Context-sensitive Analysis, II

Ad-hoc syntax-directed translation, Symbol Tables, andTypes

Remember the Example from Last Lecture?

Grammar for a basic block *(§ 4.3.3)*

$Block_0$	$\rightarrow$	$Block_1$ *Assign*
	|	*Assign*
Assign	$\rightarrow$	*Ident* = *Expr* ;
$Expr_0$	$\rightarrow$	$Expr_1$ + *Term*
	|	$Expr_1$ – *Term*
	|	*Term*
$Term_0$	$\rightarrow$	$Term_1$ * *Factor*
	|	$Term_1$ / *Factor*
	|	*Factor*
Factor	$\rightarrow$	(*Expr*)
	|	*Number*
	|	*Identifier*

Let's estimate cycle counts

- **Each operation has a COST**
- **Add them, bottom up**
- **Assume a load per value**
- **Assume no reuse**

Simple problem for an *AG*

Hey, this looks useful !

And Its Extensions

Tracking loads

- Introduced *Before* and *After* sets to record loads
- Added $\geq$ 2 copy rules per production
  - → Serialized evaluation into execution order
- Made the whole attribute grammar large & cumbersome

The Moral of the Story

- Non-local computation needed lots of supporting rules
- Complex local computation was relatively easy

The Problems

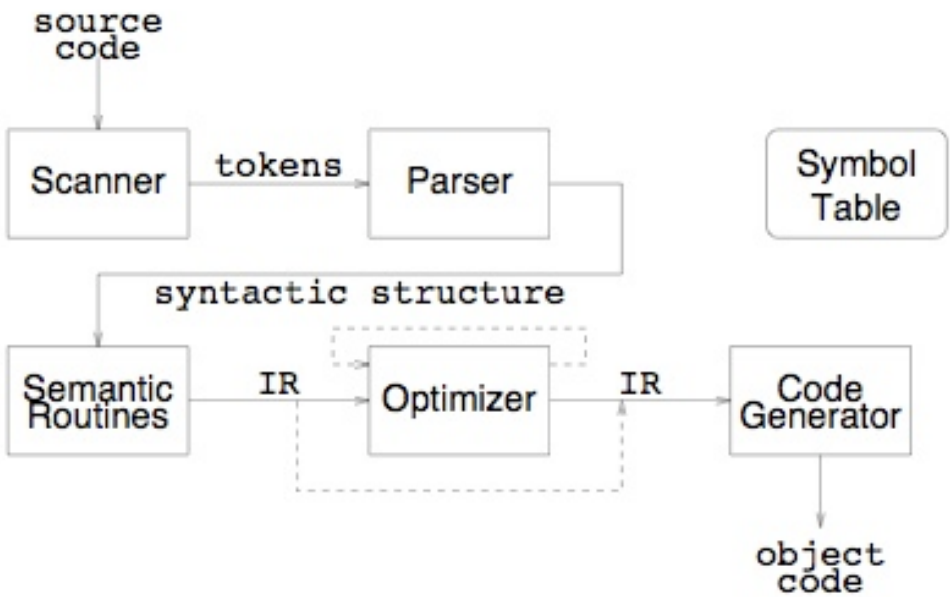
- Copy rules increase complexity
  - → Hard to understand and maintain
- Copy rules increase space requirements
  - → Need copies of attributes
  - → Can use pointers, but harder to understand

Addressing the Problem

If you gave this problem to a programmer at IBM

- 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.3)*
  - → When its done, information is in the table !
  - → Cures most of the problems
- Unfortunately, this design violates the functional paradigm
  - → Do we care?

