CS480

Syntax Directed Translation

Ch 5

Page 279-307

March 11, 2013

Semantic Rules (see also ch 2)

- Attach attributes to grammar
  - Defined by semantic rules

Source Code → Grammar w/ Semantic Rules → New Stuff

  - Grammar directs the translation
  - Attribute: piece of data associated with:
    - a node in the parse tree
    - a nonterminal in the grammar
    - each NT can have zero or more attributes.
    - terminals can get attributes from the lexer

Semantic Rules

- Annotated/decorated parse tree/grammar

- Associated with each production is:
  - semantic rules for evaluating attributes **AND/OR**
  - semantic rules for producing side-effects (e.g. updating a global variable).

Attributes

- Given: each grammar production A **->** α has associated with it a set of semantic rules

  of the form $b := f(c_1, c_2, ..., c_k)$ where f is a function

  EITHER

  – *b* is a synthesized attribute of A (LHS)

  OR

  – *b* is an inherited attribute of one of the grammar symbols on the RHS of the production

Example

- Construct a simple grammar that can represent unsigned numbers.

Grammar

Semantic Rules

Example

Production	Semantic Rules
`decl -> datatype list`	`list.att = datatype.type`
`datatype -> int`	`datatype.type := integer`
`datatype -> float`	`datatype.type := real`

`list -> list1, id`	`list1.att := list.att`
	
	`stAdd (id, list.att)`

`list -> id`	`stAdd (id, list.att)`

Inherited? Synthesized? Side effects?

Implementation

- May be achieved with a top-down parse
  - Might add parameters/return values to functions td*Nonterminal*()
  - Might add some global data structures
  - Draw the parse tree and determine how the attributes flow!
    - Be aware of initialization!

```
int foo(int);

foo(foo(foo(foo(9))));
```

Local Variables

Print the type of each local variable

Where is each piece of data available?

```
int x, y, *z;
int arrayX[100];
```

integer x
integer y
integer pointer z
integer array arrayX

```
functionstmt -> { optdecllist stmtlist }

optdecllist  -> int idorptr vardecl ; optdecllist | ε
idorptr      -> id | * id

vardecl      -> [ constant ] optinit moreinitdecls
                | optinit moreinitdecls

optinit      -> = initializer | ε

initializer  -> EXPRESSION | { initlist }

initlist     -> EXPRESSION moreinit

moreinit     -> , EXPRESSION moreinit | ε

moreinitdecls-> , idorptr vardecl | ε
```

What if we had `float`, `char`, `int`?

Practice

- Construct a CFG that allows unsigned integers to be expressed in octal or decimal notation. An octal number is succeeded with an O and a decimal number is succeeded with a D.
  - 18O
  - 10D

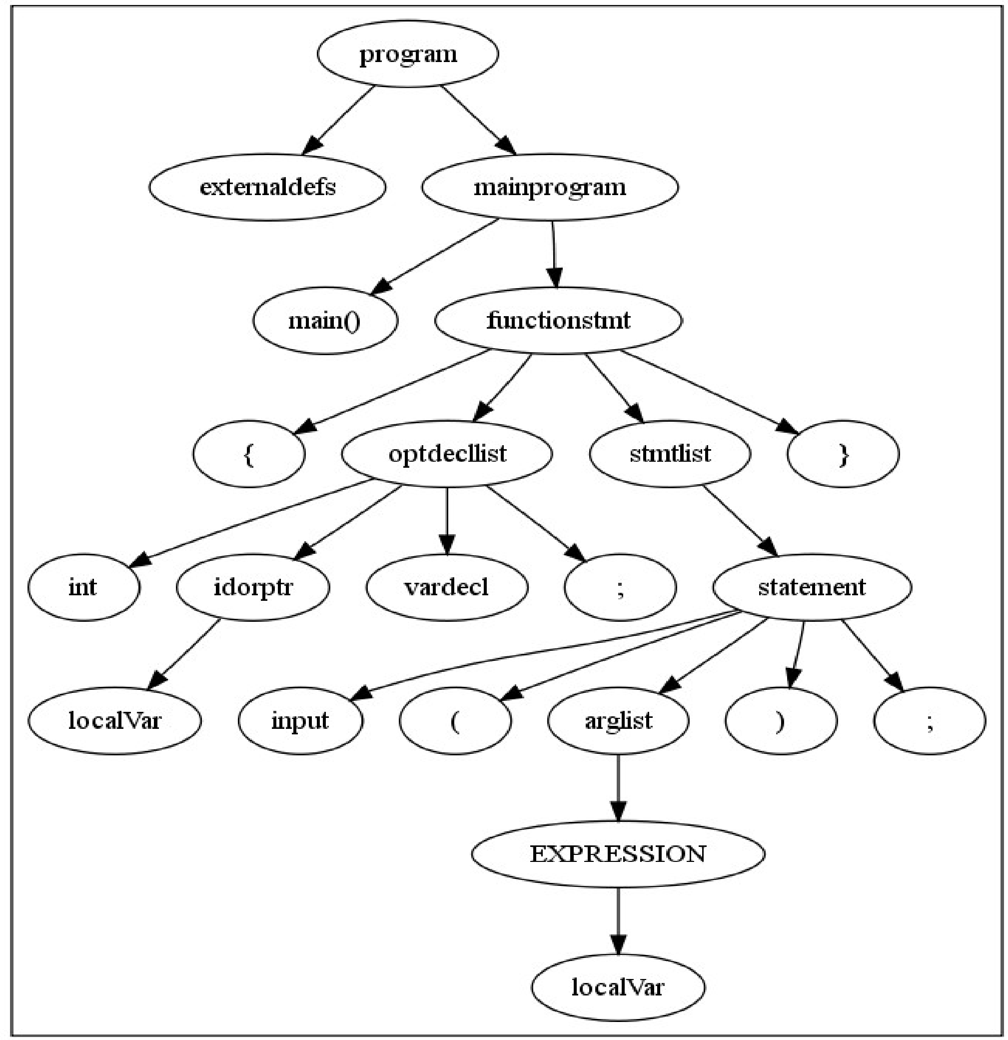
Other Examples

- Syntax Tree Printing
  - Using Dot (Graphviz) http://www.graphviz.org

```
main()
{
  int localVar;
  input(localVar);
}


digraph {

  subgraph clusterTD
  {
    program -> externaldefs;
    program -> mainprogram;
    mainprogram->main;
    mainprogram -> functionstmt;
    main [label="main()"]
    ...
```



Semantic Rules

Production	Semantic Rule
`program -> externaldefs mainprogram`	emit("program->externaldefs;"); emit("program->mainprogram;");
`mainprogram ->` **`main ( )`** `functionstmt`	emit("mainprogram->main( )"); emit("mainprogram->functionstmt");

OR

`program -> externaldefs mainprogram`	externaldefs.parent = "program" mainprogram.parent ="program"
`mainprogram ->` **`main ( )`** `functionstmt`	emit("mainprogram.parent ->mainprogram"); emit("mainprogram->main( )"); functionstmt.parent="mainprogram"

Type Checking (ch 6)

```
S -> A = E
A -> id
E -> E + T | T
T -> T * F | F
F -> id
```

Production	Semantic Rule
E → E + T	E.type = higher(E.type, T.type)
E → T	E.type = T.type
T → T * F	T.type = higher(T.type, F.type)
T → F	T.type = F.type
F → id	F.type =
A → id	
S → A = E	