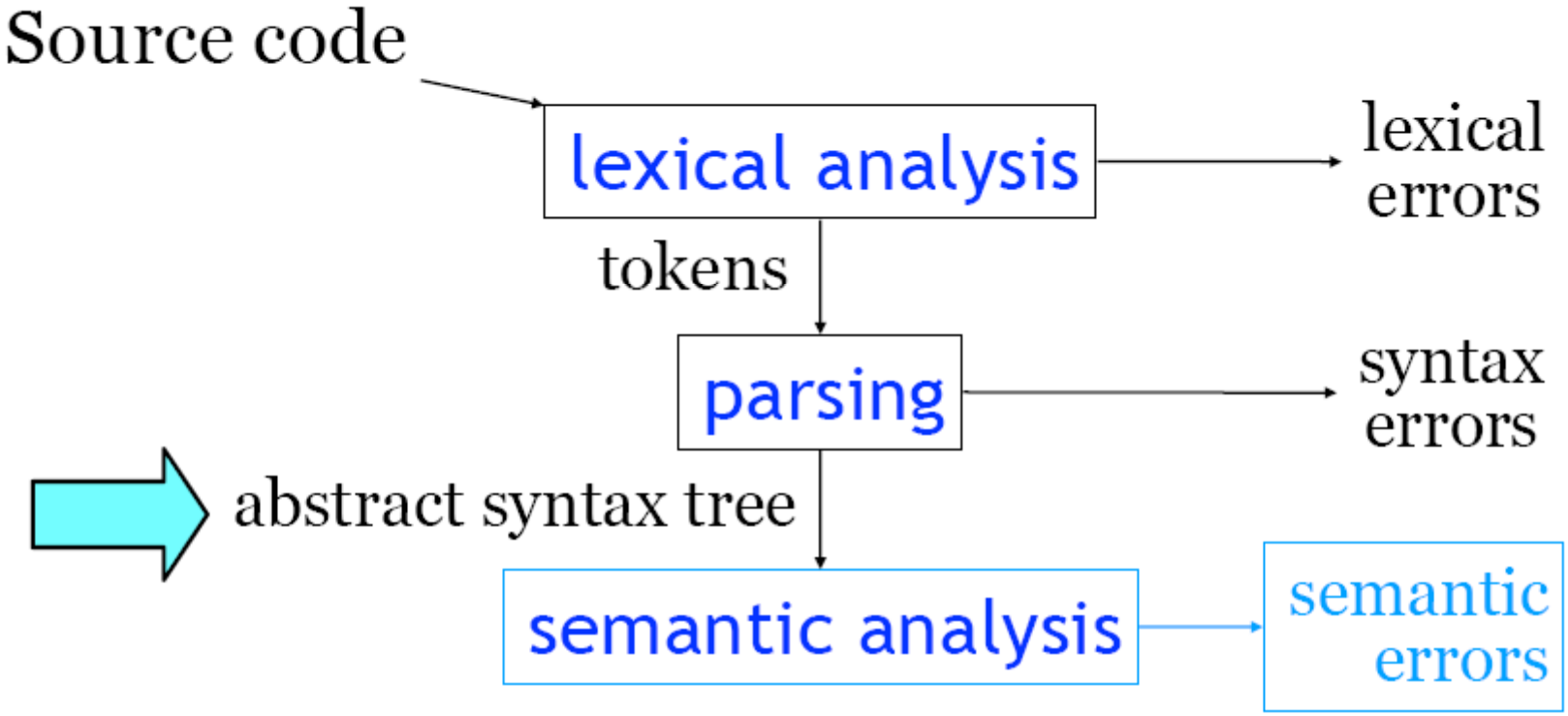
# Abstract Syntax Tree

# เฟสของคอมไพเลอร์ที่ผ่านมา



# AST

- Abstract Syntax Tree เป็นโครงสร้างข้อมูลที่ใช้แทนภาษาโปรแกรมที่จะแปลสู่ภาษาระดับล่าง
- เฟสของคอมไพเลอร์หลังจากการทำ parsing จะใช้ AST ในการปฏิบัติการหลายๆอย่าง
  - Type checking
  - Optimization ต่างๆ
  - การผลิตโค๊ดระหว่างกลาง (Intermediate Code Generation)

