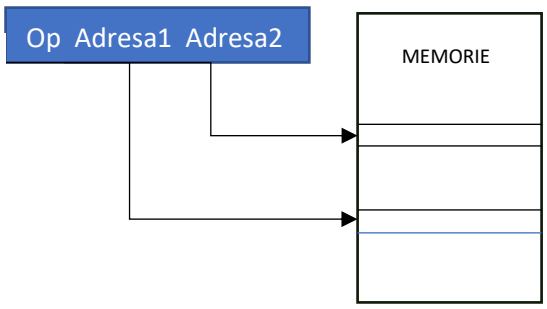
TEHNICI DE COMPILARE – CURSUL 10

GENERAREA CODULUI ORIENTATĂ CĂTRE SINTAXĂ

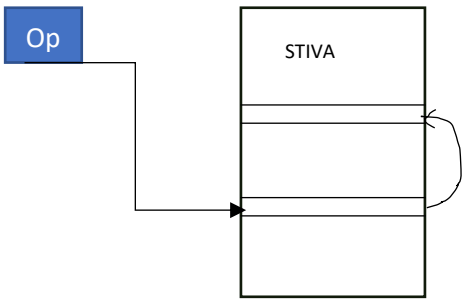
Codul-mașină generat poate fi de mai multe tipuri

- RISC (Reduced Instruction Set Computer). Folosește mai mulți regiștri, instrucțiuni cu 3 adrese, un set relativ simplu de instrucțiuni.
- CISC (Complex Instruction Set Computer). Utilizat de x86, x86-64, amd64. Folosește câțiva regiștri, instrucțiuni cu 2 adrese, un set mai complex de instrucțiuni.



2 adrese Memorie - Memorie

- Cod-mașină tip stivă. Folosit pentru generarea de cod intermediar. Mașina virtuală Java folosește un astfel de sistem.



0 adrese

- Cod-mașină bazat pe acumulator.

Itty Bitty Stack Machine

Este un calculator stivă cu un singur mod de adresare. Aceasta face ușoară translatarea anumitor limbaje.

IBSM are 4 regiștri care nu sunt direct programabili:

- Program Counter (PC) – avansează în codul mașină pe măsură ce programul este executat.
- Stack Pointer (SP) – conține adresa vârfului stivei de control. Stiva crește de la adrese mici la adrese mai mari.
- Frame Pointer (FP) – conține adresa de bază pentru instrucțiunile de acces la memorie: $load, store$.
- Limit – conține dimensiunea stivei (folosit pentru prevenirea depășirilor).

IBSM are 32 de instrucțiuni, codificate de la 0 la 31. Fiecare instrucțiune poate fi codificată pe 5 biți, astfel că 3 instrucțiuni pot fi împachetate pe un cuvânt de 16 biți. IBSM lucrează cu o instrucțiune pe cuvânt. Când sunt împachetate, se execută mai întâi instrucțiunea codificată pe cei mai puțin semnificativi 5 biți, apoi următoarea etc.

Ramificările, apelurile de procedură/funcție și revenirile din procedură/funcție fac ca execuția să continue cu prima instrucțiune a cuvântului adresat.

Nu există adresare la byte. Cuvântul de 16 biți este considerat atomic.

La lansarea în execuție, IBSM citește o singură adresă, din locația de memorie (absolută) 0. Această adresă devine SP. Inițial, IBSM extrage din stivă 3 cuvinte care reprezintă valorile regiștrilor Limit, FP, PC. Astfel:

