CS415 Compilers
Context-Sensitive Analysis
Type checking
Symbol tables

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Homework 6 due on Monday, April 9

• Homework 7 will be posted on Wednesday, April 11

• Second project has been posted; due date: Monday, April 23

• Midterm will be returned in recitation
  Midterm sample solutions (A and B versions) have been posted; Please review the sample solutions, and if you have any remaining question, please see the TA/me grader for the problem:
    1: Rajanya
    2: Ulrich Kremer
    3: Chen
    4: Jeff

• Project 1 grades will be posted soon.
Add rules to compute the decimal value of a signed binary number

<table>
<thead>
<tr>
<th>Productions</th>
<th>Attribution Rules</th>
</tr>
</thead>
</table>
| Number → Sign List | List.pos ← 0  
If Sign.neg then Number.val ← – List.val  
else Number.val ← List.val |
| Sign → ± | Sign.neg ← false |
| | Sign.neg ← true |
| List₀ → List₁ Bit | List₁.pos ← List₀.pos + 1  
Bit.pos ← List₀.pos  
List₀.val ← List₁.val + Bit.val |
| | Bit.pos ← List.pos  
List.val ← Bit.val |
| Bit → 0 | Bit.val ← 0 |
| | Bit.val ← 2^{Bit.pos} |
**Attribute Grammars**

### Productions

<table>
<thead>
<tr>
<th>Production</th>
<th>Attribution Rules</th>
</tr>
</thead>
</table>
| $List_0 \rightarrow List_1 \ Bit$ | $List_1.pos \leftarrow List_0.pos + 1$  
 $Bit.pos \leftarrow List_0.pos$  
 $List_0.val \leftarrow List_1.val + Bit.val$ |

- **LIST**
  - $List_0$
  - $List_1$
  - BIT

- **pos**
- **val**

- semantic rules define partial dependency graph
- value flow top down or across: **inherited attributes**
- value flow bottom-up: **synthesized attributes**
Attribute Grammars

Note: • semantic rules associated with production $A \rightarrow \alpha$ have to specify the values for all
  - synthesized attributes for $A$ (root)
  - inherited attributes for grammar symbols in $\alpha$ (children)
  $\Rightarrow$ rules must specify local value flow!

• terminals can be associated with values returned by the scanner. These input values are associated with a synthesized attribute.
• Starting symbol cannot have inherited attributes.
Attribute grammars can specify context-sensitive actions
- Take values from syntax
- Perform computations with values
- Insert tests, logic, ...

**Synthesized Attributes**
- Use values from children & from constants
- **S-attributed** grammars: synthesized attributes only
- Evaluate in a single bottom-up pass

Good match to LR parsing

**Inherited Attributes**
- Use values from parent, constants, & siblings
- **L-attributed** grammars:
  - $A \rightarrow X_1 X_2 \ldots X_n$ and each inherited attribute of $X_i$ depends on
    - attributes of $X_1 X_2 \ldots X_{i-1}$, and
    - inherited attributes of $A$
- Evaluate in a single top-down pass (left to right)

Good match for LL parsing

$S$-attributed $\subseteq L$-attributed
The Moral of the Story

- Non-local computation needed lots of supporting rules
- “Complex” local computation is relatively easy

The Problems

- Copy rules increase cognitive overhead
- Copy rules increase space requirements
  - Need copies of attributes
- Result is an attributed tree
  - Must build the parse tree
  - Either search tree for answers or copy them to the root
Addressing the Problem

What would a good programmer do?

