编译原理

4. 抽象语法

rainoftime.github.io

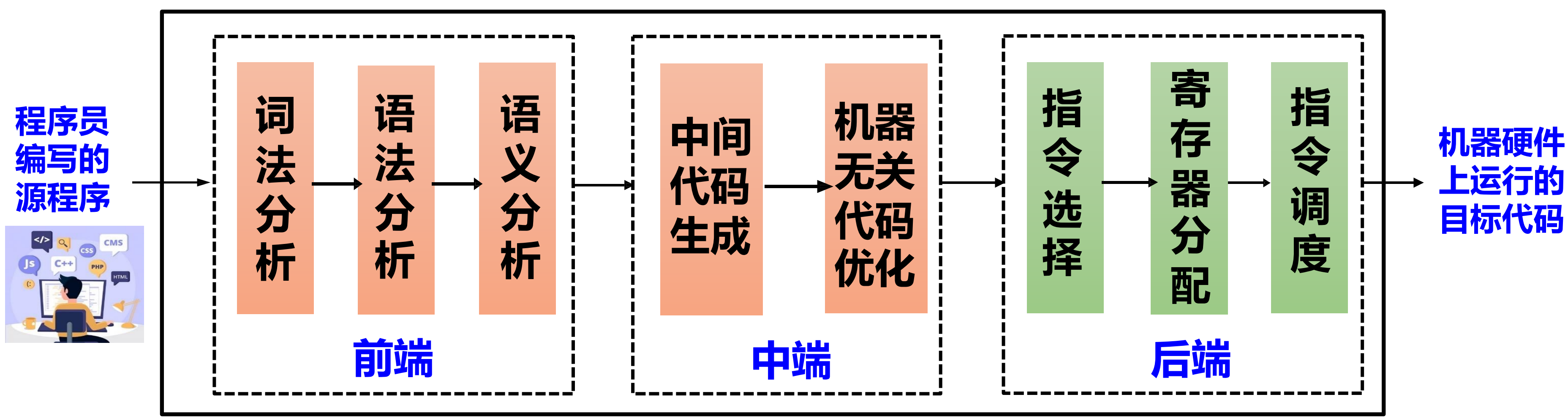
浙江大学
计算机科学与技术学院

课程内容

1. Introduction
2. Lexical Analysis
3. Parsing
4. **Abstract Syntax**
5. Semantic Analysis
6. Activation Record
7. Translating into Intermediate Code
8. Basic Blocks and Traces
9. Instruction Selection
10. Liveness Analysis
11. Register Allocation
13. Garbage Collection
14. Object-oriented Languages
18. Loop Optimizations

回顾: 编程语言 = 语法 + 语义

- **语法**: What sequences of characters are valid programs?
- **语义**: What is the behavior of a valid programs?
  - **操作语义**: How can we execute a program?
  - **公理语义**: What can we prove about a program
  - **指称语义**: What math function does the program compute?



属性文法(Attribute Grammar)

- **属性文法(Knuth, 1968)**
  上下文无关文法+属性+属性计算规则
  - **属性:** 描述文法符号的语义特征，如变量的类型、值等
    例: 非终结符E的属性E.val（表达式的值）
  - **属性计算规则(语义规则):** 与产生式相关联、反映文法符号属性之间关系的规则, 比如”如何计算E.val”

- **属性计算规则**: 仅表明属性间“抽象”关系，不涉及具体实现细节，如计算次序等

用属性描述语义信息，用语义规则描述属性之间的关系，将**语义规则与语法规则相结合**

属性文法(Attribute Grammar)的潜在应用

- **“推导类”应用: 类似程序分析**
  - 表达式的类型、值、执行代价

产生式	“语法制导”的属性计算规则
$E \rightarrow E_1$ ‘+’ E_2	$E.val := E_1.val + E_2.val$
$E \rightarrow E_1$ ‘*’ E_2	$E.val := E_1.val * E_2.val$
$E \rightarrow$ ’(‘ E_1 ‘)’	$E.val := E_1.val$
$E \rightarrow$ number	E.val := number.lex_val

- **“生成类”应用: 类似程序合成**
  - 抽象语法树生成
  - 中间代码甚至汇编生成！

由于各种局限性，属性文法并未在所有潜在应用上得到普及

属性文法(Attribute Grammar)的实现

- **属性文法: 上下文无关文法+属性+属性计算规则**
  - 属性: 描述文法符号的语义特征，如变量的类型、值等
    例: 非终结符E的属性E.val（表达式的值）
  - 属性计算规则(语义规则): 与产生式相关联、反映文法符号属性之间关系的规则, 比如"如何计算E.val"

- **可通过Parser生成器支持的"语义动作"(Semantic action)实现计算**
  - 并应用于抽象语法树生成等场景

本讲内容

1. Semantic Action
2. Abstract Parse Tree
2. Position

1. Semantic Action

Why Semantic Action

- Each terminal and nonterminal may be associated with its own type of **semantic value**.

