Derek White
- dwhite@uta.edu
- ELB 336
- MW 1:30-3
Join the mailing list

http://groups.google.com/group/uta-cse-5317
Parsing review

Recursive descent
- hand coded parser directly encodes the grammar (usually LL(1)) into a series of recursive functions

LL(k)
- top-down parser, must be able to decide which rule to use based on next k tokens in the input
  - pick A -> b or A -> c ?

LR(k)
- bottom-up parser, must be able to decide whether to apply a rule after seeing a string derived from the rule + k tokens of lookahead
  - pick A -> b or B -> b ?
ASTs and visitors

Parser turns sequence of bytes into representation of the program, usually *abstract syntax trees (ASTs)*

**ASTs**
- nodes represent constructs in the language
- node class for each syntactic construct
  - Program, Int, Var, Call, ArrayAccess, Binary, Assign, If, While, ...

**Visitors**
- classes that implement operations over the AST
- avoids having to modify the AST classes whenever a new operation is required

JTB can generate ASTs and Visitor interfaces from a grammar
Semantic analysis

- Parser discovers the *syntactic structure* of the program

- Semantic analysis:
  - interprets *meaning* of the program based on its structures
  - derives *context-sensitive* information (parser is context-free)

- Prepares the way for generating IR for the back end
Some questions to answer

- What is the type of x?
- Is x declared before it is used?
- Is x defined before it is used?
- Are there any names declared but not used?
- What declaration of x does this use refer to?
- Is an expression well-typed?
- Where can x be stored? Stack? Heap?
- Is an array access in bounds?
- Does function f produce a constant value?
- What classes implement method m?
- What classes might an object stored in x implement?
Why is semantic analysis hard?

- Answers depend on *values*, not syntax
- Answers may need non-local information
- Answers may require significant computation
Symbol tables

- Central store for information about the program

- Map from lexical *names* (symbols) to *attributes*

- What is saved in the symbol table?
  - variable names
  - types
  - defined constants
  - function and method names
  - literal constants and strings
  - labels
What else?

- textual name
- data type
- dimension information (for arrays)
- declaring method
- scope of the declaration
- storage class (heap vs. stack)
- offset in storage
- pointer to structure table (specifying how fields are laid out)
- is this parameter passed by-reference? by-value?
- can it be aliased? to what other names?
- number and type of arguments to functions
Nested scopes

- When we ask for a name, want the *most recent* declaration
- Declaration may be from the current scope or an enclosing scope
- *Innermost* scope overrides declarations from outer scopes
- OO features (inheritance) complicate things further

```java
class C {
    int x;
    void m() {
        int x = 3;
        return x + 1;
    }
    void n() { return x; }
}

class D {
    int x;
    class E {
        void q() { return x; }
    }
    class F extends C {
        void p() {
            return x + 1;
        }
    }
}
```
Nested scopes

- When using the symbol table need to keep track of the innermost scope

- Operations:
  - put(key, value) – binds key to value
  - get(key) – returns value bound to key
  - pushScope() – remembers current state of table
  - popScope() – restores table to previous scope

- Actual implementations may have a richer interface
class DupVarChecker extends Visitor {
    Env env;

    // initialize with global symtab
    DupVarChecker(Env env) { this.env = env; }

    void visit(LocalDecl n) {
        if (env.get(n.name) != null)
            error("duplicate var " + n.name);
        env.put(n.name, n.type);
    }

    void visit(MethodDecl n) {
        env.pushScope();
        n.body.accept(this);
        env.popScope();
    }
}
Using symtab in a visitor, better

```java
class DupVarChecker extends ArgVisitor<Env> {
    DupVarChecker() {
    }

    void visit(LocalDecl n, Env env) {
        if (env.get(n.name) != null)
            error("duplicate var " + n.name);
        env.put(n.name, n.type);
    }

    void visit(MethodDecl n, Env env) {
        n.body.accept(this, new Env(this));
    }
}
```

“functional push” – create a new symtab on push
Advantage: don’t have to remember to pop!
Uses call stack as the stack of scopes.
class Env {
    HashMap<String, Object> map;
    Env outer;

    Env() { outer = null; map = new HashMap(); }
    Env(Env o) { outer = o; map = new HashMap(); }

    void put(String key, Object value) { map.put(key, value); }
    Object get(String key) {
        if (map.containsKey(key)) {
            return map.get(key);
        }
        if (outer != null) {
            return outer.get(key);
        }
        return null;
    }

    Env push() { return new Env(this); }
    Env pop() { return outer; }
}
boolean dup(int[] A) {
    for (int i = 0; i < A.length; i++) {
        for (int j = i + 1; j < A.length; j++) {
            if (A[i] == A[j]) return true;
        }
    }
    return false;
}
Attributes

Internal representation of declarations
Symbol table associates names with attributes

Names may have different attributes depending on their meaning
- variables: type, procedure level, frame offset (later!)
- types: type descriptor, data size/alignment
- constants: type, value
- procedures: formals (names/types), result type, block information (local decls.), frame size
**Type checking**

Goal of type checking is to compute types for each expression and to check compatibility of types.

