ТЕХНИЧЕСКИ УНИВЕРСИТЕТ
ВАРНА

Софтуерни и интернет технологии

Програмният език Yat

Курсов Проект
по Езикови Процесори

Изготвили:
Александър Р. Георгиев
61662147 СИТ
4 курс 2б група

Даниел А. Атанасов
61662148 СИТ
4 курс 2б група

Проверил:
доц. д-р инж.
Ивайло П. Пенев

Съдържание

1 Описание на проекта

Целта на проекта е да се изработи език за програмиране от високо ниво, върху основата на LLVM, който да може да се компилира до машинен код.

LLVM е съвкупност от технологии спонсорирана от Apple, Microsoft, Google, ARM, Sony, Intel и AMD.

Фигура 1: Лого на LLVM.

Езикът е строго типизиран и поддържа целочислена и реална аритметика, функции, цикли и условни преходи.

Указатели и масиви не се поддържат, а граматиката на езика е базирана на C.

Езикът е описан от контекстно-независима граматика, не е двусмислена и се възползва от възможностите на избрания алгоритъм за парсване - LALR(1), който поддържа лява рекурсия.

Фигура 2: Лого на програмният език Yat.

1.1 Граматика на езика

```
Function -> Identifier Identifier ( ParameterList ) Scope

ParameterList -> ParameterList , Parameter
ParameterList -> Parameter
ParameterList -> %empty

// type name
Parameter -> Identifier Identifier

StatementList -> StatementList Statement
StatementList -> Statement

Statement -> Declaration
Statement -> Expression
Statement -> If
Statement -> For
Statement -> While
Statement -> Scope
Statement -> Ret

Declaration -> Identifier Identifier ;
Declaration -> Identifier Identifier = Expression ;

Expression -> Expression || And
```

```
Expression -> And

And -> And && Equality
And -> Equality

Equality -> Equality == Sum
Equality -> Equality != Sum
Equality -> Sum

Sum -> Sum + Product
Sum -> Sum - Product
Sum -> Product

Product -> Product * Call
Product -> Product / Call
Product -> Call

Call -> Call ( ArgumentList )
Call -> Term

Term -> Identifier
Term -> Number
Term -> ( Expression )

ArgumentList -> ArgumentList Expression
ArgumentList -> Expression

If -> if ( Expression ) Statement
If -> if ( Expression ) Statement else Statement

For -> for Identifier in [ Expression .. Expression ] Statement

While -> while ( Expression ) Statement

Scope-> { StatementList }

Ret -> ret Expression
```

2 Лексичен анализ

2.1 Терминални символи

Сред терминалните символи има шест ключови думи: `float`, `i32`, `void`, `if`, `else` и `ret`.

Останалите символи са стандартни оператори за артитметични и булеви операции, разделители за редове и блокове, и идентификатори.

2.2 GNU Flex

GNU Flex е генератор за бързи и лесни за поддържане лексични анализатори, създаден през 1987 година, базиран на Lex. Входният файл на flex описва гореописаните терминали под формата на регулярни изрази. Освен тях се опистват и едноредови и блокови коментари, като основната разлика, е че те се игнорират.

Тъй като регулярните изрази за могат да са двусмислени, flex ги проверява в реда в който са дефинирани. Това означава, че израза за идентификатор не трябва експлицитно да изключва ключовите думи, стига да е дефиниран след тях.

Тъй като регулярните изрази поддържани от flex са минималистични и не поддържат често срещани разширения, блоковите коментари не могат да се дефинират лесно само с един израз. От друга страна, това означава, че се могат бързо да се проверят посредством краен детерминистичен автомат.

При намирането на начало на блоков коментар (/*), се преминава в друго състояние на flex, в което се търси само края (*/), а всичко друго се игнорира. При намиране се връща обратно в началното състояние. Това не се прави в парсера е за да не се увеличава размера на парсерните таблици, които растат експоненциално с броя различни терминални/нетерминални символи.

