammar semantics

- yacc maintains a stack of “values” that may be referenced ($i) in the semantic actions

Yacc interface to lexical analyzer

- yacc invokes yylex() to get the next token
- the "value" of a token must be stored in the global variable yylval
- the default value type is int, but can be changed

```
%%
yylex()
{
        int c;

        c = getchar();

        if (isdigit(c)) {
                yylval = c - '0';
                return DIGIT;
        }
        return c;
}
```

Yacc interface to back-end

- yacc generates a function named yyparse()
- syntax errors are reported by invoking a callback function yyerror()

```
%%
yylex()
{
    ...
}
main()
{
    yyparse();
}

yyerror()
{
    printf("syntax error\n");
    exit(1);
}
```

Yacc-based expression interpreter

- input file (desk0.y)

```
%%
line : expr '\n'            { printf("%d\n", $1);}
     ;
expr : expr '+' expr        { $$ = $1 + $3;}
     | expr '*' expr        { $$ = $1 * $3;}
     | '(' expr ')'         { $$ = $2;}
     | DIGIT
     ;
%%
```

- run yacc

```
> make desk0
bison -v desk0.y
desk0.y contains 4 shift/reduce conflicts.
gcc -o desk0 desk0.tab.c
>
```

Yacc-based expression interpreter

- input file (desk0.y)

```
%%
line : expr '\n'          { printf("%d\n", $1);}
     ;
expr : expr '+' expr      { $$ = $1 + $3;}
     | expr '*' expr      { $$ = $1 * $3;}
     | '(' expr ')'       { $$ = $2;}
     | DIGIT
     ;
%%
```

- run yacc
- run desk0, is it correct? NO

```
> desk0
2*3+4
14
```

operator precedence:

Please Excuse My Dear Aunt Sally

Operator precedence in Yacc

priority from top (low) to bottom (high)

```
%token DIGIT
%left '+'
%left '*'

%%
line : expr '\n'           { printf("%d\n", $1);}
     ;
expr : expr '+' expr       { $$ = $1 + $3;}
     | expr '*' expr       { $$ = $1 * $3;}
     | '(' expr ')'        { $$ = $2;}
     | DIGIT
     ;
%%
```

Exercise (5 min.)

multiple lines:

```
%%
lines: line
     | lines line
     ;
line : expr '\n'          { printf("%d\n", $1);}
     ;
expr : expr '+' expr      { $$ = $1 + $3;}
     | expr '*' expr      { $$ = $1 * $3;}
     | '(' expr ')'       { $$ = $2;}
     | DIGIT
     ;
%%
```

Extend the interpreter to a desk calculator with registers named a – z. Example input: v=3*(w+4)

Answers

Building an AST

```
%%
lines: line
     | lines line
     ;
line : expr '\n'             { printf("%d\n", $1);}
     ;
expr : REG '=' expr          { $$ = reg[$1] = $3;}
     | expr '+' expr         { $$ = $1 + $3;}
     | expr '*' expr         { $$ = $1 * $3;}
     | '(' expr ')'          { $$ = $2;}
     | REG                   { $$ = reg[$1];}
     | DIGIT
     ;
%%
```

wanted: new semantic actions that create a list of expressions

AST datatypes

- list of expressions
- 3 types of expressions
  - registers
  - digits
  - binary operators (‘=‘, ‘+’, ‘*’)

```
%{
#include "list.h"

typedef enum {
      EXPR_REG,
      EXPR_DIG,
      EXPR_BIN
} ExprType;

typedef struct Expr {
      ExprType type;
      int reg;
      int dig;
      char op; struct Expr *left, *right;
} Expr;

%}
```

Multiple yylval types

- yylval type (union)
- type specification per token and grammar rule

```
%union {
        Expr *expr;
        List *list;
        int Int;
}

%token <Int> DIGIT
%token <Int> REG
%right '='
%left '+'
%left '*'

%type <expr> expr
%type <expr> line
%type <list> lines
```

Building expression nodes

```
expr    : REG                { Expr *e = (Expr *) malloc( sizeof(Expr));
                               e->type = EXPR_REG;
                               e->reg = $1;
                               $$ = e;
                             }
        | DIGIT              { Expr *e = (Expr *) malloc( sizeof(Expr));
                               e->type = EXPR_DIG;
                               e->dig = $1;
                               $$ = e;
                             }
```

- allocate node memory
- fill in attributes
- set rule value

Building expression nodes

```
| REG '=' expr   { Expr *e = (Expr *) malloc( sizeof(Expr));
                   e->type = EXPR_BIN;
                   e->op = '=';
                   e->reg = $1;
                   e->right = $3;
                   $$ = e;
                 }
| expr '+' expr  { Expr *e = (Expr *) malloc( sizeof(Expr));
                   e->type = EXPR_BIN;
                   e->op = '+';
                   e->left = $1;
                   e->right = $3;
                   $$ = e;
                 }
| '(' expr ')'   { $$ = $2;}
;
```

Building the AST

```
program : lines             { AST = $1;}
        ;
lines   : line              { $$ = list_append( new_list(), $1);}
        | lines line        { $$ = list_append( $1, $2);}
        ;
line    : expr '\n‘         { $$ = $1;}
```

- accumulate expressions into a list
- store into a global variable

Handling token types

```
%%
yylex()
{   int c = getchar();

    if (isdigit(c)) {
        yylval.Int = c - '0';
        return DIGIT;
    } else if ('a' <= c && c <= 'z') {
        yylval.Int = c - 'a';
        return REG;
    }
    return c;
}
```

- annotate yylval with attribute(s)

Processing the AST

```
print( List *list)
{
        int i;

        for (i=0; i < list_size(list); i++) {
            printf( "%d\n", eval( list_index( list, i)));
        }
}

main()
{
        yyparse();
        print( AST);
}
```

Exercise (5 min.)

```
eval( Expr *e)
{

        ???

}
```

- write the code to evaluate an expression

Answers