Семантика

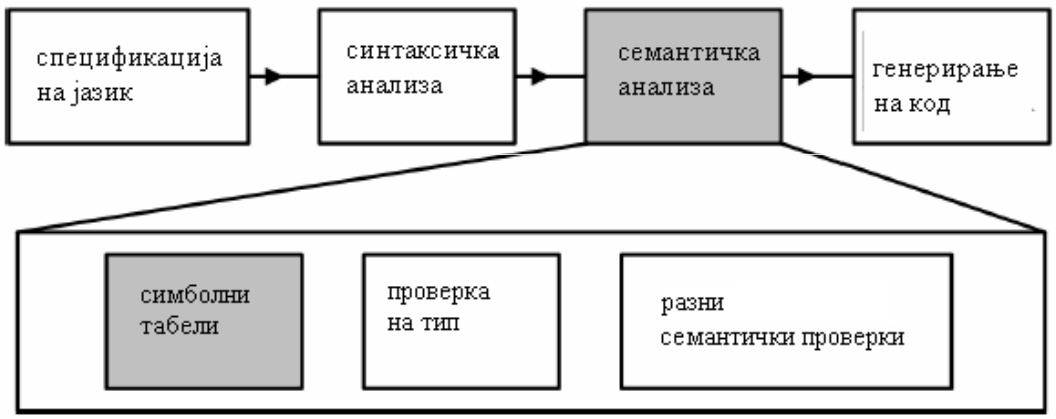
Семантика

- Што е разлика меѓу синтакса и семантика?
- Синтаксата се гриши за точноста на напишаната реченица
- Семантиката се гричи дали таа има значење.
- Пример:
  - Гласно цвеќињата зборуваат розовите - Не е ситаксички точна
  - Розовите цвеќиња зборуваат гласно –точна синтаксички, но не можат цвеќиња да зборуваат, па не значи ништо
- Семантичката анализа го проверува значењето на реченицата и ако има повеќекратно значење и ако генерира грешка ако значењето не е јасно

Семантика

- Можно е да се напише програма која е синтаксички точна, но сеуште ги крши правилата на јазикот
- Пример: myFunc() = 6;
  - Не може да се стави вредност на функција
  - Работи како оваа може да се однесуваат чудно кога ќе се извршува програмата. Не секогаш програмата работи баш онака како што мислиме дека ќе работи.
- Некои работи се комплексни за синтаксичката анализа, па тука доаѓа семантичката анализа
- Неопходна е проверка на типовите и за тоа се потребни дополнителни информации.

Симболна табела



Вовед во симболни идентификации

- Во времето на компајлирање треба да ги зачуваме различните симболи(функции или променливи) кои се декларирани во програмскиот код.
- Секој симбол мора да се идентификува со единствено име, за да подоцна истиот можеме де го најдеме
- Пример (неточна програма)

```
module awed;
start main: void -> void
{
        2*4; // rezultatot se gubi
        myfunction ( 3 ); // funkcijata ne postoi
}
```

- Без идентификација на симболите, нема да можеме да ги повикаме ниту да ги означиме

Опсег

- Дефиниција во Webster's Revised Unabridged Dictionary (1913):

  Просторија или можност за бесплатен поглед или цел; простор за акција; амплитудата на можности; слободен курс или пропустлвост; слобода; опсег на гледање, намера, или дејство.
- Во контекст на програмски јазици сметаме дека опсег е опсег на гледање.
- Опсегот го лимитира рангот во кој го бараме симболот.

Опсег

```
module example; // begin scope (global)

int a = 4;
int b = 3;

start main: void → void
{ // begin scope (main)
  float a = 0;

  { // begin scope (free block}
    char a = 'a';
    int x;
    print ( a );
  } // end of scope {free block}
  x = 1; // x is not in range of view!
  print ( b );
} // end of scope (main)

// end of scope 0 (global)
```

• 3 вгнездени повикувања на a, кои се перфектно легални, затоа што се во делови кои сеуште не содржат симбол со име `a'.

•Inger дозволува само повикување на симболи кои се дефинирани во локалниот опсег и во родителот

• Изразот x = 1; не е добар, бидејќи x се декларира во опсег кој е дете на тој во кој се јавува оваа наредба.