A rule: $A \rightarrow B\ C\ D$

  - The semantic action must return a value whose type is the one associated with the nonterminal A.
  - It can build this value from the values associated with the matched terminals and nonterminals B, C, D.

Example: CFG and its Semantic Actions

- Suppose that we want to “evaluate” the values of an expression

$E \rightarrow E_1 + T$	$\{ E.val = E_1.val + T.val \}$
$E \rightarrow E_1 - T$	$\{ E.val = E_1.val - T.val \}$
$E \rightarrow T$	$\{ E.val = T.val \}$
$T \rightarrow (E)$	$\{ T.val = E.val \}$
$T \rightarrow num$	$\{ T.val = num.val \}$

Semantic Actions in Recursive Descent Parsing

- The semantic actions are the values returned by parsing functions, or the side effects of those functions, or both.

- For each terminal and nonterminal symbol, we associate a type of semantic values representing phrases derived from that symbol.

T -> T * F

The semantic action:

```
int a = T();
eat(TIMES);
int b=F();
return a*b;
```

Semantic Actions in Yacc-Generated Parsers

- **{ … }**: semantic actions
- **$i**: the semantic values of the i_th RHS symbol
- **$$**: the semantic value of the LHS nonterminal symbol
- **%union**: difference possible types for semantic values to carry
- **<variant>**: declares the type of each terminal or nonterminal

```
%{ … %}
%union {int num; string id;}
%token <num> INT
%token <id> ID
%type <num> exp
...
%left UMINUS
%%

exp: INT {$$ = $1;}
    | exp PLUS exp {$$ = $1 + $3;}
    | exp MINUS exp {$$ = $1 - $3;}
    | exp TIMES exp {$$ = $1 * $3;}
    | MINUS exp %prec UMINUS {$$ =
-$2;}
```

Semantic Actions in Yacc-Generated Parsers

- A Yacc-generated parser keeps a stack of semantics values parallel to the state stack.

- When the parser performs a reduction, it must execute the corresponding C-language semantic action

- How to know $i for a rule **A -> Y1 … Yk**? from the top k elements of the stack
  - When the parser pops **Yk … Y1** from the symbol stack and pushes **A**, it also pops k values from the semantic value stack and pushes the value obtained by executing the C semantic action code.

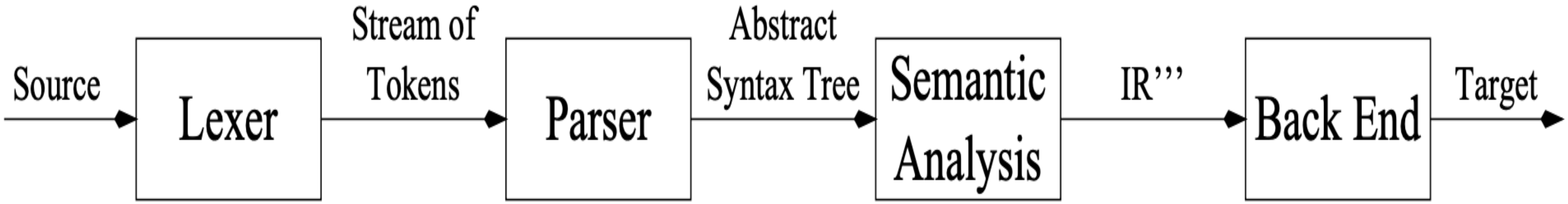
2. Abstract Parse Tree

Semantic Action的应用

Motivation

- Can write an entire compiler that fits within the semantic action phrases of a Yacc parser!
  1. Difficult to read and maintain
  2. Must analyze the program in exactly the order it is parsed