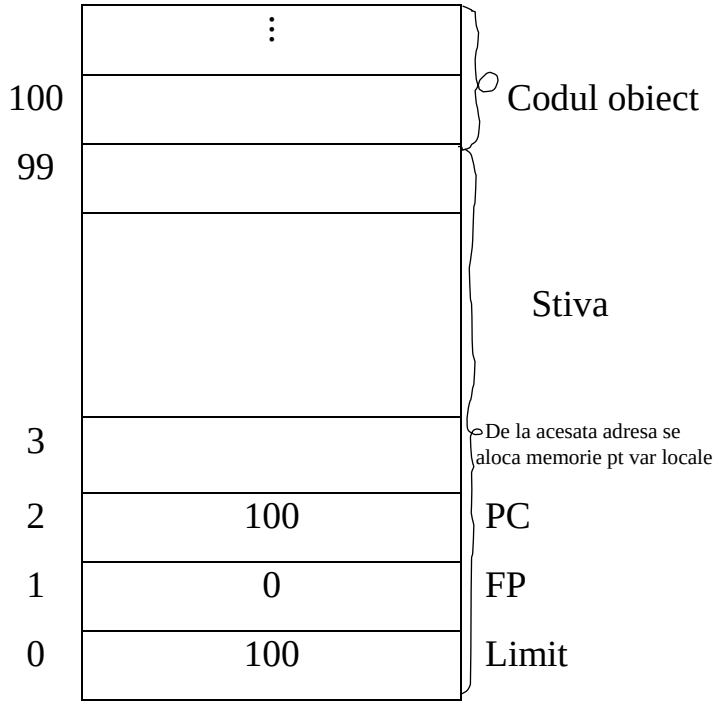
SP 3 (inițial, înainte de extragerea celor 3 cuvinte din stivă)

FP 0

PC 100

SP 0 (după extragerea celor 3 cuvinte)

Numărul -1 este utilizat ca un cod escape. Dacă este urmat de alt -1, se continuă execuția de la următorul cuvânt. Dacă numărul ce urmează după -1 este o adresă validă, execuția continuă de la acea adresă. Dacă nu este o adresă validă, se



semnalează sfârșit de fișier, execuția continuă prin încărcarea lui SP de la adresa absolută 0 și din extragerea de pe stivă a celorlalți regiștri.

Codurile operațiilor IBSM și semnificația operațiilor:

00	Nop	No operation
01	BrFalse	Branche if False. {pop a; pop b; if b=0 PC=PC+a}; executia continua cu prima instructiune a cuvantului (adresei) din PC
02	Rezervat	
03	Call	Apeleaza o procedura/functie a carei adresa este in varful stivei. Se interschimba valorile lui PC si cea din varful stivei
04	Enter	Se intra intr-o procedura/functie. Este extrasa valoarea n din varful stivei si este inlocuita de adresa continuta de FP (legatura dinamica la procedura, utilizata in header-ul sau), seteaza FP cu legatura statica (care este al doilea cuvant dupa legatura dinamica), apoi ajusteaza SP cu n cuvinte inainte (pentru rezervarea de spatiu pentru variabilele locale)
05	Exit	Iesirea dintr-o procedura in care s-a intrat cu Enter. Se extrage valoarea n din varful stivei, seteaza SP cu legatura dinamica
06	Rezervat	
07	Rezervat	
08	Dupe	Duplica varful stivei
09	Swap	pop a; pop b; push a; push b
10	Rezervat	
11	Mpy	pop a; pop b; push b*a
12	Add	pop a; pop b; push b+a
13	Xor	Sau exclusiv; pop a; pop b; push (b xor a)
14	Or	pop a; pop b; push (b or a)
15	And	pop a; pop b; push (b and a)

16	Equal	pop a; pop b; if (b=a) push 1 else push 0
17	Less	pop a; pop b; if (b<a) push 1 else push 0
18	Greater	pop a; pop b; if (b>a) push 1 else push 0
19	Not	Complementeaza bit cu bit cuvantul din varful stivei
20	Neg	Negate. Pop a; push -a
21	Rezervat	
22	Rezervat	
23	Rezervat	
24	Stop	
25	Global	Scade FP din cuvantul aflat in varful stivei
26	Store	Pop value; pop address; store value in memory (address)
27	Load	Pop address, push memory(address)
28	LoadCon	Pune pe stiva valoarea aflata la adresa din PC si incrementeaza PC cu 1.
29	Nibble	Pune pe stiva ca un cuvant valoarea urmatorilor 5 biti din cuvantul ce desemneaza instructiunea curenta, fara sa o execute. Daca mai exista alti 5 biti in cuvantul curent, atunci executia va continua cu acea instructiune, altfel continua cu instructiunea de la cuvantul ce apare in PC
30	Zero	Push 0 (ca un cuvant)
31	One	Push 1 (ca un cuvant)