• Ова ни дозволува да користиме исто име во различни опсези

Симболна табела

- Симболите кои се собираат во текот на парсирањето мора да бидат зачувани, за да подоцна се користат во текот на семантичката анализа
- Треба да се дефинира јасна структура на податоци во која ќе се зачувуваат битните информации за симболите и опсегот во кој се наоѓаат
- Ваквата структура на податоци се нарекува Симболна табела
- Симболната табела може да се имплементира на повеќе начини
  - Низа
  - Поврзани листи
  - Хаш табели
  - Бинарно пребарувачко дрво
  - n-арно пребарувачко дрво

Динамички наспроти статички табели

- Постојат два типа на симболни табели, динамички и статички
- Динамичките симболни табели можат да се користат само кога и информациите кои се собираат за даден симбол и нивното користење се врши во едно минување
  - Тие работат како стек: кога ќе се дојде до симболот, тој се пушта низ стекот
  - Кога ќе се напушти опсегот се бришат сите симболи кои припаѓаат на тој опсег
- Статичките таблеи се градат еднаш и можеме низ нив да поминуваме онолку пати колку што сакаме
  - Тие се уништуваат кога компајлерот ќе ја заврши својата работа

Динамички наспроти статички табели

```
    module example;

    int v1, v2;

5   f: int v1, v2 → int
    {
        return (v1 + v2);
    }

10  start g: int v3 → int

{
    return  (v1 + v3);
}
```

Илустрација на растење на динамичка табела

- Во линија 3 во симболната табела се ставаат симболите
  T = {v1; v2}
- После линија 5 симболната табела исто така ја содржи и функцијата f и локалните променливи v1 и v2
  T = {v1; v2; f; v1; v2}
- Во линија 9 симболната табела се враќа во глобалната форма
  T = {v1; v2}
- После линија 10 симболната табела се проширува со функцијата g и локалната променлива v3
  T = {v1; v2; g; v3}

Илустрација на растење на статичка табела.

- После линија 3 во симболната табела се ставаат симболите
  T = {v1; v2}
- После линија 5 симболната табела исто така ја содржи и функцијата f и локалните променливи v1 и v2
  T = {v1; v2; f; v1; v2}
- После линија 10 симболната табела се проширува со функцијата g и локалната променлива v3
  T = {v1; v2; f; v1; v2; g; v3}

Селекција на структура на податоци

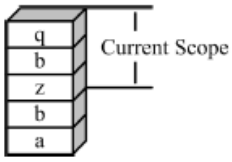
- Критериум
  - Примарни цели:
    - Зачувување на информациите
    - Брзо пребарување
  - За Inger
    - Лесна за користење
    - Брза за имплементација

Споредба на структурите на податоци

- Низа
  - прва што паѓа на ум
  - Иако е погодна има практични ограничувања
  - Имаат статичка големина(ако големината е 256 а имаме 257 симболи ќе се јави грешка)
  - Пребарување на низа е линеарен алгоритам. Може да се искористи бинарен алгоритам за пребарување, но тоа значи цело време да се сортира.
  - Би била корисна само ако се работи со динамичка табела и не се дозволени опсези

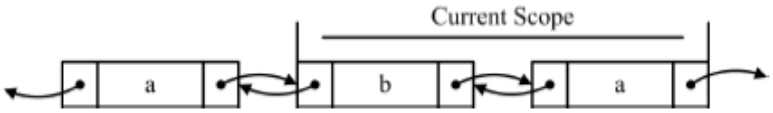
Споредба на структурите на податоци

- Стек
  - Наоѓањето на симболот е лесно, кога саканиот симбол е на врвот на стекот
  - Првата најдена променлива е последната која сме ја ставиле
  - Лесно е за опсези од различни нивоа
  - Ако се бара пребарување низ стекот, треба тој да се реконструира, а тоа е скапа операција
  - Може да биде добро за динамичка табела



Споредба на структурите на податоци

- Поврзана листа на симболи
  - Ако се имплементира како двојна поврзана листа може да се користи исто како стек (да се додава на крај и да се пребарува од крајот) без многу операции ставање и вадење.
  - Линерано време, но операциите се поевтини откотку кај стек



Споредба на структурите на податоци

- Бинарно дрво
  - Го подобрува пребарувањето, но само во сортирана форма
  - Се губи од предноста на стекот или двојно поврзаните листи во кои е лесно да се задржи опсегот (сега првиот најден симбол не е последниот додаден, туку најверојатно првиот додаден)
  - Постојано ребалансирање на дрвото

