

Semantic Analysis

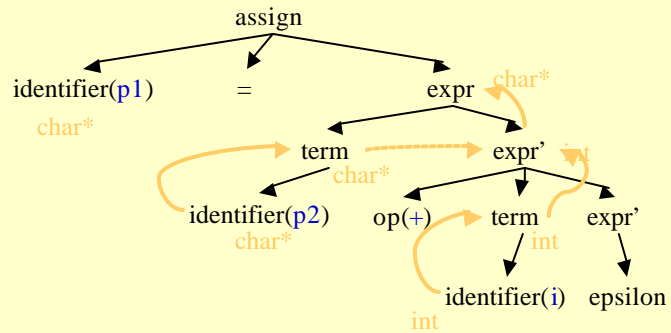
Amer Diwan

Scanning+Parsing versus semantic analysis

- In the syntax section we looked at how to recognize syntactically correct programs
- But you need to do more than that!
 - Check types
 - Pick overloaded operators
 - Resolve scopes
 - Generate code
 - ...

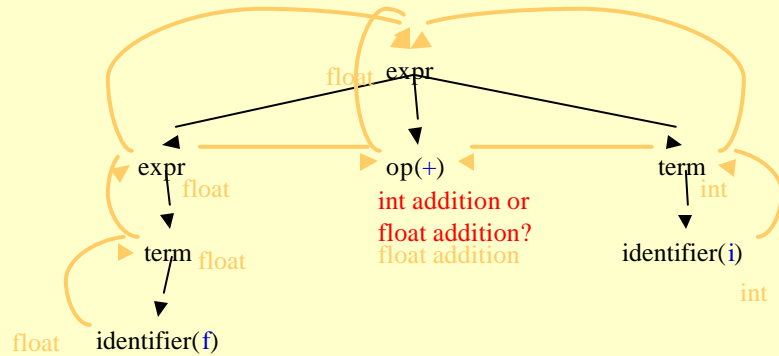
Type checking example 3

- char *p1, *p2; int i;...; p1 = p2 + i;
 expr -> term expr'
 expr' -> op term expr' | epsilon
 term -> identifier | number



Overload resolution example

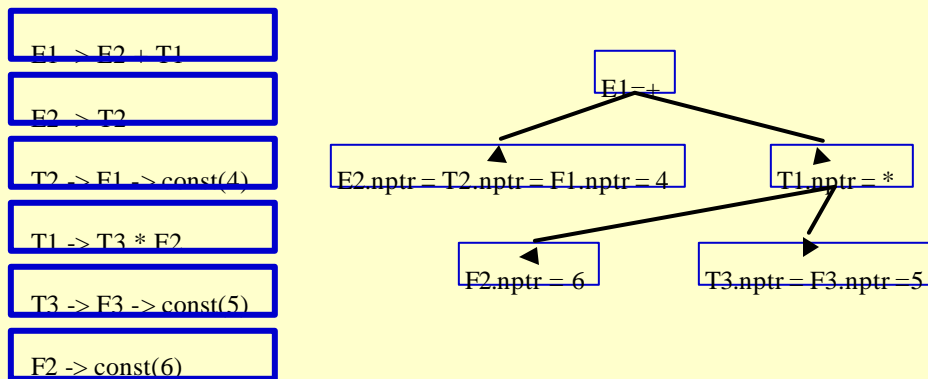
- float f; int i; ...; p1 = p2 + i;
 expr -> expr op term | term
 term -> identifier | number



Building AST

E -> E1 + T	E.nptr := mknode('+', E1.nptr, T.nptr)
E1 - T	E.nptr := mknode('-', E1.nptr, T.nptr)
T	E.nptr := T.nptr
T -> T1 * F	T.nptr := mknode('*', T1.nptr, F.nptr)
F	T.nptr := F.nptr;
F -> (E)	F.nptr := E.nptr
id	F.nptr := mkleaf(id, symtab entry of id)
const	F.nptr := mkleaf(const, value of const)

Example continued (4 + 5*6)



Type checking with the AST

char *p1, *p2; int i; ...; p1 = p2 + i;

program -> stmt

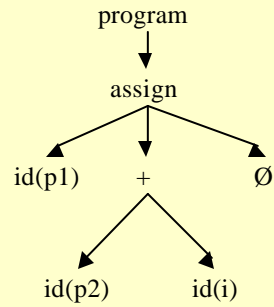
assign: stmt -> id expr stmt

'+' : expr -> node node

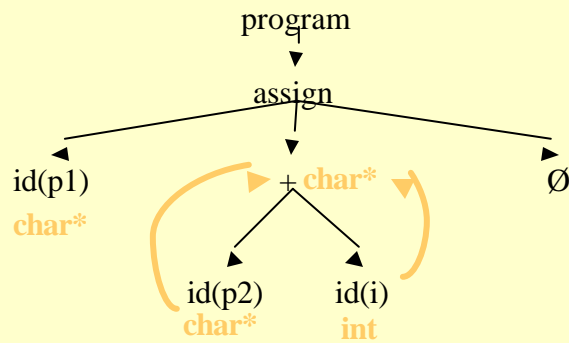
'-' : expr -> node node

id: expr -> epsilon

num: expr -> epsilon



Typechecking with the AST (cont)



Static versus dynamic semantics

- Static semantics
 - Enforced at compile time
 - Examples in previous slides are that of static semantics
- Dynamic semantics
 - Enforced at run time

Example of dynamic semantics

- Array bounds check
 - ```
for (i = 0; i < 100; ++i) {
 A[i] = 0;
}
```
  - In safe languages this translates into
  - ```
for (i = 0; i < 100; ++i) {  
    if (i in [A'first..A'last]) then  
        A[i] = 0;  
    else error;  
}
```
 - The check happens as the program is running!

