

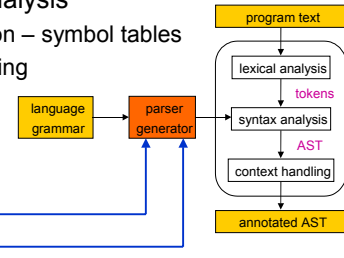
# Compiler construction in4020 – lecture 5

## Koen Langendoen

**Delft University of Technology**  
**The Netherlands**

## Overview

- semantic analysis
  - identification – symbol tables
  - type checking



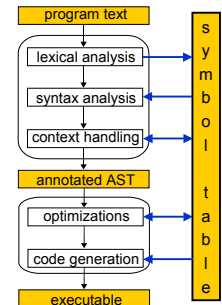
- assignment
  - yacc
  - LLgen

## 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  $\Rightarrow$  error
  - unused identifiers  $\Rightarrow$  warning
- check rules in the language definition
  - type checking
  - control flow (dead code)

## Symbol table

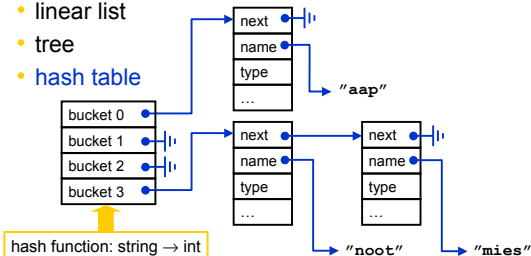
- global storage used by all compiler phases
- holds information about identifiers:
  - type
  - location
  - size
  -



## Symbol table implementation

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

graph LR
    next[next] --> node1
    subgraph node1 [ ]
        direction TB
        name1[name]
        type1[type]
    end
    name1 --> node2
    subgraph node2 [ ]
        direction TB
        name2[name]
        type2[type]
    end
    type1 --> node3
    subgraph node3 [ ]
        direction TB
        name3[name]
        type3[type]
    end
  
```



## Identification

- different kinds of identifiers
    - variables
    - type names
    - field selectors
  - name spaces
  - scopes
- ```
typed  
int j;  
void f  
{  
    st  
i: i.
```

```
typedef int i;

int j;

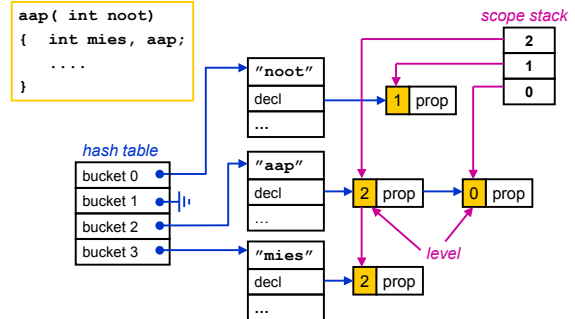
void foo(int j)
{
    struct i {i i; } i;

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



## 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;
```

- 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;
```

## Coercions

- implicit data and type conversion to match operand (argument) type
- coercions complicate identification (ambiguity)
- 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

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

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

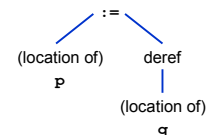
## Variable: value or location?

- two usages of variables
  - rvalue: value
  - lvalue: location

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

- insert coercion to dereference variable
- checking rules:

found	expected	
	lvalue	rvalue
lvalue	-	deref
rvalue	ERROR	-



## Exercise (5 min.)

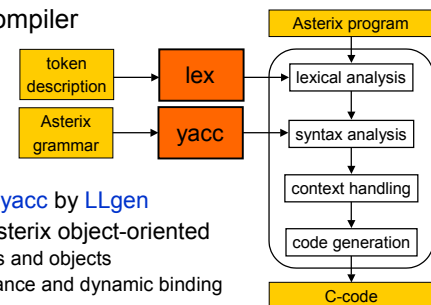
complete the table

expression construct	result kind (lvalue/rvalue)
constant	rvalue
identifier	
&lvalue	
*rvalue	
V[rvalue]	
V.selector	
rvalue + rvalue	
lvalue = rvalue	

V stands for lvalue or rvalue

## Assignment (practicum)

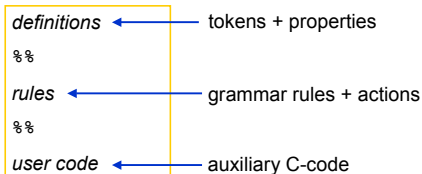
Asterix compiler



- replace yacc by LLgen
- make Asterix object-oriented
  - classes and objects
  - inheritance and dynamic binding

## Yet another compiler compiler

- yacc (bison): parser generator for UNIX
  - LALR(1) grammar → C code
- format of the yacc input file:



## 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 (\$) 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)) {
        yyval = 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 `yyperror()`

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

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

## Yacc-based expression interpreter

- input file (desk0)

```
%%
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)

```
%%
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 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 (7 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

## LLgen: LL(1) parser generator

- LLgen is part of the Amsterdam Compiler Kit
- takes LL(1) grammar + semantic actions in C and generates a recursive descent parser
- LLgen features:
  - repetition operators
  - advanced error handling
  - parameter passing
  - control over semantic actions
  - dynamic conflict resolvers

## LLgen example: expression interpreter

- start from LR(1) grammar
  - left recursion
  - operator precedence
- use repetition operators
- add semantic actions
  - attach parameters to grammar rules
  - insert C-code between the symbols

```
%token DIGIT;
main : [line]+
;
line : expr '\n'
;
expr : term [ '+' term ]*
;
term : factor [ '*' factor ]*
;
factor : '(' expr ')'
| DIGIT
;
```

LLgen

```
main : [line]+
;
line (int e;)
: expr(&e) '\n' { printf("%d\n", e); }
;
expr(int *e) (int t;)
: term(e)
[ '+' term(&t) { *e += t; }
]*
;
term(int *t) (int f;)
: factor(t)
[ '*' factor(&f) { *t *= f; }
]*
;
factor(int *f)
: '(' expr(f) ')'
| DIGIT { *f = yyval; }
;
```

grammar

semantics

## LLgen interface to lexical analyzer

- by default LLgen invokes `yylex()` to get the next token
- the "value" of a token can be stored in **any** global variable (`yyval`) of **any** type (int)

```
yylex()
{
    int c;
    c = getchar();
    if (isdigit(c)) {
        yyval = c - '0';
        return DIGIT;
    }
    return c;
}
```

## LLgen interface to back-end

- LLgen generates a user-named function (`parse`)
- LLgen handles syntax errors by inserting missing tokens and deleting unexpected tokens
- `LLmessage()` is invoked to notify the lexical analyzer

```
%start parse, main;
LLmessage(int class)
{
    switch (class) {
        case -1:
            printf("expecting EOF, ");
        case 0:
            printf("deleting token (%d)\n", LLsymb);
            break;
        default:
            /* push back token LLsymb */
            printf("inserting token (%d)\n", class);
            break;
    }
}
```

**Exercise (5 min.)**

- extend LLgen-based interpreter to a desk calculator with registers named a – z.  
Example input:  $v=3*(w+4)$

**Answers****Homework**

- study sections:
  - 2.2.4.6 LLgen
  - 2.2.5.9 yacc
- assignment 1:
  - replace yacc with LLgen
  - deadline April 9 08:59
- print handout for next week [blackboard]