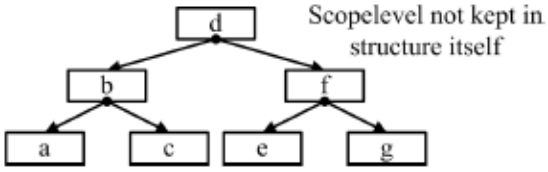


Figure 9.4: Binary Tree

Споредба на структурите на податоци

- n-арно дрво
  - Секој јазол има n деца и секој се однесува на нов опсег, како дете на опсегот родител
  - Секој јазол е опсег и сите симболи се зачувуваат во тој јазол
  - кога се потребни информации за некој симбол, треба само да пребаруваме по него и неговите родители
  - Единствена добра за имплементација статичка симболна таблела

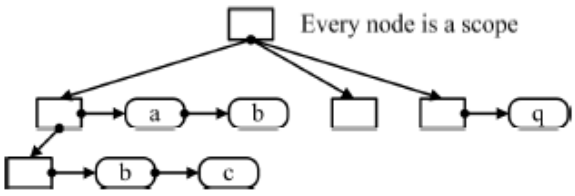


Figure 9.5: N-ary Tree

Структура на податоци за Inger

- Комбинација од n-арно дрво со поврзани листи
- Секој јазол во листата, кој репрезентира опсег, ќе содржи корен од бинарно дрво
- Главна придобивка е дека може лесно и брзо да се додаваат и одземаат опсези
  - После првото поминување, табелата веќе не се реконструира - и не треба
  - Само треба бргу да се пребарува

Типови

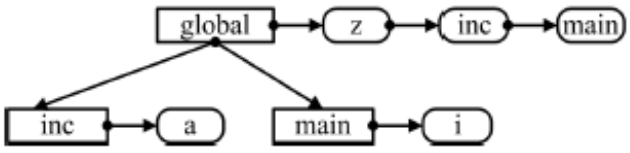
- Не е доволна само симболната табела. Таа треба да се декорира со информации, како што се типот кој го враќа
  - за таа цел се прават логички структури за симболите и типовите
  - Множество функции кои враќаат тип: CreateType(), AddSimpleType(), AddDimension(), AddModier(),. . . .

Пополнување на симболната табела

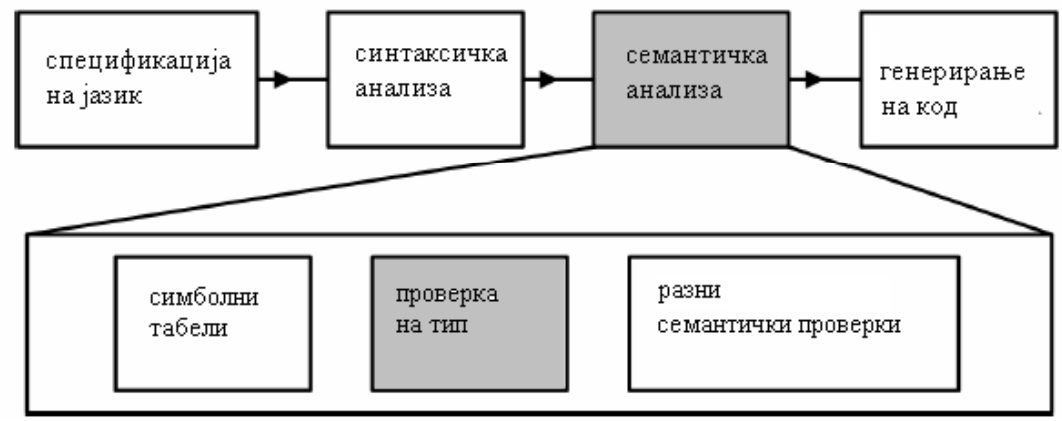
- Илустрација како симболната табела се пополнува со помош на апстракното синтаксно дрво
  1. Се започнува од коренот на АСД.
  2. За секој блок до кој се доаѓа се додава ново дете во соодветниот опсег и ова дете се прави нов опсег
  3. За секоја декларација која се наоѓа се вади
     - Име на променливата
     - Тип на променливата
  4. За секоја функција која се наоѓа се вади
     - Име на функцијата
     - Тип на функцијата, со почеток со типот кој го враќа
  5. Кога ќе се заврши со тој блок се придвишуваме кон родителот на опсегот

