Typage et sûreté (3)

- Le typage statique se base sur les **règles de typage** du langage
- Chaque opération produit un **résultat** de type R si on lui fournit des opérandes de type T_1, T_2, ...
- Déf: un opérateur est **surchargé** ("overloaded") si l'opération effectuée dépend du type des opérandes
- Exemple d'opérateurs surchargés du langage C:

T_1 + T_2	int	long	double	void*	E*	struct S
int	int	long	double	*erreur*	E*	*erreur*
long	long	long	double	*erreur*	E*	*erreur*
double	double	double	double	*erreur*	*erreur*	*erreur*
void*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*
E*	E*	E*	*erreur*	*erreur*	*erreur*	*erreur*
struct S	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*

T_1 - T_2	int	long	double	void*	E*	struct S
int	int	long	double	*erreur*	*erreur*	*erreur*
long	long	long	double	*erreur*	*erreur*	*erreur*
double	double	double	double	*erreur*	*erreur*	*erreur*
void*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*
E*	E*	E*	*erreur*	*erreur*	ptrdiff_t	*erreur*
struct S	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*	*erreur*

!T_2	int	long	double	void*	E*	struct S
	int	int	int	int	int	*erreur*

*T_2	int	long	double	void*	E*	struct S
	erreur	*erreur*	*erreur*	*erreur*	E	*erreur*

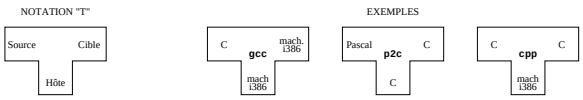
Typage et sûreté (4)

- Typiquement le langage demande de **spécifier explicitement** le type de chaque variable, paramètre et résultat de fonction (Pascal, C, Ada, Modula-3)
- Cela simplifie l'algorithme de vérification statique de type qui peut partir des feuilles d'une expression pour trouver le type global de l'expression
- Certains langages permettent **d'inférer automatiquement** le type des variables, paramètres et résultats de fonction (Haskell, ML), ce qui demande de résoudre un système de contraintes
- Exemple en Standard ML:

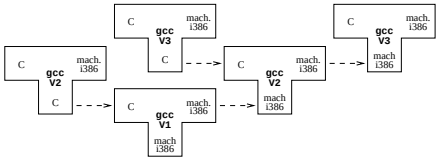
```
- fun f (x : bool, y : int) : int = if x then y else 123;
val f = fn : bool * int -> int

- fun g (x, y) = if x then y else 123;
val g = fn : bool * int -> int

- fun h (x, y) = if x then y else x;
val h = fn : bool * bool -> bool

- fun i (x) = if x then x else 123;
stdIn:4.13-4.33 Error: case object and rules don't agree [literal]
  rule domain: bool
  object: int
  in expression:
    (case x
      of true => x
       | false => 123)
```


Compilation

- Transformation d'un **programme source** en un **programme objet** "équivalent" (même input => même output, mais ne dit rien sur l'espace utilisé, le temps d'exécution, etc)
- Normalement le **langage source** est de haut niveau et le **langage cible** est de bas niveau
- Le compilateur est lui-même exprimé dans un certain **langage hôte** (langage d'écriture du compilateur, langage machine de l'exécutable, etc)

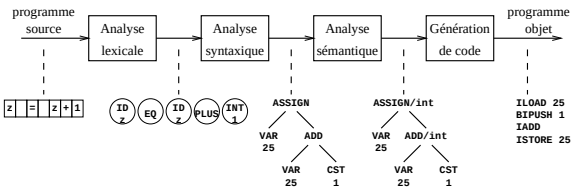


- Le compilateur étant lui-même un programme, on peut le compiler; pour faciliter le **bootstrap** on utilise L.H. = L.S. (**compilateur autogène**)



Structure d'un compilateur

- Compilateur = pipeline de petits compilateurs



- **Analyse lexicale**: lecture du programme source caractère par caractère, élimination des blancs et commentaires, grouper caractères en **symboles**
- **Analyse syntaxique**: vérification syntaxique, construction d'un **ASA**, construction d'une table de symboles
- **Analyse sémantique**: analyse de type, annotation de l'**ASA**, vérification sémantique
- Autres étapes: transformations, optimisations, etc
- **Génération de code**: production du **programme objet** par parcours postfixe de l'ASA