Exemplu pentru instrucțiunea a := a-1

	Stiva
LoadCon <adresa_a>	adresa_a
LoadCon <adresa_a>	adresa_a adresa_a
Load	val_a adresa_a
LoadCon 1	1 a adresa_a
Negate	-1 a adresa_a
Add	a-1 adresa_a
Store	(empty)

Codul IBSM pentru instrucțiune a := a-1 (adresa lui a este 3)

28
3
28
3
27
28
1
20
12
26

Generarea codului IBSM cu ajutorul gramaticilor cu atribute

Codul va fi generat cu ajutorul acțiunilor semantice. Pentru aceasta introducem neterminalul O (de la Output Object code), care are asociat un atribut moștenit a cărui valoare este un cod IBSM, cuprins între 0 și 31. Semantica neterminalului O introduce 2 rutine: $member$ – generează un întreg zecimal în fișerul de ieșire și $newline$ – generează sfârșit de linie. Sintactic, O nu introduce nimic nou: $O \rightarrow \lambda$.

$O \downarrow value \rightarrow [member \downarrow value; newline]$

Fiecare variabilă va avea alocată în tabela de simboluri adresa din memoria-stivă în care este stocată valoarea sa. Astfel, fiecare identificator va avea asociată valoarea

$$value = tip + loc \times 1000$$

Neterminalul V va mai avea asociate: un atribut sintetizat care reprezintă următoarea adresă liberă din memorie și un atribut moștenit a cărui valoare este adresa din memorie în care va fi stocată variabila curent declarată.

Gramatica atributată este:

$Z \rightarrow$ "PROGRAM" $ID \uparrow idn$ ";"

[" 3 100 0 100 -1 100"; $newline$]

"VAR" $V \downarrow \emptyset \downarrow 3 \uparrow tabOut \uparrow varOut$

$O \downarrow 28 O \downarrow varOut\ O \downarrow 4$ // pt Enter
„BEGIN" $B \downarrow tabOut$

$O \downarrow 24$ [" $-1\ -32$ ", $newline$]

"END" "."

$V \downarrow tabIn \downarrow LocIn \uparrow tabOut \uparrow locOut$

$\rightarrow ID \uparrow idn$ ":" $T \uparrow type$ ";"
$[value \leftarrow type + locIn \times 1000]$
$[into \downarrow tabIn \downarrow idn \downarrow value \uparrow tabNext]$
$V \downarrow tabNext \downarrow locIn + 1 \uparrow tabOut \uparrow locOut$
$\rightarrow [tabOut \leftarrow tabIn; locOut \leftarrow locIn]$

$T \uparrow type \rightarrow$ "INTEGER" $[type \leftarrow 1]$
$\rightarrow$ "BOOLEAN"$[type \leftarrow 2]$

$B \downarrow tabIn \quad \rightarrow ID \uparrow idn$
$[from \downarrow tabIn \downarrow idn \uparrow value]$
$[loc \leftarrow value\ div\ 1000]$
$O \downarrow 28\ O \downarrow loc$ ":=" $E \downarrow tabIn \uparrow type\ O \downarrow 26$";" // store
$[type == value\ mod\ 1000]$
$B \downarrow tabIn$
$\rightarrow \lambda$

$E \downarrow tabIn \uparrow type \rightarrow S \downarrow tabIn \uparrow stype\ C \downarrow tabIn \downarrow stype \uparrow type$

$C \downarrow tabIn \downarrow typeIn \uparrow typeOut \rightarrow$ "$=$"$S \downarrow tabIn \uparrow stype\ O \downarrow 16$ // Equal
$[stype == typeIn; typeOut \leftarrow 2]$
$\rightarrow [typeOut \leftarrow typeIn]$

$S \downarrow tabIn \uparrow type \rightarrow F \downarrow tabIn \uparrow type\ R \downarrow tabIn \downarrow type$