Пример

```
module test_module;

int z = 0;

inc: int a → int
{
    return( a + 1 );
}

start main: void → void
{
    int i;
    i = (z * 5) / 10 + 20 − 5 * (132 + inc ( 3 ) );
}
```



Проверка на тип



Проверка на тип

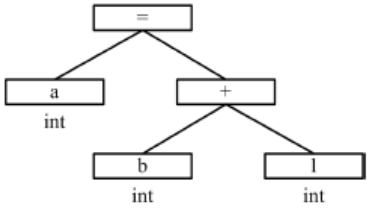
- Целта на проверката на типови е да се евалуира секој оператор, апликација или израз кој враќа вредност во АСД и да се пребарува по операндите или аргументите.
- И двата операнди или аргументите мора да бидат компатибилни типови и да формираат валидна комбинација со операторот.
  - Пример, ако имаме оператор +, лев операнд integer и десен char покажувач, собирањето не е валидно ако не се направи претопување.
- Проверувачот на типови ги проверува сите јазли и дава грешка ако не може да најде соодветно решение (со претопување) за типовите кај кои има конфликт.

Имплементација

- Процесот на проверка на типови се состои од два дела:
  - Декорирање на АСД со листови со типови
  - Поместување на овие типови нагоре по дрвото за да се пресмета:
    - Точноста на типот за операторите
    - Точноста на типот за аргументите на функциите
    - Точноста на типот за наредбите кои враќаат вредност
    - Ако типовите не се совпаѓаат во нивната форма да се обиде да направи претопување
  - Да се помести последната проверка на тип за да се осигура дали е точно нивото на индирекција (т.е. Да се означи int како покажувач променлива).

Декорирање на АСД со типови

- Се препорачува да се шета во обратен редослед и да се бараат литерални идентификатори или вредности.
  - Ако се оде во правиот редослед ќе треба дрвото да се минува двапати
  - Кога ќе се најде литерална вредност, типот треба да се лоцира во симболната табела. При тоа треба да се зачува тековниот опсег.
- Потоа се придвижуваме нагоре по дрвото и се евалуираат типовите на унарните, бинарните и апликативните јазли
- Пример: a = b + 1;. каде a и b се integer.



Декорирање на АСД со типови

- Што ќе се случи ако еден од листовите е реален број а другиот integer?
  - Еден начин да се работи со ова е да се креира табела на приоритет на конверзии. Се конвертира во типот со најголем приоритет, според табелата.
  - Најголемиот приоритет е на врвот на табелата
- Потоа се прави листа на типови во која се кажува што во што може да се конвертира
  - Во Inger може да се конвертира integer во float, но не може integer во стринг
- Оваа табела се нарекува табела на претопување

Node	Type
NODE_ASSIGN	FLOAT
NODE_ASSIGN	INT
NODE_ASSIGN	CHAR
NODE_BINARY_ADD	FLOAT
NODE_BINARY_ADD	INT

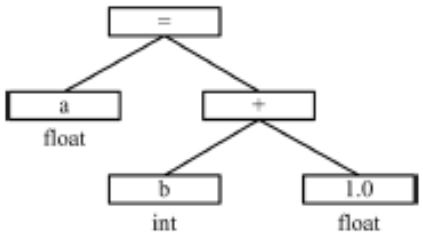
Table 10.1: Conversion priorities

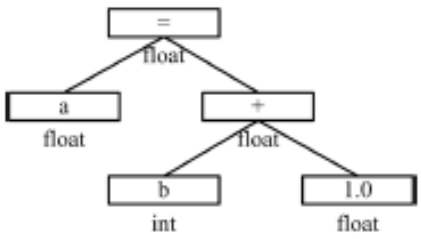
From type	To type	New node
INT	FLOAT	NODE_INT_TO_FLOAT
CHAR	INT	NODE_CHAR_TO_INT
CHAR	FLOAT	NODE_CHAR_TO_FLOAT

Table 10.2: Coercion table

Декорирање на АСД со типови

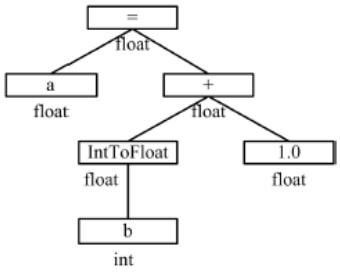
- Пример a = b + 1.0;. a-float b –integer, литералот 1.0 -float.