# การสร้าง AST

- ค่อยๆสร้างแบบ bottom-up
- ใส่โค๊ดในการสร้าง node และ link ไว้ที่ semantic action
  - Semantic action คือสิ่งที่จะถูกดำเนินการขณะที่ non-terminal หรือ terminal ด้านขวาของ CFG ได้รับการประมวลเรียบร้อยแล้ว
  - นั่นคือเวลาที่ production rule ของ CFG ได้ปฏิบัติการ rule นั้นเสร็จเรียบร้อยแล้ว

```
non terminal Expr expr; ...
expr ::= expr:e1 PLUS expr:e2
    {: RESULT = new Add(e1,e2); :}
```

*grammar production*

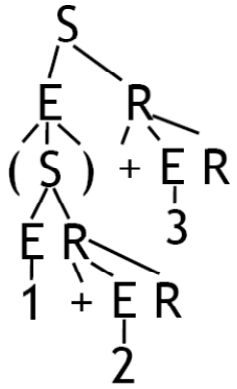
*semantic action*

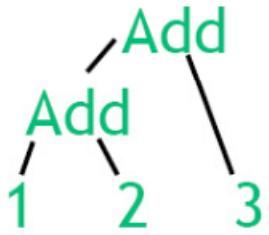
# หัวใจในการสร้าง AST

- ต้อง abstract
  - ใส่เฉพาะ node ที่จำเป็นใน AST

$$S \rightarrow E\ R$$
$$R \rightarrow \varepsilon \mid +\ E\ R$$
$$E \rightarrow \text{num} \mid (\ S\ )$$

(1 + 2) + 3




❑ เราต้องการ tree ที่ abstract ในลักษณะนี้

❑ Tree ที่ได้จากการ parse expression (1 + 2) + 3 แบบนี้ไม่ abstract เท่าไหร่

❑ ทั้งนี้เพราะเราใส่ node เข้าไปที่ AST ทุกๆครั้งที่ production rule ที่จะทำการสร้าง expression นี้ ดำเนินการเสร็จ

# คอมไพเลอร์ยุคเก่า

- รันบนเครื่องคอมพิวเตอร์ที่มีหน่วยความจำหลักน้อย
- หน่วยความจำหลักไม่เพียงพอที่จะบรรจุ AST ของทั้งโปรแกรมได้
- จะผลิตโค๊ดและทำการ type checking ที่ semantic action โดยตรงเลย
  - คอมไพเลอร์ขนาดเล็กและเข้าใจได้ง่าย แต่
  - ไม่ถูกต้องตามหลักการวิศวกรรมซอฟแวร์ที่ดี
  - ปฏิบัติการหลายๆอย่างทำได้ลำบาก
- คอมไพเลอร์ CSubset ไม่มีการสร้าง AST

# โค๊ดการสร้าง AST: node ของ tree

```
typedef struct A_exp_ *A_exp;

struct A_exp_
    {enum {A_varExp, A_intExp, A_callExp,
           A_opExp, A_assignExp, A_ifExp,
           A_whileExp, A_arrayExp} kind;
     A_pos pos;
     union {A_var var;
            /* nil; - needs only the pos */
            int intt;
            struct {S_symbol func; A_expList args;} call;
            struct {A_oper oper; A_exp left; A_exp right;} op;
            struct {A_var var; A_exp exp;} assign;
            struct {A_exp test, then, elsee;} iff; /* elsee is optional */
            struct {A_exp test, body;} whilee;
           } u;
    };
```

# โค๊ดการสร้าง AST: ตัวอย่าง prototype

- A_exp A_OpExp(A_pos pos, A_oper oper, A_exp left, A_exp right);
- A_exp A_AssignExp(A_pos pos, A_var var, A_exp exp);
- A_exp A_IfExp(A_pos pos, A_exp test, A_exp then, A_exp elsee);
- A_exp A_WhileExp(A_pos pos, A_exp test, A_exp body);