- Introduce a central repository for facts
- Table of names
  - Field in table for loaded/not loaded state
- Avoids all the copy rules, allocation & storage headaches
- All inter-assignment attribute flow is through table
  - Clean, efficient implementation
  - Good techniques for implementing the table (hashing, § B.4)
  - When its done, information is in the table!
  - Cures most of the problems
- Unfortunately, this design violates the functional, AG paradigm
  - Do we care?
The Realist’s Alternative

Ad-hoc syntax-directed translation

• Associate pieces of code with each production
• At each reduction, the corresponding code is executed
• Allowing arbitrary code provides complete flexibility
  → Includes ability to do tasteless & bad things

To make this work

• Need names for attributes of each symbol on lhs & rhs
  → Typically, one attribute passed through parser + arbitrary code
    (structures, globals, …)
  → Yacc introduced $$, $1, $2, … $n, left to right
• Need an evaluation scheme
  → Fits nicely into LR(1) parsing algorithm
parse.y :

```c
{%
#include <stdio.h>
#include "attr.h"
int yylex();
void yyerror(char * s);
#include "symtab.h"
%

union {tokentype token; }

token PROG PERIOD PROC VAR ARRAY RANGE OF
	token INT REAL DOUBLE WRITELN THEN ELSE IF
	token BEG END ASG NOT
	token EQ NEQ LT LEQ GEQ GT OR EXOR AND DIV NOT
	token <token> ID CCONST ICONST RCONST

=start program

%
program : PROG ID ';' block PERIOD { }
        ;
block : BEG ID ASG ICONST END { }
        ;
%
void yyerror(char* s) {
  fprintf(stderr,"%s\n",s);
}
int
main() {
  printf("1\t");
  yyparse();
  return 1;
}
```
You do not have to change the scanner (*scan.l*)

How to specify and use attributes in YACC?

- Define attributes as types in *attr.h*
  
  ```c
  typedef struct info_node {int a; int b} infonode;
  ```

- Include type attribute name in `%union` in *parse.y*
  
  ```c
  %union {tokentype token; infonode myinfo; ... }
  ```

- Assign attributes in *parse.y* to
  
  - Terminals: `%token <token> ID ICONST`
  - Non-terminals: `%type <myinfo> block variables procdecls cmpdstmt`

- Accessing attribute values in *parse.y*
  
  - use $$, $1, $2 ... etc. notation:
    
    ```c
    block : variables procdecls {$2.b = $1.b + 1;} cmpdstmt
    { $$a = $1.a + $2.a + $4.b;}
    ```
Summary: Is This Really “Ad-hoc”? 

Example on ilab: ~uli/cs415/spring18/examples/LexYacc

Relationship between practice and attribute grammars

Similarities
- Both rules & actions associated with productions
- Application order determined by tools
- (Somewhat) abstract names for symbols

Differences
- Actions applied as a unit; not true for AG rules
- Anything goes in ad-hoc actions; AG rules are (purely) functional
- AG rules are higher level than ad-hoc actions
Applications of SDT (Semantic Analysis)

- **Building a symbol table**
  - Enter declaration information as processed
  - At end of declaration syntax, do some post processing
  - Use table to check errors as parsing progresses

- **Simple error checking/type checking**
  - Define before use → lookup on reference
  - Dimension, type, ... → check as encountered
  - Type conformability of expression → bottom-up walk
  - Procedure interfaces are harder
    - Build a representation for parameter list & types
    - Check actual vs. formal parameter list
    - Positional or keyword associations

assumes table is global
A simple compiler

- Perform type checking (boolean and integer operations) → Error messages
- Generate ILOC code
- Lex & Yacc (C version)
**Type:** A set of values and meaningful operations on them

Types provide semantic “sanity checks” (consistency checks) and determine efficient implementations for data objects.

Types help identify:
- errors, if an operator is applied to an incompatible operand:
  - dereferencing of a non-pointer
  - adding a function to something
  - incorrect number of parameters to a procedure
  - ...
- which operation to use for overloaded names and operators, or what type coercion to use (e.g.: 3.0 + 1)
- identification of polymorphic functions
**Type system**: Each language construct (operator, expression, statement, …) is associated with a **type expression**. The type system is a collection of rules for assigning **type expressions** to these constructs.