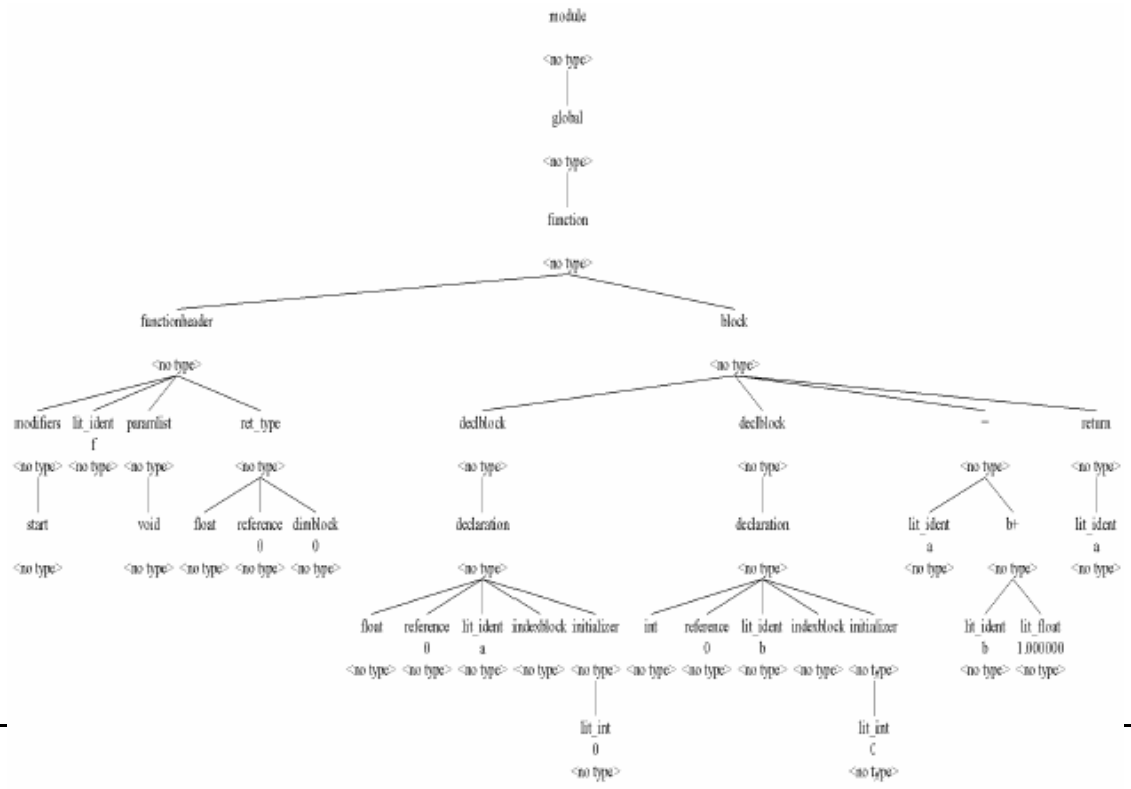


Претопување

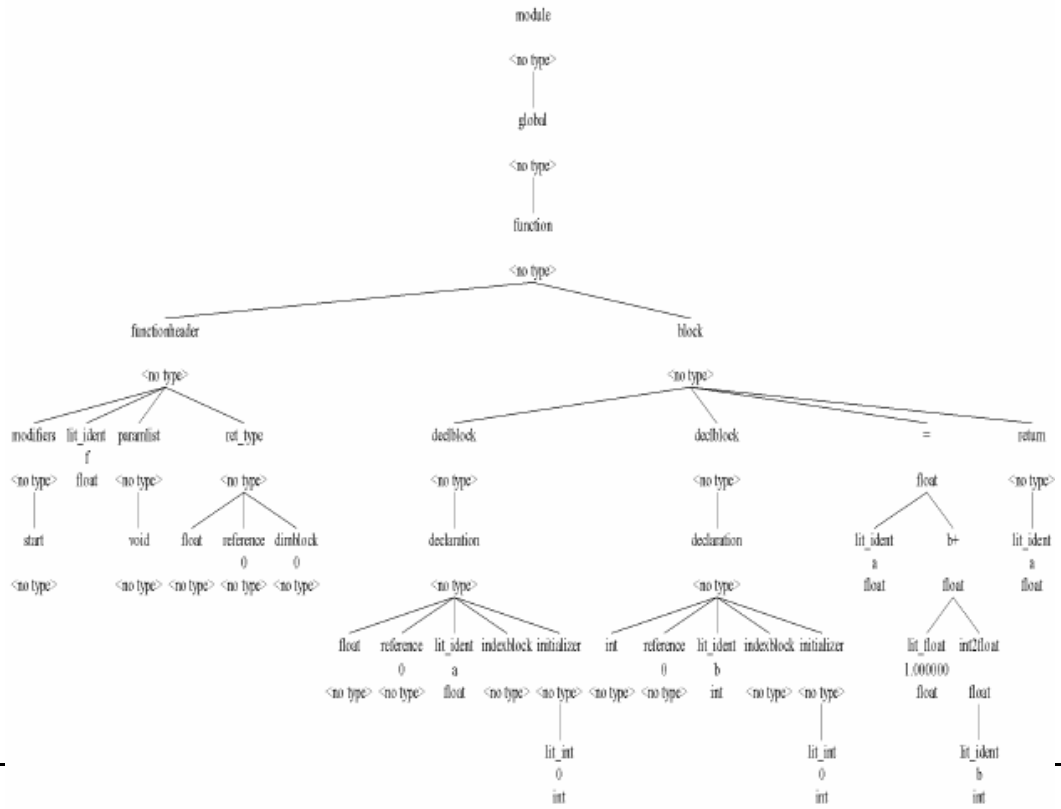
- После декорацијата на АСД и откога се направени сите проверки треба да се направи избор што понатаму
  - Може да се почне со генерирање на код-во овој случај модулот за проверка на типови е комплетно готов и одговорноста за конверзија на типовите е оставена на генераторот на код.
  - Да се спреми АСД за модулот за генерирање на код – овде одговорноста за конверзија ја има модулот за проверка на типови. Веќе дрвото е декорирано и нема потреба од повторно евалуирање на типовите од страна на генераторот на код.