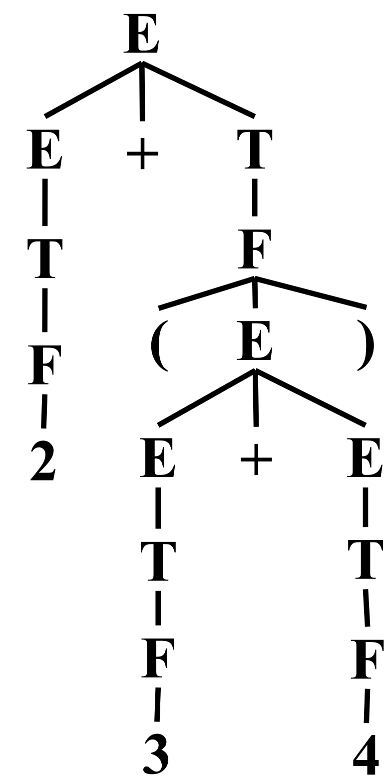
- Alternative: separate issues of syntax (parsing) from issues of semantics (type-checking and translation to machine code)
- One solution: the parser produces a parse tree that later phases can traverse.



Recap: Parse Tree

- **A parse tree** has exactly one leaf for each token of the input and one internal node for each grammar rule reduced during the parse.
  - Such a parse is called a **concrete parse tree**, representing the concrete syntax of the source language.

E -> E + T
| T
T -> T * F
| F
F -> n | (E)



- A concrete parse tree is inconvenient to use directly:
  - redundant and useless tokens for later phases
    - e.g., (,)
    - memory usage
  - depends too much on the grammar
    - The grammar changes -> parse tree changes

Abstract Syntax Trees

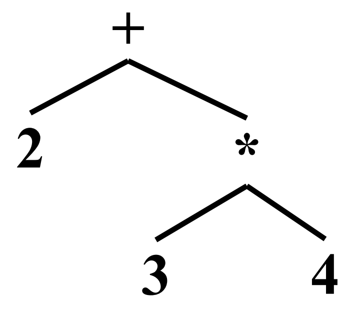
- Make a clean interface between the parser and the later phases of a compiler.
  - The semantic analysis phase takes this *abstract syntax tree*.
- The parser uses the concrete syntax to build a parse tree for the abstract syntax — abstract syntax tree

```
E -> E + T
   | T
T -> T * F
   | F
F -> n | ( E )
```

Concrete Syntax

```
E -> n
   | E + E
   | E * E
```

Abstract Syntax



Abstract Syntax Tree

Representing Abstract Syntax Trees

- To use abstract syntax tree in later phases, the compiler will need to represent and manipulate abstract syntax trees as data structures. **How?**
- A typedef for each nonterminal, a union-variant for each production.

```
typedef struct A_exp_ *A_exp;
struct A_exp_ {
  enum {A_numExp, A_plusExp, A_timesExp} kind;
  union {
    int num;
    struct {A_exp left; A_exp right;} plus;
    struct {A_exp left; A_exp right;} times;
  } u;
};

A_exp A_NumExp(int num);
A_exp A_PlusExp(A_exp left, A_exp right);
A_exp A_TimesExp(A_exp left, A_exp right);
```