Remind ourselves of Compiler Phases



Different Phases of Project

Phase I: Scanner
Phase II: Parser
Phase III: Semantic Routines
Phase IV: Code Generator

The Realist's Alternative

Ad-hoc syntax-directed translation

- Associate a snippet of code with each production
- At each reduction, the corresponding snippet runs
- 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, statics, …)
- Need an evaluation scheme
  - → Fits nicely into **LR(1)** parsing algorithm

Reworking the Example (*with load tracking*)

$Block_0 \rightarrow Block_1$ *Assign*	
	Assign	
Assign → Ident = Expr ;	cost← cost + COST(store);
$Expr_0 \rightarrow Expr_1$ + Term	cost← cost + COST(add);
	$Expr_1$ - Term	cost← cost + COST(sub);
	Term	
$Term_0 \rightarrow Term_1$ * Factor	cost← cost + COST(mult);
	$Term_1$ / Factor	cost← cost + COST(div);
	Factor	
Factor → (Expr)	
	Number	cost← cost + COST(loadi);

	Identifier	{ i← hash(Identifier);
		if (Table[i].loaded = false)
		then {
		cost ← cost + COST(load);
		Table[i].loaded ← true;
		}
		}

This looks simpler than the Attribute Grammar solution!

One missing detail: initializing cost

Reworking the Example (*with load tracking*)

Start	→	Init Block	
Init	→	ε	cost ← 0;
$Block_0$	→	$Block_1$ Assign	
	|	Assign	
Assign	→	Ident = Expr ;	cost ← cost + COST(store);

… and so on as in the previous version of the example …

- Before parser can reach Block, it must reduce Init
- Reduction by Init sets cost to zero

This is an example of splitting a production to create a reduction in the middle — for the sole purpose of hanging an action routine there!

Example — Building an Abstract Syntax Tree

- *Assume constructors for each node*
- *Assume stack holds pointers to nodes*

Goal	→	*Expr*	Goal.node = E.node;
Expr	→	*Expr + Term*	E_0.node= MakeAddNode(E_1.node,T.node);
	|	*Expr - Term*	E_0.node= MakeSubNode(E_1.node,T.node);
	|	*Term*	E.node = T.node;
Term	→	*Term * Factor*	T_0.node= MakeMulNode(T_1.node,F.node);
	|	*Term / Factor*	T_0.node= MakeDivNode(T_1.node,F.node);
	|	*Factor*	T.node = F.node;
Factor	→	<u>(</u> *Expr* <u>)</u>	F.node = Expr.node;
	|	<u>number</u>	F.node= MakeNumNode(token);
	|	<u>id</u>	F.node = MakeIdNode(token);

Reality

Most parsers are based on this *ad-hoc* style of context-sensitive analysis

Advantages

- Addresses shortcomings of Attribute Grammar paradigm
- Efficient, flexible

Disadvantages

- Must write the code with little assistance
- Programmer deals directly with the details

Most parser generators support a yacc/bison-like notation

Typical Uses

- 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

assumes table is global

- 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
    - ♦ Create list of sites to check
    - ♦ Check offline, or handle the cases for arbitrary orderings

Symbol Tables

- For compile-time efficiency, compilers use symbol tables
  - → Associates lexical names (symbols) with their attributes

- What items go in symbol tables?

  - → Variable names
  - → Defined constants
  - → Procedure/function/method names
  - → Literal constants and strings

  - → Separate layout for structure layouts
    - ♦ Field offsets and lengths

- A symbol table is a compile-time structure
- More after mid-term!

Attribute Information

- Attributes are internal representation of declarations
- Symbol table associates names with attributes
- Names may have different attributes depending on their meaning:
  - Variables: type, procedure level
  - Types: type descriptor, data size/alignment
  - Constants: type, value
  - Procedures: Signature (arguments/types) , result type, etc.

Type Systems

- Types
  - Values that share a set of common properties
  - Defined by language (built-ins) and/or programmer (user-defined)
- Type System
  - Set of types in a programming language
  - Rules that use types to specify program behavior

- Example type rules
  - If operands of addition are of type integer, then result is of type integer
  - The result of the unary "&" operator is a pointer to the object referred to by the operand
- Advantages
  - Ensures run-time safety
  - Provides information for code generation

Type Checker

- Enforces rules of the type system
- May be strong/weak, static/dynamic

- Static type checking
  - → Performed at compile time
  - → Early detection, no run-time overhead
  - → Not always possible (e.g., A[I], where I comes from input)

- Dynamic type checking
  - → Performed at run time
  - → More flexible, rapid prototyping
  - → Overhead to check run-time type tags

Type expressions

- Used to represent the type of a language construct
- Describes both language and programmer types

- Examples
  - → Basic types (built-ins) : integer, float, character
  - → Constructed types : arrays, structs, functions

A simple type checker

Using a synthesized attribute grammar, we will describe a type checker for arrays, pointers, statements, and functions.

