Abstract Syntax

Mooly Sagiv

html://www.cs.tau.ac.il/~msagiv/courses/wcc08.html

Outline

- The general idea

- Cup

- Motivating example
  Interpreter for arithmetic expressions

- The need for abstract syntax

- Abstract syntax for arithmetic expressions

Semantic Analysis during Recursive Descent Parsing

- Scanner returns "semantic values" for some tokens

- The function of every non-terminal returns the "corresponding subtree value"

- When $A \rightarrow B\ C\ D$ is applied the function for A can use the values returned by B, C, and D

  – The function can also pass parameters, e.g., to $D()$, reflecting left contexts

$E \rightarrow \textbf{num } E'$

$E' \rightarrow \varepsilon$

$E' \rightarrow + \textbf{num } E'$

```
int E() {
  switch (tok) {
    case num :   temp=tok.val; eat(num);
                      return EP(temp);
    default: error(…); }}
int EP(int left) {
  switch (tok) {
    case $: return left;
    case + :   eat(+);
               temp=tok.val; eat(num);
               return EP(left + temp);
    default: error(…) ;}}
```
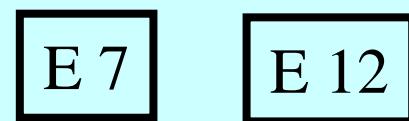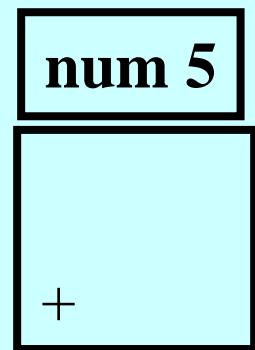
Semantic Analysis during Bottom-Up Parsing

- Scanner returns "semantic values" for some tokens

- Use parser stack to store the "corresponding subtree values"

- When $A \rightarrow B\ C\ D$ is reduced the function for A can use the values returned by B, C, and D

- No action in the middle of the rule

Example

$$E \rightarrow E + \textbf{num}$$

$$E \rightarrow \textbf{num}$$

num 5
+

| E 7 | | E 12 |

```
terminal Integer NUMBER;
terminal PLUS,MINUS,MULT,DIV;
terminal LPAREN, RPAREN;
terminal UMINUS;
non terminal Integer expr;
precedence left PLUS, MINUS;
precedence left DIV, MULT;
Precedence left UMINUS;
%%
expr ::= expr:e1 PLUS expr:e2
        {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
        | expr:e1 MINUS expr:e2
        {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
        | expr:e1 MULT expr:e2
        {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
        | expr:e1 DIV expr:e2
        {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
        | MINUS expr:e1 %prec UMINUS
        {: RESULT = new Integer(0 - e1.intValue(); :}
        | LPAREN expr:e1 RPAREN
        {: RESULT = e1; :}
        | NUMBER:n
        {: RESULT = n; :}
```
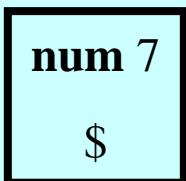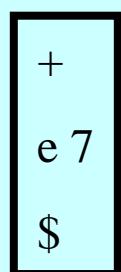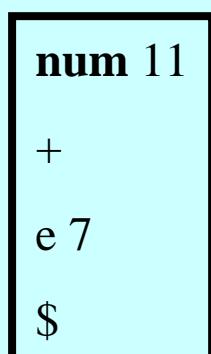
stack	input	action
$	7+11+17$	shift
num 7 $	+11+17$	reduce e → **num**
e 7 $	+11+17$	shift
+ e 7 $	11+17$	shift
num 11 + e 7 $	+17$	reduce e→ **num**
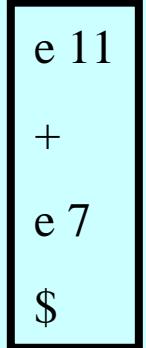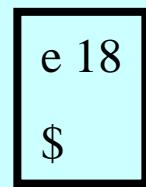
stack	input	action
e 11 + e 7 $	+17$	reduce e → e+e
e 18 $	+17$	shift
+ e 18 $	17$	shift
num 17 + e 18 $	$	reduce e→ num
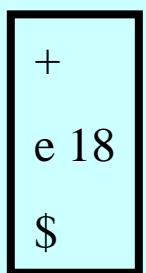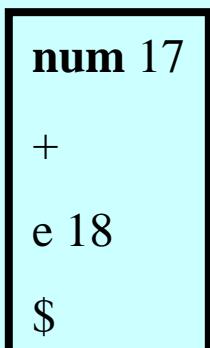
stack	input	action
e 17 + e 18 $	$	reduce e::= e+e
e 35 $	$	accept