# โค๊ดการสร้าง AST:  OpExp

```
A_exp A_OpExp(A_pos pos, A_oper oper, A_exp left, A_exp right) {
        A_exp p = malloc(sizeof(*p));
        p->kind=A_opExp;
        p->pos=pos;
        p->u.op.oper=oper;
        p->u.op.left=left;
        p->u.op.right=right;
        return p;
}
```

# โค๊ดการสร้าง AST: AssignExp

```
A_exp A_AssignExp(A_pos pos, A_var var, A_exp exp) {
      A_exp p = malloc(sizeof(*p));
       p->kind=A_assignExp;
       p->pos=pos;
       p->u.assign.var=var;
       p->u.assign.exp=exp;
       return p;
}
```

# โค๊ดการสร้าง AST: IfExp

```
A_exp A_IfExp(A_pos pos, A_exp test, A_exp then, A_exp elsee) {
        A_exp p = malloc(sizeof(*p));
        p->kind=A_ifExp;
        p->pos=pos;
        p->u.iff.test=test;
        p->u.iff.then=then;
        p->u.iff.elsee=elsee;
        return p;
}
```

# โค๊ดการสร้าง AST: WhileExp

```
A_exp A_WhileExp(A_pos pos, A_exp test, A_exp body) {
        A_exp p = malloc(sizeof(*p));
        p->kind=A_whileExp;
        p->pos=pos;
        p->u.whilee.test=test;
        p->u.whilee.body=body;
        return p;
}
```

# สร้าง AST ที่ semantic action

- จาก production: expr = expr + expr ใส่ semantic action (ให้ expr ด้านขวาคือ e1 และ e2 ตามลำดับ):

  {return A_OpExp(A_pos pos, A_oper A_plusOp, A_exp e1, A_exp e2);}
- จาก production: stmt = if (expr) stmt else stmt ใส่ semantic action (ให้ expr และ stmt ทั้งสองที่อยู่ด้านขวาเป็น e1 s1 และ s2 ตามลำดับ):

  {return A_IfExp(A_pos pos, A_exp e1, A_exp s1, A_exp s2);}

# โค๊ดการสร้าง AST ใน parser

```
A_exp parseStmt() {
    A_exp result;
    switch (next_token) {
        case IF: consume(IF); consume(LPAREN);
        A_exp e1 = parseExpr();
        consume(RPAREN);
        A_exp s1, s2;
        s1  = parseStmt();
        if (next_token == ELSE) {
            consume(ELSE);
            s2 = parseStmt();
        }
        else s2 = EmptyStmt();
        result = A_IfExp(A_pos pos, A_exp e1, A_exp s1, A_exp s2);
        break;
        case ID: ...
        case WHILE: ...
```

# บทนำเกี่ยวกับ semantics analysis

# พิจารณาโค๊ดต่อไปนี้

```
fie(a,b,c,d) {
        int a, b, c, d;
      …
}
fee() {
        int f[3],g[1], h, i, j, k;
        char *p;
        fie(h,i,“ab”,j, k);
        k = f * i + j;
        h = g[17];
        printf(“<%s,%s>.\n”,p,q);
        p = 10;
}
```

# มีอะไรที่ผิดพลาดบ้าง

- จำนวน argument ของฟังก์ชั่น fie
- ประกาศ g[1] ใช้ g[17]
- ab ไม่ใช่ int
- f ไม่ใช่เป็นตัวแปรที่จะนำมาทำกระบวนการทางคณิตศาสตร์
- ไม่ได้ประกาศ q
- 10 ไม่ใช่ string
- สิ่งเหล่านี้ไม่สามารถตรวจจับได้โดยใช้ CFG

# ข้อจำกัดของ CFG

- ภาษาโปรแกรมไม่ใช่ context free แต่เป็น context sensitive
  - Assignment จะมีความหมายก็ต่อเมื่อปริมาณทั้งสองด้านเป็น type เดียวกัน
  - ชื่อ “x” ในโปรแกรมเป็นตัวแปรแบบ scalar หรือ array หรือ function
  - มีการประกาศ “x” มาก่อนหรือไม่
  - ใน expression x * y + z นั้น ตัวแปรแต่ละตัวมี type ที่สอดคล้องกันหรือไม่
  - จำนวน argument ของ function เมื่อถูกเรียกเท่ากับที่ประกาศไว้หรือไม่