- **if (e) S1 else S2**: is e a boolean?
- **y = z**: is z’s type a subtype of y’s?
- **x + 4**: if x is an int, x+4 is also an int

Use symbol table to store type of declaration for later use.
Representation of types

Type descriptor
- Compile-time structure that represents a type expression
- Think: AST for a type expression

```plaintext
int                     Int
C*                     Pointer(TypeName("C")))
double[]               Array(Double)
int => char * char     Function(Int, Product(Char, Char))
```
Type compatibility

Part of type-checking requires testing if two types are compatible

Often, can just do structural equivalence of the type descriptors

In OO languages, type compatibility involves subtyping

Gets more complicated with type aliases (typedef)
or in languages with more than one way to write the same type
- Java: java.lang.Object vs. Object
- X10: Array{length==10} vs. Array{length==5*2}
Subtyping

Set-theoretic view of types
- A type is a set of values
- \( T_1 \) is a subtype of \( T_2 \) if \( T_1 \)'s set of values is a subset of \( T_2 \)'s

Substitution principle:
- Can always substitute an instance of a subtype for an instance of a supertype and the program will still type-check

\[
\text{new Bird()} \quad \rightarrow \quad \text{new Penguin()}
\]

Liskov Substitution Principle:
- if \( P(x) \) is true about objects \( x \) of type \( T \), then \( P(y) \) is true for objects \( y \) of type \( S \), a subtype of \( T \)
- “behavioral subtyping”
- usually too strong to be enforceable
Notation:
- $T_1 <: T_2$ – $T_1$ is a subtype of $T_2$

$:=$ is:
- reflexive: $T <: T$
- transitive: if $T_1 <: T_2$ and $T_2 <: T_3$, then $T_1 <: T_3$
- antisymmetric: if $T_1 <: T_2$ and $T_2 <: T_1$, then $T_1 = T_2$
### Type-checking assignment

<table>
<thead>
<tr>
<th>if x has type T1 e has type T2 then require T2 &lt;: T1 x = e has type T1</th>
</tr>
</thead>
</table>

Let $T <: S$

$$S \ s = \ldots;$$
$$T \ t = \ldots;$$

$s = t; \quad // \text{allowed}$
$t = s; \quad // \text{not allowed}$
Type-checking assignment

x has type T1
if
e has type T2
T2 <: T1
then
x = e has type T1

Let T <: S

S s = ...;
T t = ...;

s = t; // allowed
t = s; // not allowed
Subtyping in Java

Primitives:
- <: is =
- int <: int
- *Why not* int <: long?

Classes:
- class C
  - C <: Object
- class C extends D
  - C <: D

Interfaces:
- interface I
  - I <: Object
- class C implements I
  - C <: I

Arrays:
- int[] <: Object
- C[] <: Object
- if T1 <: T2 then T1[] <: T2[]
  - Note: this rule is broken
  - Why?
Covariant array subtyping

In Java:
- `Integer[] <: Object[]`

```java
Integer[] x = new Integer[10];
Object[] y = x;
y[0] = "this is a string, not an integer";
```

Will cause an ArrayStoreException at run-time
Typing judgments

Can formalize type-checking as a set of *typing rules*

Expressions are type-checked in an *environment A*

- environment is a function from variables to types
- implemented as part of the symbol table

Typing judgment:

\[ A \vdash e : T \]

“in environment A, expression e has type T”
Typing environment

aka “environment” or “typing context”

environment $A$ is a function from variables to types

in $A$, a declaration `int x`, adds $x \rightarrow \text{int}$ to $A$

usually write $A$ as a set:
- $A = \{\} \rightarrow \text{the empty environment}$
- $A = \{x_1: T_1, \ldots, x_n: T_n\} \rightarrow \text{environment where } A(x_i) = T_i$

environment implemented as part of the symbol table
Typing rules

Can formalize type-checking as a set of *typing rules*

Inference rule:

\[
\begin{array}{c}
P1 \\
\vdots \\
\text{...} \\
Pn \\
\end{array} \quad \Rightarrow \quad Q
\]

“if \( P1 \) and \( P2 \) and ... and \( Pn \), then \( Q \)”