So why can't we write

all the compiler code

in Bison/CUP?

Historical Perspective

- Originally parsers were written w/o tools
- yacc, bison, cup... make tools acceptable
- But it is still difficult to write compilers in parser actions (top-down and bottom-up)
  - Natural grammars are ambiguous
  - No modularity principle
  - Many useful programming language features prevent code generation while parsing
    - Use before declaration
    - gotos

Abstract Syntax

- Intermediate program representation

- Defines a tree - Preserves program hierarchy

- Generated by the parser

- Declared using an (ambiguous) context free grammar (relatively flat)

  – Not meant for parsing

- Keywords and punctuation symbols are not stored (Not relevant once the tree exists)

- Big programs can be also handled (possibly via virtual memory)

Issues

- Concrete vs. Abstract syntax tree

- Need to store concrete source position

- Abstract syntax can be defined by:

    - Ambiguous context free grammar

    - C recursive data type

    - "Constructor" functions

- Debugging routines linearize the tree

Abstract Syntax for Arithmetic Expressions

Exp →**id**	(IdExp)
Exp →**num**	(NumExp)
Exp →Exp Binop Exp	(BinOpExp)
Exp →Unop Exp	(UnOpExp)
Binop → +	(Plus)
Binop → **-**	(Minus)
Binop → *	(Times)
Binop → /	(Div)
Unop → -	(UnMin)

```
package Absyn;
abstract public class Absyn { public int pos ;}
Exp extends Absyn {} ;
class IdExp extends Exp { String rep ;
    IdExp(r) { rep = r ;}
}
class NumExp extends Exp { int number ;
    NumExp(int n) { number = n ;}
}
class OpExp {
    public final static int PLUS=1;  public final static int Minus=2;
    public final static int  Times=3;  public final static int  Div=4;
}
final static int OpExp.PLUS, OpExp.Minus, OpExp.Times, OpExp.Div;
class BinExp extends Exp {
    Exp left, right;    OpExp op ;
    BinExp(Exp l, OpExp o, Bin Exp r) {
        left = l ;  op = o; right = r ;
        }
}
```

```
terminal Integer NUMBER;
terminal PLUS,MINUS,MULT,DIV;
terminal LPAREN, RPAREN;
terminal UMINUS;
nonterminal Exp expr;
precedence left PLUS, MINUS;
precedence left DIV, MULT;
Precedence left UMINUS;
%%
expr ::= expr:e1 PLUS expr:e2
          {: RESULT = new BinExp( e1, OpExp.PLUS, e2); :}
          | expr:e1 MINUS expr:e2
          {: RESULT = new BinExp(e1, OpExp.MINUS,  e2); :}
          | expr:e1 MULT expr:e2
          {: RESULT = new BinExp(e1, OpExp.PLUS, e2); :}
          | expr:e1 DIV expr:e2
          {: RESULT = new BinExp(e1, OpExp.DIV,  e2); :}
          | MINUS expr:e1 %prec UMINUS
          {: RESULT = new BinExp(new NumExp(0), OpExp.MINUS,  e1); :}
          | LPAREN expr:e1 RPAREN
          {: RESULT = e1; :}
          | NUMBER:n
          {: RESULT = new NumExp(n.intValue()); :}
```

Summary

- Jlex(Jflex) and Cup simplify the task of writing compiler/interpreter front-ends
- Abstract syntax provides a clear interface with other compiler phases
  - Supports general programming languages
- But the design of an abstract syntax for a given PL may take some time