Exemple d'un petit compilateur

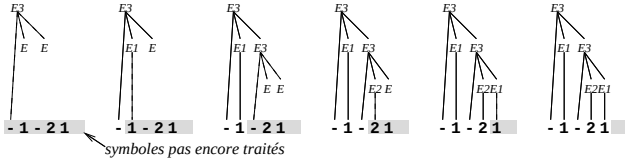
- Langage source = variante minimale de C

```
<program> ::= <statement>
<statement> ::= "if" <paren_expr> <statement>
             |  "if" <paren_expr> <statement> "else" <statement>
             |  "while" <paren_expr> <statement>
             |  "do" <statement> "while" <paren_expr> ";"
             |  "{" { <statement> } "}"
             |  <expr> ";"
             |  ";"
<paren_expr> ::= "(" <expr> ")"
<expr> ::= <test> | <id> "=" <expr>
<test> ::= <sum> | <sum> "<" <sum>
<sum> ::= <term> | <sum> "+" <term> | <sum> "-" <term>
<term> ::= <id> | <int> | <paren_expr>
<id> ::= "a" | "b" | "c" | "d" | ... | "z"
<int> ::= <an_unsigned_decimal_integer>
```

- Langage cible = variante de la machine virtuelle Java (JVM)
  - machine à pile (les opérandes des instructions sont sur le dessus d'une pile, le résultat remplace les opérandes)
  - 26 variables globales (`ILOAD` n et `ISTORE` n)
  - instructions: `BIPUSH` n, `DUP`, `POP`, `IADD`, `ISUB`, `GOTO` $dest$, `IFEQ` $dest$, `IFNE` $dest$, `IFLT` $dest$, `RETURN`

Analyse syntaxique descendante (1)

- L'A.S. se base sur la grammaire pour bâtir l'arbre de dérivation (ou l'**ASA**) à partir des symboles
- Certaines grammaires (dites LL(k)) permettent de faire l'analyse avec l'algorithme de l'**analyse syntaxique descendante**:
  - l'arbre est construit de gauche à droite
  - à tout moment l'algorithme se base sur les k prochains symboles (l'**anticipation** ou "look-ahead") pour déterminer quelle est la prochaine dérivation à ajouter à l'arbre
- Plusieurs langages sont LL(1), i.e. l'analyse est guidée uniquement par le prochain symbole
- Exemple: <*E*> ::= 1 | 2 | - <*E*> <*E*>



Analyse syntaxique descendante (2)

- Notre grammaire n'est pas LL(1), par exemple le `if` et = demandent plus qu'un symbole d'anticipation
- Il est possible de modifier la grammaire pour qu'elle accepte (presque) le même langage tout en étant (presque) LL(1)
- Nouvelle grammaire:

```
<program> ::= <statement>
<statement> ::= "if" <paren_expr> <statement> <if_tail>
             |  "while" <paren_expr> <statement>
             |  "do" <statement> "while" <paren_expr> ";"
             |  "{" { <statement> } "}"
             |  <expr> ";"
             |  ";"
<if_tail> ::= | "else" <statement>       <== ambiguïté!!!  pas LL(1)
<paren_expr> ::= "(" <expr> ")"
<expr> ::= <test> <expr_tail>           <== généralisation!!!
<expr_tail> ::= | "=" <expr>
<test> ::= <sum> <test_tail>
<test_tail> ::= | "<" <sum>
<sum> ::= <term> <sum_tail>             <== s'éloigne de l'ASA!!!
<sum_tail> ::= | "+" <sum> | "-" <sum>
<term> ::= <id> | <int> | <paren_expr>
<id> ::= "a" | "b" | "c" | "d" | ... | "z"
<int> ::= <an_unsigned_decimal_integer>
```

- À l'aide de règles supplémentaires, l'analyseur syntaxique règlera l'ambiguïté, rejettera les affectations illégales et construira un ASA de la bonne forme