$R \downarrow tabIn \downarrow type \rightarrow$ "$*$"$F \downarrow tabIn \uparrow ftype\ O \downarrow 11$ // Mpy
$[ftype == type; type == 1]$
$R \downarrow tabIn \downarrow ftype$
$\rightarrow$ "AND" $F \downarrow tabIn \uparrow ftype\ O \downarrow 15$ // And
$[ftype == type; type == 2]$
$R \downarrow tabIn \downarrow ftype$
$\rightarrow \lambda$

$F \downarrow tabIn \uparrow type \rightarrow$ "(" $E \downarrow tabIn \uparrow type$ ")"
$\rightarrow ID \uparrow idn$
$[from \downarrow tabIn \downarrow idn \uparrow value]$
$[type \leftarrow value\ mod\ 1000; loc \leftarrow value\ div\ 1000]$
$O \downarrow 28\ O \downarrow loc\ O \downarrow 27$ // Load <loc>
$\rightarrow NUM \uparrow val\ O \downarrow 28\ O \downarrow val\ [type \leftarrow 1]$
$\rightarrow$ "TRUE" $O \downarrow 28\ O \downarrow 1\ [type \leftarrow 2]$ // One
$\rightarrow$ "FALSE" $O \downarrow 28\ O \downarrow 0\ [type \leftarrow 2]$ // Zero

Semantica instrucțiunii *if*

IF b THEN x ELSE y

- b expresie booleană
- x, y secvențe de instrucțiuni

<cod de evaluare a lui b>
LoadCon <offset e-t>
BranchFalse
t: <cod pentru x>
LoadCon 0

LoadCon <offset j-e> // pentru salt la adresa j
BranchAlways
e: <cod pentru y>
j: <cod pentru ce urmeaza dupa IF>

Problema ramificării înainte

Există două moduri de a aborda această problemă.
1) Programul este compilat de 2 ori. Prima dată, toate adresele ramificațiilor sunt înregistrate pentru a fi completate la cea de-a doua trecere. În cea de-a doua trecere, se generează cod.
Compilatoarele în doi pași sunt necesare pentru acele limbaje în care variabilele nu sunt declarate înainte de utilizare, deoarece nu se știe de la primul pas ce cod trebuie generat pentru o referință la o variabilă arbitrară.
2) Prin a doua metodă, se păstrează o înregistrare pentru fiecare adresă incompletă și când această adresă devine cunoscută se face un jump în cod pentru a completa această adresă sau offset-ul corespondent. Această metodă, deși necesită o anumită nesecvențialitate în fișierul ce conține codul obiect, are avantajul compilării într-un singur pas.
Această tehnică se numește $backpatching$. Calculul unui offset se face pe baza adreselor instrucțiunilor deja generate, ceea ce înseamnă că aceste adrese sunt accesibile.
Vom folosi un atribut sintetizat pentru completarea offset-urilor.

$Stmts \downarrow LocIn \uparrow locOut$
$\rightarrow$ $"IF"\ BoolExp \downarrow locIn \uparrow locEx$
$EMIT \downarrow 28 \downarrow locEx \uparrow locOff1$ // adresa pentru primul offset, e-t
$EMIT \downarrow 0 \downarrow locOff1 \uparrow locBrf$ // este 0 pentru ca inca nu se cunoaste e-t
$EMIT \downarrow 1 \downarrow locBrf \uparrow locThen$
$"THEN"\ Stmts \downarrow locThen \uparrow locLdz$
$EMIT \downarrow 30 \downarrow locLdz \uparrow locLdc$ // se pune 0 pe stiva, pt salt neconditionat
$EMIT \downarrow 28 \downarrow locLdc \uparrow locOff2$ // adresa pentru al doilea offset, j-e
$EMIT \downarrow 0 \downarrow locOff2 \uparrow locBra$
$EMIT \downarrow 1 \downarrow locBra \uparrow locElse$
$"ELSE"\ Stmts \downarrow locElse \uparrow locOut$
$Backpatch \downarrow locOut \downarrow locOff1 \uparrow locElse - locThen$
$Backpatch \downarrow locOut \downarrow locOff2 \uparrow locOut - locElse$
$\rightarrow OtherStmts \downarrow locIn \uparrow locOut$