Grammar for source language:

$$
\begin{array}{lll}
\mathrm{P} & ::= & \mathrm{D} \; ; \mathrm{E} \\
\mathrm{D} & ::= & \mathrm{D} \; ; \mathrm{E} \mid \textsf{id:}\ \mathrm{T} \\
\mathrm{T} & ::= & \textsf{char} \mid \textsf{integer} \mid \textsf{array [num] of } \mathrm{T} \mid \ \uparrow \mathrm{T} \\
\mathrm{E} & ::= & \textsf{literal} \mid \textsf{num} \mid \textsf{id} \mid \ \mathrm{E} \textsf{ mod } \mathrm{E} \mid \ \mathrm{E}[\mathrm{E}] \mid \ \mathrm{E} \uparrow
\end{array}
$$

- Basic types *char*, *integer*, *typeError*
- assume all arrays start at 1, *e.g.*,
  array [256] of char
  results in the type expression $array(1\ldots256, char)$
- $\uparrow$ builds a pointer type, so $\uparrow$ integer
  results in the type expression *pointer(integer)*

Type checking example

Partial attribute grammar for the type system

D	::=	id: T	$\{\ addtype(\text{id}.entry,\ T.type)\ \}$
T	::=	char	$\{\ T.type \leftarrow char\ \}$
T	::=	integer	$\{\ T.type \leftarrow integer\ \}$
T	::=	$\uparrow \mathrm{T}_1$	$\{\ T.type \leftarrow pointer(T_1.type)\ \}$
T	::=	array [num] of T_1	$\{\ T.type \leftarrow array(1\ldots\text{num}.val,\ T_1.type)\ \}$

Type checking expressions

Each expression is assigned a type using rules associated with the grammar.

E ::=	literal	$\{ E.type \leftarrow char \}$
E ::=	num	$\{ E.type \leftarrow integer \}$
E ::=	id	$\{ E.type \leftarrow lookup(\mathsf{id}.entry) \}$
E ::=	E_1 mod E_2	$\{ E.type \leftarrow$ if $E_1.type = integer$ and $E_2.type = integer$ then $integer$ else $typeError \}$
E ::=	$E_1[E_2]$	$\{ E.type \leftarrow$ if $E_2.type = integer$ and $E_1.type = array(s,t)$ then t else $typeError \}$
E ::=	$E_1 \uparrow$	$\{ E.type \leftarrow$ if $E_1.type = pointer$ then t else $typeError \}$

Type checking statements

Statements do not typically have values, therefore we assign them the type *void*. If an error is detected within the statement, it gets type *typeError*.

$$S \ ::= \ \mathsf{id} \leftarrow E \qquad \{\, S.type \leftarrow \ \mathsf{if}\ \mathsf{id}.type = E.type \ \mathsf{then}\ void \ \mathsf{else}\ typeError \,\}$$

$$S \ ::= \ \mathsf{if}\ E\ \mathsf{then}\ S_1 \qquad \{\, S.type \leftarrow \ \mathsf{if}\ E.type = boolean \ \mathsf{then}\ S_1.\mathrm{type} \ \mathsf{else}\ typeError \,\}$$

$$S \ ::= \ \mathsf{while}\ E\ \mathsf{do}\ S_1 \qquad \{\, S.type \leftarrow \ \mathsf{if}\ E.type = boolean \ \mathsf{then}\ S_1.\mathrm{type} \ \mathsf{else}\ typeError \,\}$$

Is This Really "Ad-hoc" ?

Relationship between practice and attribute grammars

Similarities

- Both rules & actions associated with productions
- Application order determined by tools, not author
- (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 functional
- AG rules are higher level than *ad-hoc* actions