3 Граматичен анализ

3.1 Описание

С някои изключения, езикът се парсва както е описано в граматиката. В останалите случаи се прави допълнително разделение на нетерминал-

ните символи с цел лесно структуриране на изходния код. Главната причина е предварително да се раздели езика на блокове, които отговарят на условията на LLVM - всеки блок да съдържа точно една инструкция за преход (br/ret) и тя да е последната инструкция от блока.

3.2 GNU Bison

GNU Bison е генератор на бързи таблични bottom-up парсери, създаден през 1985 година и базиран на Yacc. Bison поддържа три алгоритъма за парсване: LALR, LR и GLR. LALR е подмножество на LR, което е достатъчно за нашият език, а GLR се използва главно ако LR не е достатъчно.

Входният файл на bison описва граматиката на езика с гореописаните промени. За да може да се изгради AST, всеки нетерминален символ е дефиниран като C++ промелнива с определен тип, а за всяко правило има действие, което трябва да инициализира лявата си страна когато бъде намерено.

Ключовите настройки на bison са опцията за генериране на C++ код, вместо C и използването на `std::variant<...>` вместо union... за вътрешен тип на парсера. Това позволява използването на типове с нетривиални деструктори.

За унифициране на типовете на подобните конструкции в AST-то ни използваме `std::variant<...>` за да избегнем проблеми с алокиране и деалокиране на памет, които биха възникнали ако ползвахме стандартен полиморфизъм;

4 Генериране на изходен код

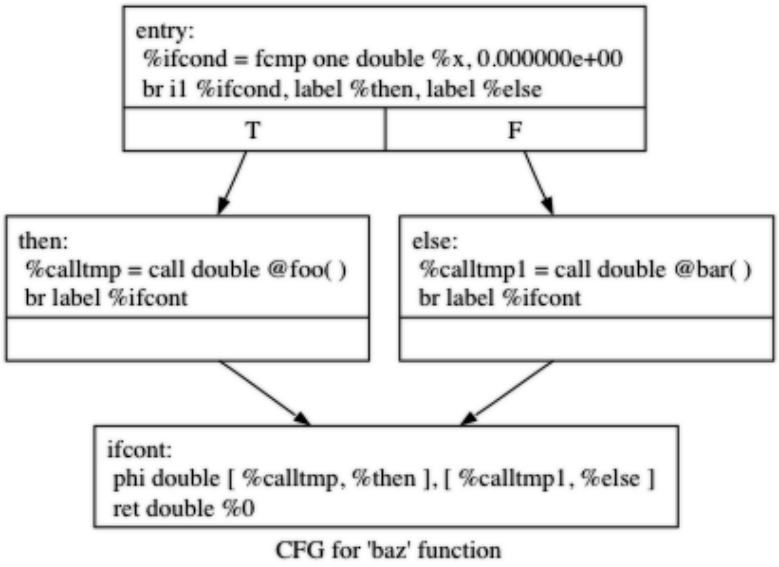
4.1 Предварителна обработка

Преди да започне генерирането на код се проверява валидността на кода. Това е сравнително просто, понеже повечето проблеми се обработват по време на граматичния анализ, а езика не поддържа сложни типове или оператори. Единственият частен случай е когато операндите в една операция се различава, при което целочисленият се преобразува в реален. Булвият тип автоматично се преобразува в целочислен когато е нужно.

4.2 LLVM

Генерирането на кода става при обхождане на синтактичното дърво, като блоковете се обхождат два по два понеже всеки блок трябва да знае кой е следващия. Това е нужно защото вътрешен блок трябва да знае къде да отиде когато стигне до последната си инструкция.
Поддържат се някои инструкции по подразбиране:

- Добавя се преход в края на блок в зависимост от колко блока има след него на същото и предишното ниво.
- Преобразования между примитивни типове(`bool->i32->float`).
- Създаване на нови блокове при разклонения.



Фигура 3: Графични представяне на условен преход посредствон LLVM IR блокове.