Analyseur lexical

```
1.  enum { DO_SYM, ELSE_SYM, IF_SYM, WHILE_SYM, LBRA, RBRA, LPAR, RPAR,
2.         PLUS, MINUS, LESS, SEMI, EQUAL, INT, ID, EOI };
3.
4.  char *words[] = { "do", "else", "if", "while", NULL };
5.
6.  int ch = ' ';
7.  int sym;
8.  int int_val;
9.  char id_name[100];
10.
11. void syntax_error() { fprintf(stderr, "syntax error\n"); exit(1); }
12.
13. void next_ch() { ch = getchar(); }
14.
15. void next_sym()
16. { again: switch (ch)
17.     { case ' ': case '\n': next_ch(); goto again;
18.       case EOF: sym = EOI; break;
19.       case '{': next_ch(); sym = LBRA; break;
20.       case '}': next_ch(); sym = RBRA; break;
21.       case '(': next_ch(); sym = LPAR; break;
22.       case ')': next_ch(); sym = RPAR; break;
23.       case '+': next_ch(); sym = PLUS; break;
24.       case '-': next_ch(); sym = MINUS; break;
25.       case '<': next_ch(); sym = LESS; break;
26.       case ';': next_ch(); sym = SEMI; break;
27.       case '=': next_ch(); sym = EQUAL; break;
28.       default:
29.         if (ch >= '0' && ch <= '9')
30.           { int_val = 0; /* missing overflow check */
31.             while (ch >= '0' && ch <= '9')
32.               { int_val = int_val*10 + (ch - '0'); next_ch(); }
33.             sym = INT;
34.           }
35.         else if (ch >= 'a' && ch <= 'z')
36.           { int i = 0; /* missing overflow check */
37.             while ((ch >= 'a' && ch <= 'z') || ch == '_')
38.               { id_name[i++] = ch; next_ch(); }
39.             id_name[i] = '\0';
40.             sym = 0;
41.             while (words[sym] != NULL && strcmp(words[sym], id_name) != 0)
42.               sym++;
43.             if (words[sym] == NULL)
44.               if (id_name[1] == '\0') sym = ID; else syntax_error();
45.           }
46.         else
47.           syntax_error();
48.     }
49. }
```

Analyseur syntaxique (1)

```
enum { VAR, CST, ADD, SUB, LT, ASSIGN,
       IF1, IF2, WHILE, DO, EMPTY, SEQ, EXPR, PROG };

struct node { int kind; struct node *o1, *o2, *o3; int val; };
typedef struct node node;

node *new_node(int k)
{ node *x = (node*)malloc(sizeof(node)); x->kind = k; return x; }

node *paren_expr(); /* forward declaration */

node *term()  /* <term> ::= <id> | <int> | <paren_expr> */
{ node *x;
  if (sym == ID) { x=new_node(VAR); x->val=id_name[0]-'a'; next_sym(); }
  else if (sym == INT) { x=new_node(CST); x->val=int_val; next_sym(); }
  else x = paren_expr();
  return x;
}

node *sum()  /* <sum> ::= <term> | <sum> "+" <term> | <sum> "-" <term> */
{ node *t, *x = term();
  while (sym == PLUS || sym == MINUS)
    { t=x; x=new_node(sym==PLUS?ADD:SUB); next_sym(); x->o1=t; x->o2=term(); }
  return x;
}

node *test()  /* <test> ::= <sum> | <sum> "<" <sum> */
{ node *t, *x = sum();
  if (sym == LESS)
    { t=x; x=new_node(LT); next_sym(); x->o1=t; x->o2=sum(); }
  return x;
}

node *expr()  /* <expr> ::= <test> | <id> "=" <expr> */
{ node *t, *x;
  if (sym != ID) return test();
  x = test();
  if (x->kind == VAR && sym == EQUAL)
    { t=x; x=new_node(ASSIGN); next_sym(); x->o1=t; x->o2=expr(); }
  return x;
}

node *paren_expr()  /* <paren_expr> ::= "(" <expr> ")" */
{ node *x;
  if (sym == LPAR) next_sym(); else syntax_error();
  x = expr();
  if (sym == RPAR) next_sym(); else syntax_error();
  return x;
}
```