# โครงสร้างและความหมายของโปรแกรม

- CFG และ parser ที่สร้างจาก CFG ตรวจจับความถูกต้องของโครงสร้างของภาษาโปรแกรม (syntactic analysis)
- แต่การผลิตโค๊ดสุดท้ายที่ถูกต้องจะต้องคำนึงถึงความหมายของโปรแกรมด้วย (semantic analysis)
- ทำอย่างไรเราถึงจะตรวจจับเรื่องความหมายของโปรแกรมได้

# เทคนิคการทำ semantic analysis

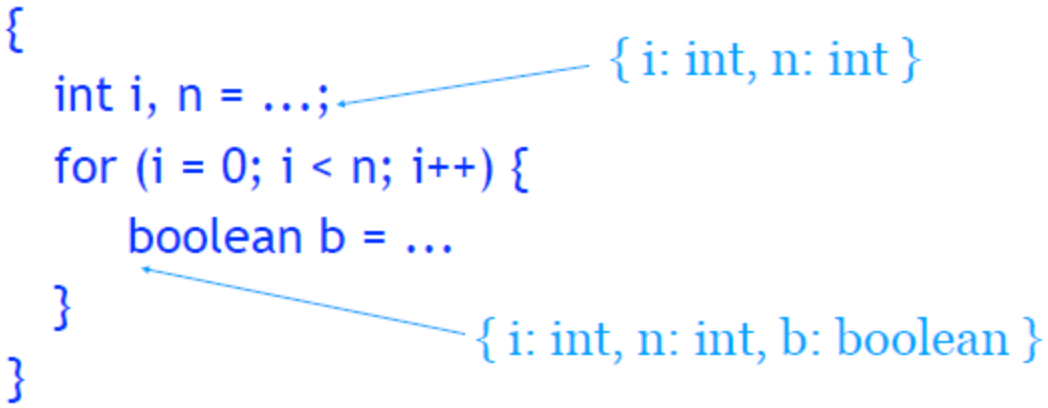
- ใช้เทคนิคที่มีพื้นฐานมาจากคณิตศาสตร์และทฤษฎีคอมพิวเตอร์ เหมือนเช่นที่เราใช้ CFG กับ syntactic analysis (formal method)
  - Context Sensitive Grammar (CSG)
- ใช้เทคนิคเฉพาะกิจ (ad-hoc technique)
  - Symbol table
  - Semantic action
- ในทางปฏิบัติเราใช้ ad-hoc technique กัน
- การวิเคราะห์ CSG เป็นปัญหา P-SPACE Complete

# Symbol Table

- ข้อมูลสำคัญในการทำ semantics analysis คือชนิดของข้อมูล หรือ type ของ identifier ในภาษาโปรแกรม
  - Identifier เช่น ตัวแปร ชื่อของฟังก์ชั่น ชื่อของ struct หรือ array
- เราต้องมี environment ที่บ่งบอก type ของ identifiers
- Environment นี้เรียกว่า symbol table (ตารางสัญลักษณ์)

# Symbol Table

- โดยแนวคิดหลัก symbol table มีโครงสร้างข้อมูลหลักเป็นเซ็ทของคู่ identifier : type
  - เช่น { x: int, y: array[string] }
  - มีแนวคิดในเรื่องของ scope (ขอบเขตที่ identifier มีความหมาย สามารถอ้างอิงได้) เข้ามาเกี่ยวข้องด้วย



# Symbol Table ADT

- มองแบบโครงสร้างข้อมูล abstract data type (ADT) symbol table ประกอบไปด้วย:
  - เซ็ทของคู่ลำดับ identifier : type
  - ปฏิบัติการพื้นฐานต่อไปนี้:
    - การค้นหา identifier ในเซ็ทของคู่ลำดับ identifier : type
    - การเพิ่มคู่ identifier : type เข้าในเซ็ทนี้

```
class SymTab {
  Type lookup(String id) …
  void add(String id, Type binding) …
 }
```