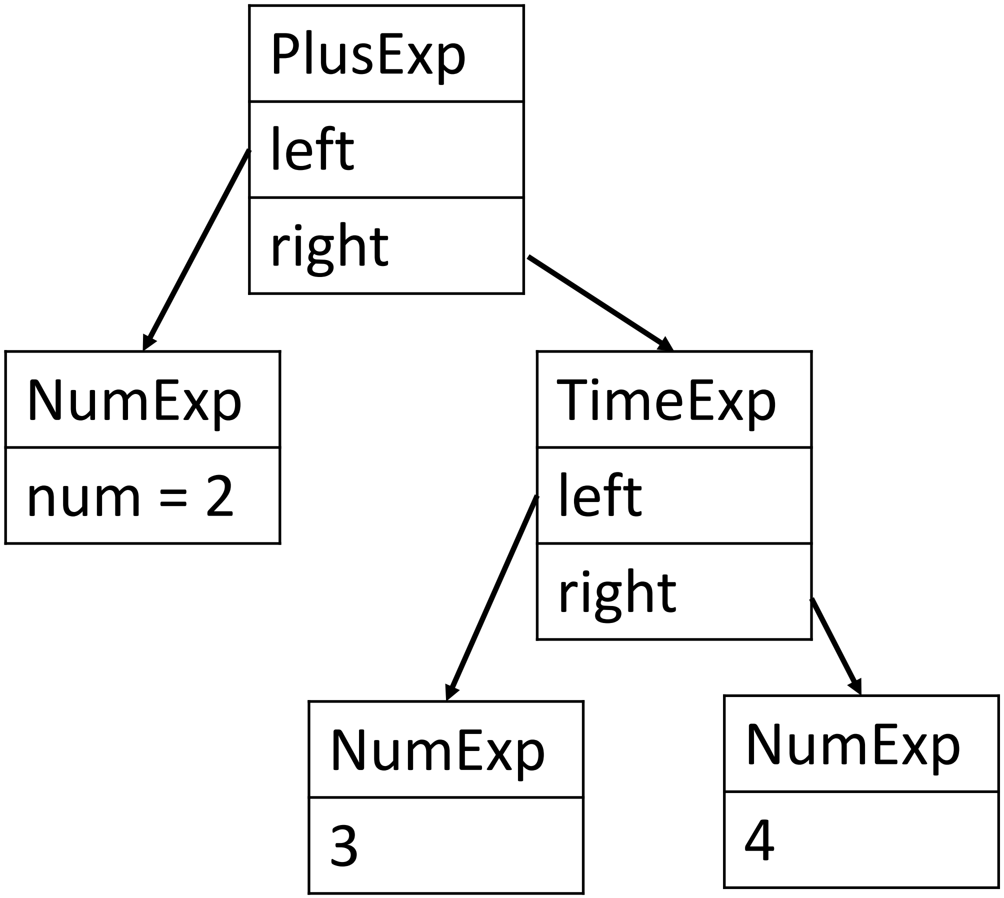
E -> n
| E + E
| E * E

Representing Abstract Syntax Trees

- Construct the abstract syntax tree of $2 + 3 * 4$ using these data structures

```
e1 = A_NumExp(2);
e2 = A_NumExp(3);
e3 = A_NumExp(4);
e4 = A_TimesExp(e2, e3);
e5 = A_PlusExp(e1, e4);
```

```
A_exp A_PlusExp(A_exp left,
A_exp right) {
  A_exp e =
checked_malloc(sizeof(*e));
  e->kind = A_plusExp;
  e->u.plus.left = left;
  e->u.plus.right = right;
  return e;
}
```



Building Abstract Parse Trees Automatically

- Can we construct abstract syntax trees automatically?
- The Yacc (or recursive-descent) parser, parsing the concrete syntax, constructs the abstract syntax tree.
- How?

```
%left PLUS
%left TIMES

%%
exp : NUM                       {$$=A_NumExp($1);}
        | exp PLUS exp          {$$=A_PlusExp($1, $3);}
        | exp TIMES exp         {$$=A_TimesExp($1, $3);}
```

Applications of AST Traversals

- Pretty print
- Desugaring
- Inlining
- High-level optimizations (e.g., 删除公共子表达式)
- Symbolic execution!(e.g., Clang Static Analyzer)
- **Semantic analysis , e.g., type checking**
- **Translation to intermediate representations**
- …

注意：编译相关书籍/课程中通常会说“基于AST翻译到中间语言/表示”. 但是在很多其他场合，我们可以认为AST本身也是一种“中间表示”

3. Position

Semantic Action的更多应用

Positions

- In a one-pass compiler, lexical analysis, parsing, and semantic analysis are all done simultaneously.
- If there is a type error that must be reported to the user, the *current position* of the lexical analyzer is a reasonable approximation of the source position of the error.
- In a one-pass compiler, the lexical analyzer keeps a "current position" global variable.
- For a compiler that uses abstract-syntax-tree data structures:
  - It need not do all the parsing and semantic analysis in one pass
  - The lexer reaches the end of file before semantic analysis even begins.
- How can we know the error position if there is a type error?

Positions

- The source-file position of each node of the abstract syntax tree must be remembered.
- How?
- The abstract-syntax data structures must be sprinkled with pos fields, which indicate the position, within the original source file, of the characters from which these abstract syntax structures were derived.
- How to set the pos fields?
- First, the lexer must pass the positions of the beginning and end of each token to the parser
- Then, for parsers:

Positions

- Ideally, the parser should maintain a *position stack* along with the *semantic value stack*, making the position of each symbol available for the semantic actions to use.
  - Bison can do this, **Yacc does not**
- Yacc: one solution is to define a nonterminal symbol pos whose semantic value is a source location (line number, or line number and position within line).
  - e.g., access the position of the PLUS

```
%{ extern A_OpExp (A_exp,A_binop,A_exp,position); %}
%union { int num; string id; position pos;….};
%type <pos> pos

pos: { $$ = EM_tokpos; }
exp: exp PLUS pos exp {$$= A_OpExp($1, A_plus, $4, $3); }
```

Thank you all for your attention