An aside on dynamic semantics

- Compilers try hard to minimize effort to enforce dynamic semantics at run time:

```
- if [0..100) in [A'first..A'last] {  
    for (i = 0; i < 100; ++i) {  
        A[i] = 0;  
    }  
    else {  
        for (i = 0; i < 100; ++i) {  
            if (i in [A'first..A'last])  
                A[i]=0;  
            else error;  
        }  
    }  
}}
```

How to do semantic analysis

- Use attribute grammars
 - A formal system
 - Can generate code to propagate attributes automatically
- Ad-hoc: action routines
 - Sprinkle pieces of code that are executed when a terminal/non-terminal is matched
 - More operational than attribute grammars

Attribute grammars

- Productions + propagation of attributes
 - ‘+’: $\text{expr}_1 \rightarrow \text{expr}_2 \text{expr}_3$
 - $\text{op} := \text{selectOp}(\text{'+'}, \text{expr}_2.\text{ty}, \text{expr}_3.\text{ty})$
 - $\text{expr}_1.\text{ty} := \text{op.ty}$
 - selectOp : operator x type x type \rightarrow type
 - “ty” is an attribute that the above propagates

Attribute grammars: type of rules

- Copy
 - $F \rightarrow (E)$
 - $F.\text{val} := E.\text{val}$
- Invoke semantic functions
 - ‘+’: $\text{expr}_1 \rightarrow \text{expr}_2 \text{expr}_3$
 - $\text{op} := \text{selectOp}(\text{'+'}, \text{expr}_2.\text{ty}, \text{expr}_3.\text{ty})$
 - $\text{expr}_1.\text{ty} := \text{op.ty}$
 - selectOp : operator x type x type \rightarrow type
 - Semantic functions: primitives specified by the language designer

When to compute the attributes?

- ‘+’: $\text{expr}_1 \rightarrow \text{expr}_2 \text{expr}_3$
 $\text{expr}_1.\text{ty} := \text{op.ty}$
 $\text{op} := \text{selectOp}(\text{'+'}, \text{expr}_2.\text{ty}, \text{expr}_3.\text{ty})$
- The attribute grammar processor figures out the order in which to compute attributes:
 - e.g., op.ty before $\text{expr}_1.\text{ty}$

Attribute grammars:

Use them on abstract syntax or concrete syntax?

- Concrete syntax has many details that are irrelevant to semantics:
 - e.g., $F \rightarrow (E)$ useful for parsing with the correct precedence but otherwise useless...
- Abstract syntax
 - eliminates all the “useless” syntactic details
 - attribute grammar can be more concise!
- **Book incorrectly emphasizes AG on concrete syntax**

Using action routines

- An ad-hoc alternative to attribute grammars
 - Interleave code (action routines) with parsing
 - Often used to build AST during a parse

Example

- $E \rightarrow T \{ EE.arg := T.ptr \} EE \{ E.ptr := EE.ptr \}$
- $EE_1 \rightarrow + T \{ EE_2.arg := make_plus(EE_1.arg, T.ptr) \} EE_2$
 $\{ EE_1.ptr := EE_2.ptr; \}$
 | **epsilon** $\{ EE_1.ptr := EE_1.arg; \}$
- $T \rightarrow (E) \{ T.ptr := E.ptr; \}$
 | **number** $\{ T.ptr := make_leaf(number.val); \}$

Note that programmer has to specify order of evaluation

Relationship to reading

- Reading gives additional/longer examples of both attribute grammars and action routines
 - Helpful for project 1 and 2
- Discusses the differences between different kinds of attribute grammars in more detail

Relationship to other classes

- CSCI 4555
 - You will get experience with action routines when you use a parser generator
 - You will get to use attribute grammars for a compiler generator!

Next topic: Control flow

- Read chapter 6 (except 6.5.3 and 6.7)

Example building an abstract syntax tree

- $E1 \rightarrow E2 + T$ [E1.ptr := make_bin_op("+", E2.ptr, T.ptr)]
 $E \rightarrow T$ [E.ptr := T.ptr]
 $T1 \rightarrow T2 * F$ [T1.ptr := make_bin_op("*", T2.ptr, F.ptr)]
 $T \rightarrow F$ [T.ptr := F.ptr]
 $F \rightarrow (E)$ [F.ptr := E.ptr]
 $F \rightarrow \text{const}$ [F.ptr := make_leaf(const.val)]