Analyseur syntaxique (2)

```
node *statement()
{ node *t, *x;
  if (sym == IF_SYM)  /* "if" <paren_expr> <statement> */
    { x = new_node(IF1);
      next_sym();
      x->o1 = paren_expr();
      x->o2 = statement();
      if (sym == ELSE_SYM)  /* ... "else" <statement> */
        { x->kind = IF2;
          next_sym();
          x->o3 = statement();
        }
    }
  else if (sym == WHILE_SYM)  /* "while" <paren_expr> <statement> */
    { x = new_node(WHILE);
      next_sym();
      x->o1 = paren_expr();
      x->o2 = statement();
    }
  else if (sym == DO_SYM)  /* "do" <statement> "while" <paren_expr> ";" */
    { x = new_node(DO);
      next_sym();
      x->o1 = statement();
      if (sym == WHILE_SYM) next_sym(); else syntax_error();
      x->o2 = paren_expr();
      if (sym == SEMI) next_sym(); else syntax_error();
    }
  else if (sym == SEMI)  /* ";" */
    { x = new_node(EMPTY); next_sym(); }
  else if (sym == LBRA)  /* "{" { <statement> } "}" */
    { x = new_node(EMPTY);
      next_sym();
      while (sym != RBRA)
        { t=x; x=new_node(SEQ); x->o1=t; x->o2=statement(); }
      next_sym();
    }
  else  /* <expr> ";" */
    { x = new_node(EXPR);
      x->o1 = expr();
      if (sym == SEMI) next_sym(); else syntax_error();
    }
  return x;
}

node *program()  /* <program> ::= <statement> */
{ node *x = new_node(PROG);
  next_sym(); x->o1 = statement(); if (sym != EOI) syntax_error();
  return x;
}
```


Générateur de code

```
enum { ILOAD,ISTORE,BIPUSH,DUP,POP,IADD,ISUB,GOTO,IFEQ,IFNE,IFLT,RETURN };

typedef signed char code;
code object[1000], *here = object;

void g(code c) { *here++ = c; } /* missing overflow check */
code *hole() { return here++; }
void fix(code *src, code *dst) { *src = dst-src; } /* missing overflow check *

void c(node *x)
{ code *p1, *p2;
  switch (x->kind)
    { case VAR   : g(ILOAD); g(x->val); break;
      case CST   : g(BIPUSH); g(x->val); break;
      case ADD   : c(x->o1); c(x->o2); g(IADD); break;
      case SUB   : c(x->o1); c(x->o2); g(ISUB); break;
      case LT    : g(BIPUSH); g(1);
                   c(x->o1);
                   c(x->o2);
                   g(ISUB);
                   g(IFLT); g(4);
                   g(POP);
                   g(BIPUSH); g(0); break;
      case ASSIGN: c(x->o2);
                   g(DUP);
                   g(ISTORE); g(x->o1->val); break;
      case IF1   : c(x->o1);
                   g(IFEQ); p1=hole();
                   c(x->o2); fix(p1,here); break;
      case IF2   : c(x->o1);
                   g(IFEQ); p1=hole();
                   c(x->o2);
                   g(GOTO); p2=hole(); fix(p1,here);
                   c(x->o3); fix(p2,here); break;
      case WHILE : p1=here; c(x->o1);
                   g(IFEQ); p2=hole();
                   c(x->o2);
                   g(GOTO); fix(hole(),p1); fix(p2,here); break;
      case DO    : p1=here; c(x->o1);
                   c(x->o2);
                   g(IFNE); fix(hole(),p1); break;
      case EMPTY : break;
      case SEQ   : c(x->o1);
                   c(x->o2); break;
      case EXPR  : c(x->o1);
                   g(POP); break;
      case PROG  : c(x->o1);
                   g(RETURN); break;
    }
}
```