**Type expressions** for
- basic types: integer, char, real, boolean, typeError
- constructed types, e.g., one-dimensional arrays:
  \[
  \text{array}(lb, \ ub, \ elem\_type)
  \]
  where elem_type is a **type expression**

A **type checker** implements a type system. It computes or “constructs” **type expressions** for each language construct
Example type inference rule:

\[
\begin{align*}
E & \vdash e_1 : \text{integer} & E & \vdash e_2 : \text{integer} \\
\hline
E & \vdash (e_1 + e_2) : \text{integer}
\end{align*}
\]

where \( E \) is a type environment that maps constants and variables to their type expressions.

Questions: How to specify rules that allow type coercion (type widening) from integers to reals in arithmetic expressions?

\[3.0 + 1 \quad \text{or} \quad 1 + 3.0\]
Example type inference rule pointer dereferencing:

$$\frac{E \vdash e : ???}{E \vdash *e : ???}$$

where $E$ is a type environment that maps constants and variables to their type expressions.
Example type inference rule pointer dereferencing:

\[
E \vdash e : \text{pointer(integer)} \\
\hline
E \vdash *e : \text{integer}
\]

where \( E \) is a type environment that maps constants and variables to their type expressions.

\text{pointer(...)} \text{ is now part of the type expression language such as array(...).}
Example type inference rule pointer dereferencing:

\[
\frac{E \vdash e : \text{pointer}(\beta)}{E \vdash *e : \beta}
\]

where \( E \) is a type environment that maps constants and variables to their type expressions.

Type expressions may also contain type variables such as \( \beta \). Type variables can denote any type expression.

Type variables are needed to express polymorphic types.
Example type inference rule address computation:

\[
E \vdash e : \text{integer} \\
E \vdash \&e : ???
\]

where \( E \) is a type environment that maps constants and variables to their type expressions.

What about a polymorphic version of this rule?
Formal proof that a program can be typed correctly.

```c
int a;
...
...*(&a) + 3 ...
```
Programmers may define their own types and give them names:

```c
type my_int is int;
...
int a;
my_int b;
...
... a + b ...
```

*Type names* can also be part of the *type expression language*. Note: *type names* and *type variables* are different!
Type Equivalence

Structural -- type equivalence: type names are expanded
Name -- type equivalence: type names are not expanded

Example:

\[
\begin{align*}
\text{type } A & \text{ is } \text{ array}(1..10) \text{ of integer}; \\
\text{type } B & \text{ is } \text{ array}(1..10) \text{ of integer}; \\
a & : A; \\
b & : B; \\
c, d & : \text{ array}(1..10) \text{ of integer}; \\
e & : \text{ array}(1..10) \text{ of integer};
\end{align*}
\]

Answer: structural equivalence:
name equivalence:
Type Equivalence

Structural -- type equivalence: type names are expanded
Name -- type equivalence: type names are not expanded

Example:

type A is  array(1..10) of integer;
type B is  array(1..10) of integer;
a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;

Answer: structural equivalence:    (a, b, c, d, e)
        name equivalence:          (a); (b); (c, d, e);
Revisit our type inference rule for “+”.

```c
exp : exp ' +' exp { if ($1 == TYPE_INT && $3 == TYPE_INT)
    $$ = TYPE_INT;
    else {
    $$ = TYPE_INT;
    printf("\n***Error: illegal operand types\n");
    }
}
```

PROJECT HINT: The definition of type expression as C types (structs) should be done in `attr.h`. `attr.c` may contain helper functions. The assignment of type expression C types to terminals and nonterminals of the grammar is done in `parse.y`. 
Lexically-scoped Symbol Tables

The problem
• The compiler needs a distinct record for each declaration
• Nested lexical scopes admit duplicate declarations

The interface
• `insert(name, level)` - creates record for `name` at `level`
• `lookup(name, level)` - returns pointer or index
• `delete(level)` - removes all names declared at `level`

Many implementation schemes have been proposed (see § B.4)
• We’ll stay at the conceptual level
• Hash table implementation is tricky, detailed, & fun

Symbol tables are compile-time structures the compiler use to resolve references to names. We’ll see the corresponding run-time structures that are used to establish addressability later.
procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure r {
            int x, y, z
            ...
        }
        procedure s {
            int x, a, v
            ...
        }
        ...
    }
    ...
}  

... q ...