Each \( P, Q \) is a typing judgment or another proposition
Assignment rule

So the rule for assignment can be written:

\[
\begin{align*}
A \vdash e_1 : T_1 & \quad A \vdash e_2 : T_2 & \quad T_2 \lessdot T_1 \\
\hline
A \vdash e_1 = e_2 : T_1
\end{align*}
\]
More rules

Constants

\[ A \vdash n : \text{int} \]
\[ A \vdash \text{true} : \text{boolean} \]
\[ A \vdash \text{false} : \text{boolean} \]

Variables

\[ A \vdash x : A(x) \]
Implementing the rules

class TypeChecker extends FuncVisitor<Env,TypeObj> {
  TypeObj visit(Var n, Env env) {
    TypeObj t = env.get(n.name);
    if (t == null)
      throw new TypeError("undeclared var");
    return t;
  }

  TypeObj visit(Int n, Env env) {
    return TypeObj.INT;
  }

  TypeObj visit(Bool n, Env env) {
    return TypeObj.BOOLEAN;
  }

  ...
}

TypeChecker visits the AST

visit method for node class N implements the typing rule for N, returning the computed type
Variables

\[
A \vdash e_1 : \text{int} \quad \quad A \vdash e_2 : \text{int} \\
\hline
A \vdash e_1 + e_2 : \text{int}
\]

```java
TypeObj visit(Add n, Env env) {
    // visit the the children first to get their types
    TypeObj t1 = n.left.accept(this, env);
    TypeObj t2 = n.right.accept(this, env);
    if (t1.equals(TypeObj.INT) && t2.equals(TypeObj.INT))
        return TypeObj.INT;
    else
        throw new TypeError("Cannot add " + t1 + " and " + t2);
}
```
Checking if

\[ A \vdash e : \text{boolean} \quad A \vdash s_1 : \text{void} \quad A \vdash s_2 : \text{void} \]

\[ \text{if (e) s}_1 \text{ else } s_2 : \text{void} \]

Introduce void “type” to indicate that a statement is well-typed

```java
TypeObj visit(If n, Env env) {
    // visit the the children first to get their types
    TypeObj t0 = n.cond.accept(this, env);
    n.then.accept(this, env); // can discard the result
    n.elsx.accept(this, env);

    if (t0.equals(TypeObj.BOOLEAN))
        return TypeObj.VOID;
    else
        throw new TypeError("Condition of if must be boolean");
}
```
Checking arrays

A ⊨ e : int

T is a type

A ⊨ new T[e] : T[

TypeObj visit(NewArray n, Env env) {
  TypeObj t0 = n.size.accept(this, env);

  if (t0.equals(TypeObj.INT))
    return TypeObj.arrayType(n.baseType);
  else
    throw new TypeError("Array size must be int");
}

Checking arrays

\[ A \vdash e_1 : T[ \] \quad A \vdash e_2 : \text{int} \]

\[ A \vdash e_1[e_2] : T \]

```
TypeObj visit(ArrayAccess n, Env env) {
    TypeObj t1 = n.array.accept(this, env);
    TypeObj t2 = n.index.accept(this, env);

    if (t1.isArray() && t2.equals(TypeObj.INT))
        return t1.baseType();
    else
        throw new TypeError("Array size must be int");
}
```
Checking field accesses

\[ A \vdash e : C \]
\[ \text{C has field T f} \]
\[ \text{----------------------------------------} \]
\[ A \vdash e.f : T \]

Details next week
Checking calls and new

\[ A \vdash e_0 : C \quad A \vdash e_i : T_i \quad \text{C has method} \quad T \ m(T_1, \ldots, T_n) \]

\[ A \vdash e_0.m(e_1, \ldots, e_n) : T \]

\[ \text{C is a class} \quad A \vdash e_i : T_i \quad \text{C has constructor} \quad C(T_1, \ldots, T_n) \]

\[ A \vdash \text{new } C(e_1, \ldots, e_n) : T \]
Checking a method declaration

T, all Ti are types \{this: C, x1: T1, ..., xn: Tn\} S returns T
\[ \vdash S : \text{void} \]
\[ \text{T } m(T1 x1, ..., Tn xn) \{ S \} \text{ is well-formed in class C} \]

More on this next week
Symbol table needs to keep track of:
- currently enclosing class
- whether in a static method, or non-static method
  - since “this” is not allowed in static methods
- information about each class
  - usually in a separate class table
Maps class names to class table entries:

- class name
- enclosing class, package (if any)
- superclass name (used to lookup class table entry)
- superinterface names
- method names, parameter types, return type, access flags, exceptions thrown, other modifiers (synchronized, native)
- field names, types, constant values, access flags
- member class names
Questions?