- За да се спреми АСД за генерирање на код треба да се искористи техника претпоување, што значи конверзија од еден во друг тип.
  - Пример $N \subseteq R$
  - Ова се прави со додавање на други јазли, наречени претопоувачки јазли.



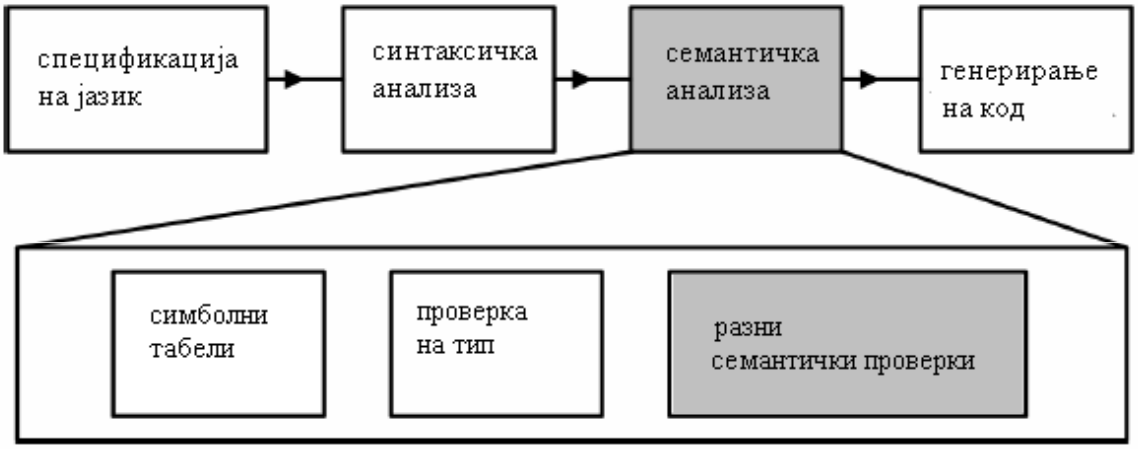
АСД декорација-пред



АСД декорација-после



Разни семантички проверки



Разни семантички проверки – вредност од левата страна

- Lvalue (вредност од левата страна) е изразот од левата страна на еднаквоста.
- Една од неопходните проверки. Се проверува дали има неточни означувања во изворниот код.
- Ова може да се направи ако се дефинира што е добра лева вредност а што не, па да се проверува по таа дефинирана листа