Machine virtuelle et programme principal du compilateur

```
int globals[26];

void run()
{ int stack[1000], *sp = stack; /* missing overflow check */
  code *pc = object;
  again: switch (*pc++)
    { case ILOAD : *sp++ = globals[*pc++];              goto again;
      case ISTORE: globals[*pc++] = *--sp;              goto again;
      case BIPUSH: *sp++ = *pc++;                       goto again;
      case DUP   : sp++; sp[-1] = sp[-2];               goto again;
      case POP   : --sp;                                goto again;
      case IADD  : sp[-2] = sp[-2] + sp[-1]; --sp;      goto again;
      case ISUB  : sp[-2] = sp[-2] - sp[-1]; --sp;      goto again;
      case GOTO  : pc += *pc;                           goto again;
      case IFEQ  : if (*--sp == 0) pc += *pc; else pc++; goto again;
      case IFNE  : if (*--sp != 0) pc += *pc; else pc++; goto again;
      case IFLT  : if (*--sp <  0) pc += *pc; else pc++; goto again;
    }
}

int main()
{ int i;

  c(program());

  for (i=0; i<26; i++)
    globals[i] = 0;
  run();
  for (i=0; i<26; i++)
    if (globals[i] != 0)
      printf("%c = %d\n", 'a'+i, globals[i]);

  return 0;
}
```

Règles de compilation (1)

- C(<*expr*>) = code qui aura empilé la valeur de <*expr*> lorsque le point de contrôle atteint la fin du code

C(<*int*>)	= `BIPUSH` V(<*int*>)
C(<*id*>)	= `ILOAD` P(<*id*>)
C(<*sum*> `+` <*term*>)	= C(<*sum*>) C(<*term*>) `IADD`
C(<*sum*> `-` <*term*>)	= C(<*sum*>) C(<*term*>) `ISUB`
C(<*id*> `=` <*expr*>)	= C(<*expr*>) `DUP` `ISTORE` P(<*id*>)

C(<*sum*$_1$> `<` <*sum*$_2$>)	= `BIPUSH 1`
	C(<*sum*$_1$>)
	C(<*sum*$_2$>)
	`ISUB`
	`IFLT` *fin*
	`POP`
	`BIPUSH 0`
	fin:

Règles de compilation (2)

- Exemple

C(a+b<10)

= `BIPUSH 1`
C(a+b)
C(10)
`ISUB`
`IFLT` *fin*
`POP`
`BIPUSH 0`
fin:

= `BIPUSH 1`
C(a)
C(b)
`IADD`
`BIPUSH 10`
`ISUB`
`IFLT` *fin*
`POP`
`BIPUSH 0`
fin:

= `BIPUSH 1`
`ILOAD 0`
`ILOAD 1`
`IADD`
`BIPUSH 10`
`ISUB`
`IFLT` *fin*
`POP`
`BIPUSH 0`
fin:

Règles de compilation (3)

- C(<*statement*>) = code qui aura produit l'effet du <*statement*> lorsque le point de contrôle atteint la fin du code (la pile reste inchangée)

C(`if` <*paren_expr*> <*statement*>) = C(<*paren_expr*>) `IFEQ` *fin* C(<*statement*>) *fin:*
C(`if` <*paren_expr*> <*statement*$_1$> `else` <*statement*$_2$>) = C(<*paren_expr*>) `IFEQ` *else* C(<*statement*$_1$>) `GOTO` *fin* *else:* C(<*statement*$_2$>) *fin:*
C(`while` <*paren_expr*> <*statement*>) = *début:* C(<*paren_expr*>) `IFEQ` *fin* C(<*statement*>) `GOTO` *début* *fin:*
C(`do` <*statement*> `while` <*paren_expr*> `;`) = *début:* C(<*statement*>) C(<*paren_expr*>) `IFNE` *début*
C(<*expr*> `;`) = C(<*expr*>) `POP`

Règles de compilation (4)

- Exemple

C(`while (a+b<10) a = a+b;`)

= *début:*
C(a+b<10)
`IFEQ` *fin*
C(a = a+b;)
`GOTO` *début*
fin:

= *début:*
C(a+b<10)
`IFEQ` *fin*
C(a = a+b)
`POP`
`GOTO` *début*
fin:

= *début:*
`BIPUSH 1`
`ILOAD 0`
`ILOAD 1`
`IADD`
`BIPUSH 10`
`ISUB`
`IFLT` *f*
`POP`
`BIPUSH 0`
f:
`IFEQ` *fin*
`ILOAD 0`
`ILOAD 1`
`IADD`
`DUP`
`ISTORE 0`
`POP`
`GOTO` *début*
fin: