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 ⇒ 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 = 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

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

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

A scoped hash-based symbol table

- 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

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;
```

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

Variable: value or location?
• two usages of variables
  rvalue: value
  lvalue: location
• insert coercion to dereference variable
• checking rules:

<table>
<thead>
<tr>
<th>found</th>
<th>expected</th>
</tr>
</thead>
<tbody>
<tr>
<td>lvalue</td>
<td>rvalue</td>
</tr>
<tr>
<td>rvalue</td>
<td>-</td>
</tr>
<tr>
<td>ERROR</td>
<td>-</td>
</tr>
</tbody>
</table>

Exercise (5 min.)
complete the table

<table>
<thead>
<tr>
<th>expression</th>
<th>construct</th>
<th>result kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>rvalue</td>
<td>rvalue</td>
</tr>
<tr>
<td>identifier</td>
<td>lvalue</td>
<td>lvalue</td>
</tr>
<tr>
<td>#value</td>
<td>rvalue</td>
<td>rvalue</td>
</tr>
<tr>
<td>V[value]</td>
<td>rvalue</td>
<td>rvalue</td>
</tr>
<tr>
<td>V.selector</td>
<td>lvalue</td>
<td>lvalue</td>
</tr>
<tr>
<td>rvalue + rvalue</td>
<td>rvalue</td>
<td>rvalue</td>
</tr>
<tr>
<td>lvalue + rvalue</td>
<td>rvalue</td>
<td>rvalue</td>
</tr>
</tbody>
</table>

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

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

Assignment (practicum)
Asterix compiler
1) replace yacc by LLgen
2) make Asterix object-oriented
   • classes and objects
   • inheritance and dynamic binding

Yacc-based expression interpreter
• input file

```
token DIGIT
"
line : expr \"\n" { printf("%d\n", $1);}
expr : expr '+' expr { $0 =$1 + $3;}
  | expr '*' expr { $0 =$1 * $3;}
  | '\(' expr ')\)' { $0 =$2;}
  | DIGIT
"
```
• yacc maintains a stack of “values” that may be referenced ($1) 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

```c
%%
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()`

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

### Yacc-based expression interpreter

- input file (desk0)
- run yacc

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

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

%%
```

#### Example

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

### Operator precedence in Yacc

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

---

### Exercise (7 min.)

- extend the interpreter to a desk calculator with registers named `a` – `z`. Example input: `v=3*(w+4)`
**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
- make grammar LL(1)
  - left recursion
  - operator precedence
- use repetition operators
- add semantic actions
  - attach parameters to grammar rules
  - insert C-code between the symbols

```c
%token DIGIT;

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

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

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 (yylval) of any type (int)

```c
yylex()
{
  int c;
  c = getchar();
  if (isdigit(c)) {
    yylval = 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

```c
%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]