- Пример за лошо дефинирана лева вредност
  function () = 6;
  2 = 2;
  "somestring" = "somevalue";

- Пример за добро дефинирана лева вредност
  int a = 6;
  name = "janwillem";

Разни семантички проверки –вредност од левата страна

- За да се провери валидноста на левата вредност треба да се прошета по целото АСД.
- Пример за алгоритам за проверка:

```
Pocni od korenot na ASD
for sekoj jazol najden vo ASD do
    if jazolot e operator '=' then
5     proveri go negovoto najlevo dete vo ASD
          koja e nejzinata lvalue I vidi if e validna.
      if ne e validna javi greshka

    else go to naredniot jazol
else go to naredniot jazol
```

Разни семантички проверки – параметри на функција

- Параметрите на функцијата мора да се проверат пред да се почне со генерирање на кодот.
- Треба да се провери точниот број на параметри и дали нивниот тип е точен.
- Проверката на бројот на аргументи е директна и се состои од два чекори
  - Прво се собираат сите насловни јазли од AST и се зачувуваат во листа
  - Потоа се споредува дали бројот на аргументи кои се користат во секоја функција и бројот на аргументи кои се бараат од секоја од функциите е еднаков

Разни семантички проверки –клучните зборови return

- Механизамот за проврка на тип во компајлерот на Inger проверува дали функцијата го враќа точниот тип со тоа што како тип што го враќа функцијата се означува дека е типот кој е return вредност.
- Inger е направен така да било која функција мора да враќа вредност, без разлика дали е таа non-void функција или return не е последната наредба во функцијата.

Разни семантички проверки –клучните зборови return

- Код до кој не се стига: кодот после клучниот збор **return** во Inger нема да се извршува. Ако постои такво нешто треба да се генерира предупредување.

- За да се направи оваа проверка се оди по АСД од озгора надолу и во секој блок се пребарува дали го има зборот **return**.
- Ако јазолот дете кој го содржи овој збор не е последен јазол, до остатокот од кодот нема да се стигне (недостиглив)

Пример за недостиглив код

```
start main : void -> int
{
    int a = 8;
    if ( a == 8 )
    {
        print ( 1 );
        return( a );
        print ( 2 );
    }
}
```

Разни семантички проверки –клучните зборови return

- Non-void функциите враќаат нешто: Последната наредба во било која non-void функција мора да биде клучниот збор '**return**' со цел да се врати некоја вредност. Ако го нема тој збор треба да се генерира предупредување (во Inger се генерира предупредувањето `control reaches end of non-void function').

- Функциите кои враќаат вредност но не прават ништо можат да направат проблем во извршувањето на програмата.

Пример Non-void функција која враќа вредност

```
module functionreturns;
start main : void -> void
{
    int a;
    a = myfunction();
}
    myfunction : void-> int
    {
        int b = 2;
    }
```

- Овде ќе се генерира предупредување, затоа што нема каква вредност да и се додели на променливата a

Разни семантички проверки – дупликат случаи

- Синтаксички е точно ако во код за **case** се случи да има двапати иста вредност.
- Семантичката проверка во Inger генерира предупредување ако се случи вакво нешто (може да враќа и грешка)

Пример (за дупликат case вредности)

```
module duplicate cases ;
start main : void ! void
{
        int a = 0;
        switch( a )
        {
           case 0
          {
            printf ( "This is the rst case block" );
          }
           case 0
         {
            printf ( "This is the second case block" );
         }
         default
          {
          printf ( "This is the default case" );
          }
         }
       }
}
```

- Алгоритамот оди рекурзивно по AST, почнувајќи од коренот и ги бара NODE SWITCH јазлите. За секој ваков јазол гледа дали вредноста која е тука веќе се јавила и ако е така ќе генерира предупредување

Разни семантички проверки – Goto лабели

- Користење на Goto наредбите честопати е штетно
  - Dijkstra ([3]) stated: премногу се примитивни и можат да направат каша
- Во Inger се имплементирани
  - Мора да се декларираат пред да се користат, но ова не може да се провери со синтаксичката проверка
  - Не е направено
- Проверката може да се направи со тоа што декларациите би се собирале во симболната табела
- Дали некоја повикана лабела е веке декларирана ќе се пребарува во АСД