B0: {  
    int a, b, c  
B1: {  
    int v, b, x, w  
B2: {  
    int x, y, z  
    ...
    }  
B3: {  
    int x, a, v  
    ...
    }  
    ...
    }
    ...
}  

...
Example

Picturing it as a series of Algol-like procedures

procedure p {
    int a, b, c
    procedure q {
        int v, b, x, w
        procedure r {
            int x, y, z
        }
    }
    procedure s {
        int x, a, v
    }
}

B0: {
    int a_0, b_1, c_2
    B1: {
        int v_3, b_4, x_5, w_6
        B2: {
            int x_7, y_8, z_9
        }
    }
    B3: {
        int x_{10}, a_{11}, v_{12}
    }
}

... r ... s

... q ...

a_{11}, b_4, c_2,
v_{12}, w_6, x_{10},
no y or z
Lexically-scoped Symbol Tables

High-level idea
• Create a new table for each scope
• Chain them together for lookup

“Chain of tables” implementation
• insert() may need to create table
• it always inserts at current level
• lookup() walks chain of tables & returns first occurrence of name
• delete() throws away table for level $p$, if it is top table in the chain

Individual tables can be hash tables.
Lexically-scoped Symbol Tables

High-level idea

- Create a new table for each scope
- Chain them together for lookup

Remember

If we add the subscripts, the relationship between the code and the table becomes clear.

The names visible in $s$:

- $a_{11}, b_4, c_2$
- $v_{12}, w_6, x_{10}$
- no $y$ or $z$
Implementing Lexically Scoped Symbol Tables

Stack organization

Implementation
- **insert ()** creates new level pointer if needed and inserts at nextFree
- **lookup ()** searches linearly from nextFree-1 forward
- **delete ()** sets nextFree to the equal the start location of the level deleted.

Advantage
- Uses **much** less space

Disadvantage
- Lookups can be expensive
Implementing Lexically Scoped Symbol Tables

Stack organization

\[ a_{11}, \ b_4, \ c_2, \ v_{12}, \ w_{6}, \ x_{10}, \ \text{no } y \ or \ z \]

Implementation

- **insert ()** creates new level pointer if needed and inserts at nextFree
- **lookup ()** searches linearly from nextFree-1 down stack
- **delete ()** sets nextFree to the equal the start location of the level deleted.

Advantage

- Uses **much** less space

Disadvantage

- Lookups can be expensive
Threaded stack organization

Implementation

- **insert()** puts new entry at the head of the list for the name
- **lookup()** goes direct to location
- **delete()** processes each element in level being deleted to remove from head of list

Advantage
- lookup is fast

Disadvantage
- delete takes time proportional to number of declared variables in level
Threaded stack organization

- Implementation
  - `insert()` puts new entry at the head of the list for the name
  - `lookup()` goes direct to location
  - `delete()` processes each element in level being deleted to remove from head of list

- Advantage
  - `lookup()` is fast

- Disadvantage
  - `delete()` takes time proportional to number of declared variables in level

```
<table>
<thead>
<tr>
<th></th>
<th>v 12</th>
<th>a 11</th>
<th>x 10</th>
<th>w 6</th>
<th>x 5</th>
<th>b 4</th>
<th>v 3</th>
<th>c 2</th>
<th>b 1</th>
<th>a 0</th>
</tr>
</thead>
</table>
```

```
h(x)
a_{11}, b_4, c_2, v_{12}, w_6, x_{10}, no y or z
```
Start working on the project!

Intermediate representations
Read EaC: Chapter 5.1 - 5.3

Code generation
Read EaC: Chapter 7.1 - 7.5