$Backpatch \downarrow locn \downarrow locOff \uparrow value$
$\rightarrow$ $[" - 1";\ number \downarrow locOff;\ newline]$
$[number \downarrow value; newline]$
$[" - 1";\ number \downarrow locn;\ newline]$

$EMIT \downarrow value \downarrow locIn \uparrow locOut$

$\rightarrow [number \downarrow value;\ newline; locOut \leftarrow locIn + 1]$

Neterminalul $Backpatch \downarrow locn \downarrow locOff \uparrow value$ generează un cuvânt (16 biți) cu valoarea $value$ (la care inițial fusese încărcată valoarea 0) la adresa $locOff$, după care se întoarce (jump, cu ajutorul secvenței "-1") la adresa $locn$.

Neterminalul $EMIT \downarrow value \downarrow locIn \uparrow locOut$ este folosit pentru generarea la adresa curentă, în fișierul care conține codul obiect, a constantei $value$, după care actualizează adresa.

Exerciții

1) Care este codul generat pentru fragmentul de program următor?

$a := 3;$

$IF\ a < b\ THEN\ b := 1\ ELSE\ b := 5;$

$c := 0\ ;$

2) Folosind tehnica Backpatching, care sunt producțiile atributate pentru generarea codului pentru instrucțiunea $WHILE\ b\ DO\ x$

- b expresie booleană
- x secvență de instrucțiuni

```
t:      <cod de evaluare a lui b>
        LoadCon <offset e-b>
        BranchFalse        // daca b este false, salt la adresa e
b:      <cod pentru x>
        LoadCon 0                  // salt neconditionat
        LoadCon <offset t-e>
        BranchAlways               // ne intoarcem la adresa t
e:      <cod pentru ce urmeaza>
```

Anexa 1: Revenirea din eroare într-un parser de tip LR

Pentru fiecare situație de eroare se implementează o rutină de eroare.
Există două modalități:
1) Se analizează starea curentă pentru care s-a obținut eroare. Dacă pentru respectiva stare există un token pentru care avem deplasare sau reducere, atunci se consideră că în intrare ar exista acel token și se efectuează acțiunea corespondentă, de regulă deplasare (adică se plaseaza pe stiva de lucru acel token și noua stare a parser-ului), după care se continuă parsarea. Eventual, putem

repeta aceasta dacă în noua stare se obține din nou eroare pentru token-ul curent. Există un număr finit de astfel de repetări posibile.
2) Se ignoră token-ul curent și pentru starea curentă se verifică dacă se poate continua parsarea fără eroare pentru următorul token. Repetând acest pas, în caz că se obține în continuare eroare, se va afișa întregul șir de erori, fie până la întalnirea unei configurații din care se poate continua fără eroare, fie până la finalul fișierului.

Se da gramatica G cu productiile E-> E+E | E*E | n
1: 2: 3:
Sa se construiasca tabela SLR(1) pentru G.
Extindem G: E'->E: adaugam simbolul terminal #
Calculam Follow(X), X neterminal; se initializeaza Follow(E)={#}

Follow(E)	#, +, *

E->E+E Follow(E)+=First(+E.Follow(E))={+}
E->E*E Follow(E)+=First(*E.Follow(E))={*}
Se calculeaza multimile canonice LR(0)

Tabela SLR(1), după eliminarea ambiguităților (conflictelor), este:

	+	*	n	#	E
0	Error1()	Error1()	Shift 2	Error2()	1
1	Shift 3	Shift 4	Error3()	accept	
2	Reduce 3	Reduce 3	Error3()	Reduce 3	
3	Error1()	Error1()	Shift 2	Error1()	5
4	Error1()	Error1()	Shift 2	Error1()	6
5	Reduce 1	Shift 4	Error3()	Reduce 1	
6	Reduce 2	Reduce 2	Error3()	Reduce 2	

Error1()={ print("se asteapta un operand"; push 'n'; push 2;}
Error2()={print("Expresie vida");}
Error3()={print("Se asteapta un operator"); push '+'; push 3;}