编译原理/Compiler Abstract Syntax Trees

周雅倩
复旦大学计算机科学技术学院
zhouyaqian@fudan.edu.cn
2018/11/9

References

编 译 原 理

- http://en.wikipedia.org/wiki/Abstract_syntax_tree
- http://www.stanford.edu/class/cs143/
  – Partial contents are copied from the slides of Alex Aiken

Outline

编 译 原 理

- Abstract syntax trees
- Construct syntax trees
  – By hand
  – Semantic rule
  – Translation Scheme

Intermediate Representation

编 译 原 理

- Source programs are sequences of ASCII characters
  – Sequence of characters can not be processed flexibly
  – Suitable data structures must be introduced to compilers
- There are several levels of intermediate representations in modern compilers
- Now we only introduce the first level

Intermediate Representation

编 译 原 理

- Two important data structures:
- Abstract syntax tree
  – Usually, it is used to represent statements and expressions
- Symbolic table
  – Often, it is used to represent information about symbols in the programs

Abstract Syntax Tree (1)

编 译 原 理

- Abbreviated as AST
- Fact#1
  – program constructs are recursive defined
- Fact#2
  – CFG is used to define program constructs
- Fact#3
  – Parse tree can be used to represent the internal structures of the program

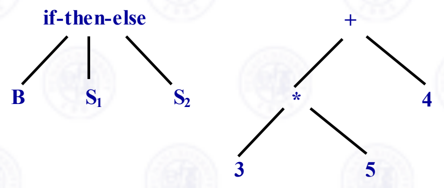
Abstract Syntax Tree (2)

编译原理

- Trees can be used to represent hierarchical structures
- It is very natural to use a tree as an intermediate representation
- An abstract syntax tree is a condensed form of parse tree

Abstract Syntax Tree (3)

编译原理



Syntax Tree Node (1)

编译原理

- We are very familiar with binary trees
- However, the syntax tree is different from binary tree
- Usually, a point to parent is used to facilitate traversing
- Two subtrees are not enough to represent most program constructs such as if_then_else, while, for, …

Syntax Tree Node (2)

编译原理

```
typedef sturct node {
    struct node *parent;
    enum node_type node_id;
    union {
        struct if_node              if_node;
        struct binary_node          binary_node;
        struct identify_node        id_node;
        struct num_node             num_node;
        .....................
    } u
} syn_tree_node ;
```

Subnodes (1)

编译原理

```
struct if_node {
                struct node *if_cond;
                struct node *then_part;
                struct node *else_part;
        } ;

struct binary_node {
        struct node *left_expr;
        struct node *right_expr;
} ;
```

Subnodes (2)

编译原理

```
struct id_node {
                struct sym_entry *id;
} ;

struct num_node {
        struct num num;
} ;
```

Syntax Tree Node (3)

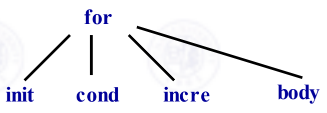
编 译 原 理

- For a binary expression, the syntax tree is almost a binary tree
- For other constructs, n-nary trees are introduced according to the number of sub-components in the constructs

Syntax Tree Node (4)

编 译 原 理

- A compound statement
  - A list of statements
- A comma expression
  - A list of expressions
- A for statement

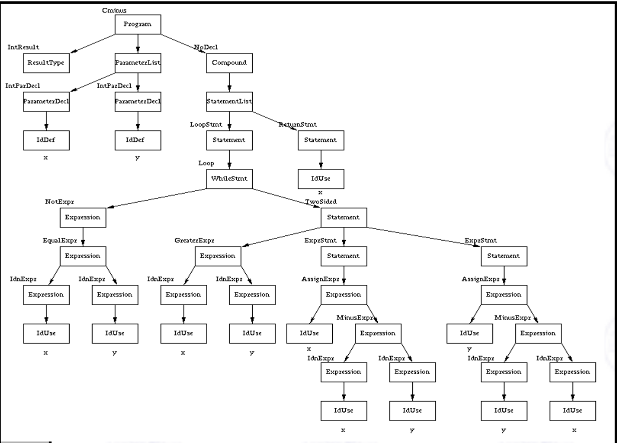


Example

编 译 原 理

- C function:

```
int GCD(int x, int y)
{
    while (!(x == y))
        if (x > y) x = x - y;
        else y = y - x;
    return x;
}
```



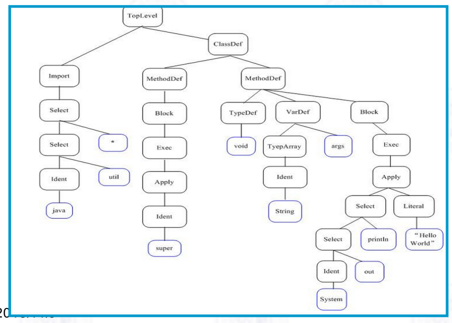
编 译 原 理

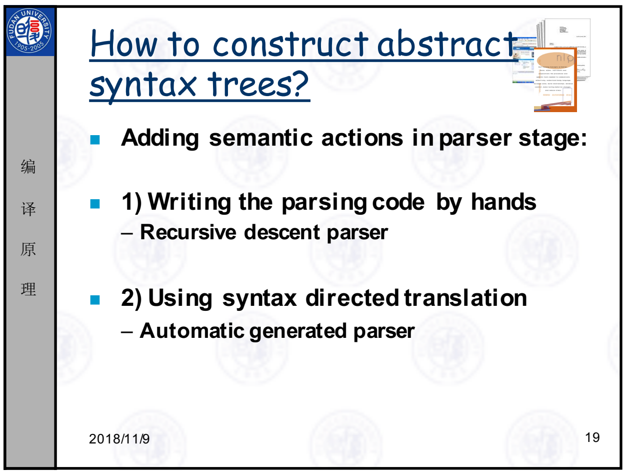
An abstract syntax tree for the following code for the Euclidean algorithm:

```
while b ≠ 0
if a > b
        a := a − b
else
        b := b − a
return a
```

“hello world” in Java

编 译 原 理



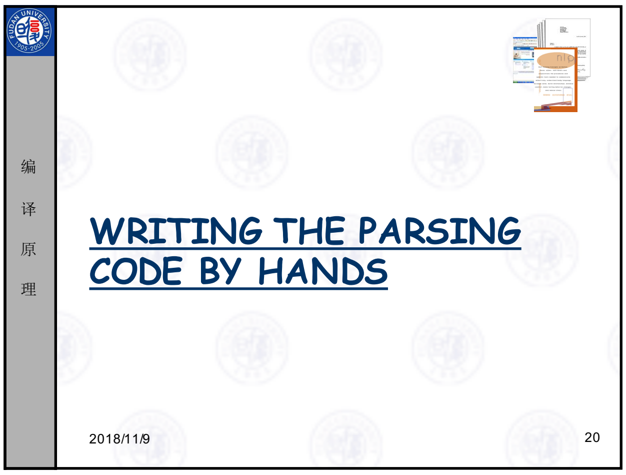


How to construct abstract syntax trees?

- Adding semantic actions in parser stage:
- 1) Writing the parsing code by hands
  – Recursive descent parser
- 2) Using syntax directed translation
  – Automatic generated parser

编 译 原 理

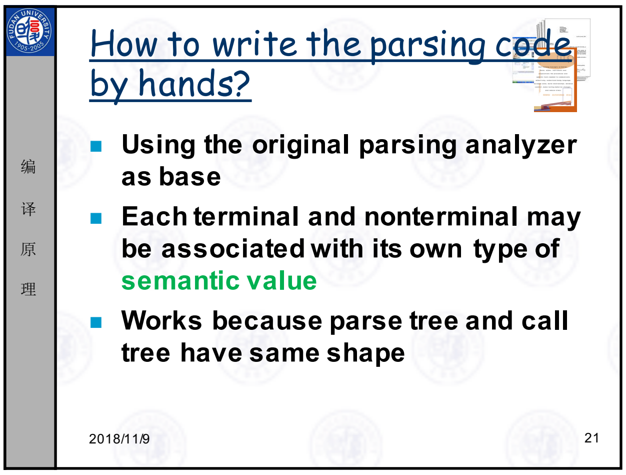
2018/11/9 19



WRITING THE PARSING CODE BY HANDS

编 译 原 理

2018/11/9 20

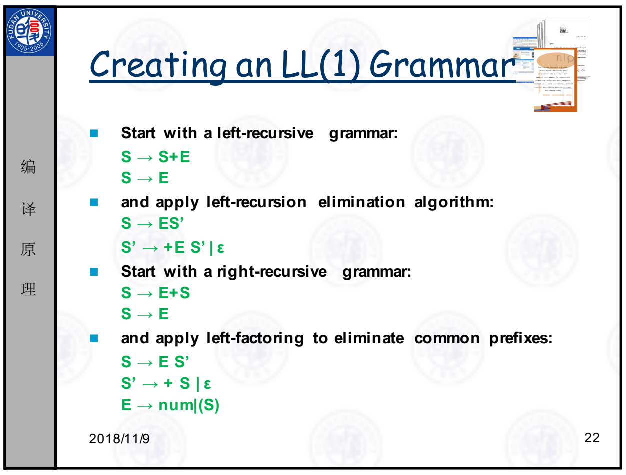


How to write the parsing code by hands?

- Using the original parsing analyzer as base
- Each terminal and nonterminal may be associated with its own type of semantic value
- Works because parse tree and call tree have same shape

编 译 原 理

2018/11/9 21

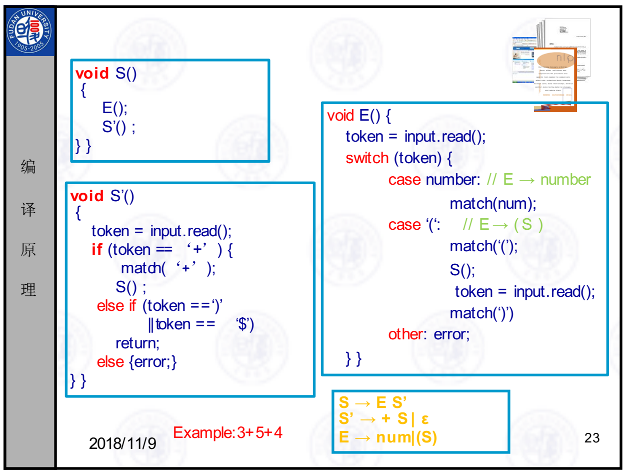


Creating an LL(1) Grammar

- Start with a left-recursive grammar:
  S → S+E
  S → E
- and apply left-recursion elimination algorithm:
  S → ES'
  S' → +E S' | ε
- Start with a right-recursive grammar:
  S → E+S
  S → E
- and apply left-factoring to eliminate common prefixes:
  S → E S'
  S' → + S | ε
  E → num|(S)

编 译 原 理

2018/11/9 22



```
void S()
 {
     E();
     S'() ;
} }
```

```
void S'()
 {
   token = input.read();
   if (token ==  '+'  ) {
        match(  '+'  );
        S() ;
    else if (token =='))'
              ||token ==    '$')
        return;
    else {error;}
} }
```

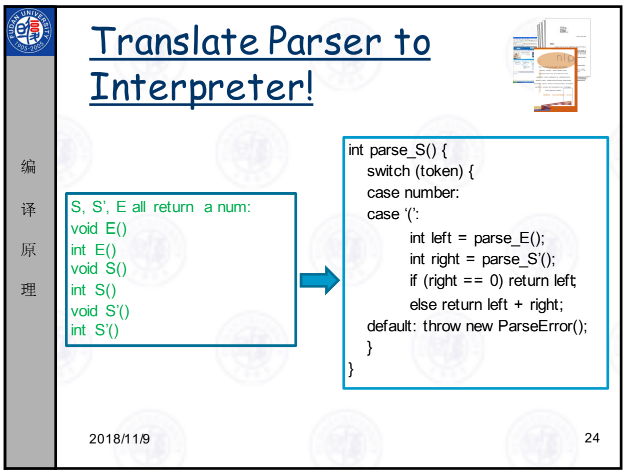
```
void E() {
   token = input.read();
   switch (token) {
         case number: // E → number
                match(num);
         case '(':    // E→ ( S )
                match('(');
                S();
                 token = input.read();
                match(')')
         other: error;
   } }
```

S → E S'
S' → + S | ε
E → num|(S)

Example:3+5+4

编 译 原 理

2018/11/9 23



Translate Parser to Interpreter!

S, S', E all return a num:
void E()
int E()
void S()
int S()
void S'()
int S'()

```
int parse_S() {
   switch (token) {
   case number:
   case '(':
           int left = parse_E();
           int right = parse_S'();
           if (right == 0) return left;
           else return left + right;
   default: throw new ParseError();
   }
}
```

编 译 原 理

2018/11/9 24

编 译 原 理

```
int parse_E() {
 switch(token) {
 case number:
          int result =token.value;
          token = input.read(); return result;
 case '(':
          token = input.read();
          int result =parse_S();
          if (token != ')') throw new ParseError();
          token = input.read(); return result;
 default: throw new ParseError();
 }
}
```

```
int parse_S'()
 {
    token = input.read();
    if (token ==  '+'  ) {
         match(  '+'  );
       return parse_S ();
    else if (token == ')'
                  ||token ==    '$')
       return 0;
     else {error;}
} }
```

Example:3+5+4

编 译 原 理

Top-Down Translation

- **For a grammar G:**

S → E S'
S' → + S | ε
E → num|(S)

S, S', E all return an Expr:

void E()	Expr E()
void S()	Expr S()
void S'()	Expr S'()

编 译 原 理

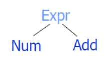
Creating the AST

```
abstract class  Expr { }

class  Add extends
   Expr {
   Expr  left, right;
   Add(Expr L, Expr R)
   {
       left = L; right =
   R;
   }
}
```

```
class Num extends Expr {
   int value;
   Num (int v)  { value = v; }
}
```

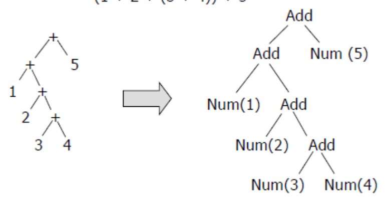
Class Hierarchy



编 译 原 理

AST Representation

(1 + 2 + (3 + 4)) + 5



编 译 原 理

```
Expr parse_E() {
  switch(token) {
      case num:   // E → number
              Expr result = Num (token.value);
              token = input.read(); return result;
      case '(':    // E → ( S )
              token = input.read();
              Expr result = parse_S();
              if (token != ')') throw new ParseError();
              token = input.read(); return result;
      default:  throw new ParseError();
  }
}
```

编 译 原 理

```
Expr parse_S() {
  switch (token) {
      case num:
      case '(':
        Expr left = parse_E();
        Expr right = parse_S'();
        if (right == null) return left;
        else return new Add(left, right);
      default: throw new ParseError();
      }
}
```

EBNF: Extended BNF Notation

编 译 原 理

- Extended Backus-Naur Form = a form of specifying grammars which allows some regular expression syntax on RHS
  – *, +, ( ), ? operators (also [X] means X?)

S → ES'
S' → ε |+S

$\Rightarrow$

S → E(+E)*

- EBNF version: no position on + associativity

Top-down Parsing EBNF

编 译 原 理

- Recursive-descent code can directly implement the EBNF grammar:

  S → E(+E)*

编 译 原 理

```
void S () {  // parses sequence of E+E+E ..
        E ();
        while (true) {
                token = input.read();
                switch (token) {
                case '+':  E();
                        break;
                case ')':
                case EOF: return;
                default: throw new ParseError();
                }
        }
}
```

Reassociating the AST

编 译 原 理

```
Expr S() {
    Expr result = E();
    while (true) {
        token = input.read();
        switch (token) {
        case '+':
                result = new Add(result, E());
                break;
        case ')':
        case EOF: return result;
        default: throw new ParseError();
        }
    }
}
```

编 译 原 理

USING SYNTAX DIRECTED TRANSLATION

Attribute grammar

编 译 原 理

- Synthesized attributes
  – The value of an attribute at a node is computed from values of the attributes at the children of that node in the parse tree
  – The synthesized attributes are the result of the attribute evaluation rules, and may also use the values of the inherited attributes
- Inherited attributes
  – The inherited attributes are passed down from parent nodes.

Constructing Syntax Trees for Expressions

Production	Semantic Rules
$E \rightarrow E_1 + T$	E.nptr := mknode('+', E_1.nptr, T.nptr)
$E \rightarrow E_1 - T$	E.nptr := mknode('-', E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow (E)$	T.nptr := E.nptr
T → **id**	T.nptr := mkleaf(**id**, **id**.entry)
T → **num**	T.nptr := mkleaf(**num**, **num**.val)

2018/11/9 37

Semantic rules (1)

- A parse-tree node is associated with the production
- This node is constructed according to the production
- Now, each production is also associated with a semantic rule

2018/11/9 38

Semantic rules (2)

- Semantic rules define how the value of an attribute at a parse-tree node is computed
- A semantic rule may have side effects
  – Printing a value or updating a global variable

2018/11/9 39

Constructing Syntax Trees for Expressions

- Define operations on the data structures
  – mknode(op, left, right)
  – mkleaf(id, entry)
  – mkleaf(num, val)

2018/11/9 40

USING SYNTAX DIRECTED TRANSLATION

2018/11/9 41

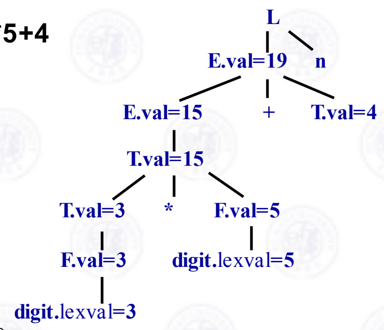
Example

Production	Semantic Rules
L → E **n**	Print(E.val)
$E \rightarrow E_1 + T$	E.val := E_1.val+ T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	T.val := T_1.val* F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
F → **digit**	F.val := **digit**.lexval

2018/11/9 42

Example

- 3*5+4



Bottom-Up Evaluation of S-Attributed Definitions

- Modify the data structure for parser stack

	token	val
	…	…
	X	X.x
	Y	Y.y
top →	Z	Z.z
	…	…

Example

Production	Semantic Rules
L → E $	print(val[top])
E → E_1 + T	val[top] := val[top-2]+val[top]
E → T	
T → T_1 * F	val[top] := val[top-2]*val[top]
T → F	
F → (E)	val[top] := val[top-1]
F → **digit**	

Example

Input	State	Val	Production used
3*5+4 $	-	-	
*5+4 $	3	3	
*5+4 $	F	3	F → digit
*5+4 $	T	3	T → F
5+4 $	T *	3 –	
+4 $	T * 5	3 – 5	
+4 $	T * F	3 – 5	F → digit
+4 $	T	15	T → T * F

Example

Input	State	Val	Production used
+4 $	E	15	E → T
4 $	E +	15 –	
$	E + 4	15 – 4	
$	E + F	15 – 4	F → digit
$	E + T	15 – 4	T → F
$	E	19	E → E + T
	E $	19 –	
	E $	19	L → E $